Systems Analysis & Design in an Age of Options

Edition 2.0

Deeper Dives

Gary Spurrier

Bentley University

Heikki Topi

Bentley University

Prospect Press

Founded in 2014, Prospect Press serves the academic discipline of Information Systems by publishing essential and innovative textbooks across the curriculum, including introductory, emerging, and upper-level courses. Based in Burlington, Vermont, Prospect Press distributes titles worldwide. We welcome new authors to send proposals or inquiries to Beth.golub@prospectpressvt.com.

Editor: Beth Lang Golub
Associate Editor: Dave Williams
Production Management: Peter Holm, Sterling Hill Productions
Copyeditor: Carl Quesnel, Assurance Editorial
Cover Design and Illustrations: Annie Clark

eTextbook: supplement to: ISBN: 978-1-958303-19-1
Available from Redshelf, VitalSource, and Perusall

For more information, visit https://www.prospectpressvt.com/textbooks/spurrier-systems-analysis-and-design-2-0.

CONTENTS

2. Initial Visioning and Business Analysis

2.5 Deeper Dives: Advanced Topics Related to Initial Visioning and Business Analysis 481

3. Conceptual Data Modeling

3.5 Deeper Dives: Advanced Topics in Conceptual Data Modeling 498

4. User Stories and User Interface Models

4.5 Deeper Dives: Advanced Topics Related to Business Analysis and Use of Its Results..... 506

5. Use Case Narratives and Functional Testing

5.5 Deeper Dives: Advanced Topics in Use Cases and Testing..... 512

6. Designing the User Experience and User Interfaces

6.5 Deeper Dives: Advanced Topics in Designing User Experience and User Interfaces 518

7. Systems Development and Resourcing Approaches

7.5 Deeper Dives: Advanced Topics on System Acquisition 523

8. System Cost Estimation

8.5 Deeper Dives on Advanced System Estimating Topics 530

9. Business Benefits Estimation and Cost/Benefit Analysis

9.5 Deeper Dives: Advanced Benefits Estimation and Cost/Benefit Analysis Topics 549

10. Project Approach Selection

10.5 Deeper Dives: Advanced Project Approach Selection Topics 565

11. Feasibility Analysis, Statement of Work, and Business Case

11.5 Deeper Dives: Advanced Topics on Project Planning and Approval..... 576

12. Up-Front Project and Release Planning

12.5 Deeper Dives: Up-Front Project Management and Release Planning..... 580

13. System Architecture

13.5 Deeper Dives: Advanced Topics on Architecture..... 584

14. Technical Design of Data and Logic

14.5 Deeper Dives: Advanced Topics in Technical Design and Implementation 592

15. Leading Iterative Systems Development

15.5 Deeper Dives: Advanced Iterative Development Topics..... 609

16. Deployment

16.5 Deeper Dives: DevOps Deployment in an Age of Options 616

Initial Visioning and Business Analysis

2.5 Deeper Dives: Advanced Topics Related to Initial Visioning and Business Analysis

In this section, we do deeper dives on several advanced topics that all build on and expand coverage from earlier in this chapter:

- Deeper Dive 2.1 compares and contrasts plan-driven and hybrid versions of the Systems Development Process Framework, including noting their relationship to agile concepts. It provides a more in-depth context for the initial visioning and business analysis activities.
- Deeper Dive 2.2 describes the history and context of the Unified Modeling Language.
- Deeper Dive 2.3 expands the brief representation of how IT solutions can help organizations achieve their goals.
- Deeper Dive 2.4 presents a conceptual connection between systems analysis and design processes and organizational strategy, business models, and tactics.
- Deeper Dive 2.5 introduces additional, more advanced, UML activity diagram modeling constructs.
- Deeper Dive 2.6 discusses the role of AI-based components in organizational information systems.
- Deeper Dive 2.7 presents an example that illustrates how generative AI can be used to support a BA's work.

These sections are independent of each other, but they assume earlier coverage of the related fundamental topics in this chapter.

2.5.1 Deeper Dive 2.1: Contrasting Plan-Driven and Hybrid Versions of the Systems Development Process Framework

In Chapter 1 we introduced the plan-driven approach to systems projects, portrayed in Figure 1-16, and in this chapter, we described a hybrid approach, portrayed in Figure 2-1. In this Deeper Dive, we explain the similarities and differences between the two approaches.

We start with a quick review of the plan-driven approach. In short, in the plan-driven approach, the BA formally models all system requirements up front, then hands them to developers to program, test, and deliver to the client. This is very similar to how a building architect creates blueprints, then hands them over to a construction crew. In fact, when developing new system features, we call the programming phase "software construction."

The plan-driven System Development Process Framework portrayed in Figure 1-16 shows a linear, "one thing at a time, with each thing done once" approach. This is simple and logical. However, unfortunately, it's also a path to failure in many systems projects. Why is that? In most systems projects, you can't fully understand the requirements up front. They're not completely clear and/or they may change as the project executes; they're somewhat unstable. This results in you delivering a system that may meet the documented up-front requirements but doesn't meet *actual client needs at the time of system delivery.* This outcome is so common that it is sometimes called the "Yes, but…" syndrome. Specifically, the IT team delivers a system to a client, who, once they start using the system, finds that it isn't working for them in important ways. In effect, the client says, "Yes, the system meets the up-front requirements, but it doesn't meet my actual needs now." The

upshot of this is that the project either fails or has to be restarted to make additional changes, costing more time and money. Neither of these outcomes is a good thing.

Because of these problems with the plan-driven approach, the IT community has worked hard since the 1990s to come up with new, more effective approaches. One of those approaches is agile, essentially a complete rejection of the plan-driven approach. In agile, the IT team documents only a few, informal requirements up front—typically user stories—and then quickly starts development, even though they know the requirements are highly incomplete. The idea here is to determine the detailed requirements by building working software for a short period of time—typically one to four weeks—and then collecting client feedback on that software. The client feedback provides more requirements, which the team uses to build another, more complete version of the software, again programming for a short period of time. This iterative "build a little, revise a little" process occurs again and again in the agile process. Each iteration is called a "sprint." Over the course of multiple sprints, the software becomes more and more complete and mature, and each version of the software targets the client's current needs based on their ongoing feedback. As such, the agile approach emphasizes requirements that *emerge* as the project builds software in sprints. In this way, agile emphasizes flexibility and responsiveness to client needs.

The agile approach has gained significant popularity, with hundreds of books written about it. Despite this, most systems teams working in large, complex environments don't actually use the agile approach, at least not in its "pure" form, with minimal up-front requirements. Instead, these teams find that agile doesn't do *enough* up-front requirements work. The features needed are too complex to build without careful planning. Also of great concern, when enhancing an existing system, the agile approach tends to break the existing software code.

So, in many circumstances, neither "pure" plan-driven nor "pure" agile approaches work well. What to do? Most teams employ an approach that combines aspects of plan-driven and agile: a hybrid approach. In hybrid, you do a lot of formal, up-front requirements work. However, you *also* preserve flexibility and responsiveness to changing requirements by then building the software in sprints, including collecting client feedback at the end of each sprint. This is the approach shown in Figure 2-1. In this way, you do enough up-front planning to support complicated requirements and avoid breaking the existing code base, while avoiding the "Yes, but . . ." syndrome by being able to revise requirements as the project executes.

What does this mean for initial visioning and business analysis? First, note that the top of Figures 1-16 and 2-1 are highly similar. Initial visioning and business analysis work in a highly similar way in both plan-driven and hybrid projects. The main difference is that, in hybrid, you acknowledge that the requirements are likely to evolve to some degree during sprint-based software construction. This is the difference you see at the bottom of Figures 1-16 and 2-1.

Does this mean that there's no place for the agile approach? Not at all. The agile approach is highly appropriate in certain circumstances. For example, when requirements are highly unclear and/or changing very rapidly, it makes little sense to do significant up-front requirements. You'll do some initial visioning, but much less business analysis. For example, agile is typically used in start-up companies implementing new business ideas or when creating a new, small, simple system in an existing organization. We'll introduce the agile version of the Systems Development Process Framework in Chapter 10, discussing how to choose the optimal project approach (plan-driven or hybrid or agile) for any given project. By comparing all versions of the framework, you'll find that all project approaches will need to include a certain level of planning and specification of desired changes in how the business is conducted. Process modeling is an essential tool for that purpose.

2.5.2 Deeper Dive 2.2: Unified Modeling Language (UML)

Activity diagrams are just one of several specific modeling techniques you'll learn in this book that are part of the **Unified Modeling Language (UML)** family of tools. UML consists of an integrated set of graphical modeling techniques divided into two categories:

- **Structural models:** There are seven structural diagram types and seven behavioral diagram types. Structural models describe the static characteristics of the area of interest, addressing questions such as "What are the key concepts and their relevant characteristics?", "How are these concepts related to each other?", and "How is the system organized?" For example, a structural model would, in the context of the ValueForAll (VFA) retail store, include concepts such as Customer, Product, Sale, and so on. A structural model would also cover the attributes (such as customerNumber, customerFirstName, customerLastName, etc.) and relationships (a customer may, over time, purchase multiple products in multiple sale events) of the concepts. The class diagram is one of UML's most important structural diagram types. We'll cover the class diagram in detail in Chapters 3 (for real-world models) and 14 (for technical design).
- **Behavioral models:** These models describe what actions and activities take place. Activity diagrams are one of the most important behavioral model types. As you saw earlier in this chapter, a behavioral model can, for example, be used to specify the steps of the purchase process in a store like VFA. Our first example described it from when a customer enters a retail store to when the customer exits with purchased and paid goods.

UML is important because it provides standard approaches to SA&D modeling. Prior to the 1990s, several different modeling experts created and pushed different, incompatible modeling approaches. To resolve this, UML emerged in the 1990s via a compromise agreement between three well-known modeling approach authors: Grady Booch, Ivar Jacobson, and James Rumbaugh, who all worked for the same company (Rational Software) in 1995. In a nutshell, each expert had created their own set of modeling techniques, and the IT industry needed greater standardization.

UML is intended to be a "graphical language for visualizing, specifying, constructing, and documenting the artifacts of distributed object systems." UML is governed by an organization called the Object Management Group (OMG). The first OMG version of the language was 1.1, published in 1997, and the current version (2.5.1) became available in 2017. As a compromise, UML is an imperfect and incomplete set of requirements standards. While it provides several helpful modeling tools, it also is lacking in many essential areas. The missing elements include user interface modeling, security models, detailing low-level business logic, nonfunctional requirements, and others, all of which you'll learn about later in this book using non-UML techniques. Also, many UML diagram types are seldom used. In this book, we focus only on those that are widely used. Still, where UML does provide guidance, it has established its role as the de facto standard of graphical modeling languages.

Note that the same UML diagram type can be used for multiple purposes. Specifically, structural and behavioral diagrams can describe either the real world (the domain of interest) with context models or the hardware and software that enable the real world to operate using an IT solution with design models. Though there certainly is a connection between the two, they still are clearly different. A real-world customer is not the same as the representation of that customer in a data store. A real-world business document (say, an invoice or an insurance claim) doesn't behave in the same way that its representation in a software system might.

2.5.3 Deeper Dive 2.3: Positive Effect of IT on Organizational Performance

This section expands on Section 2.1.3, providing a more in-depth look at how IT solutions can positively affect organizational performance. In Section 2.1.3, you started your SA&D process at the level of a specific project. While this is valid, you need to understand that many large organizations use higher level planning processes intended to align different IT projects with the organization's overall strategy. Together, these aligned systems projects create an overall plan, or roadmap, for IT investments.

IT solutions aligned and integrated with organizational strategies and processes can serve an essential role in increasing organizational output and reducing organizational costs. Systems solutions can positively affect organizational performance in four primary ways:

- **Increase top-line revenues or other key results:** IT solutions can help a for-profit business in different ways to increase revenues. For one, IT could enable it to sell more units of output. For example, a chain of retail stores could implement a retail sales website in addition to its brick-and-mortar stores. Or IT could increase its capability to charge more per unit. For example, a car manufacturer could increase revenue by selling more cars of the same type or cars with more features and higher quality at a higher price point. Also, a company could find access to entirely new markets (or create entirely new business opportunities) with the help of IT. This is particularly true given the global communication infrastructure that the internet provides. Prominent examples include firms that have used IT for online platforms that enable individuals to sell goods and services via the internet, including eBay and Etsy for tangible products and Airbnb and Vrbo for rental accommodations.

  For noncommercial enterprises, different measures might be relevant in this same category. For example, a government healthcare clinic might consider patient care outcomes as its key output. Even a for-profit company could strive to improve nonrevenue output measures, such as, for example, customer satisfaction. Many organizations have created entirely new business models by innovatively combining technology solutions and human contributions. Think about Amazon. Over its history, it has increased top-line revenue by dramatically increasing its volume (number of units sold), broadening the scope of goods it sells and services that it offers, and expanding to new businesses (such as its AWS cloud service that is currently very important for Amazon's profitability). IT-based solutions enable all these gains.
- **Increase bottom-line profits:** The organization can also increase efficiency by decreasing the costs needed to create the desired output. IT can also reduce resource costs in several ways: by negotiating better prices with suppliers, using fewer hours of labor through automation, reducing rework by increasing initial quality, lowering component costs by changing the design of the product, and so on. Decreasing costs by improving organizational processes is one of the oldest and most widely recognized ways to use IT to increase organizational performance. For example, some of the earliest well-known case studies on the use of IT were about speeding up **business processes**. An organization might also want to reduce nonmonetary costs of production, such as environmental pollution or employee turnover. PwC describes a large insurance company that gained annualized savings of $250 million through an IT-driven expense reduction program (PwC, 2016).
- **Stay in business:** In many cases, organizations can no longer survive without IT solutions. The systems have become a necessity. Bill Gates, founder of Microsoft, once famously said, "If your business is not on the internet, then your business will be out of business." So, the third fundamental way of supporting organizational performance is ensuring *existing* systems can operate effectively without interruption. This could lead systems projects to replace obsolete technologies or add functionality mandated by law (such as calculating new sales taxes). Fixing security vulnerabilities or updating the system to enable continued business with buyers or suppliers would be in the same category.

  In addition to increasing output and reducing resources required to create output, IT-based solutions serve in a third role. Increasingly often, systems are a necessary resource for being able to do business in the first place. For example, many companies must fully integrate into various supply chains with their suppliers and customers. A large, powerful supplier or customer can effectively dictate to a smaller organization how its information systems need to be able to communicate with those of its supply chain partners. Imagine, for example, that your company is a vendor to Costco Wholesale, a global warehouse retailer. In practice, a large client such as Costco can specify the

rules for commercial electronic communication (electronic data interchange or EDI). If Costco changes its EDI requirements, its vendors have no choice but to comply.

- **Achieve social and environmental goals**. Increasingly, many organizations (businesses, government units, and not-for-profit organizations) have concluded that they want to (or must) support social and environmental goals. These goals exist either in parallel with their main objectives (such as financial profit) or as one of them. Information systems solutions frequently play an essential role in this process. These solutions may allow organizations to, for example, reduce energy or materials used in manufacturing. They can also improve the efficiency of transportation of goods and services, prevent premature failures of mechanical components through enhanced monitoring, etc.

Information Systems Can Serve Organizational Goals in Four Primary Ways

- ***Increasing revenues or other key results*** that are essential for organizational performance
- Decreasing the resources that are needed to produce the output (such as labor or parts), thereby ***increasing bottom-line profits***
- Satisfying a requirement set by external stakeholders (such as a government unit or supply chain partners) or expectations by customers simply to stay in business
- Helping organizations to achieve their ***social and environmental goals***

So far, we've been talking about IT supporting the goals of a business that is not itself primarily in the business of IT. Of course, the IT solution itself can be a direct source of revenue. For example, thousands of software vendors sell or rent their software systems. Prominent examples include enterprise resource planning (ERP) vendors that provide financial, payroll, and supply chain management systems to other firms. Alternatively, a software solution can be given away to organizations or consumers in exchange for charging for advertising. Google search and Facebook social media are significant examples of such an advertising model.

2.5.4 Deeper Dive 2.4: Process Innovation, Digital Transformation, and Business Strategy

This section aims to introduce the business context that will ultimately determine if the IT capabilities developed for the organization are truly valuable and aligned with the organization's goals. Throughout history, many terms have been used to describe the actions and processes a company or other organization can use to look for ways to transform itself and how it operates. For example, already in 1990, Michael Hammer, a well-known consultant, published a *Harvard Business Review* article titled "Reengineering Work: Don't Automate, Obliterate." In it, he stated that we should "reengineer our businesses: use the power of modern information technology to radically redesign our business processes in order to achieve dramatic improvements in their performance" (p. 104). Several books on the same topic followed a few years later, including Hammer and Champy's (1993) *Reengineering the Corporation* and Tom Davenport's (1993) *Process Innovation*. The authors disagreed about how extensive and fast changes should be (Davenport, 1993), but the fundamental idea was still the same. IT-based solutions can be highly effective mechanisms to improve organizational efficiency and effectiveness through revised business processes.

Business process A set of structured actions an organization executes to achieve a goal or set of goals required as part of the organization's way of doing business.

About a decade later, other related, widely applied mechanisms to address the same questions appeared. The names were different, such as business process management, modeling, and design (e.g., Becker, Kugeler, and Rosemann, 2013). Becker et al. (2013) start the preface of their book by stating simply, without hesitation: "Managing business processes is a necessity for every organization." More recently, digitalization and **digital transformation** have become widely used for discussing the same phenomenon. For example, Hess et al. (2016) write about digital transformation strategy. In that context,

Examples of Labels for Approaches to Changing Organizations with Information Technology

- Business process reengineering (1990s)
- Business process management, modeling, and design (2000s and later)
- Digital transformation (2010s)
- Artificial Intelligence-driven systems (2020s)

Multiple Levels at Which Information Systems Can Affect Organizations

- Business model
- Strategy
- Tactics
- Operations

they define digital transformation as a phenomenon "concerned with the changes digital technologies can bring about in a company's business model, which result in changed products or organizational structures, or the automation of processes" (p. 124).

Many experts see digital transformation as a broader concept than business process redesign or management. It also involves more substantive changes, such as those related to business models and organizational structures. In numerous cases, digital transformation creates products or services that, practically speaking, could only exist after the introduction of information technology. Capabilities such as social media, internet search engines, crowdsourcing funding, and widespread use of generative AI are just a few examples of such disruptive transformations.

Digital transformation
Using digital technologies to change an organization's business model, including new or modified products, structures, or processes.

A business model and changes in it are not limited only to businesses. For example, in their widely used book *Business Model Generation*, Osterwalder and Pigneur (2010, p. 14) define a *business model* as follows: "A business model describes the rationale of how an organization creates, delivers, and captures value." They also outline the nine component elements of business models, as specified in Table 2-1. This interpretation goes beyond just the processes (which would be included in key activities). The key idea is that digital transformation may change the entire organization's identity and relationships with its stakeholders.

Table 2-1: Business model elements (Osterwalder and Pigneur, 2010, pp. 16–17)

1. Customer Segments	2. Value Propositions
3. Channels	4. Customer Relationships
5. Revenue Streams	6. Key Resources
7. Key Activities	8. Key Partnerships
9. Cost Structures	

This chapter considers all these terms as pointing to the same fundamental idea: using information systems to transform how organizations work to achieve their goals. Ever-improving information technology capabilities constantly offer new opportunities to change how organizations operate to reach their goals most effectively. These new opportunities can sometimes be so significant that the changes are no longer related to improving current ways of operating. Instead, organizations must rethink which businesses they will stay in or out of. For example, Amazon started as an online bookstore but has exploded into vastly different businesses. For instance, through Amazon Web Services (AWS), the company has become a dominant pioneer in cloud computing.

Strategy addresses questions about the businesses the organization should be in (and stay out of). Strategic decisions are important, rare, and provide boundaries for tactical and operational decisions.

In other words, information technology capabilities combined with suitable types of organizational processes and structures create opportunities for strategic change instead of only tactical or operational change. **Strategic decisions** are typically finally approved at the highest level of the organization (Board of Directors, top executive management), have a long time horizon, and have significant operational-level consequences. They are important and rare and provide boundaries for future downstream decisions. Imagine a university that moves its graduate programs entirely to an asynchronous online delivery mode from its current on-campus programs. This decision is strategic in that it: (a) changes the type of business the university is in (from place-based specialized service to geographically unlimited commodity business), (b) is essential, and (c) is likely a rare decision. For Uber and Lyft, it has been a strategic decision from the beginning to focus on connecting those who offer rides and those who need rides. So far, both companies are very actively fighting the notion that they are in the business of offering rides, because that would create a variety of legal obligations they want to avoid. This decision is undoubtedly strategic. They couldn't have chosen it without understanding how to use IT capabilities to build a platform for connecting riders and drivers. It is particularly noteworthy how the emergence of global connectivity through the internet, cloud-based flexible availability of computing resources, and ubiquitous mobile clients has opened up new opportunities for small- and medium-sized businesses (SMBs). The scope of

Strategic decision
Long-term, rare, and important decisions about which high-level activities the organization will continue to be engaged in and which ones it will abandon.

these opportunities is much broader than what these companies traditionally could access.

It is also essential to see that strategy and a **business model** are different, although they are interconnected. An organization may apply other business models in different contexts as its strategic choice. A business model is a more generic concept than a strategy (Teece, 2010). Multiple organizations can select the same business model but apply it uniquely to different specific concepts. The business model articulates the logic of the business, the value proposition it offers to its customers, and the basic parameters of how it plans to achieve its overall goals. Strategy tells a story about how an organization will do better than its rivals. As Ovans (2015) describes, a strategy could involve either developing a new business model or applying the generic elements of a business model in a specific context. The same strategy can be pursued with many different business models. A business model is much more abstract than a business process. Executing a business model often requires multiple business processes, and the same business model can be implemented with various processes.

Business model
Articulation of the logic of the business and the value proposition that the business offers to its customers.

Tactical and operational decisions and actions are necessary to implement high-level strategy (see Figure 2-23). At the strategic level, an organization makes decisions regarding the businesses it should be in (and stay out of), asking questions about *what* it should do. At the **tactical decision** level, the organization must ask *how* it will achieve its strategic objectives (and, ultimately, its vision). Assume a company has a strategic goal of being a market leader in a critically important product segment within a region. It could employ one or several tactics to achieve this goal. The tactics could include creating a solid online presence, opening its own brick-and-mortar stores, or working to get products on the shelves of major existing retailers. Notice that none of these paths are goals alone; they are ways to achieve a higher level (strategic) goal.

Tactical decision
Determining how an organization will achieve its strategic goals.

Tactical decisions determine at a high level how the organization will reach its strategic goals.

Operational decisions determine how an organization acts in its day-to-day operations to achieve its tactical and strategic goals.

At the same time, these tactics are still at quite a high level. If, for example, the company decides to focus on creating an outstanding online presence as the mechanism (a tactic) for achieving its (strategic) goal, it still must execute the tactical plan. This process of execution requires many **operational decisions** and actions, leading ultimately to the detailed implementation of the company's online presence. As you can see, the strategy, tactics, and operations are all interconnected. The best strategy doesn't mean anything unless it is implemented through a set of successful tactical actions. Tactical actions don't happen unless the company takes appropriate operational actions.

Operational decision
Decisions related to everyday operations of an organization.

IT and IT-based systems are essential to all modern organizations' business models, strategies, tactics, and operations. It's hard to think of any goal-driven human activity today that would not in some way benefit from an IT-based system operationally (if it didn't use it already). In many cases, organizations don't have a choice. For example, their external stakeholders might require specific IT-based systems. Achieving a reasonable level of operational efficiency is often not possible without the use of a computing solution. Imagine a modern high-volume grocery store. Without an effective IT system, there's no way a store could determine the amounts customers owe, keep records of payments, and maintain records of its inventory at the required speed.

Most modern business models incorporate elements entirely dependent on IT, such as creating and maintaining channels, enabling key value propositions, and enabling core activities. Developing a functioning business model for home delivery of groceries was an elusive goal for many companies for a couple of decades after the emergence of the consumer internet in the 1990s. Colossal early failures did not prevent others from trying to reach the goal. All attempts have been heavily dependent on the use of IT. Recently, several established grocery chains (such as Whole Foods and Stop & Shop) have been successful in developing a functioning, sustainable home-delivery model.

What does all of this have to do with systems analysis and design, the critical topic of this book? Plenty, as you probably can already see.

- First, *any system an organization decides to develop should serve a purpose aligned with the organization's strategy* and chosen tactics. It should be operationally efficient, supporting all the actions the organization and its employees must take to implement the tactics.
- Second, *IT-based systems and innovative use of IT are highly flexible multipurpose tools that enable entirely new means of operation*. Many categories of IT-based capabilities have radically changed how businesses interact with their customers and produce value for them. These include, for example:
  - Enhanced reporting and querying capabilities of early data management systems
  - Large systems enabling sophisticated decision-making in the context of complex operations (such as scheduling and resource planning for airlines or underwriting of mortgages or insurance products)
  - E-business systems enabling companies to reach customers across all types of boundaries
  - Artificial intelligence–based systems that perform sophisticated expert analysis work better than human experts

  It is essential for an innovative organization to keep exploring new opportunities that new IT capabilities offer and to analyze the level at which they could affect the organization positively, *including new strategies and business models*.
- Third, an organization's strategy, business models, and tactics need to inform business analysis as one of the cornerstones of SA&D. Conversely, the results of business analysis need to inform strategy, business model development, and tactics.

As illustrated in Figure 2-23, organizational SA&D processes need to be aligned with the chosen strategy. (In this case, the organization has determined that business model option 2 fits in with the strategy.) The tactics and operational actions must follow the strategy and business model choice.

In the same way, an organization needs to make decisions regarding its systems development activities at multiple levels. Any organizational decision to develop or acquire IT-based systems requires resources (competent people, money, time). Those resources always have alternative uses. So, all SA&D activities must be justified based on strategic, tactical, or operational needs. Whether

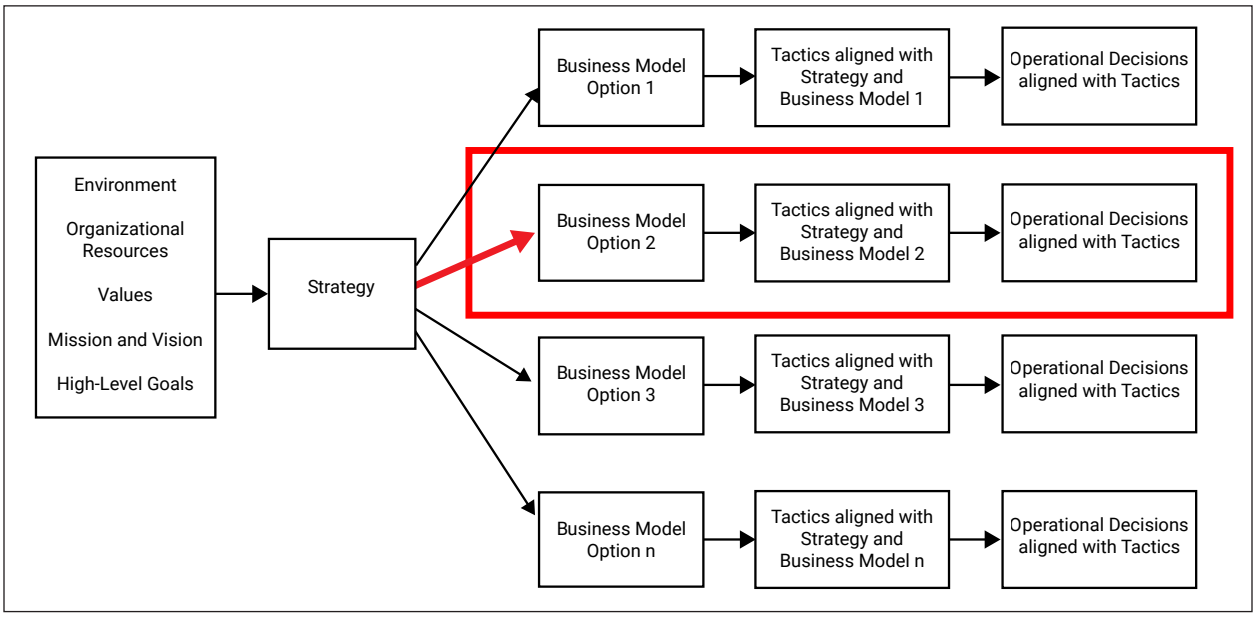


Figure 2-23 Strategy, business model(s), tactics, and operational decisions

the IT-related resource allocations are aligned with the organization's strategic direction and goals is determined at the highest level in the management of IT resources. Are the organization's available human resources appropriate for what it needs to accomplish? Does it have the correct type of IT infrastructure in place or accessible through some flexible (cloud-based) arrangement? Are its mechanisms for selecting the projects it will be working on guided by the organizational strategy? Leffingwell (2018) refers to these as *portfolio-level* decisions, providing overall guidance to the organization's IT investments.

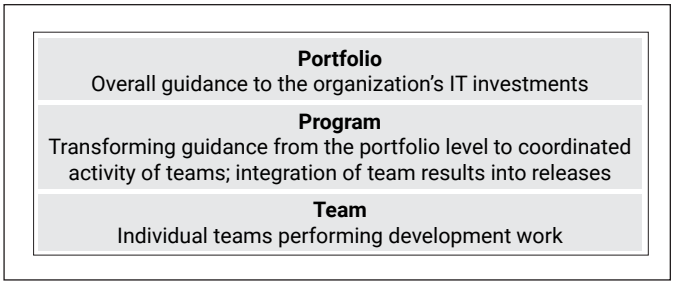


Figure 2-24 A simplified schematic representation of the Scaled Agile Framework (Leffingwell, 2018)

At the next level, the organization's IT development and infrastructure resources must be organized and guided to collectively produce results aligned with the portfolio-level guidance. Leffingwell calls this level the *program level*. Particularly in large organizations, a highly complex structure may be needed. This structure must convert the high-level directions from the portfolio level to detailed actions. These actions, in turn, allow the systems development teams to produce results that can be integrated into desired deliverables. In Leffingwell's model, the actual development work occurs at the *team* level. This three-level approach is called the Scaled Agile Framework, shown in Figure 2-24 (Leffingwell, 2018).

2.5.5 Deeper Dive 2.5: Using an Activity Diagram for Process Modeling—Advanced Structures

This section covers additional activity diagram modeling structures (see Figure 2-25), ones that are used less frequently than the foundational concepts introduced in Section 2.2.2 but provide essential tools for more complex modeling situations.

Two related concepts are an *object node* and an *object flow*. With an object node (modeled with a regular rectangle), it is possible to show the movement of a document or a set of data from one action to another. Beyond that, activity diagrams can include object nodes that belong to the data store category (marked with the <<datastore>> keyword, also called *stereotype* in the UML context). As the name suggests, data stores are IT-based structures that can store nontransient (i.e., permanent or persistent) data. In practice, modern information systems use databases to connect process elements and store nontransient data. Because of this and the large number of such data stores in most systems, in this book we tend to emphasize modeling data separately from activity diagrams. Instead, we model data with UML class diagrams (or a similar notation, entity relationship diagrams). Still, using a data store to represent this process structure is a valuable modeling tool in many cases. You can also show the movement of an object from one action directly to another.

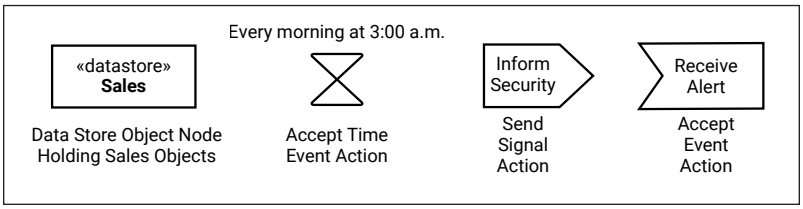


Figure 2-25 Advanced activity diagram notation

Figure 2-26 presents an example demonstrating object nodes and flows, including a data store. It shows a more detailed representation of the "Make Payment" action from the VFA example. (Note that pure object flows are marked with dashed arrows, but arrows representing both control and object flows are all solid.) It also highlights that activities can be presented with a hierarchical structure. This, in practice, allows you to decompose many actions into lower level activities. In this case, the example represents the internal structure of the "Make Payment" action at a much more detailed level. With this enhanced detail, we've reduced the level of abstraction from the earlier

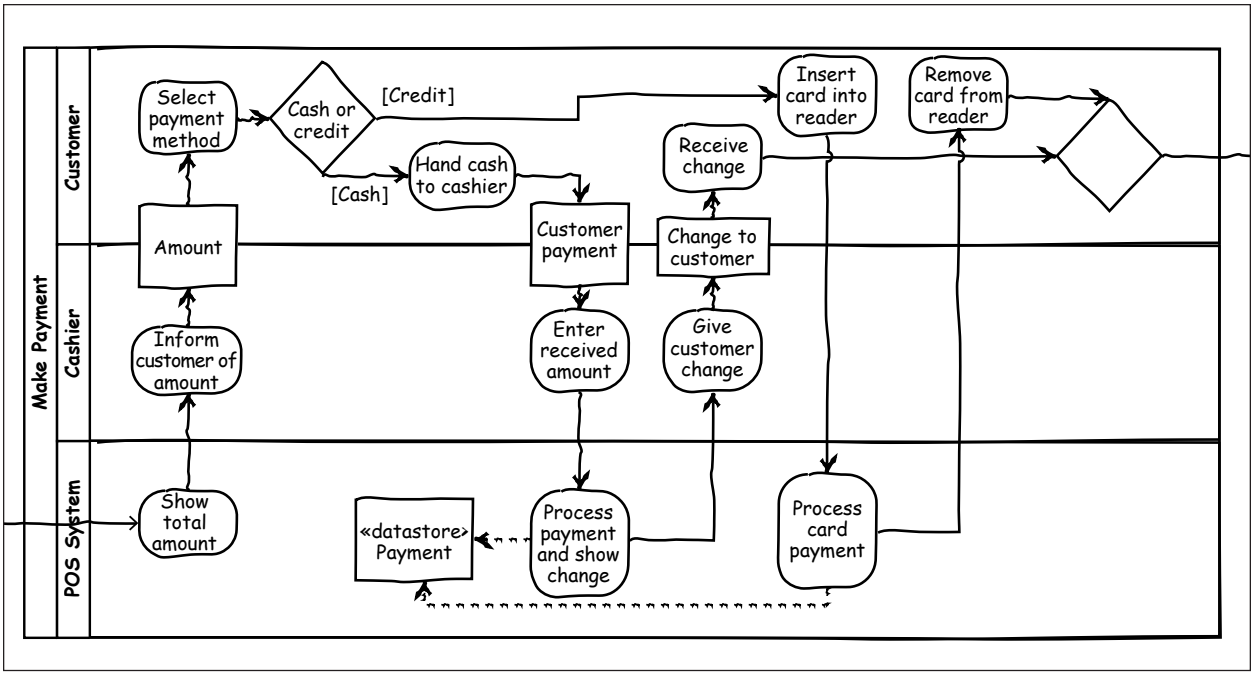


Figure 2-26 "Make Payment" activity diagram with object node and object flows added

model. It recognizes that "Make Payment" is not only an action of the customer; it also requires contributions from the cashier and the point-of-sale (POS) system.

Another vital activity-modeling construct group consists of nodes that send signals, react to (accept) signals, or respond to (accept) time events. These nodes can indicate that under certain conditions, a signal informs another action regarding a need to take action. Time events can be wait times or circumstances that occur regularly (every 30 minutes, once a day at a particular time, whenever certain data is received from another system, and so on). Time-related event triggers, in particular, are prevalent in real-world systems.

Each of these structures is included in Figure 2-27. On the top of the diagram is an *accept time event* symbol (the hourglass) with the notation "60 seconds." Together with the surrounding dashed line, it indicates that if the "Remove Card from Reader" task has not been completed in 60 seconds, it will be interrupted. The control will move to the cashier, to the "Ask for Customer Action" action. The cashier will ask the customer to remove the card, after which the control returns to the "Remove Card from Reader" task. Note that no mechanism is specified here to deal with the situation where a customer doesn't remove the card, despite numerous requests. This situation would need to be considered in an actual POS application. The diagram also includes a *send* signal event, in the case of a stolen credit card, and an *accept* signal specifying the procedures that security will follow when notified of a stolen card.

As you can see, the representation of the single "Make Payment" action in our simple example from earlier in the chapter has suddenly become increasingly complex. This happens relatively easily, and it is not a problem, per se. It is, however, essential that you, as a business analyst using activity diagrams as a tool, don't let their usability for their desired purpose suffer. This is particularly important if a specific model is intended for communication with users. The model should focus on the characteristics of the real-world situation that are relevant for the planned communication partners. A full specification of a complete business process is not always needed. The hierarchical structure of the activity diagram can be used to visualize multiple levels of detail. That way, an appropriate level can be chosen, depending on the situation. We hope you've noticed

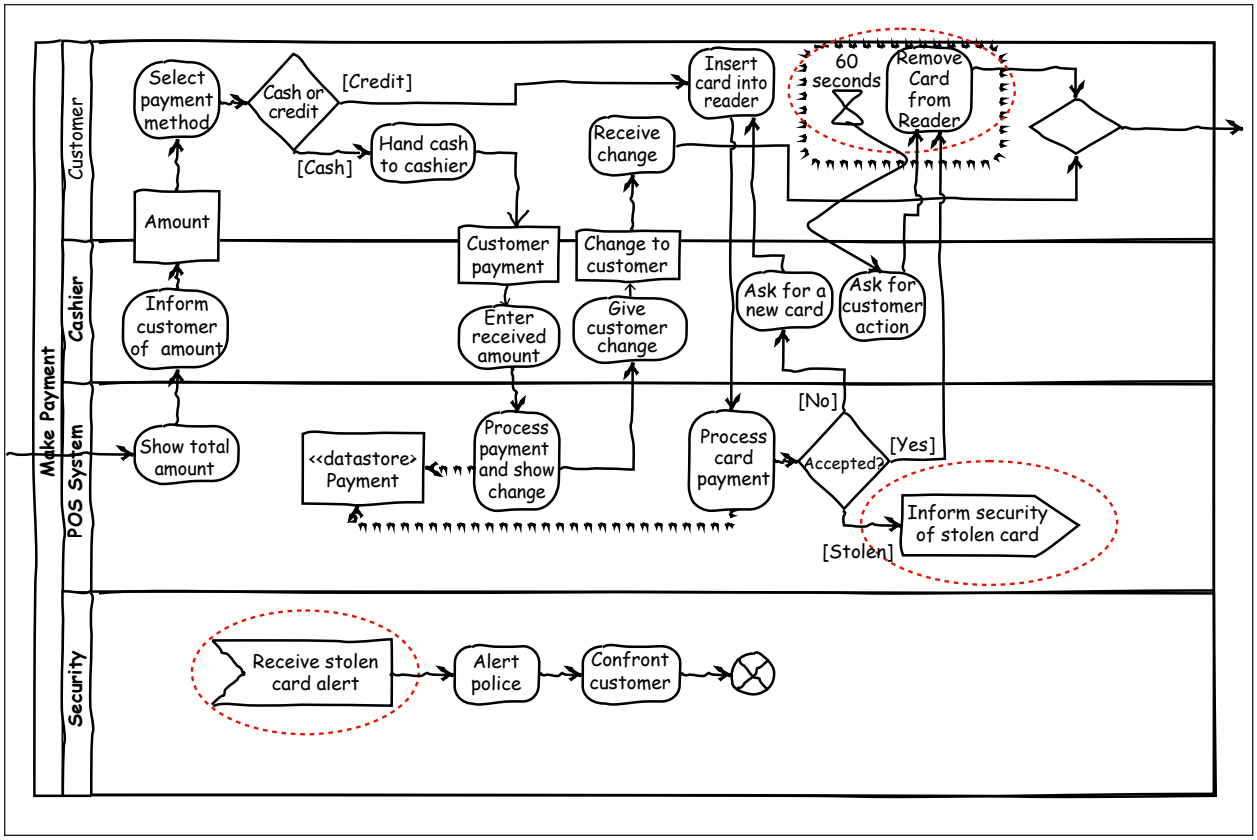


Figure 2-27 Activity diagram with signals added

how easily exploring the current processes leads to meaningful and essential questions about how the organization's business is currently conducted. At times, the findings also suggest that no current uniform process exists. This is a valuable outcome and an issue that the organization should actively rectify.

2.5.6 Deeper Dive 2.6: Analyzing AI Components as Part of an Overall Systems Solution

System components based on artificial intelligence (AI) technologies are increasingly valuable to organizations. But, for the BA, the potential that these capabilities offer raises key questions:

- When should you use AI to power system components (rather than conventional systems, meaning those programmed by human developers)?
- Will those AI-powered components replace entire conventional systems or work in tandem with them?
- What does a combination of AI and conventional systems look like?
- How does the use of AI-based system components impact your SA&D process?

The easiest way to answer these questions is to explore them via a software application example. We'll look at a property insurance company that must evaluate roof damage claims after hailstorms. When this happens, homeowners file claims that are stored in the company's insurance claims system. In response, the company must examine each homeowner's roof using an on-site inspector. Sometimes there will be damage, and sometimes there won't. When there's damage, the company must determine if it was caused by hail (meaning the company owes the homeowner payment for their claim) or by other problems that aren't the company's fault. For example, Figure 2-28 shows images of a variety of types of roof damage to composition shingles. The first image


Hail damage


Cracking from normal aging and weathering


Blistering from manufacturing or installation defects


Curling and cupping from normal aging and weathering

Figure 2-28 Images showing various types of shingle damage (From shutterstock.com/JLStock; iStock.com/RonTech2000; iStock.com/xphotoz; iStock.com/BanksPhotos)

shows hail damage, the only type of damage that the company would be liable for, while the remaining images show damage from other causes: improper roof installation, shingle manufacturing defects, and simple weathering and aging over time.

The current state of the Property Insurance Roof Damage Evaluation process, which we abbreviate as epic user story ROOFEVAL, is shown in Figure 2-29. Note that any given town may go a long time between major hailstorms, although smaller storms may damage a few homes at a time more frequently. The upshot of this is that the company doesn't keep a large number of inspectors on hand in any given town. When claims do come in, an insurance adjuster in the office arranges for an inspector to visit impacted properties on-site. Making these visits happen can be slow and inefficient for several reasons:

- Human inspectors are expensive, and the houses with claims may be far apart. As such, the insurance claims system is programmed to, in between major storms, send a list of claims by postal code only once a week (letting claims accumulate into a pile) or when there are three or more new claims in a single postal code.
- Also, we only inspect houses where claims have been filed and in the order the claims come in. So two houses sitting right next to each other might be inspected on different days, depending on when each homeowner decides to file a claim, increasing travel time and expenses.
- When large hailstorms do hit, the local inspectors may be overwhelmed, and the company may have to call in inspectors from other states to travel to the impacted area, taking more time and costing more money.
- Last but not least, examining a house's roof requires an inspector to climb a ladder onto and then walk across the roof—a slow and dangerous process.

The company believes that there are opportunities for AI-controlled drones to improve the process. First, drones could fly to impacted areas more quickly and with higher inspection capacity.

Second, drones could be dispatched to claim sites automatically. Third, rather than waiting for each individual claim to come in, drones could automatically evaluate the roofs of all customers within, say, 100 meters of the roof of a claimant. Finally, drones could eliminate the danger human inspectors face. These and other opportunities for improvement we'll discuss are shown on the right of Figure 2-29.

The company also believes that separate AI-based capabilities could be used to automatically categorize roofs: no damage, hail damage, damage from improper roof installation, etc. This image categorization capability is a familiar one. It is the same type of capability used, for example, to categorize photos as containing cats or dogs. Importantly, these types of systems are *not programmed*, because there's no easy way to create an algorithm in a conventional programming language to accomplish that. Instead, image categorization systems are *trained* by feeding a large number of already-categorized images into the AI-based module. This module then attempts to sort the training image set into the correct categories. In the first training run, the module will likely do a very poor job. But after each run, the module is told which images it categorized correctly or incorrectly. The module then adjusts itself for the next run. Over time, such modules can learn to be highly accurate in their categorizations.

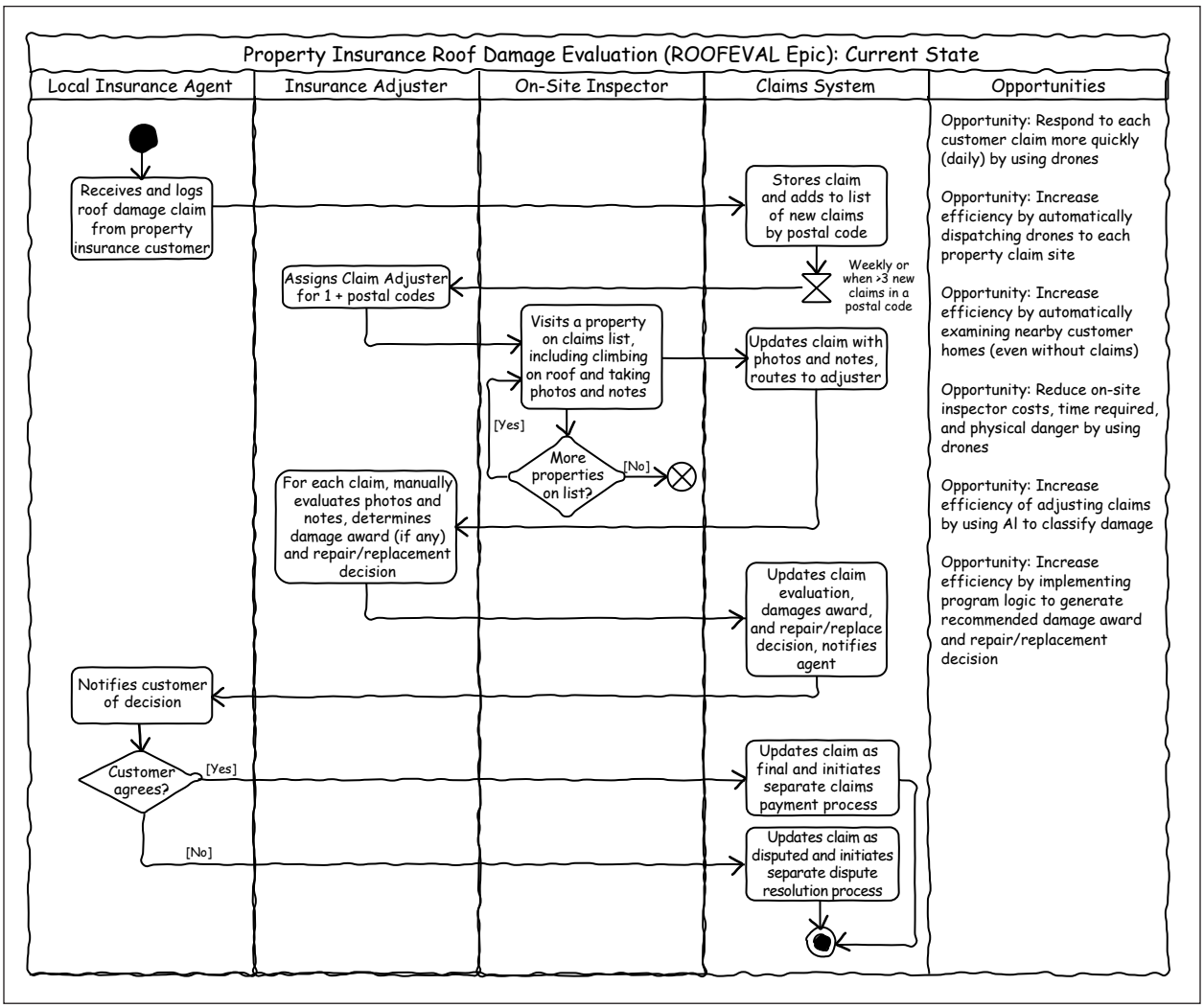


Figure 2-29 Current state activity diagram for the roof evaluation process

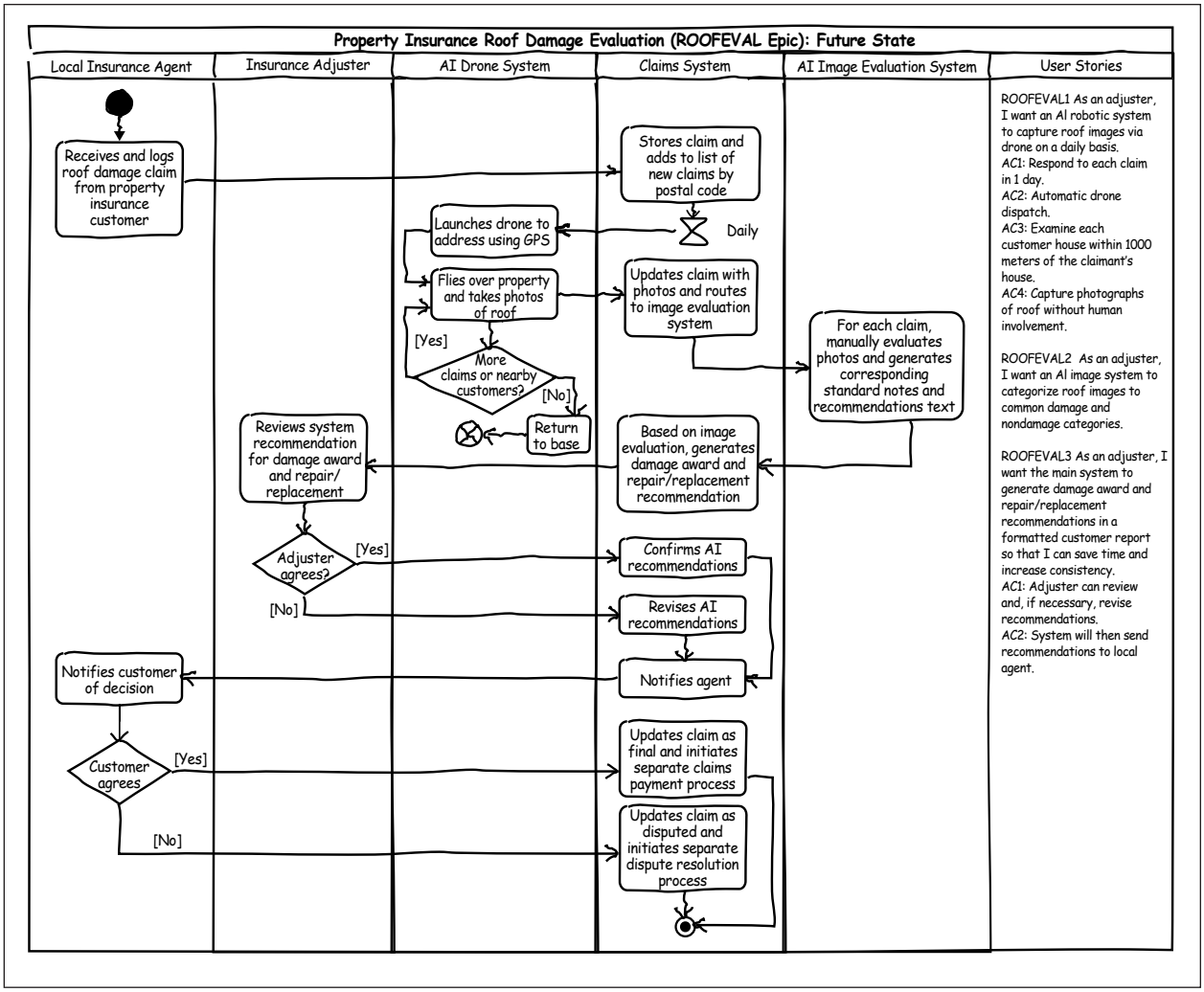


Figure 2-30 Future state activity diagram for the roof evaluation process

Still, it is important to realize that even the most accurate AI-based image evaluation systems occasionally make mistakes. So, in situations like this, where a misclassified insurance claim could be a serious problem for both the company and the claimant, you retain a human insurance adjuster in the loop to correct machine categorization mistakes.

Finally, the company realizes that its current claims system could be programmed with new features to automatically perform more of the insurance claim damages calculations and drafting of the report to the claimant.

The improved future state business process is shown in the activity diagram in Figure 2-30. As we've done previously, the future state is modifying a copy of the current state diagram. In this diagram, we converted the opportunities from the current state to features expressed as user stories and acceptance criteria.

Considering the questions posed at the start of this subsection, what have you learned? Key insights include:

- **Use AI-based system capabilities for problems that are difficult for humans to program:** There's no obvious way, for example, for a human developer to write code to

categorize roofing shingle images. And autonomous vehicles like drones do use some conventional coding but also require advanced AI capabilities.

- **AI-based capabilities won't replace conventional systems:** In our example, you still need the original claims system to capture claims data and images and to process claims data. In fact, the user story to automate generating damage award and repair/replacement reports would likely be implemented via conventional, human-programmed enhancements to that system.
- **AI-based capabilities and conventional systems will work together:** Each type of system will provide functionality that it is uniquely suited to. Note, however, that the application architecture, by which we mean the individual systems and how they interact with each other, in the future state is significantly more complex than in the current state (that is, the current state just has the claims system, while the future state adds two AI systems).
- **AI-based capabilities and conventional systems can use the same SA&D tools and techniques:** While AI systems are quite different from a conventional system in how they function, you were still able to use the same activity diagrams and user stories to understand the requirements. Adding or improving activities in system swim lanes identifies a user story. On the other hand, you did have to remember that AI systems can make mistakes (for example, miscategorizing shingle images) so you need to be sure to include human checks for AI mistakes.

As we go through additional chapters, we'll explore other types of AI systems that can be used to address problems that are difficult to solve with conventional systems. We'll also see how those AI systems can be integrated into overall application architectures to improve business processes.

2.5.7 Deeper Dive 2.7: Using AI Capabilities to Support BA Work

In addition to offering exciting new capabilities for organizational systems, AI has the potential to support the work of business analysts directly. For example, BAs can use generative AI systems such as ChatGPT or Bard to analyze materials that they have collected from various stakeholders and provide useful drafts of different requirements documents. In these types of roles, these AI tools act as potentially useful but at times error-prone assistants whose work you must evaluate and verify carefully.

In this section, we'll review an example that demonstrates one possible type of generative AI use by a BA. In this example, we used ChatGPT to create a vision statement based on a set of meeting minutes. These minutes are available on the book's website. The request to ChatGPT was as follows:

> Based on the meeting minutes below for the Wayback Public Library, generate a project vision document that specifies key stakeholders, organizational problems and opportunities, key system capabilities and project deliverables, expected business benefits, and a definition of project success. Here are the minutes: . . .

The resulting project vision document is included in Figure 2-31. Here are key observations about this draft and its implications for BA work:

1. **Useful first draft:** The ChatGPT-produced vision statement is reasonably aligned with the meeting minutes, including summarizing those minutes into the project vision categories. This demonstrates the potential for a generative AI tool to be a helpful, productivity-enhancing partner for the BA.
2. **But imperfect and needing human revision:** The version that ChatGPT created is far from perfect and needs to be carefully reviewed and edited by a BA. To demonstrate the differences between a ChatGPT-generated draft and a professionally finished project vision document, please compare Figures 2-31 and 2-14.
3. **A BA had to organize and conduct the requirements meeting and then create the meeting minutes:** A key point is that ChatGPT couldn't have organized and conducted

Project Vision Document: Library Information System for Wayback Public Library

1. Key Stakeholders:

- Wayback Public Library (WPL) organization
- Library patrons (members)
- Librarians (including head librarian, branch librarians, and other staff)
- Capstone team (development and implementation)

2. Organizational Problems and Opportunities:

- Manual, inefficient processes for checkout, check-in, and item management.
- Expired patron library cards not being checked during checkout.
- Inefficient collection of overdue fines and inability to identify patrons with outstanding fines.
- Difficulty in enforcing maximum item checkout limit.
- Inconsistent and error-prone checkout process.
- Lack of scanning and automation in the check-in process.
- Inefficient data collection for returns and item tracking.
- Need for centralized management of item additions to the library collection.

3. Key System Capabilities and Project Deliverables:

- Membership management: Track patrons' membership status, including renewal dates and contact information.
- Checkout process automation: Automate item checkout, verify patron eligibility, check for fines, and enforce item limits.
- Fine management: Efficiently collect and track overdue fines, preventing patrons with outstanding fines from checking out.
- Efficient checkout process: Streamline and standardize the checkout process, reducing errors and improving speed.
- Check-in process automation: Implement barcode scanning and automate check-in process, including fine calculation for late returns.
- Item tracking and routing: Track item returns, determine if they were returned to the correct branch, and route items accordingly.
- Centralized item management: Enable the head librarian to add item definitions and assign copies to different branch libraries.
- Label generation: Generate item labels for efficient tracking and placement on reshelving carts.

4. Expected Business Benefits:

- Increased efficiency and accuracy in checkout, check-in, and item management processes.
- Improved revenue collection through efficient fine management.
- Enhanced patron satisfaction with streamlined processes and reduced errors.
- Enhanced data collection for returns, enabling better analysis and decision-making.
- Improved item tracking and routing, reducing misplaced items and ensuring efficient shelving.

5. Definition of Project Success:

- Successful implementation and deployment of the library information system within the agreed timeline and budget.
- Adoption and acceptance of the system by librarians and patrons.
- Reduction in manual errors and improvement in process efficiency.
- Increased revenue collection through efficient fine management.
- Positive feedback from library staff and patrons on the system's usability and effectiveness.
- Achievement of the project's defined key system capabilities and deliverables.

Note: This project vision document serves as an initial outline based on the provided meeting minutes. Further discussions with stakeholders, detailed requirements gathering, and analysis may be necessary to refine and finalize the vision.

Figure 2-31 Process vision document generated by ChatGPT

the interview with the clients and then produced the detailed meeting minutes. And without those meeting minutes, ChatGPT couldn't generate a meaningful project vision draft, so a BA's communication and social skills are still critical and not being replaced (at least not yet) by generative AI.

4. **Use of AI is new and rapidly evolving:** Finally, while generative AI has potential to support certain BA tasks, before using it, you need to ensure that doing so follows an organization's institutional policies and any legal and regulatory requirements related to the use of these tools.

3.5 Deeper Dives: Advanced Topics in Conceptual Data Modeling

In this section, we do deeper dives on several advanced topics that all build on and expand coverage from earlier in this chapter:

- **Deeper Dive 3.1** provides general principles and practical guidance for using the UML class diagram notation for conceptual data modeling.
- **Deeper Dive 3.2** discusses the connections between the process model and the conceptual data model and how findings from process modeling may be used to support conceptual data modeling.
- **Deeper Dive 3.3** briefly discusses the effect of real-time AI components on organizational data needs.
- **Deeper Dive 3.4** illustrates, using a practical example, the use of generative AI tools (such as ChatGPT) for conceptual data modeling.

3.5.1 Deeper Dive 3.1: UML Class Diagram: Another Approach to Modeling the Problem Domain

In Section 3.2, you learned the fundamental elements and modeling practices of EERDs. EER notation is very widely used for conceptual data modeling.

Conceptual data modeling focuses on the characteristics of the real-world domain of interest, not on the characteristics of any technical system used for enabling processes within the domain of interest.

The second type of notation, class diagrams, is a UML standard associated with the object-oriented approach (Fowler and Scott, 2003; Larman, 2004). (We introduced UML in Chapter 2, Section 2.2.) As noted in this chapter's Introduction, for conceptual data modeling, both notations are roughly equivalent, with EERDs being more popular for data modeling. However, when designing your program logic, you often switch from EERDs to class diagrams, which is easy to do (see later in this section how the models offer equivalent notations for data modeling). Class diagrams enable us to extend our data model to also include our system's program logic, which these days is typically object-oriented.

The key takeaway here is that EERDs and class diagrams offer the same conceptual data modeling power. You can use either one equally effectively. But when some organizations reach the internal (technical) program design stage, they switch to class diagrams because of their broader range of features often used for technical design.

We'll return to the issue of the internal (technical) design of software applications later in this book (primarily in Chapter 14). Right now, our focus is on data modeling of the external world.

When you structure the results of your domain modeling process with the UML class diagram notation, your terminology will be somewhat different from that of the EER notation discussed previously. Table 3-1 outlines the mapping between the key concepts in EERD and UML class diagram (used for conceptual data modeling) notation. Discussion of the similarities and differences between the concepts follows.

Table 3-1: Mapping between EERD and UML class diagram concepts

EERD	UML Class Diagram
Entity Instance	Object
Entity Type	Class
Attribute	Attribute
Relationship	Association
Associative Entity	Association Class
Cardinality	Multiplicity
Generalization/Specialization Hierarchy	Inheritance Hierarchy
Supertype/Subtype	Superclass/Subclass

3.5.1.1 Classes

UML calls individual things *objects* and collections of objects with shared characteristics *classes*. So, Clemenza Poserina and Kofa Owusu, the fictitious employees from Section 3.2.1, would be objects in the Employee class, and our earlier examples of entity types (Product, Patient, Employee, Musical Instrument, Musician, Trip, Shipment, Purchase, Insurance Policy) would here be identified and labeled as classes. Figure 3-16 shows the UML class diagram notation for the classes Sale and Product. UML class diagram notation doesn't have a requirement for an identifier attribute similar to what EER notation requires because UML assumes that each object has a built-in identifier that separates it from all other objects.

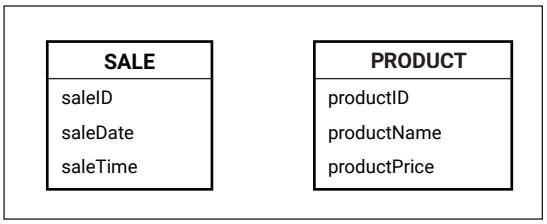


Figure 3-16 UML class diagram notation for classes

3.5.1.2 Attributes

Class diagrams and EER notation both use the word *attribute* to refer to a descriptor of an entity or a class. Even in terms of the notation, they resemble each other very closely (as you can see by comparing Figure 3-16 with Figure 3-3).

3.5.1.3 Associations

An EERD relationship is the same thing as a class diagram association. Here, we consistently use the term *association* because we're talking about class diagrams.

The UML class diagram approach offers a richer variety of modeling options compared with EERDs. In UML, there are two special types of *structural* associations: part-whole association and inheritance hierarchy. They are structural because they convey information about the structures to which the objects (instances of classes) belong.

A part-whole association describes how a whole is built of component parts. There are two types of part-whole associations: aggregation and composition. Here, we describe all four types of associations in UML: aggregation, composition, inheritance hierarchy, and general association.

- To illustrate aggregation, assume that a team at a ValueForAll (VFA) store consists of one supervisor and multiple employee members. The supervisor and the member employees are all parts of a larger whole: the team. In this case, the part-whole association should be used to describe the relationship between Supervisor and Team and Member and Team. Note that a Team is not a permanent structure, since its membership can be changed at any point: the identity or the existence of a team member or a supervisor is not dependent on the team. This is an example of a specific type of part-whole association called **aggregation** (see Figure 3-17a).
- Some part-whole structures denote a structure that links parts to the whole in a much more permanent way. **Composition** refers to whole-part structures in which the parts are entirely dependent on the whole and are assumed to belong to only one whole. For example, in most cases (unless they are intentionally built to be modular), a building consists of rooms that cease to exist if the building is destroyed; a room gets its full identity from the building. In the VFA context, an example of composition could be a line item of a sale: each line item is uniquely associated with a single sale (at a specific time with a specific customer). An example of this is shown in Figure 3-17b.
- In UML, an *inheritance hierarchy* is a mechanism for expressing specialization/generalization. Conceptually, the idea is the same as in EERDs: a superclass may have multiple subclasses, each of which inherits the attributes and the associations of the superclass. In addition, each subclass may have its own attributes and associations. Using the same example as above in the context of EERDs, this notation is featured in Figure 3-17c.
- Finally, *general association* is used in situations when objects in two classes are connected with each other but not in a way that would form a whole-part structure or an

Aggregation A part-whole association in which each part is inherently associated with a single instance of the whole.

Composition A whole-part association in which the parts are entirely dependent on the whole and are assumed to belong to only one whole.

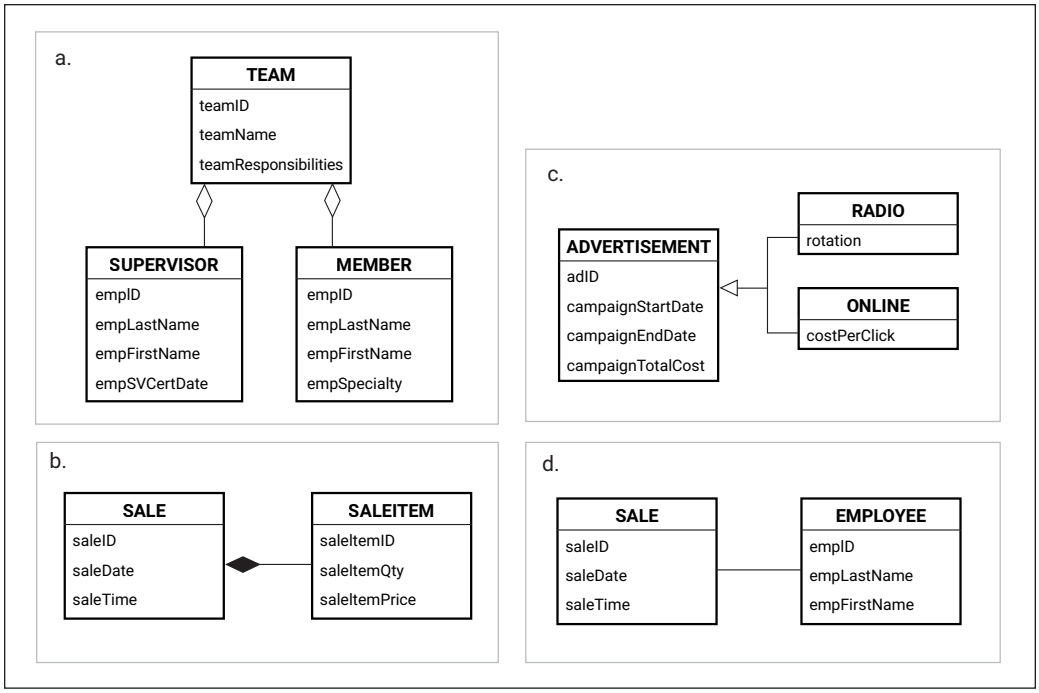


Figure 3-17 Class diagram notations for various relationship types

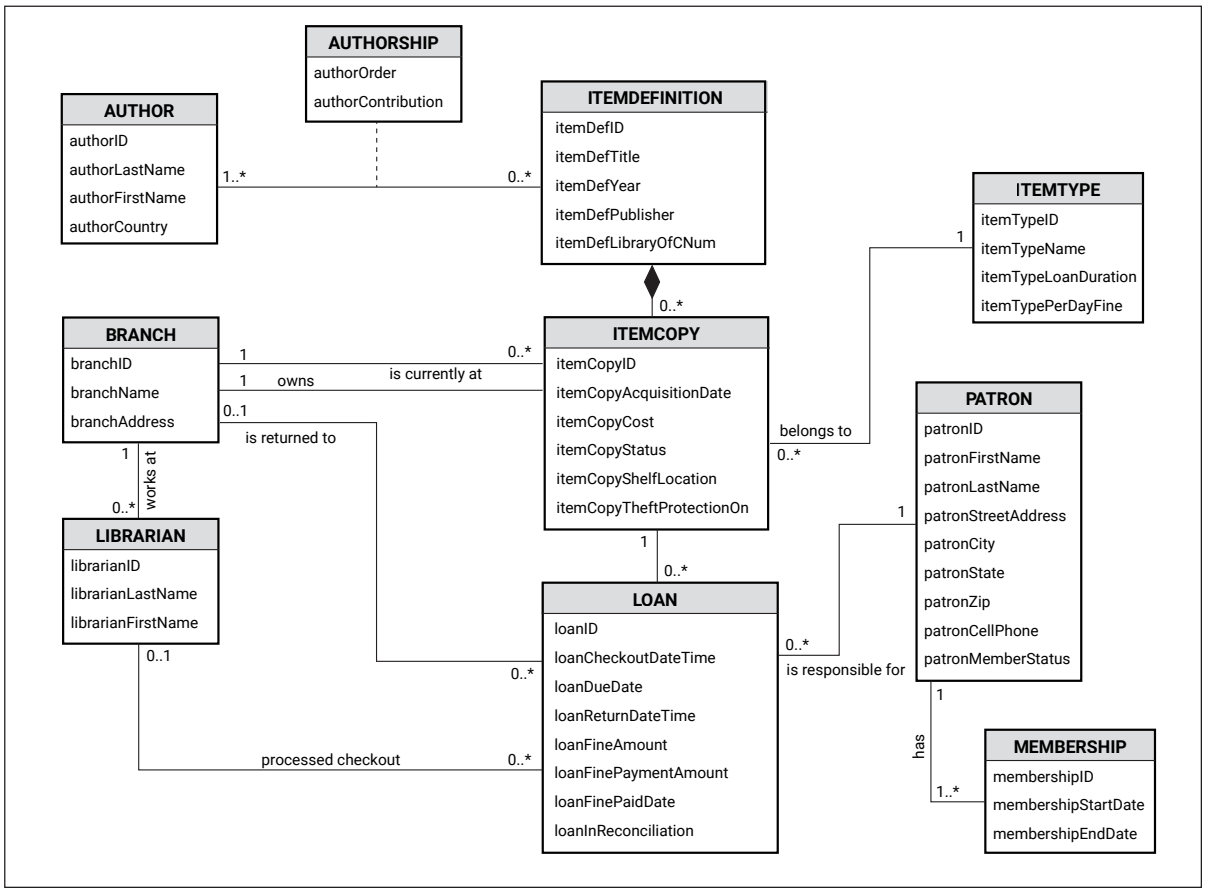


Figure 3-18 WPL domain model expressed with the UML class diagram notation

inheritance hierarchy. These are very typical and important in modeling associations among concepts. The syntax is easy: a general association is presented with a simple line between the associated classes (see Figure 3-17d). Figure 3-18 is a UML representation of the integrated Wayback Public Library (WPL) model (expressed with an EERD in Figure 3-15). In this figure, most of the relationships are general associations. For example, when analyzing the Loan–Patron association, we find out that a patron is not a part of a loan, the loan is not a part of the patron, and they are not each other's subtypes/supertypes. The relationship between Loan and Patron is, however, still important and has to be included as a general association. We need to maintain information regarding which Patron was responsible for each of the Loan instances.

3.5.1.4 Multiplicity

Multiplicity The number of class instances associated with an instance of another class in an association.

The UML class diagram notation gives us the same tools as EERDs for specifying the characteristics of a one-to-one, one-to-many, or many-to-many association. Specifically, in UML, the term **multiplicity** means essentially the same thing as *cardinality* in the EER context. Figure 3-19 shows the most used multiplicity options. Please note that, as with cardinality, multiplicity is specified separately for each direction of the association.

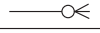
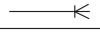
UML	Description	EERDs
0..1	Optional one	
1	Mandatory one	
0..*	Optional many	
1..*	Mandatory many	
*	Many (use not recommended—vague)	
n..m	Many with specific lower and upper bounds	Not formally specified

Figure 3-19 Multiplicity notations

3.5.1.5 Association Classes

Finally, you also need a structure for specifying the concept that is expressed with an associative entity in EERDs. In the UML class diagram notation, this is called an association class, and its expression in UML is presented in Figure 3-20. Note the difference between EERDs and UML here. In EERDs, the associative entity is included as an essential component of the relationship, whereas in UML, the association class is connected to the association with a dashed line. Fundamentally, the concepts are still the same.

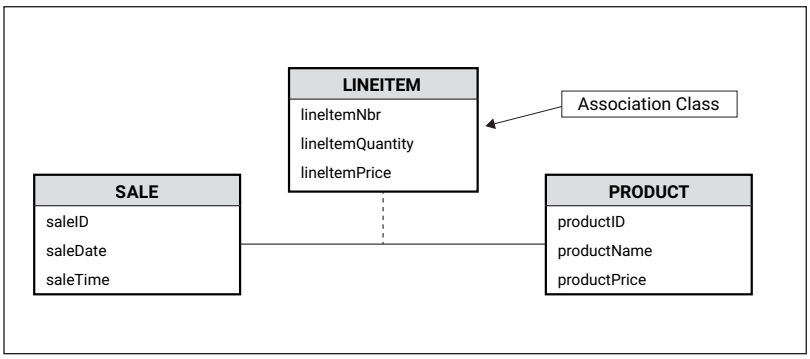


Figure 3-20 Association class notation in UML

3.5.2 Deeper Dive 3.2: Connections Between Process Models and the Conceptual Data Model

Associated closely with Section 3.2.4, this section provides a deeper dive into the questions regarding the connections between process models and the conceptual data model.

In Chapter 2, you learned to model organizational processes with the UML activity diagram notation. In this chapter, you worked toward understanding and developing skills in conceptual data modeling. These two activities are tightly interconnected and, in many ways, dependent on each other. Even though the connection between the two is not always modeled exhaustively, the organizational processes specified with activity diagrams need specific and well-understood data to function and achieve their goals. This data can't be captured in a reliable way unless the underlying conceptual structures are well understood and internally consistent. This type of conceptual

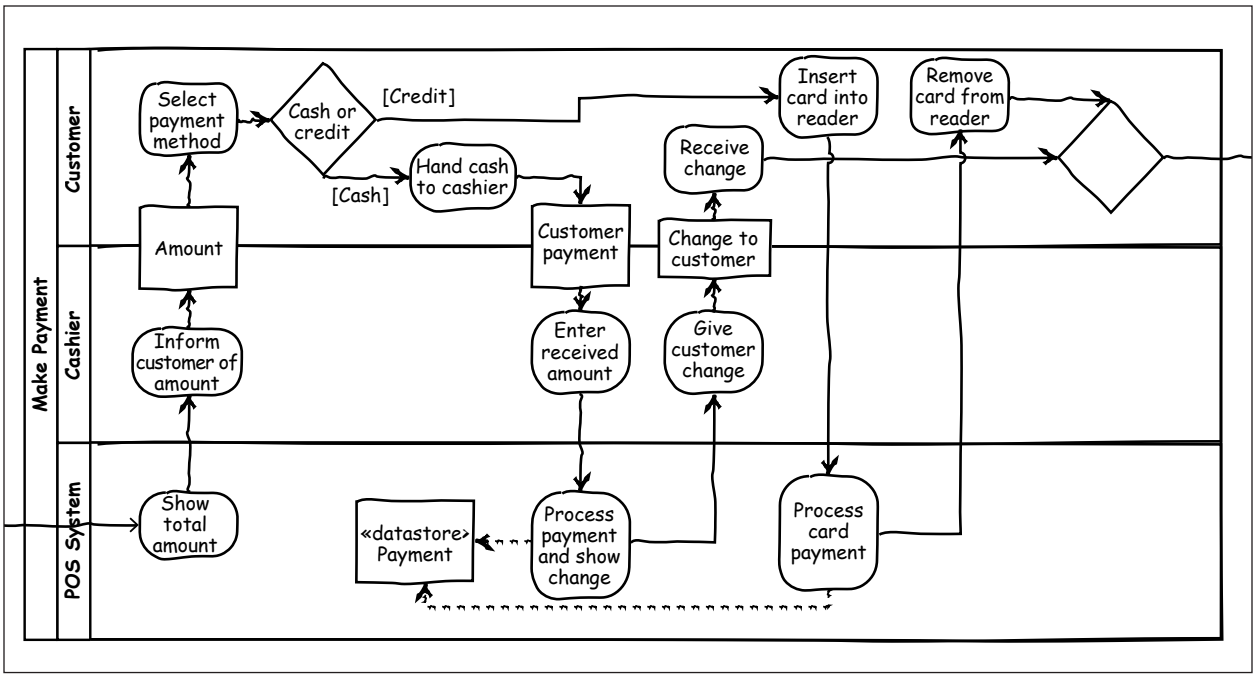


Figure 3-21 Evaluating the conceptual data modeling effect of effect changes

understanding is very difficult to achieve without conceptual data modeling. Conceptual data modeling, in turn, is informed by process modeling in terms of the relative importance of the concepts. In general, understanding organizational processes is a key part of the discovery process in conceptual data modeling.

We can explore this through a simple example and evaluate, from the perspective of conceptual data modeling, a segment of a VFA process model initially presented in Figure 2-26 and repeated in Figure 3-21. We can ask, for example, the following question: What are the data elements that we can derive from this process model segment so that we can either add them to the model or verify that they already are there?

Let's analyze the model one activity at a time. It assigns the task of showing the "Total Amount" to the cashier and the customer. The system needs to maintain a running total of all the items that are included in the sale so that it can show the total to the cashier/customer at the start of the Make Payment process. It's also very likely that the total will be maintained as a persistent data item. Total Amount would likely be an attribute of Sale. Further analysis of Total Amount reveals that we'll need subtotals for each line item. The subtotal, in turn, is determined by the quantity of items sold multiplied by the price at which the item was sold. All these data items would need to be included in a conceptual data model.

When a customer makes a payment in cash, the cashier enters the amount into the system, creating another data item that the system needs to maintain at least during the execution of this functionality. This figure is needed to determine how much money the cashier should give back to the customer as change. More importantly, the process model indicates that there's a data store called Payment in the system. Based on this, we know that the system will need to store the payment amount (whether paid in cash or by credit card). It's likely that the company also wants to know the exact time and date of the payment and the payment method. There might also be other items that need to be captured in the case of a credit card payment, such as the payment processor that was used. The specifics would need to be determined in conversations with the experts. At any rate, the process model strongly suggests that we should have an entity called Payment. Even in

this simple case, the process model revealed these two important details: the need for numerous Line Item attributes and the need for a Payment entity. Obviously, you might already have considered these factors without the process model, but even in those cases, the process model would have served as a useful checklist.

How about including system users as entity types in the conceptual data model? Let's consider customers of VFA. We've already included Customer as an entity in our conceptual data model, and the future state process models include Customer as a system user because they will directly use our system to make purchases—for example, via a website, kiosk, or mobile app. As a result, the customers are also users of the system and would be portrayed as actors in activity diagrams.

How does the modeling process just illustrated differ from what you learned in the earlier Fundamentals section? In Fundamentals, we used our *textual* knowledge of the cases to create conceptual data models. Here, we used our *process* knowledge to the same end. Note that we created the process models themselves based on the textual knowledge. In reality, we used the same underlying knowledge to get to the same data modeling goal. The difference was whether we started from a text-oriented or process-oriented perspective.

Given that they have the same origin, we see that all process models identified for the retail store would be associated with the *same* conceptual data model; the concepts and their associations are the same for the entire retail store (or store chain) context. In your role as a business analyst, you need to make sure that the terminology used across various phases of business process modeling and conceptual data modeling stays consistent. In practice, this process is often highly iterative because the models affect each other. On the one hand, business process models are important sources of information for the conceptual data modeling process. On the other hand, the conceptual data modeling process will often clarify the domain so that new business process approaches are easier to discover. This also means that the order of business process modeling and conceptual data modeling that the chapter sequence in this book suggests (business process modeling first, then conceptual data modeling) is not by any means fixed. In some cases, it makes sense to start with conceptual data modeling, particularly if the concepts of the domain are not well known and well defined.

Note that the conceptual data model has a much longer lifetime in many cases than the process models within the same domain. Organizational concepts tend to have a longer lifetime than processes and are more stable (they change less frequently). In many cases, an organizational current state conceptual data model already exists when a business process modeling activity starts. Also particularly within an industry sector, organizational conceptual data models have a lot of similarities. One result of these similarities is packaged data models (see Hoffer, Ramesh, and Topi, 2019, p. 135), which are data models available for organizations within a specific industry domain. VFA might license a packaged data model from a company that specializes in the retail industry and the characteristics of retail stores. It would do so knowing that any of its competitors could license the same packaged model. In most cases, this would not be a problem because all stores or chains would adapt the packaged model for their own purposes.

Although the sequencing of business process modeling and conceptual data modeling activities is not set in stone, it's essential that you as a business analyst understand the relationships between the two and keep them aligned, as appropriate.

3.5.3 Deeper Dive 3.3: Identifying the Data Needs of AI Components

AI systems and AI-based components integrated into traditional systems benefit from very large amounts of data, which is used for training machine learning models. Effective integration of AI models into organizational systems that serve real-time operations requires real-time interaction between the AI components and the databases that serve them. McFadin (2023) emphasized the importance of scalability, reliability, and speed requirements for the data management infrastructure used to support AI-based applications. McFadin used companies such as Netflix, Uber, and FedEx as examples of enterprises that are feeding millions of daily transactions immediately back

to their AI systems, allowing them to maintain an up-to-date view of optimal viewing recommendations, waiting time estimates and route recommendations, and estimated delivery times, respectively. Depending on the context, storing data regarding the recommendations provided to various stakeholders might also be necessary, further adding to the data requirements. Traditional relational databases are not sufficiently fast and scalable for many real-time AI component needs. NoSQL databases, such as Apache Cassandra, a document database, are designed to deal with very large workloads with fast response times. Other popular databases used for this purpose are Couchbase, Redis, and DynamoDB.

Fast response times and flexibility are not sufficient. An organization that wants to integrate AI capabilities into its organizational systems still must document and maintain an understanding of the data that the AI components need to operate and the elements of AI output that need to be stored either temporarily or permanently. Modeling these data requirements can be done with the conceptual modeling tools introduced in this chapter.

Ensuring the quality of the data is essential for the value of the AI-based components and systems. Heus (2023) writes: "High-quality data ensures that the AI model can make accurate and reliable predictions. It's not just about quantity; it's about having the right data." Along with ensuring access to the raw data, it's necessary to maintain high-quality metadata. Heus continues: "This data must be accompanied by relevant metadata, which provides context and makes the data understandable and usable." Data governance is beyond the scope of this discussion, but, as Janssen et al. (2020) state, trustworthy AI needs to be built on the foundation of solid data governance.

3.5.4 Deeper Dive 3.4: Using AI Capabilities to Support Conceptual Data Modeling

In this section, we'll present a relatively simple but illustrative example of current (Fall 2023) capabilities of generative AI (specifically, ChatGPT) to support conceptual data modeling using a narrative description generated based on stakeholder interview(s). The following prompt was used:

The following are meeting minutes from a meeting with the Wayback Public Library concerning the requirements for a conceptual data model. Based on these minutes, create a conceptual data model for the new system:

The prompt was followed by the text of minutes and a resulting textual description. Both are available at https://www.prospectpressvt.com/textbooks/spurrier-systems-analysis-and-design-2-0 (scroll to the horizontal red menu line, then click on Student Resources). Figure 3-22 provides a graphical representation of the textual version. Comparing this with the model shown in Figure 3-15 reveals that ChatGPT generated a model that is closely aligned with the one that we created. The key differences are as follows:

- The ChatGPT output did not provide complete information for identifying all the cardinalities. The missing elements are identified with arrows.
- The ChatGPT model identified the Patron–Membership relationship as a one-to-one relationship, which would prevent the system from maintaining a membership history for a patron.
- The ChatGPT model did not separate Item Definitions from Items. As a result, the model doesn't allow direct identification of the item copies that are, in practice, copies of the same book.
- In the Item context, the ChatGPT model did not separate Item Type into its own entity.
- Instead of an integrated Loan concept, the ChatGPT model included two separate entities: Checkout Transaction and Checkin Transaction.
- Instead of tracking Fine payments at the level of a Loan, the ChatGPT model tracks them at the level of a Patron, preventing a detailed level of tracking of fine payments.

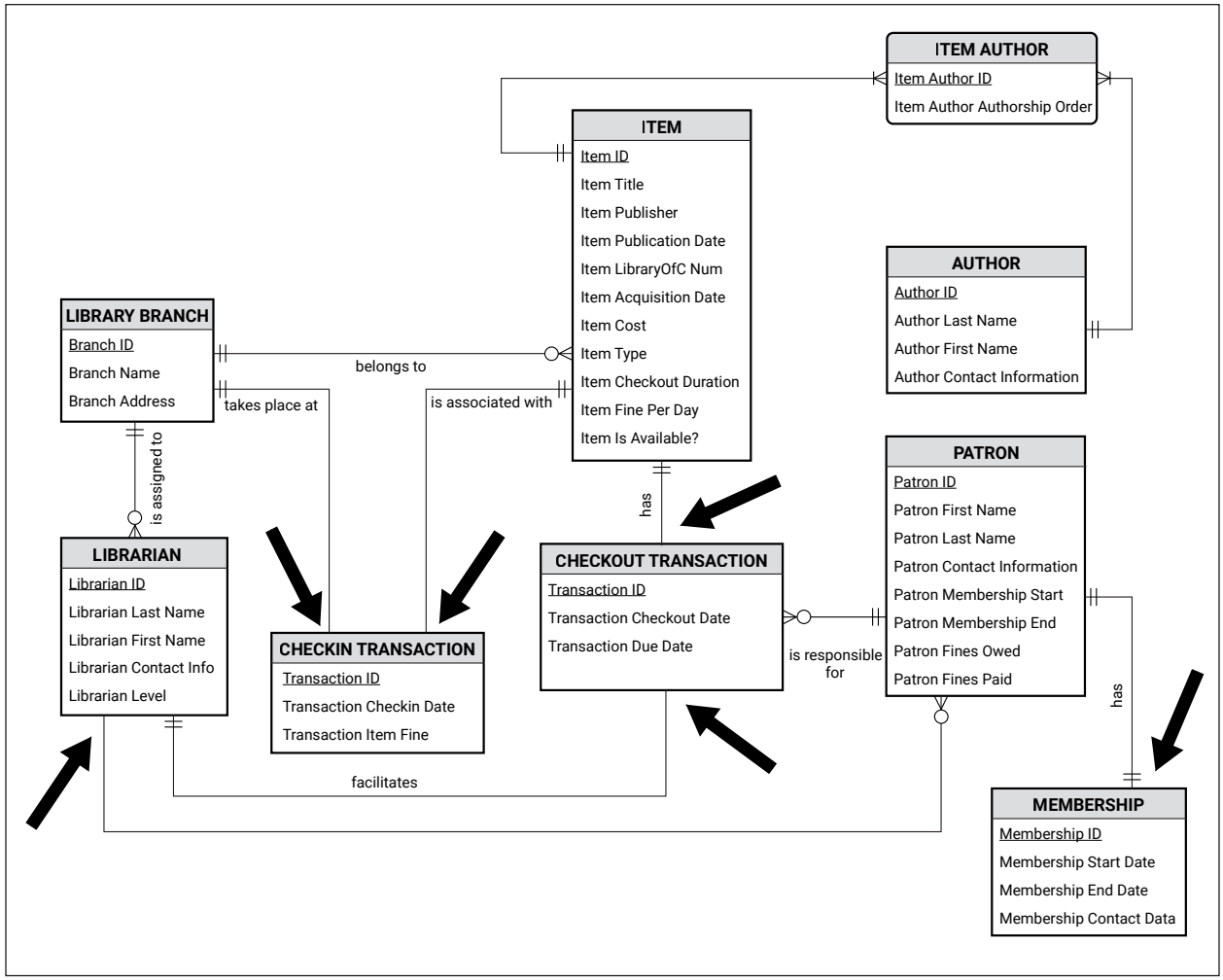


Figure 3-22 WPL conceptual data model generated by ChatGPT

The model is clearly not ready for use as a foundation for database design, but it forms a reasonable first draft.

It's too early to provide detailed predictions of how generative AI deals with complicated, large-scale real-world models. Based on this simple example, however, it's clear that the generative AI tools demonstrate significant potential for serving as an effective assistant for human modelers.

On the other hand, it's also clear that human beings must possess deep BA skill sets to recognize which parts of the AI-generated model are correct and which need to be corrected.

Chapter 4

User Stories and User Interface Models

4.5 Deeper Dives: Advanced Topics Related to Business Analysis and Use of Its Results

In this section, we do deeper dives on advanced topics that build on and expand coverage from earlier in this chapter:

- **Deeper Dive 4.1** discusses the importance of analyzing the implications and potential consequences of planned information systems capabilities.
- **Deeper Dive 4.2** describes the use of the INVEST model to evaluate user stories.
- **Deeper Dive 4.3** explores the use of user stories and UI models to express AI-based capabilities.
- **Deeper Dive 4.4** presents an example that illustrates how generative AI can be used in the development of user stories and initial UI models.
- **Deeper Dive 4.5** explores the relative importance of UI models in greenfield versus brownfield development.

4.5.1 Deeper Dive 4.1: Ethical Implications of IT-Powered Change

Historically, the SA&D process often neglected the ethical implications of systems. But that's changing, given the reach and strength of the effect of systems solutions powered by new enabling technologies such as the Internet of Things (IoT); big data and analytics; and various forms of artificial intelligence (AI), including deep learning, machine learning, robotics, and so on. Identifying and addressing the moral impacts of such systems requires systematic, principled ethical analysis of the harms and benefits that new systems capabilities are likely to cause stakeholders. Stakeholders include anyone potentially affected by a system (not just direct users of the system), which could mean customers, employees, management, owners, governments, and so on.

Job elimination from system automation is a classic example of an ethical issue. It's a concern because increasing efficiency via automation has always been a key goal of new systems. We examine that topic in Chapter 9.

However, the range of ethical issues is vastly broader than that. For example, in the ValueForAll (VFA) case, we described a feature enabling customers to search for and quickly locate products. That clearly benefits the customer, but would a store like VFA actually want to provide it? Some stores might prefer to expose their customers to as many of their products as possible while in the store—for example, by arranging products in a certain way. In this case, implementing a product search/locate feature might depress sales. On the other hand, perversely, a store could design the feature so that it would intentionally guide the customer, without their knowledge, past products that they are most likely to purchase.

Would such a system violate general ethical principles? Should the store disclose such a practice to its customers? In this example, VFA needs to weigh its obligation to shareholders to maximize sales against its obligation to provide the best possible service to customers.

AI provides additional examples. AI-based solutions need training data sets, and in practice those data sets are often biased. For example, consider an AI system that filters resumes using the resumes of existing job holders as training data. If those existing job holders are mostly white and male, the AI system may be biased against resumes from women and minorities. How do we identify, evaluate, and mitigate this and many other ethical challenges?

In general, we must acknowledge that identifying ethical questions and addressing them in a

systematic, thoughtful, and principled way is part of our responsibilities as BAs. This analysis should start with initial visioning and continue throughout the system's lifetime.

4.5.2 Deeper Dive 4.2: Evaluating User Stories with the INVEST Model

One of the ways many analysts evaluate the quality of their user stories is the so-called INVEST model (Cohn, 2004, Chapter 2; Rubin, 2012). The INVEST model was created specifically for agile software development methods, like Extreme Programming and Scrum. This model defines a good user story as one that's:

- **I**ndependent (of other user stories). If stories are independent, the team will have much more freedom to determine the order in which they are built. Dependencies between stories complicate planning and estimation significantly. Independence between stories is fully aligned with a well-known "loosely coupled, highly cohesive" principle.
- **N**egotiable (one that leaves space for negotiation during construction). The idea underlying this characteristic is that, particularly in agile development, stories should not be considered as detailed specifications set in stone but, as Rubin (2012, p. 201) eloquently states it, as "placeholders for conversation."
- **V**aluable (for the client and/or to the user). This characteristic should be self-evident: There's no point in including stories that don't provide value to the key stakeholders of the project. The prioritization of the stories addresses the question of how valuable a particular story is.
- **E**stimable (one that provides a sufficient foundation for a good approximation of effort). If a team can't estimate the size of the story in the context of the project, either the story is too large to be used as a foundation for construction or the team doesn't have sufficient expertise to work on it.
- **S**mall (allows its completion within a specific construction cycle). By the time a team is working to construct the capabilities specified in a story, each story should be small enough that it can comfortably fit the boundaries of the regular sprint size of the project.
- **T**estable (particularly when evaluated together with acceptance criteria). Each story should have acceptance criteria that make it possible to give a definitive answer to this question: Are we done constructing the story?

In general, the user story and acceptance criteria skills you learned in this chapter support many of these points. In particular, attaching acceptance criteria to a regular user story and then prioritizing both that base user story and each acceptance criterion attached to it helps you ensure that:

- You understand the story's *value*.
- Each piece is *estimable* because it's decomposed into *small* chunks of functionality via regular user stories and their acceptance criteria.
- You have enough detail to perform good *testing*.

However, in large, complex projects with clear, stable requirements, you may find you have to modify the INVEST model in two ways:

- You may encounter significant and unavoidable *interdependencies* (the opposite of independence) among various user stories; stories may depend on each other and need to be designed and built in a particular order.
- You may find it valuable and necessary to do significant, formal up-front requirements (BRUF) rather than *negotiate* detailed requirements during agile development.

This isn't a defect in the stories. Rather, it recognizes differences that tend to arise in user stories as projects move from small and simple projects suitable for agile to large and complex projects where a hybrid approach (see Chapter 1) will be more effective. We explore this topic extensively in Chapter 10.

Also, in Chapter 5, you'll learn how to further elaborate user stories using a technique called *use case narratives*.

4.5.3 Deeper Dive 4.3: Including AI-Based System Capabilities in User Stories and UI Models

Recent advances in AI enable the creation of systems (or system components) that would be difficult or impossible to create using a conventional programming language like Java, C#, or JavaScript. Examples include image recognition and categorization, chatbots, robotics, and many others. Given these advances, many organizations are strongly interested in incorporating AI-based system capabilities into their organizational systems.

The question at this point is, how are AI-based capabilities best included in user stories and UI models? The answer isn't simple because it depends on the type of capability that needs to be included in the requirements.

4.5.3.1 User Stories for AI-Based Capabilities

In the fundamentals part of this chapter, we used a user story example that requires the system to be able to provide a customer with the location of a product in a VFA store based on an image of the product the customer is interested in buying. Here's the user story as it was expressed earlier:

> As a customer in a retail store, I want to be able to use my own portable device to identify the physical location of a specific product so I don't waste time searching for the products that I want to buy.
>
> AC1 Ability to search by product name.
>
> AC2 Ability to search by product image.

This is a legitimate user story that, despite its seeming simplicity, describes the user's need accurately and with a sufficient level of detail. This is a user requirement, and from the user's perspective, it's ultimately irrelevant how the capability is technically implemented.

However, important additional conditions might be expressed with other acceptance criteria that may have a significant effect on the implementation and so can't be ignored. For example, an acceptance criterion might be associated with this user story that requires that no image match will take more than, say, 0.5 seconds. Another acceptance criterion could specify the level of confidence that needs to be reached before a match is declared and the discovered product is presented to the customer. Yet another criterion could suggest a relationship between confidence in the quality of the match and the number of options presented to the customer. All of these acceptance criteria would provide additional information that would help select the specific AI technique for an internally developed matching solution or an externally developed application or service.

But this just reinforces the point that user stories (and their acceptance criteria) that are implemented via AI are specified in the same way as those that are implemented via conventional programming.

Chapter 2 included another example that integrates AI technologies with a traditional business process. In it, a property insurance company is looking to improve its claims management process with capabilities that would increase its efficiency and quality.

These capabilities were specified with three user stories:

> ROOFEVAL1 As an adjuster, I want an AI robotic system to capture roof images via drone on a daily basis.
>
> AC1 Respond to each claim in one day.
>
> AC2 Automatic drone dispatch.
>
> AC3 Examine each customer house within 100 meters of the claimant's house.
>
> AC4 Capture photographs of the roof without human involvement.
>
> ROOFEVAL2 As an adjuster, I want an AI image system to categorize roof images into common damage and nondamage categories.

ROOFEVAL3 As an adjuster, I want the system to generate damage award and repair/replacement recommendations in a formatted customer report so I can save time and increase consistency.

AC1 Adjuster can review and, if necessary, revise recommendations.

AC2 System will then send recommendations to local agent.

Note that the references to AI ("AI robotic," "AI image") could be removed from these stories without a direct effect on the requirement specification. The key purpose of these user stories is to express the user's needs; the technology selection can come later. However, if it's clear that the capabilities will be AI-driven, it can be specified.

This discussion provides two significant takeaways:

1. The feature requirements expressed as user stories don't necessarily specify that the capabilities be built using an AI solution (or via conventional programming, or any other specific technology). From the users' perspective, what matters is the service that the system will be providing. In many cases, AI-based solutions are one option among many, and technical design will ultimately determine the specification of how the service will be provided.
2. However, user story acceptance criteria may describe constraints that will, in practice, mandate a specific design solution. In these cases, it's highly valuable to recognize those constraints early so their impact on project feasibility can be evaluated without delays.

4.5.3.2 UI Models for AI-Based Capabilities

When we consider the effect of the inclusion of AI-based capabilities on UI models, in many cases, these capabilities are implemented as services that aren't directly part of the core application but still have an important role in determining the application's behavior. For example, AI-based capabilities could be used for analyzing the status of a stakeholder (customer, student, etc.), identifying potential problems, confirming identities, estimating the probability of success, and providing many other similar services.

But users need to interact with AI-based capabilities via a user interface, no different from conventional capabilities. So, as with user stories, the fact that the application under consideration is based on the use of AI doesn't change our approach for developing the initial UI model. Put plainly, a user interface screen for an AI-based capability is still a screen, no different from a screen created for a conventional capability.

The UI model must, however, be aligned with the required input and output parameters of the AI service. For example, in the case of the image search feature of the VFA application for locating a product in a store, the AI service needs to (a) accept a user-provided image as input, (b) accept the identification of the search space (a specific collection of product photos) as input, and (c) provide a set of potential matches and the likelihood of these matches as output. The UI model, thus, must capture the image for user input, determine the search space (products available at the relevant store) aligned with the user's context, and accept output from the AI-based application.

4.5.4 Deeper Dive 4.4: Using Generative AI to Identify User Stories and UI Models

In Chapters 2 and 3, we explored using generative AI tools like ChatGPT or Gemini to assist us in SA&D tasks. Here, we discuss the use of generative AI tools to help create user stories and UI models.

As of this writing, several AI-based applications focus on generating user stories for a specific industry context. These tools work based on relatively brief and open-ended prompts. The two best known tools currently appear to be userstorygenerator.ai and AgileStory.app. There's little academic research on these tools currently, but professional reviews (such as Steele, 2023) and personal experimentation suggest that these services can already serve in a limited role. They still have a long way to go before they threaten to replace well-educated, creative human business analysts.

Steele makes the following observations regarding the four AI-based user story generators that he tested:

- AI models create relatively simplistic, generic user stories and miss many of the intricate details that human BAs learn to understand through thorough exploration of the domain of interest.
- The AI-generated stories are at times so polished that they become an obstacle for the type of detailed conversations that user stories are intended to generate.
- The stories are frequently technically correct but out of the scope for the project.
- The stories are unavoidably based on textual material from the past used to train the model. So the stories reflect what used to be state-of-the-art features rather than a specification of genuinely creative features of future system requirements.

In his review, Steele identifies three potential uses for the current generative AI-based user story generator tools: (1) serving as a "creative collaborative partner" for a business professional responsible for generating candidate stories, (2) creating a checklist of stories that can be used to help ensure that nothing essential is missing, and (3) providing models of technically correct story writing.

In addition to creating user stories, generative AI tools can be used to analyze and organize sets of user stories. This can be useful, particularly with hundreds of stories within the same application.

Generative AI tools don't yet show as much promise for UI modeling. However, numerous user experience and user interface development tools benefit from AI-based components as engines for collaborative design, prototyping, wireframing, and technical AI design. With these tools, as with any tools, it's essential to understand how the AI-based capabilities support the specific types of projects that you, as a BA, are focusing on.

It's important to make one observation regarding all AI-based generators of planning and design artifacts: training the models with rich product-, company-, or industry-specific data is likely to lead to better outcomes than relying on generic models. It's also essential that computing professionals who use AI-based tools follow the ethical guidance and disclosure rules provided by their employers, schools and universities, and professional societies regarding the use of generative AI tools.

4.5.5 Deeper Dive 4.5: The Relative Importance of UI Models in Greenfield versus Brownfield Development

UI models are widely used in system projects, but their value varies based on whether the project is a greenfield project or a brownfield project.

Greenfield development
Development of an entirely new system.

Early-stage UI modeling is particularly important in **greenfield development**, where a brand-new system is created. The term *greenfield* derives from building construction, specifically the case where a building is being constructed on land where no structure has previously existed. In this case, the lack of any existing system interfaces creates an acute need for users to be able to visualize the new capabilities. This makes modeling the UI especially useful. In addition, carefully aligning the UI models with user stories enhances their power. User stories express what the system will do, and the UI models demonstrate an initial visual representation of the mechanism through which the users will get their work done with the system.

Brownfield development
Enhancement of an existing system.

In contrast to greenfield development, enhancing an existing system is called **brownfield development**. In this approach, construction occurs where an existing structure or its foundations exist and must be considered. In brownfield development, there's already an overall UI design that users will be familiar with. When making minor enhancements to such an existing system, extensive UI modeling may not be particularly useful. Here, UI modeling can be reduced or skipped altogether. For example, minor changes could include adding a few fields to an existing screen or creating a new screen that's modeled after an existing one. However, even in brown-

field development, systems that involve significant new or updated UI capabilities will benefit from UI modeling.

UI modeling is even useful in cases when the organization intends to acquire required software capabilities from external software vendors or open-source projects (as will be discussed in Chapter 7 in more detail). In these cases, the organization won't construct the software. Rather, it will select and configure an existing software product that, of course, already has a well-defined design, including a UI. However, it may still make sense to build early-stage UI models so key stakeholders within an organization can reach an agreement regarding the desired systems capabilities and functioning. In this way, UI models help identify the external software that best fits the need. This is similar to greenfield development, where users may experience difficulty envisioning a new system and its capabilities without the help of a visual model that resembles the eventual system.

Use Case Narratives and Functional Testing

5.5 Deeper Dives: Advanced Topics in Use Cases and Testing

In this section, we do deeper dives on several advanced topics that all build on and expand coverage from earlier in this chapter:

- **Deeper Dive 5.1** explains alternative terminologies for some use case narrative concepts.
- **Deeper Dive 5.2** discusses the level of detail that may be specified in use case narratives, ranging from the highly detailed use case narratives we've focused on in this chapter to "casual" use cases that offer an intermediate level of detail to "brief" use cases that resemble user stories in their minimal level of detail.
- **Deeper Dive 5.3** introduces use case diagrams, including the relationship of use case diagrams to use case narratives and why use case narratives provide more value and are more widely used than use case diagrams.
- **Deeper Dive 5.4** discusses the relationship between use case narratives and the technical design of software programs.

5.5.1 Deeper Dive 5.1: Alternative Use Case Terminologies

Given how important use case narratives are to most systems projects, it's perhaps surprising that there's no single, standard format for them specified by the Object Management Group (i.e., in UML) or other standards bodies. This is true even though use case narratives have been around since the mid- to late 1980s, when they were first introduced by Ivar Jacobson (1993) and then substantially influenced by the writings of Alistair Cockburn in 2001.

Somewhat confusingly, Jacobson and Cockburn use different terms for two key use case concepts. Table 5-3 shows equivalent terms between these two authors:

This is a bit like the situation in Chapter 3, where we noted that EERDs and class diagrams provide different notations and terminology for creating conceptual data models.

Even though Jacobson invented use cases, we've used Cockburn's terminology in our examples because of the wide popularity of his book (2001).

Beyond that, we note a couple of other points about terminology:

- **Use Case Slices:** Were invented by Jacobson and Ng (2005) after Cockburn's book (2001) was published. Because of that, we use their term for this concept.
- **A Use Case by any other name:** Many teams that write out logic steps in the general approach of a use case narrative may not call them use cases or use either of the terminologies shown in Table 5-3. Those teams may use different terms, like "scenarios" or "specs." But what those teams are creating is fundamentally the same as use case narratives. Because of that, you can easily adapt the skills you learned in this chapter to those situations.

Table 5-3: Alternative use case narrative terminologies

Use Case Narrative Concept	Alistair Cockburn Terminology	Ivar Jacobson Terminology
Simplest, most basic version of a use case that omits variations and exceptions	Main Success Scenario (MSS)	Basic Flow
A variation or exception to the most basic version	Extension	Alternative Flow

5.5.2 Deeper Dive 5.2: Use Case Level of Detail—Brief, Summary, and Fully Dressed

As noted earlier, use cases can be written in a range of formats that vary significantly in detail (Cockburn, 2001). On one hand, a so-called **casual use case** might be only slightly more elaborate than a user story, perhaps expressing system behavior in one or two paragraphs, while the **brief use case** may be as terse as a one-line user story. In contrast, **fully dressed use cases** are highly detailed and describe the user-system interaction in a comprehensive way.

Casual use case Use case narrative offering a moderate level of details: more than a brief but less than a fully dressed use case.

Brief use case A use case that provides only a highest level summary description of the user's goal.

Fully dressed use case The most comprehensive use case narrative format.

Because we're primarily focused on using use cases to expand on the details of user stories and link together other related requirements specifications, we focus on the "fully dressed" use case narrative format.

Table 5-4 provides a comprehensive description of the sections that may be found in such a "fully dressed" use case. Some of the elements describe user story features we have discussed extensively earlier in this chapter.

Table 5-4 describes the great majority of sections that you might expect to find in a highly detailed, "fully dressed" use case. We've stated that using such a fully dressed use case format will be typical, given that our general approach is to utilize use cases to expand on "one-liner" user stories.

Table 5-4: "Fully dressed" use case narrative sections, showing the most commonly used sections

Use Case Section	Comments
Story	This is a statement of the key requirement driving the need for the use case narrative in the first place. As you will see, the relationship between a story and a use case need not be one-to-one. In many development approaches, use cases are created without corresponding user stories.
Use case title	A label typically starting with a verb.
Story owner	Name of a business customer with primary approval responsibility. In agile development approaches, this is often the product owner.
Story creator	Name of the person (typically a business analyst) who is creating the story.
Revisions and approvals	Use cases will typically undergo multiple revisions, both before software construction (if BRUF techniques are used) and after construction (based on user feedback of the delivered, executable software). Revision history typically includes version numbers, version dates, the name of the person making the revisions, and a note explaining the gist of the revision.
Stakeholders and interests	These people include all individuals named in the project who have a reason to review and comment on this use case. They may include both business customers and IT team members. Typically, this includes the stakeholder name and function with respect to the project. It's also common to include a specification of a stakeholder's interest(s) regarding a use case in a use case narrative.
Scope, context, and background	This generally includes a statement of the software and release number, and often some textual background explaining the context and rationale for the change. If a single story generates multiple use cases (e.g., because of large size and/or complexity), then this is a place to explain how this use case fits into the larger requirements picture. Scope specification also includes the name of the system to which the user story belongs ("system under discussion"; Cockburn, 2001).
Dependencies	This is a place to describe other changes that must be completed before or at the same time as this use case for the functionality to work correctly. This can include either other functional requirements (i.e., other use cases) or changes to nonfunctional requirements, such as implementing architectural changes to improve security, scalability, or reliability. For example, before defining how data will be summarized in a use case, it may be first necessary to build an ETL (extract, transform, and load) data capability use case.
Actor role(s)	What type of actor uses this functionality? If more than one role uses this system capability, typically the primary actor type is highlighted.
Precondition(s)	These are states that must be true or activities that must be completed for the functionality to work correctly. For example, a monthly data update may need to be completed prior to executing a data summarization routine. Preconditions are necessary but not sufficient conditions for the use case to be executed; only a trigger can actually launch a use case.
Success guarantee	Outcome if the use case is executed successfully.
Minimal guarantee	Outcome under the worst-case scenario. This typically states that the permanent state of the system should not be changed if the use case can't be completed successfully and that errors should be logged.
Trigger(s)	Distinct from preconditions, a trigger articulates what specifically causes this functionality to begin executing. In the case of a data summarization routine, for example, the trigger could be any number of things, including the data load completing, a timing issue such as reaching the first day of the month, or simply having a user start the process.

Table 5-4, continued

Use Case Section	Comments
Main success scenario (or basic flow)	This is a description of how the process should work when everything works correctly. Synonyms for *main (success) scenario* include *main flow*, *basic flow*, and *happy path* (Leffingwell, 2011). Scenarios, both the main success scenario described here and extensions explained later in the table, can be expressed in many ways, including structured English, pseudocode, activity diagrams (or other forms of process models), or any other form of process description that communicates effectively.
Extensions or alternative flows	In addition to the main success scenario, in most cases it's useful to describe other versions of the process beyond the happy path. These can include: • How to handle situations in which errors occur • Significant variations in successful functionality that may be triggered by conditions or selected by the user • Additional functionality that may fall into the "nice to have" category of acceptance criteria (see the next row)
Acceptance criteria	These criteria are critically important, as they are what the functionality must actually accomplish to be considered acceptable. Using the MoSCoW prioritization model, they come in three varieties, all of which are typically specified at project inception, but which may, with appropriate coordination between business customers and the IT team, be modified: • **Minimum viable product (MVP):** These acceptance criteria are the MoSCoW model "must have" (MH) capabilities. By definition, if any MVP items are omitted or fail to function correctly when the executable software is delivered, the use case is not "done." • **Should have (SH) capabilities:** These are capabilities that ultimately may be as important as the MVP items but, if necessary, could be delayed to a subsequent release (e.g., because a manual workaround could be implemented to compensate in the short-term). • **Could have (CH) capabilities:** These are "nice to have" capabilities that are generally included in budget estimates but that the business and IT teams agree can be omitted and the solution will still add value.
Test cases	Test cases are closely related to both user stories and corresponding use cases. The difference is that test cases (or test scripts, as they are sometimes called) provide "worked examples" of specific inputs, system usage process steps, and expected intermediate and final outputs. While test cases are often not included directly in use cases, in certain agile approaches, including in test-driven development (TDD), the test cases *are* the key requirements. In TDD, the key users are typically colocated with a small team of developers, such that the changes to be made can be effectively described informally. The only documentation in TDD may be the test cases, which explicitly establish whether the delivered, executable software is working correctly.
Exclusions/ out-of-scope items	These are MoSCoW "won't have" (WH) capabilities that are explicitly agreed to be omitted from scope (at least for the current project release). As such, they are, by definition, not acceptance criteria but are often explicitly included immediately after acceptance criteria to provide further clarity to the scope of the use case.
Assumptions	These are statements that specify dependencies that must be successfully addressed to ensure the success of the use case. They are generally focused on business issues or items occurring outside the control of the IT team. Examples include the following: • Business customers who are known to use different processes or calculations in different locations in the current situation will standardize on a single approach that's compatible with the use case in the context of the new system/ functionality. • Infrastructure upgrades needed to accommodate expansions of users served will be available by the time of the external release of functionality. • Another IT team will meet a specific deadline for delivering production functionality that this use case depends on.
References and attachments	Based on its focus on the process of the main success scenario and its extensions, one might conclude that a use case is fundamentally a process- or logic-oriented document, generally coupled with preconditions and triggers for executing that logic. However, a use case can reference other related and equally important perspectives. This is the hub-and-spoke requirements model explained earlier. As noted in the introduction to this chapter, constructing a complete software solution (whether new or an enhanced version of an existing application) requires establishing a detailed understanding of not just the process but several other key aspects of the problem domain and corresponding software solution. These aspects may typically include the following: • Data models: The pertinent domain entities and their relationships, whether specified as a new collection of entities or as an extension to an existing data model. • User interface/user experience (UI/UX): In most cases, the design of the user interface (UI) and its user experience (UX) "look and feel" is a central concern for business customers. Indeed, the usability and attractiveness of the UI/UX is often a critical factor in determining whether end users will be willing to actually use the system on an ongoing basis. As such, presenting realistic UI/UX models (wireframes, mock-ups, and even working prototypes) can be a critical factor for ensuring the overall success of the software project. While it's sometimes possible to embed data models and UI/UX models directly in the use case, it's usually more desirable in the use case to simply reference separate files containing those models. It makes sense to do so because often multiple use cases depend on those same data and UI/UX models.

Table 5-4, continued

Use Case Section	Comments
Nonfunctional requirements	As opposed to functional requirements, which were specified previously (especially via the acceptance criteria and scenarios), nonfunctional requirements pertain to general performance capabilities of the system. Based on their names, nonfunctional requirements are sometimes called the "ibilities," including: • **Scalability:** How many users and/or the volume of data that the system can support within a given level of performance, such as response times or time to process large jobs. • **Reliability:** The times that the system is supposed to be available and the amount of time (typically expressed as a percent) that the system may be down. • **Extensibility:** The extent to which the system is designed and constructed in ways that enable the addition of new, often specific capabilities in the future. • **Criticality:** This refers to areas of privacy and security risk that the system must be capable of handling. Typical criticality issues include the following: ▪ **Processing, storing, and transmitting sensitive data:** This can include sensitive corporate client data or (even more critically) sensitive data associated with individual human beings, especially when that data includes key identifiers or data pertaining to healthcare conditions and treatments. ▪ **Facing the public internet:** This interacts with the public internet and is especially pertinent when coupled with the handling of sensitive data. ▪ **Subject to audit:** For example, systems that handle or affect money (financial accounting, payroll, banking, trading systems, and so on) are generally considered high risk and subject to in-depth audits to ensure that they are functioning correctly with sufficient controls (including controls over both their operation and their development). ▪ **Human safety critical:** Although this is typically not as much of an issue with administrative information processing systems as with systems controlling tangible objects operating in three-dimensional space (e.g., embedded systems, autopilot systems, robotics), this can be an issue when failure to issue a correct and timely response might interfere with a stakeholder's receipt of critical services. For example, the inability of a health insurance system to verify coverage could cause a patient needing urgent services to have them be delayed or to have to shift to a different healthcare facility. Requirements are also sometimes denoted via the acronym FURPS, which stands for the classification "functionality, usability, reliability, performance, and supportability" (Larman, 2005). Functionality clearly refers to functional requirements, whereas the remaining items are nonfunctional in nature. The general correspondence between the "ibilities" and the nonfunctional elements of FURPS is apparent. (For example, "performance" roughly equates to "scalability," as "supportability" does to "extensibility," and so on.) Usability is considered a nonfunctional requirement dimension because the same functional requirements can be implemented in a variety of different ways, leading to different user experiences. However, user experience/user interface prototypes are often used as a mechanism to discover and even document functional requirements (as discussed in Chapter 4), so the lines are often blurred in this context.
Story details, open issues, and conversations	As noted previously, use cases (and software requirements, more generally) tend to evolve iteratively over time. As such, it's often useful to provide a space for comments from various team members to raise and respond to open issues and questions surrounding the use case. As issues and questions are addressed, they can be closed out in this section. This approach can be especially effective when the comments are referenced back to updates to the use case version number in the revisions section.

However, to be clear, it's not necessary to fill out every single section of the fully dressed format for every single use case. This, in turn, begs the question as to how the IT team, in general, and the BA, in particular, should determine which sections to populate in any given situation.

In general, as there's no specific "official" guidance to the format and contents of use case narratives, you may choose an approach based on organizational practice or whatever works most effectively in a specific situation. In short, if you find that your project would benefit from including a specific fully dressed use case section, then you should include it. Otherwise, following the principles of intelligent modeling, you should exclude it.

5.5.3 Deeper Dive 5.3: Use Case Narratives versus Use Case Diagrams

Another potential source of confusion is that use case narratives can be confused with a different (but related) requirements model, the UML **use case diagram**. A use case diagram is a UML standard diagram that portrays the different categories of actors (typically, users performing different roles with a software system) and their relationship to the use cases (e.g., which actors are associated with which use cases). If one use case is included as a component within another use case or extends another use case, then those relationships will be shown as well. By convention,

Use case diagram A UML diagram type that provides a summary view of a system's use cases, their relationships, and actors.

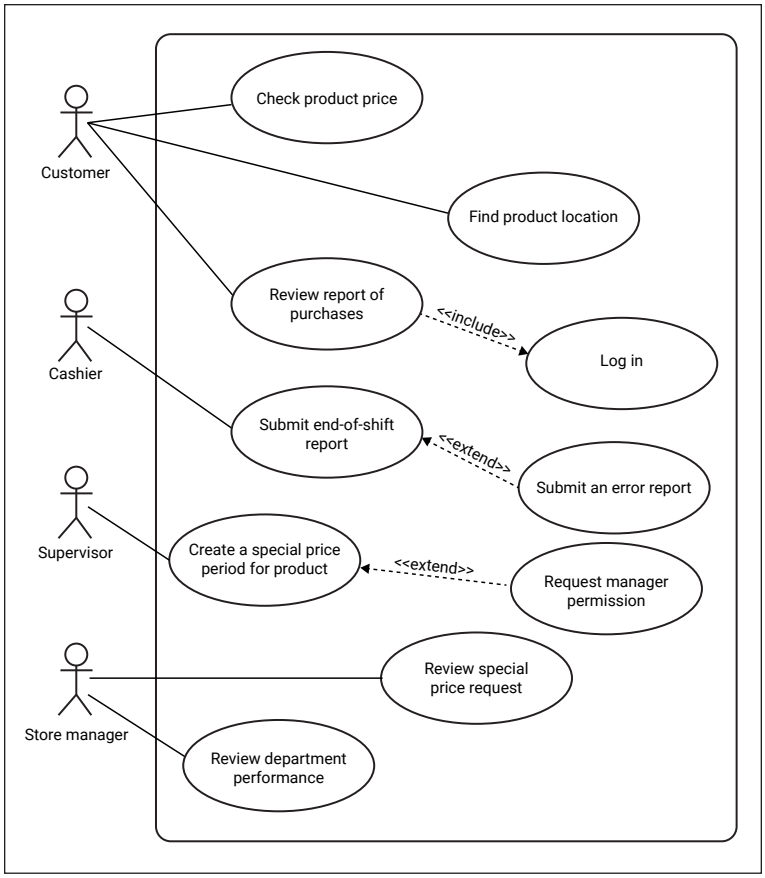


Figure 5-15 Example use case diagram

actors are shown as stick figures, use cases are shown as ovals, and the system boundary is represented by a box surrounding the use cases. See Figure 5-15 for an example.

A quick review of Figure 5-15 reveals several issues. First, use case diagrams are fairly simplistic. As such, they convey relatively little information, as they only include a brief title for each use case. In fact, use case diagrams are largely redundant with and provide less overall information than user stories. For example, per Chapter 4, in addition to specifying the "who" and the "what" indicated by use case diagrams, user stories specify the "why." Indeed, a use case diagram can be considered merely an alternative way of listing a product backlog of either user stories or brief use case narratives. Given these points, there's seldom reason to invest a lot of time in use case diagrams (Fowler, 2004).

As such, it's not surprising that use case diagrams are not used at all in more than 90% of software projects (Gelbard et al., 2010). All told, use case diagrams add little value beyond their nature as a visual catalog of use cases, so we don't emphasize them. Still, this Deeper Dive briefly describes use case diagrams so you aren't confused if you do encounter them being used in a project.

In closing, you might wonder: Given these issues, why does UML specify a standard for use case diagrams but not for use case narratives? The answer is that UML focuses on diagramming notations, not textual descriptions (Cockburn, 2001, p. xxi), so UML didn't bother with creating a standard for use case narratives, even though they are far more useful and more widely used than use case diagrams.

5.5.4 Deeper Dive 5.4: The Relationship Between Use Case Narratives and Program Technical Design

If you've taken a programming class, you might be wondering how and if you can utilize use case narratives to create your technical designs, including object-oriented (OO) classes and methods. First, as illustrated in Section 5.4.2, you can utilize use case narratives to identify or extend classes (or entities) in a conceptual data model. Many of these conceptual data model classes (or entities) may evolve into OO programming classes during technical design.

However, it's not the case that a single use case typically maps to a single OO programming class. Instead, most use cases will utilize multiple classes (entities), both in the conceptual data model and in OO program code. You can see this in our "Calculate fines for late items" use case by reviewing the test data needed to power this functionality. Specifically, you see that you need a whole series of data model classes (or entities) to make this use case functionality work: Item Definition, Item Type, Item Copy, Patron, Loan, and so on. In OO terms, you use these conceptual data model entities/classes as the starting point for designing your OO programming classes. Also, you'll typically allocate different use case narrative logic steps to the various OO program-

ming classes, with those logic steps becoming OO methods. For example, in calculating fines for late items, the steps for calculating the fine could become a method in the Loan class, while the steps to update the overall fines owed by the Patron could become a method in the Patron class.

Note that the process of OO programming design also typically includes other class types beyond conceptual data model classes. For example, you need presentation classes for defining the user interface of the application, which presents business data and logic to the user.

However, use case narratives and your conceptual data model together provide the understanding of the system functionality needed to design programming classes. We explore program technical design in Chapter 14.

Chapter 6

Designing the User Experience and User Interfaces

6.5 Deeper Dives: Advanced Topics in Designing User Experience and User Interfaces

In this section, we do deeper dives on advanced topics that build on and expand coverage from earlier in this chapter:

- **Deeper Dive 6.1** discusses the issues related to the design of interfaces primarily intended for retrieving and analyzing data and presenting it in a way that supports organizational decision-making at various levels.
- **Deeper Dive 6.2** explores situations in which users may want to achieve the same interaction goal using multiple different channels (such as different device types).
- **Deeper Dive 6.3** covers UX/UI design challenges that are typical in situations when a system is designed for use in global organizations that are dealing with cultural, political, legal, and regulatory differences among the countries and regions in which the organization operates.
- **Deeper Dive 6.4** discusses the primary approaches to usability evaluation, including usability testing and usability inspections.

6.5.1 Deeper Dive 6.1: Interaction Design—Designing Interfaces for Retrieving and Analyzing Data

In addition to transaction-oriented screens, there are many other categories of user interfaces. A very common one includes interfaces that are primarily intended for retrieving and analyzing data and presenting it in a way that supports organizational decision-making at various levels: in short, data analytics.

The two main types of traditional output interfaces are reports and queries. The days are now gone when decision-makers at various levels would get huge piles of paper at regular intervals, including reports that would provide information regarding a specific aspect of an organization. The same information (and much more) is still available, but it's now made available through reporting systems that allow users to select the information they want at the time they want it. Predefined and structured reports are still valuable because they can provide the needed information in a carefully designed format that fits its purpose. However, equally important are flexible query systems that allow users to ask and get answers to questions predefined reports don't address.

Increasingly, both predefined reports and data queries are offered through specialized analytical and business intelligence (BI) systems, such as Tableau, Microsoft Power BI, and Qlik (the top three systems in the *2023 Gartner Magic Quadrant for Analytics and Business Intelligence Platforms*). These platforms are quite powerful and can easily integrate with both in-house and third-party systems. So, today it seldom makes sense to develop a data analytics system for a single system (or even a single organization). However, integrating the analytics and BI platforms with the specific data of other systems is not trivial and requires careful analysis and planning before implementation.

Regardless of the tool used, both reports and queries should follow relatively simple and straightforward guidelines, including:

- Ensure that you understand and consider all factors affecting the quality of the data that forms the foundation for the reports or the queries.
- Use tools for query and report production that you understand at a sufficient level of detail and verify your results with test materials. Errors in reports and queries can be very costly if they lead to suboptimal business decisions.

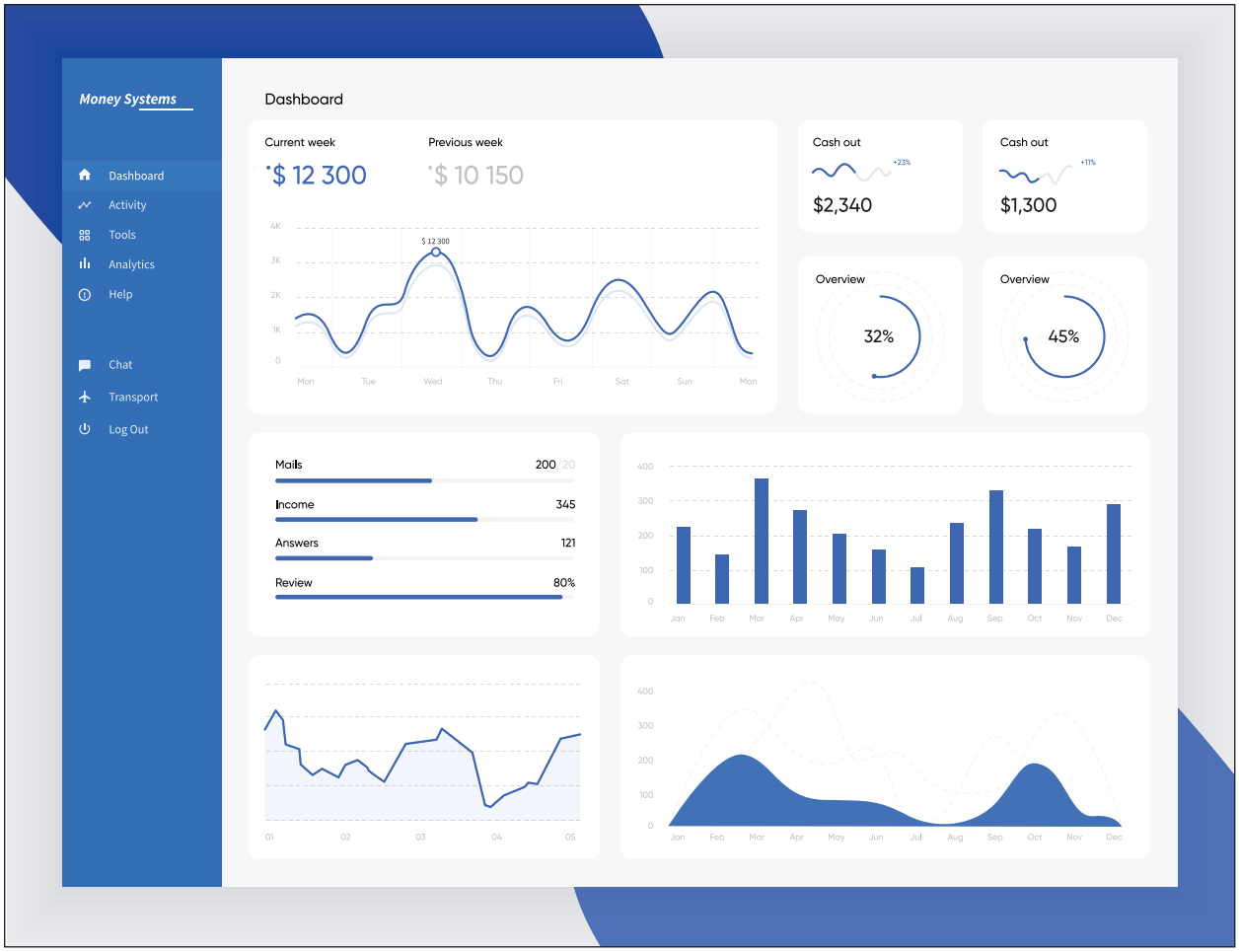


Figure 6-15 Dashboard example (iStock.com/Delices_89)

- For reports:
  - Include the necessary contextual information that makes the reports easy to read, follow, and understand, such as one or both of the following:
    - A report header and/or footer with a title, date produced, date range to which the report applies, and contact information for a person responsible for the report
    - A page header and/or footer with a page number (out of # of total pages) and appropriate abbreviated report level header/footer data repeated
  - Use a clear, easily readable hierarchical structure that includes headers, subtotals, and totals at appropriate levels of grouping (such as, for example, product, product group, product line, and company as a whole).
  - Properly format the numeric and textual data that emphasizes the role of the data included and make it easy to read. For example, make sure monetary values are formatted with a proper currency sign and appropriate number of decimals (typically two).
- For queries:
  - Include in the output as much information as possible regarding the specification (i.e., the executed query itself) that led to the result, including any specific parameters used to produce this query.

- Find the appropriate balance between sophisticated formatting and the expected lifetime of the query. For example, if the query is intended for one-time use, don't spend a lot of time on formatting.

Dashboards have become an important part of modern systems design. Their primary purpose is to provide an overview of an organization's performance based on key performance indicators (KPIs). The KPIs should represent the most important measurements of organizational performance, and the dashboard makes them available quickly and transparently, ensuring that all key decision-makers have the same information. Figure 6-15 presents a dashboard example.

6.5.2 Deeper Dive 6.2: Achieving the Same Goal with Multiple Interaction Channels

Chapters 4 and 5 intentionally left user stories and use cases abstract enough so we could implement the user-system interaction identified in them using a variety of different technical mechanisms. For example, the search for products was expressed with a user story in Chapter 4:

> As a customer in a retail store, I want to be able to use my own portable device to identify the physical location of a specific product so I don't waste time searching for the products that I want to buy.

In Chapter 4, this user story was illustrated with an early-stage smartphone interface prototype, but it's possible to implement it with multiple device types, including a tablet, smartphone, or a smartwatch. Instead of text input, we could have the user use voice commands. And if we ignored the "to use my own portable device" part of the user story, we could also implement this story using a kiosk. All of these approaches address the same user goal, and each one should be designed with a consistent UX/UI, to the extent the interface type differences allow.

As an example, let's consider how devices using mobile interfaces—smartphones, tablets, smartwatches, and kiosks, all of which in recent years have seen explosive growth—differ from those used on desktop and laptop computers.

The design of mobile interfaces requires close attention to the distinctive characteristics of various types of mobile devices, including:

- **Small screen size.** Despite significant increases in the size of the displays of modern smartphones, the screens are still considerably smaller than those used in stationary devices.
- **Poor screen visibility.** Particularly outdoors, even the best screens can be difficult to read and respond to if, for example, the glare of the sun is unavoidable.
- **Unreliability of connectivity.** Again, the reliability and geographic reach of mobile connectivity have improved significantly during the past 10 to 15 years. Still, in many parts of the world, wireless connectivity based on typical wireless carrier networks is far from perfect.
- **Use of the devices while multitasking.** It's not uncommon to see mobile application users try and balance the demands of using an application and simultaneously moving in a city from one place to another. Even more common is the use of a mobile application surreptitiously while communicating with other individuals face-to-face (e.g., in a meeting).

Comprehensive guidance for mobile interface design is beyond the scope of this chapter, but we'll provide a brief set of guidelines. For an in-depth understanding of the details of mobile interface design, we'd recommend you consult the applicable additional resources listed in Section 6.5.

Babich (2016) provides a good summary of key UX principles for designing high-quality mobile user experiences. The bolded labels are his, whereas the explanatory text is ours:

- **Cut out the clutter.** An uncluttered, focused interface is particularly important for mobile use, both to help the user in situations when attention is divided and to allow effective focus on key operations on a small screen.

- **Make navigation self-evident.** This is consistent with our earlier guidance for all environments to provide users with navigation consistent with the path(s) users need to follow to achieve their goals.
- **Design finger-friendly tap-targets.** Mobile interfaces should enable users to easily select intended targets even with large fingers, while moving, and with divided attention.
- **Text content should be legible.** Don't squeeze everything into a single screen by using a font size that makes text illegible for many users.
- **Make interface elements clearly visible.** It's important to use color and contrast in a thoughtful, intentional way so the resulting interface is legible and provides appropriate guidance, even under the constraints typical for mobile interfaces.
- **Design controls based on hand position.** Single-handed use of mobile applications is more typical than expected, and most mobile applications should make this possible.
- **Minimize the need for typing.** Despite advances in input technologies, mobile input still needs to be improved for users, particularly when moving and multitasking. Users will significantly appreciate mechanisms that reduce the need for typing.

These principles (in the same way as with Nielsen's heuristics or Shneiderman's golden rules) are not rocket science, but following them will lead to significantly better and more widely used mobile applications.

6.5.3 Deeper Dive 6.3: Designing User Experiences for Global Organizations

Systems intended for global use need UX/UI designs that support differences in users' language, legal/regulatory environment, and culture.

For languages, the screens and controls need to be translatable to multiple languages in a configurable manner (without coding). This is called *language localization*.

Legal and regulatory requirements may vary by geography. For example, at the time of this writing, the European Union's General Data Protection Regulation (GDPR) requirements require disclosures regarding the use of cookies and a service provider's use of personal data that don't exist in the United States. So, the UX/UI for systems used in both the EU and elsewhere need to either support GDPR specifically for use in EU countries or implement designs with that support everywhere (even where they aren't required).

Cultural differences should also be reflected in the design of systems targeted to a broad range of different markets, particularly when working on customer-facing systems. Providing comprehensive coverage regarding how to address cultural differences in user experience design is a broad topic and beyond the scope of this chapter. Both the academic research community (such as Cyr & Trevor-Smith, 2004, and Alexander et al., 2021) and design professionals (such as Shen, n.d.) discuss these topics at length, and we highly recommend that you learn more about this important topic area. For example, Shen's research (Shen, 2016) provides several intriguing examples based on a travel site that operated in 17 countries and customized its context for all of those markets. For instance, Dutch users of the site would provide incorrect name information in travel reservations despite a detailed warning against doing so. Shen concluded that users in this market would likely be motivated by loss aversion and, so, recommended using a fee in case of incorrect name data. Shen also found out that German users preferred to see estimations of auto travel expressed in terms of distance instead of time (as would have been the preference in the United States).

Usability evaluation and measurement of the impact of design changes in different markets are essential parts of the global design processes. Trusting one's initial assumptions is insufficient.

6.5.4 Deeper Dive 6.4: Usability Evaluation

Usability evaluation is a critically important element of UX/UI design. Without it, there's no evidence to suggest that users will use the system at all, let alone use it for its intended purpose. The two primary forms of usability evaluation are *usability testing* and *usability inspection*. The former involves current or intended users of the system, whereas experts perform the latter.

Either form of usability evaluation can be conducted throughout the systems development process. In the Systems Development Process Framework (Figure 6-1), usability evaluation can be used as early as the end of business analysis, but it will provide more detailed results after functional design. The test results are closest to reality after the iterative development cycles (sprints) and ultimately after the sprint results are integrated into releases.

Both forms of usability testing can be performed by either the development team/organization responsible for the product or an external consulting organization specializing in usability evaluation. An external organization can be highly beneficial because it can offer specialized expertise in testing and an unbiased analysis of the test results, but using a consultancy can be costly. Having usability evaluation expertise within the development team is highly beneficial.

6.5.4.1 Usability Testing

Usability testing A process in which intended system users perform specific tasks with the new system to identify potential problems with it.

Usability testing is a process in which intended system users perform specific tasks with the new system, and through this process identify potential problems with it. According to Nielsen (1994, p. 165), "User testing with real users is the most fundamental usability method and is in some sense irreplaceable, since it provides direct information about how people use computers and what their exact problems are with the concrete interface being tested." Indeed, conducting usability testing with users has several benefits: it will give strong evidence regarding whether the users will be able to complete the intended tasks successfully, provide information about typical errors and the time it will take for the users to complete the tasks, show whether the users are satisfied with their experience with the system, and give guidance regarding the changes that are needed (https://www.usability.gov/how-to-and-tools/methods/usability-testing.html). Of course, usability testing is not without its challenges: it's a costly way to evaluate a system component, it has to be planned correctly and executed carefully to maintain the reliability and validity of the tests, and it will require a significant amount of time to do well. The specifics of comprehensive usability testing are beyond the space available in this book, but, fortunately, excellent sources of information are available for free—for example, at https://www.usability.gov.

6.5.4.2 Usability Inspection

Usability inspection is a process that user experience professionals conduct following a predefined inspection protocol; in these methods, no intended users are involved. Several different usability inspection methods are available, including the two most widely used ones: heuristic evaluation and the cognitive walkthrough.

Heuristic evaluation A usability inspection process in which a small group of experts evaluates a set of system capabilities against a set of heuristics.

In **heuristic evaluation**, a group of three to five experts evaluates the target (an application, system component, website, etc.) against a set of predefined principles (such as Nielsen's heuristics or Shneiderman's golden rules, discussed earlier) to find out where the system is not aligned with the heuristics. Typically, it's not possible to evaluate the entire system, so it's vital to prespecify the scope of the evaluation. Research evidence (Hollingsed & Novick, 2007) suggests that heuristic evaluation is effective in finding most of the usability problems with the target. Still, usability testing often reveals additional problems and is more likely to lead to the identification of all severe problems.

Cognitive walkthrough A usability inspection process that focuses on the learnability of the new system. Conducted by a group of evaluators guided by a conceptual framework.

The **cognitive walkthrough** is also an evaluation method that uses nonuser evaluators, but its main focus is on the system's learnability for new or infrequent users. As such, it's beneficial for evaluating systems that offer little or no training (such as e-commerce sites, kiosks, ATMs, etc.) or the first experiences for all types of systems. A vital element of the cognitive walkthrough process is the identification of the tasks to be evaluated, followed by a guided process in which evaluators walk through the sequence of steps that a typical target user would likely take. The method provides detailed forms for documenting the results of the evaluation.

7.5 Deeper Dives: Advanced Topics on System Acquisition

In this section, we do deeper dives on advanced topics that build on and expand coverage from earlier in the chapter:

- **Deeper Dive 7.1** discusses the implementation and maintenance processes that are required for the effective use of various types of purchased systems.
- **Deeper Dive 7.2** describes the benefits and disadvantages of the use of open-source software.
- **Deeper Dive 7.3** explores various software contract types.
- **Deeper Dive 7.4** discusses how software and human resource sourcing decisions affect systems development methodology choices.
- **Deeper Dive 7.5** explores specific issues related to the acquisition of AI capabilities and sourcing AI expertise.

7.5.1 Deeper Dive 7.1: Implementation and Maintenance Processes for Third-Party Systems

Few third-party systems (with the possible exception of traditional packaged software) are ready to serve any organization without a significant amount of implementation work. This work is typically executed as a collaborative project with the client organization, the vendor, and often a separate implementation partner (i.e., a consultant helping with the process). During these processes, a broad range of issues needs to be addressed, belonging to the following categories:

- **CONS (Construction):** Activities that lead to software code-level changes to the purchased system solution or around it. Such activities can include the creation of glue code to integrate the product with the rest of the application portfolio or, less commonly, changes to the core product itself.
- **CONF (Configuration):** Virtually all purchased system solutions require a significant amount of configuration work, in which the system's options are set in a way that best supports the client organization's needs.
- **CM (Change management):** Organizational change activities that enable the organization to effectively use the COTS-based solution. These activities typically include data preparation, updating organizational policies and procedures, creating training for system users, and user acceptance testing.
- **DEP (Deployment):** Activities that move the organization from its current systems to the COTS-based solution. We explain deployment approaches in Chapter 16.
- **MAN (Maintenance):** Activities that fix errors and introduce new features to a functioning application that has already been deployed.

As noted in Chapter 1 and as discussed further in Chapter 12, implementing the same software package repeatedly in different organizations can often best be done by creating a repeatable series of implementation steps. This often leads to a highly plan-driven process, both for requirements analysis and for implementation via configuration. On the other hand, the construction of glue code may still be done via a more agile, sprint-based approach operating in parallel with the configuration of the software. Often today, the configuration steps are also scheduled to be completed within development sprints. This is useful in providing a regular checkpoint to review and verify the just-completed configuration steps, as well as helping to keep configuration and construction work in sync with each other.

An incomplete but representative list of the issues that need to be addressed regarding third-party systems is provided as follows (and summarized in Figure 7-6):

- Is there a need to *tailor* the purchased systems somehow for the organization? (Typically, the goal is to do as little tailoring as possible.) [CONS]
- How will the purchased systems be *integrated* with each other and existing systems within the organization? [CONS]
- How will the purchased systems need to be *configured* to make sure they perform as specified? [CONF]
- How will the purchased systems be used to *support the organization's business processes*, and how much will the *processes need to be changed* to get full benefits out of the best-practice support and other features of the purchased systems? How will the new processes be *documented*? [CM]
- What type of *training* will employees need to be able to use the purchased systems appropriately? [CM]
- How will *user acceptance testing* (UAT) and *data preparation* be coordinated with other implementation activities? [CM]
- Will new *organizational support mechanisms* be needed because of the purchased systems? How will support personnel and superusers be trained? [CM]
- How will the purchased systems be deployed? How will the *transition* between existing and new systems take place? How will the existing *data* be copied to the purchased systems? How will *conflicts between existing and new business processes* be addressed? [DEP]
- How will the *ongoing relationship with the vendor* be organized? [MAN]
- How frequently will *new purchased system features be received and implemented*? What is the expected release cycle, and how will the new releases be introduced with as little interruption as possible? [MAN]

Construction (CONS)	
	Tailoring
	Integration of component applications and systems (glue code)
Configuration (CONF)	
	Configuration of systems features with parameter setting
Change management (CM)	
	Business process changes
	Business process documentation
	Training
	Support mechanisms
Deployment (DEP)	
	Transition period
	Transferring data to the new system
	Managing business process transition
Maintenance (MAN)	
	Managing relationships with vendors
	Release management

Figure 7-6 Third-party system implementation activities

The code in square brackets after each item on the list denotes the category of work it belongs to, as indicated in Figure 7-6.

Adopting a third-party system doesn't mean the organization can simply negotiate a price with a vendor, install the code on a server, and ask the employees to point their browsers to a new URL. Fortunately, very large numbers of organizations have addressed these and other questions over the past 40-plus years, so support is available in the form of both literature (e.g., Finney & Corbett, 2007) and implementation partners. Still, both academic literature and practical anecdotes indicate that the risks of a failed implementation of a third-party, mission-critical system are real, and the costs of failure can be significant, sometimes even catastrophic. Such a system implementation process must be planned very carefully, and it requires organization-wide participation and commitment at all organizational levels, including top management.

Open-source software (OSS) Software that developers make available (together with its source code) for free.

7.5.2 Deeper Dive 7.2: Open-Source Software

One of the truly unique features of the software industry compared to most other businesses is the existence of the **open-source software** (OSS) model. Software developers following OSS make available for free not only software capabilities but also the source code, which users and user

organizations can then freely modify and apply for specific purposes. Often the licenses specified for OSS dictate that all derivatives of a specific OSS solution also need to be available to the public using the same license under which the original software was made available. The motivations of individuals and organizations to contribute to the development of OSS vary significantly:

- Some individuals find it highly satisfying to see the results of their work being used broadly around the world.
- Some consider work on OSS to be an excellent learning opportunity.
- Others believe that their contributions to the software allow them to build a business based on consulting and support services around it.
- Some organizations find it cost-effective to share the costs and benefits of development work with other organizations through employee contributions to OSS.

OSS as a model has been highly successful in the context of systems software, including the Linux operating system and its distributions (such as Ubuntu and Red Hat), other open-source UNIX variants (FreeBSD, OpenBSD, and NetBSD), MySQL and MariaDB database management systems, Apache HTTP Server, and so on. In that list, 6 projects were categorized as IT Operations (including Linux, #1) and 11 were categorized as Data & Analytics (for example, MySQL, #3; Hadoop, #6; MongoDB, #9; Redis, #12; PostgreSQL, #16; HBase, #22; Cassandra, #25). Both categories consist primarily of systems software solutions.

Open-source software categories

- Systems software for IT operations
- Systems software for data and analytics
- Development tools and environments
- Application software

A significant portion of the internet infrastructure runs on OSS. In addition, OSS products in the systems software category are vitally important for the internal infrastructure of many organizations around the world and even in space—the computing infrastructure of the International Space Station uses Linux.

However, in this book, we're primarily focused on developing application software rather than systems software.

The application software packages currently available as open source are nowhere near as important as the systems software solutions. However, interesting OSS applications do exist and are used by millions of people around the world every day. They include ERP systems (such as ERPNext and Odoo), CRM systems (SugarCRM), data mining packages (RapidMiner), learning management systems (Sakai), and so on. Some of these applications, such as RapidMiner and Sakai, are clear success stories within specific application categories. It always makes sense to know what the available options are. Still, it's relatively unlikely that you'll be designing an architecture that is built around an open-source enterprise system or even a set of administrative open-source applications.

Somewhere between the two prior categories are open-source development tools and environments, including the PHP and Python programming languages, the Eclipse integrated development environment (IDE), and many open-source application development frameworks and toolkits, such as Ruby on Rails, Django, Bootstrap, and Node.JS. On the TechCrunch list discussed previously, 8 products out of 25 are in the development and operations tool category (for example, DevOps, which we explore in Chapter 16), including Git (code repository, #2) and Node .JS (backend framework, #4). It's highly likely that you'll use multiple products from this category if you're involved in systems development projects as a developer. Some of the open-source data analytics tools are equally likely to be in your development toolkit, including some of the database environments listed previously, but also Apache Spark and Apache Kafka. Many enterprise application integration tools are also open source, such as Apache Camel.

Note that many companies in the open-source world have built very successful businesses around "free" software, selling, for example, specialized enterprise versions of software products, consulting related to the deployment of OSS for specific organizational purposes, or providing ongoing support. These commercial services address a major risk that organizations face in using OSS. Specifically, in OSS, any programmer or other participating organization can update the

solution. This openness introduces risks that the software may break or suffer from security vulnerabilities. Also, when using OSS, an organization often lacks software support when problems arise. By using an enterprise version of an OSS solution, an organization can often still benefit from low or zero licensing costs while addressing these control and support risks for a reasonable ongoing support fee.

For example, Red Hat, which offers the most popular enterprise Linux solution, development tools for the Linux environment, and infrastructure management solutions, was purchased in 2019 by IBM for $34 billion, after reaching annual revenues of $3.4 billion and an operating income of $512 million as an independent company. Most enterprises using open-source products at the core of their technical architecture, pay for support to ensure security and continuous availability.

What does all this mean to you as a business analyst? You'll continue to see OSS used as a key component of computing infrastructure, particularly in the operating system and data management categories. Open-source products will also continue to form a significant portion of the software development tool category, including actual development environments and communication and coordination tools. Many of the systems that are used to integrate application software are also available open source. However, you're much less likely to find (or want to use) open-source application software that could directly support your business processes.

7.5.3 Deeper Dive 7.3: Types of Contracts with a Vendor

7.5.3.1 Application Software Development and Maintenance

Vendor contract types vary significantly, depending on the type of service being contracted. In application software development and maintenance outsourcing, a key choice is between fixed price (FP) and time and materials (T&M). In a fixed-price contract, the client and the vendor agree on a specific set of services and a price for them, whereas in a T&M contract, the client pays for the vendor's work and associated expenses as the vendor expends resources on that work. The cost of the work is generally based on the vendor's number of work hours, days, or another time unit. In FP, the client pays for specific outcomes of the development and maintenance work, but in T&M, the payment is for the vendor's work inputs. A key factor for both parties in determining the form of the contract is the extent to which the work can be specified at a detailed level: a vendor is unlikely to entertain the idea of an FP contract if the project is highly open-ended, but for a client, that would be an optimal choice.

7.5.3.2 Enterprise Systems and Other Third-Party Software

When implementing an enterprise system or other third-party software systems, the contracts are different from tailored application software development. These contracts cover issues such as software licenses, software acceptance, intellectual property rights, confidentiality, liability, insurance, warranties, and dispute resolution as part of the master contract. Additional documents specify work related to a particular part of the system life cycle (such as implementation), including statements of work, consultancy and support arrangements, and the details of the modules included in the contract at a particular point in time (Donagher, n.d.). Marcus Stephen Harris (n.d.) identifies, among others, the following specific aspects that are essential to cover in an enterprise systems contract:

- In addition to the client's employees, who else will use the system?
- If third-party products are used, are they included in the contract, and under which terms?
- What is the licensing basis, such as site licenses (for a specific geographic location or a company); the number of (simultaneous) users; or revenue?
- What are the policies regarding disaster recovery?
- Regarding maintenance and life cycle issues, what is included in the base price, and what is covered by the (typically annual) maintenance fee?
- What is the specific meaning of any warranty and actions in the context of a warranty breach?

- What are the vendor's indemnity obligations in case of disputes from a third party? Who should be responsible for the dispute resolution process if a third party claims the system violates its intellectual property rights?
- What are the policies related to keeping the software source code in escrow and the situations when it will be released (typically a vendor's failure as a company)?

Harris also recommends separating the enterprise systems *licensing contract* (discussed previously) from the *implementation contract*. The latter will address the work that needs to be done to deploy the software code owned by the vendor to the client's organization in a way that will allow the client to achieve its goals for the system. The implementation contract will cover the allocation of responsibilities for configuration, business process design, data preparation and transfer to the new system, system evaluation, training, and so on, among the vendor, the client, and the implementation consultant(s).

7.5.3.3 Cloud Infrastructure for Systems Development and Deployment

Increasingly, IT organizations are buying essential systems software services from cloud service providers (such as Amazon's AWS, Microsoft's Azure, Google's Cloud Platform, etc.). Gilbert (2011) recommends the following elements be included in a cloud computing contract:

1. Rights related to modifying the contract terms. Is the vendor allowed to change the contract terms unilaterally?
2. A clear description of the included services, including service-level agreements.
3. Restrictions related to the use and reuse of the client's data (often essential even from the legal perspective), including an agreement regarding the location of the data.
4. Requirements related to confidentiality and security.
5. Relevant provisions regarding intellectual property rights. Will the service provider have any right to the clients' data maintained on the servers? What specific services will the client have access to?
6. Representations and warranties, which specify the recourses when one of the parties has made untrue statements or not fulfilled promises.
7. Indemnification—specifying that a party in a contract is not liable under specific circumstances (such as inappropriate use of a third party's intellectual property).
8. Limitation of liability and damages.
9. Term and renewal of the contract. Which time frame does the contract cover, and under which conditions can it be renewed?
10. Termination of service. What happens if the service is terminated? For example, what are the terms for the customer to move their data to another service provider? This is not necessarily trivial, given that there could be hundreds of terabytes or even multiple petabytes of data.

7.5.4 Deeper Dive 7.4: The Effect of Software and Resource Sourcing on Development Methodology

The decisions discussed throughout this chapter are closely associated with decisions that any organization must make regarding software development approaches and methodologies, as will be explored further in Chapter 10. For example:

- **Agile and resource locations:** Suppose an organization has followed a "pure" agile methodology. In that case, the methodology itself expects the software development resources to be colocated or at least capable of being connected with communication technologies that create an illusion of colocation. Consequently, significant time-zone differences will unavoidably create challenges for "pure" agile development.
- **Agile and fixed-price contracts:** Using agile development also makes FP contracts difficult because agile uses a flexible scope of work. This makes it tough to set a fixed price for a fixed application scope. If an FP contract is used in agile development, it's

more likely to specify a fixed budget and timing, with the contract being reviewed frequently to verify that the client organization is realizing enough value from the work for it to continue.

- **Requirements when the team is in multiple locations:** Suppose an organization uses an outsourcing arrangement with geographically and organizationally distributed resources. In that case, it's often impossible to use them without some type of formal and relatively detailed up-front requirements (BRUF) of at least functional system capabilities (for example, using methods such as fully dressed use cases and system sequence diagrams, which we covered in Chapter 5 and will cover in Chapter 14, respectively).
- **Functional designs and third-party software:** Suppose a solution that an organization chooses is largely based on third-party software. In that case, it's possible that no detailed functional design specifications (such as those covered in Chapter 5) are needed because, in many cases, the detailed features of the system are determined by what the licensed software provides and can't be easily changed. However, functional design specifications can be used as a point of reference when evaluating various purchased systems against each other.

These four points are just examples, but it's important to consider the connection among the decisions regarding the chosen development methodology, the sourcing of software (components), and the sourcing of development resources.

7.5.5 Deeper Dive 7.5: Acquisition of AI Capabilities and Sourcing AI Expertise

As discussed in Chapters 2 through 5, new systems capabilities increasingly require the use of artificial intelligence (AI) technologies of various types. In this section, the key question is how sourcing and locating AI capabilities and expertise might differ from conventional (programmed) system capabilities.

Many factors affecting the configure versus construct decision regarding AI capabilities are the same as those for conventional software. For example, Walch (2021) identified the following dimensions for these decisions:

- Availability of specialized expertise
- The urgency of the need
- Compatibility with existing systems
- Resulting dependency on vendors (vendor lock-in)
- Total cost of ownership of different options
- The number and variety of solutions that are available for purchase
- Industry and regulatory considerations

Every one of these factors applies to configure versus construct decisions for both AI and conventional software. However, there are differences within each factor, and we'll discuss some of them here.

A factor that is especially important for AI is the availability of human expertise. There's a shortage of competent experts in both developing and applying AI. The client organization needs technical experts to identify and create AI capabilities *and* organizational experts to design the integration of AI with other, conventional system components. This includes considering the nondeterministic nature of the AI components. (AI software may not always generate the same outputs for a given set of inputs.)

AI expertise is not cheap, but not having it may become even more expensive, particularly if an organization decides to construct some of its AI capabilities internally. Starting internal development without a strong group of development resources can lead to wasted time and money. The (lack of) availability of internal expertise is also a major factor affecting an organization's dependency on AI vendors and vendor lock-in (the extent to which AI capabilities integrated with organizational systems lead to a long-term dependency on external vendors). It's essential to

understand (a) who owns the AI training data and the models underlying the AI capabilities and (b) what happens if the vendor fails.

It's also essential to consider how close the required expertise is to the organization's core competencies. For example, if we think about the property insurance company discussed in Chapter 2 and its AI-based roof inspection system, it's likely that this company would not develop specialized expertise internally related to the software used to control and monitor the drones required in the inspection process. Instead, it would be natural for the company to use purchased solutions and their built-in expertise (or a specialized consulting firm) to gain these capabilities. However, the insurance company might conclude that the AI capabilities needed to evaluate the images of the roofs are at the very core of what they do and, so, decide to develop these system capabilities internally.

Suppose an organization identifies an opportunity for the use of AI. In that case, it's essential to determine how critical it is to be a first mover using the discovered capability. How likely is it that the first users of a capability will gain a significant advantage over those who follow later? If the first mover advantage within a certain industry or market is clearly present, finding a solution sold by an external vendor that is not yet widely known within the industry might be the only way to move fast enough.

As with any software-based system, it's important to understand the available solutions from external vendors. The availability of a healthy market of existing capabilities suggests that building one's own capabilities might not make sense. At the same time, the availability of a broad range of applications in an area indicates that the organization might have overlooked an interesting opportunity and should act quickly to evaluate the value of widely available capabilities.

Within AI, there's a tendency for a specific area (such as, say, robotics, computer vision, expert systems, large language models, etc.) to become a focus of attention, pulling focus away from others. However, it's useful to remember that even though the latest discussion has been largely focused on generative AI and large language models, many other areas of AI are also potentially highly valuable and moving forward at a high speed.

When considering the construct versus configure decision for AI-enabled capabilities, it's essential to understand the current and future regulatory requirements within the relevant industries. For many organizations in heavily regulated industries (such as healthcare or financial services), maintaining sufficient internal capabilities to fully understand the regulatory requirements might be too much to make it possible to depend on internal staff. One of the benefits of using an external vendor is gaining access to a shared understanding of those requirements.

Chapter 8

System Cost Estimation

8.5 Deeper Dives: Advanced System Estimating Topics

In this section, we do deeper dives on several advanced estimating topics that all build on and expand coverage from earlier in this chapter:

- **Deeper Dive 8.1** discusses the critical difference between estimates and commitments.
- **Deeper Dive 8.2** analyzes the importance of aligning our estimation approach with the characteristics of projects and organizations.
- **Deeper Dive 8.3** presents other estimating approaches beyond planning poker.
- **Deeper Dive 8.4** introduces advanced principles for improving system cost estimates.
- **Deeper Dive 8.5** presents factors that need to be considered while estimating configuration projects
- **Deeper Dive 8.6** discusses the process of estimation in projects that integrate AI-based components.

8.5.1 Deeper Dive 8.1: Estimates versus Commitments

We've noted that there's a systemic problem in the IT industry of underestimating projects. On top of this, the Cone of Uncertainty shows that it's impossible to be highly accurate in your software labor estimates prior to creating detailed requirements. This is compounded by the fact that a system cost estimate can only be overestimated by a fixed amount of money but technically can be underestimated by an unlimited amount of money. For example, if the cost of a project is estimated at $500,000, then the maximum overestimate is $500,000, because a project can't cost less than $0. Practically speaking, the real maximum overestimate would likely be far less than $500,000 because, even if overestimated, the cost of the project is likely to be much higher than $0. On the other hand, unfortunately there's no reason why this project couldn't end up costing (and so be underestimated by) millions of dollars, a result that has occurred far too many times in our industry.

Commitment In software projects, a commitment is a promise to the organization to deliver a certain amount of scope in a specific amount of time and within a certain budget.

Because of all of these issues, you need to avoid presenting an early, inherently inaccurate estimate to business clients and then allow them to press you into agreeing to that estimate as a **commitment**. To be clear, in contrast to an estimate, which is merely an informed prediction of the project cost, a commitment is a *promise* by the team to deliver the specified functionality by a particular date and for a not-to-exceed budget (fixed scope/fixed time/fixed budget). As such, prematurely committing to a (likely too low) system cost estimate is an all-too-likely way of locking your team into a no-win situation, where it's impossible to meet scope/time/budget promises. As such, you need to be clear with your clients that estimates and commitments are two very different things.

Instead, when a client asks for a commitment, the Cone of Uncertainty suggests that you defer making it until as late in the requirements process as possible. Ideally, to reach ±25% estimate uncertainty, this would include completion of all functional requirements, up to and including user interface designs. The commitment budget should ideally be the "most likely" estimate plus 25%.

On the other hand, there are organizations and projects—agile ones—that don't require commitments to estimates. We discuss this issue in depth in Section 8.5.2.

In the case of large, complex software projects, where the dominant hybrid project approach typically will include a high degree of big requirements up-front (BRUF), you should ideally (re)estimate the project at each project requirements milestone, using planning poker and other estimating approaches as noted in Figure 8-1. We explore those additional estimating approaches in Section 8.5.3.

8.5.2 Deeper Dive 8.2: Tuning Estimation to the Needs of Different Organizations and Projects

How accurate do our estimates need to be? Is a commitment, as described in the previous section, even necessary? There's no simple answer to that question: organizations and projects within those organizations vary greatly in terms of the approach and outcomes they value.

For example, **traditional organizations** place high value on predictable, high-assurance delivery of software and typically do seek a commitment from the systems team. Such organizations are often large, mature, and operate in stable market environments. As such, they will tend to put a lot of effort into creating detailed up-front requirements that result in highly accurate estimates, typically today using a hybrid project approach (rather than a plan-driven approach). This lowers the risk of delivering a project that is over budget and fails to deliver a positive return on investment.

Traditional Organization: An organization that values highly predictable and low-risk creation of system capabilities.

On the other hand, **agile organizations** focus on achieving rapid, responsive delivery of value. Often, they are business start-ups or organizations operating in turbulent, rapidly evolving industries, where a high degree of responsiveness is needed to survive and thrive. Such organizations may be considered "agile" in terms of their overall management, not just their software project approaches. As such, they tend to use agile project approaches that use emergent requirements and place less emphasis on the importance of up-front estimates. In this way, they can be highly responsive while maintaining an eye toward ensuring that the systems they build meet their evolving business needs over time.

Agile Organization: An organization that values rapid delivery of software capabilities in a highly responsive manner with respect to changing system requirements.

Note that today both types of software projects may start requirements and estimating in a similar place, by initially envisioning software requirements in only general terms, as a series of epic user stories. These epics are so general that they can't be estimated with a high degree of accuracy. Instead, the best that can be accomplished are "ballpark" estimates using techniques such as "T-shirt sizing." (This and other estimating techniques introduced in this section are explained in detail in Section 8.5.3.)

Moving beyond this high-level vision, typically both types will then work to decompose the epics into a series of regular user stories, often annotated with acceptance criteria, enabling a higher (but still low overall) level of estimating accuracy. A typical estimating approach at this point would include planning poker or "expert judgment."

However, at this point, the two types diverge in key ways, affecting the types of estimates and level of estimation accuracy possible. For example, the traditional approach develops BRUF in greater and greater detail, enabling higher and higher estimating accuracy prior to software construction, per the Cone of Uncertainty. In contrast, the agile approach emphasizes launching software construction projects based on the regular user stories, determining designs throughout the project using many informal, face-to-face conversations.

We offer advice for estimating in traditional and agile organizations in the next subsections.

8.5.2.1 Traditional Organizations and Projects: Handling Commitments with Progressive Estimates

Traditional projects using plan-driven or hybrid approaches tend to continue to expand and refine user stories using the BRUF approach, including detailed functional designs of the domain model, user interface, and software logic. Assuming this approach is used in appropriate situations (meaning projects where requirements can be clearly understood prior to construction and are generally stable over time) this can produce highly accurate system cost estimates. Additional estimating techniques that can be employed with highly detailed requirements are described in Section 8.5.3.

However, when you're asked to make a commitment prior to having completed detailed requirements, the natural tendency is to increase your "most likely" estimate by a significant amount, ranging from 20% to 100% or even more, as you try to create a "safe" estimate you can commit to. This is sometimes called "padding" the estimate and can be viewed dimly by business clients who must actually provide the funds for the project. Unfortunately, that padding often turns out to be

Table 8-2: Ranges of estimation error at project phases			
	Estimation Error		
	Low Side	High Side	Range of High to Low Estimates
Initial Concept	0.25× (-75%)	4.0× (+300%)	16×
Approved Product Definition	0.50× (-50%)	2.0× (+100%)	4×
Requirements Complete	0.67× (-33%)	1.5× (+50%)	2.25×
User Interface Design Complete	0.80× (-20%)	1.25× (+25%)	1.6×
Detailed Design Complete	0.90× (-10%)	1.10× (+10%)	1.2×

inadequate, given that, per the Cone of Uncertainty, at the outset of a project the variability around the most likely estimate can range far beyond ±100%.

This creates tension between the client's' desire for an early commitment versus your team's need to defer committing until highly detailed requirements are in hand.

A better approach is to educate business clients about the Cone of Uncertainty, showing them that a high degree of accuracy in the early stages of a software project is simply impossible. Rather than estimating and committing to a single estimate early in a project, a better approach is for the overall team, both business clients and the IT team, to agree to an approach where estimates are progressively refined as requirements become more detailed. This can be an attractive approach because in these types of projects, the BRUF typically are created early in the project by a few BAs, requiring a relatively small amount of money compared to the subsequent costs of software construction by the entire team. Table 8-2 summarizes suggested acceptable ranges of error at each phase of a project, per the Cone of Uncertainty (McConnell, 2006). Before launching major software construction, you should reach the level of estimation error, with corresponding padding, that the business can tolerate.

The lesson here is simple: if you're working in a traditional environment with a hybrid approach, then focus on creating detailed up-front requirements. This could include detailed domain models, user interface models, and use cases. This could be followed by increasingly accurate estimates based on planning poker or one of the other, more detailed estimating approaches described in the Deeper Dives section.

On the other hand, if you're working for an agile organization, start coding as quickly as you reasonably can with less detail, perhaps not more than user stories and some informal notes about the activity diagrams and domain models.

8.5.2.2 Estimation and Cost Management for Hybrid Projects

Regarding traditional environments, you must remember that most traditional environments today use the hybrid approach, where BRUF is then subject to a degree of requirements revisions during sprints. How do you take this into account in your estimates?

The hybrid approach provides you with flexibility to manage scope (and, therefore, estimates and costs) in two ways:

- **Guardrails scope:** While plan-driven and hybrid approaches both include a significant amount of BRUF, in a hybrid approach you can use "guardrails" scope to gain flexibility in managing estimates and budgets. First, your minimum viable product (MVP) scope consists of all "Must Have" (MH) items, which you should estimate and include in the budget. You should estimate "should have" (SH) and "could have/nice to have" (CH) items, but you can omit them if the project budget appears to be too high. Put plainly, omitted items become "won't haves" (WH). Note that the inclusion of SH and CH items in the budget allows you to manage the project to a successful overall outcome even if overall costs threaten to exceed the original estimates. If necessary, you can exclude some SH and CH items after the project starts to deliver your MVP within the overall project budget. These SH and CH items may end up being included in a subse-

quent project (implying a later software release) under a separate budget for that project. Also, note that SH and CH items to be omitted could include specific acceptance criteria within user stories, in addition to the option of omitting entire user stories.

- **Requirements revisions during sprints:** Given the use of iterative or sprint-based construction, hybrid projects include the ability to allow a degree of scope revisions, or "tweaks," to requirements. Typically, these revisions emerge at the end of each sprint during sprint reviews. Revisions may occur because the original requirements are unstable or you identify new ideas for additional, valuable features as the project executes (McConnell, 2006, pp. 42–44). If you anticipate a significant need for revisions, it's wise to include a degree of additional project budget to accommodate them. You can best do this by assessing the additional effort and cost of requirements revisions from prior projects. For example, if you know that in previous projects you've ended up spending about 30% more than original estimates to accommodate requirements revisions, you can add that to your current project's estimates based on BRUF.

8.5.2.3 Avoid Commitments in Agile Projects

Finally, if an organization executes its software projects in a truly agile fashion (combining emergent requirements with iterative or sprint-based development) that organization shouldn't attempt to implement a budget commitment for a set amount of scope. When "pure" agile is used, it's likely that the project has unclear and rapidly changing requirements, as is the case, for example, in many start-up or proof-of-concept (PoC) projects. Because of this need to support flexible, rapidly changing scope, the budget and timing of a project may be fixed (based on X number of IT team members allocated for Y number of weeks or months), but you can't reasonably commit to any guaranteed minimum scope (resulting in flexible scope/fixed time/fixed budget).

Put more plainly, making commitments in an agile project is a recipe for project failure. Don't do it!

Rather, in this situation, it's appropriate at the end of each sprint to review the value of features constructed and delivered in relation to costs incurred. Remember that agile approaches may be most appropriate for organizations that are themselves agile, in the sense that they value rapid, highly responsive software projects.

8.5.3 Deeper Dive 8.3: Beyond Planning Poker—Other Estimating Approaches

This section explains several additional software labor estimating approaches:

- T-shirt sizing
- Expert judgment
- Function point analysis

Roughly speaking, these approaches are presented in order from least effort and least accuracy to most effort and most accuracy. Also, given that more accurate system cost estimates generally come later in a systems project (because of greater, more detailed elaboration of requirements), this order also tends to correspond to the stages of the project where estimates are generally performed, per Figure 8-1.

T-shirt sizing and expert judgment are like planning poker in that they are based on the subjective judgment of the IT team members.

In contrast, function point analysis replaces subjective judgment with objective estimation methods (fundamentally, counting software design items like screens, tables, and reports and then using formulas to transform those counts into estimates).

T-shirt sizing System cost estimation approach in which very high-level requirements are used to generate rough estimates using category labels similar to T-shirt sizes: Extra Small, Small, Medium, Large, Extra Large, etc.

8.5.3.1 T-Shirt Sizing

In **T-shirt sizing**, the IT team assigns very rough estimates to a user story or feature (or, more typically, an epic or even several epics together), based on their subjective judgment. These estimates are designated using category labels similar to the sizes of a T-shirt: Extra Small, Small, Medium, Large, Extra Large, and so on. You should use T-shirt sizing early in a project (during

Table 8-3: Example T-shirt sizing labels and labor amounts	
T-Shirt Size	**Labor**
X-Small	0.25 FTE Month
Small	0.5 FTE Month
Medium	1 FTE Month
Large	3 FTE Month
X-Large	6 FTE Month
XX-Large	12 FTE Month

Initial Visioning) when you have little requirements detail and the Cone of Uncertainty tends to be at its widest (McConnell, 2006, p. 145). As such, T-shirt sizing is typically used to help organizations evaluate features based on "directionally correct," ballpark estimates. This denotes estimates that are not highly accurate, but, rather, should be of the same order of magnitude as what the actual costs would be (Ranadive, 2013). In this way, T-shirt sizing estimates are sufficiently accurate to help determine which system features are worth additional requirements analysis and which ones are obviously too expensive to continue thinking about.

Table 8-3 presents one example of T-shirt sizing categories and labor amounts. These categories and amounts are not standardized, so different organizations and teams will choose values that are meaningful to them. A larger organization could use much larger labor amounts—for example, 6 FTE months might be a Medium-sized project.

As an example, if a feature is evaluated as XX-Large using Table 8-3 and hourly labor rates are $75, then software labor costs might be in the order of magnitude of:

12 FTE Months × 160 hours per month × $75 per hour = $144,000

Given this estimate, if the business value is ballparked at, say, $300,000, the project might be worth further analysis. On the other hand, if the business value is only $75,000, the feature could be quickly weeded out from further consideration.

8.5.3.2 Expert Judgment

Expert judgment
An approach to system cost estimation in which expert(s) in requirements and development generate estimates by comparing a current user story or product backlog item to similar items built previously.

Although lacking a catchy label like "T-shirt sizing" or "planning poker," individual **expert judgment** is among the most frequently used software estimating techniques (McConnell, 2006, Chapter 9). Like planning poker, expert judgment is based on the subjective judgment of team members. However, unlike planning poker, rather than the whole team estimating every user story, in expert judgment each user story estimate involves only those team members who have the deepest involvement and expertise with that story's requirements and program code. For functional requirements, this typically means the Product Owner, the BA, or both. For the technical design and coding, this typically means a technical Scrum Master or Development Lead and the developer who will actually program that user story.

Because expert judgment involves fewer team members, this approach is sometimes preferred over planning poker for large product backlogs and large development teams, simply because it requires less time overall from the team members. And because expert judgment generates each user story estimate based on the expertise of team members with the deepest and most relevant experience, it can be as effective as planning poker.

Expert judgment can be performed at about the same points in a project as planning poker (they both estimate based on a product backlog consisting of user stories) so they can be considered alternatives to each other.

The same techniques that help generate accurate planning poker estimates work with expert judgment. For example, the accuracy of expert judgment improves as your requirements become more detailed and you decompose the work into smaller chunks.

Best Case vs. Most Likely Case vs. Worst Case
A technique to improve system cost estimates by systematically exploring different scenarios using different levels of optimism and pessimism. The various scenarios are combined into a single estimate using formulas.

Beyond that, expert judgment often uses some additional techniques pertaining to Best Case, Most Likely Case, and Worst Case scenarios. These techniques can be especially useful early on, before the team has decomposed the user stories into smaller chunks:

- **Estimate Best Case, Most Likely Case, and Worst Case:** It's a good idea to estimate Best Case, Most Likely Case, and Worst Case estimates (McConnell, 2006, pp. 107–108). One specific approach is to start with the Most Likely Case estimates for each user story, then set them aside to consider complications that would generate Worst Case estimates. What is a Worst Case? Prompting the estimator(s) to consider what

would happen if "everything went wrong" or "everything turned out to be as complicated as possible" is one way to frame it. Even so, research has found that such Worst Case estimates are often not really the true Worst Case outcomes. Finally, after creating the Worst Case estimates, go back to generate Best Case estimates. Interestingly, after creating Worst Case estimates, it's not unusual to find that the Best Case estimates end up being higher than the original Most Likely Case estimates, because the Worst Case estimates often cause developers to identify significant additional work and complications that they must deal with. Obviously, when this happens, the Most Likely Case estimates should be revised.

- **Use formulas to combine Best Case, Most Likely Case, and Worst Case into an "Expected Case" point estimate:** To combine these estimates to produce a single estimate for each item, use the following formulas, which are specified by the Program Evaluation and Review Technique (PERT) (McConnell, 2006, pp. 108–109). To create an "Expected Case" point estimate, McConnell suggests using the following formula:

$$\text{Expected Case} = [\text{Best Case} + (4 \times \text{Most Likely Case}) + \text{Worst Case}]/6$$

Although this takes into account Worst Case scenarios, in situations without a previous track record of the accuracy of estimates (sometimes called "reference data sets"), it may be worthwhile to use a slightly more pessimistic (and so safer) formula to generate the Expected Case estimate:

$$\text{Expected Case} = [\text{Best Case} + (3 \times \text{Most Likely Case}) + (2 \times \text{Worst Case})]/6.$$

8.5.3.3 Function Point Analysis

Function point analysis A system cost estimation approach using a standardized method of counting system design elements and transforming those counts into estimates using mathematical formulas.

So far, we've been discussing subjective estimation approaches. We now turn to **function point analysis**, an objective estimating approach based on counting system design elements, then transforming those counts into estimates using mathematical formulas.

These system design elements, technically called *elementary processes* (EPs) or *program characteristics*, include the following categories:

- External inputs: Screens, forms, and dialogs through which a user or other program adds, deletes, or changes data.
- External outputs: Screens, reports, and graphs that the system generates for use by a user or other program. Typically provides complex summaries of different types of data, which are presented using sophisticated formatting.
- External queries: Retrieving data directly from a database with minimal data summarization or formatting.
- Internal logical files: Typically a single flat file or a relational data table within the system.
- External interface files: Data groupings in an outside application that enter into or leave from the system.

Of these five categories, the first three represent "transactions," and the last two represent "data."

Note that you can only use function points with detailed, functional designs, and, in fact, their accuracy will benefit from the completion of initial technical designs. Importantly, you also can only use them in plan-driven or hybrid projects using BRUF. You can't use them effectively in agile projects where detailed requirements only emerge sprint-by-sprint.

Unadjusted function points Output of function point analysis indicating the amount of work in a software project's scope. "Unadjusted" means that the count of the function points has not been revised to account for nondesign elements.

Table 8-4 is an example of a function point analysis for a software project. For each program characteristic category, there are counts for elements in that category for three complexity categories: low, medium, and high. For example, in this case, there are six low complexity, two medium complexity, and three high complexity external input items. For each complexity level/characteristic combination, you will multiply a conversion value with the number of items to get the function points. The total from the table is the number of **unadjusted function points**, a key output of function point analysis indicating the amount of work resulting from the software project's design elements. The number of unadjusted function points is used to estimate project effort, cost, and

Table 8-4: Example function point analysis using complexity ratings

	Low Complexity		Medium Complexity		High Complexity	
Program Characteristic	**Project Value**	**Conversion**	**Project Value**	**Conversion**	**Project Value**	**Conversion**
External Inputs	6	3	2	4	3	6
External Outputs	7	4	7	5	0	7
External Queries	0	3	2	4	4	6
Internal Logical Files	0	4	2	10	3	15
External Interface Files	2	5	0	7	7	10
Unadjusted Function Points	(6*3 + 7*4 + 0*3 + 0*4 + 2*5) + (2*4 + 7*5 + 2*4 + 2*10 + 0*7) + (3*6 + 0*7 + 4*6 + 3*15 + 7*10) = **284**					

Table 8-5: Example function point analysis using all medium complexity ratings

	Low Complexity		Medium Complexity		High Complexity	
Program Characteristic	**Project Value**	**Conversion**	**Project Value**	**Conversion**	**Project Value**	**Conversion**
External Inputs	0	3	11	4	0	6
External Outputs	0	4	14	5	0	7
External Queries	0	3	6	4	0	6
Internal Logical Files	0	4	5	10	0	15
External Interface Files	0	5	9	7	0	10
Unadjusted Function Points	11*4 + 14*5 + 6*4 + 5*10 + 9*7 = **251**					

duration. "Unadjusted" means that the count of the function points has not been revised to account for nondesign elements, such as team characteristics, data communications, performance, multiple sites, etc.

One issue that may trouble you is determining whether each program characteristic item is low, medium, or high complexity. Function point estimation, including extensive rules for determining complexity, is standardized by the International Function Point Users Group (ifpug.org). These rules are complex, so some authorities recommend simplifying the approach so that all program characteristics are classified as "medium complexity" (McConnell, 2006, p. 201; Jones, 1997, Chapter 2).

Taking the example in Table 8-4 and applying the simplified approach shown in Table 8-5, the overall number of function points is somewhat lower (251 rather than 284) but still in the same ballpark. This level of accuracy could still be useful, especially as a way to check against subjective estimates.

In this book, we'll use this simplified counting approach. For more accurate counting later in a large, complex project, we recommend using the complexity adjustments defined in the *Counting Practices Manual* of the International Function Points User Group (Counting Practices Manual, Release 4.3.1).

Putting function points in context, they are, on one level, quite easy and efficient to use, as it's possible to rapidly count the elements of a BRUF design, rather than having to rely on subjective estimation judgments. However, function point analysis involves some important assumptions and, therefore, limitations:

- Require (fairly) detailed designs: As you saw earlier in the book, creating elaborated, detailed designs is quite effort- and time-intensive. When taking into account the effort of creating them, the overall effort to execute function point analysis can, in fact, be quite large.

- Limited to Software Project Approaches using BRUF: Function point analysis is appropriate for hybrid or plan-driven approaches using BRUF. It can't be easily applied to agile approaches using emergent requirements.

8.5.3.4 Converting Unadjusted Function Points to Staff Months

By themselves, function points don't tell you how much time (and therefore money) a project should take. For example, per the current example, if a project is estimated at 251 function points, how much time and money will that require?

Function point analysis provides formulas to directly estimate the amount of effort in a project in terms of "staff months." A staff month is assumed to be 132 productive hours from a single IT team member in a calendar month. Estimating staff months in this way depends on two factors: the number of unadjusted function points and the size of the IT team (team size).

Assume you have a seven-member team. A conversion of the 284 unadjusted function points to staff months is shown in Table 8-6, with formulas specific to the type of project. ("General," "enhancement," and "new development" formulas are shown; others are available in McConnell, 2006.) For example, the "general" formula estimates 21.8 staff months. This suggests a project duration of about 3 months (21.8 staff months divided by 7 team members).

Table 8-6: Converting function points (FPs) to staff months

Project Type	Formula	Computed Staff Months
General	=0.512 * FPs^0.392 * TeamSize^0.791 =0.512 * 284^0.392 * 7^0.791	21.8
Enhancement	=0.669 * FPs^0.338 * TeamSize^0.759 =0.669 * 284^0.338 * 7^0.759	19.8
New development	=0.520 * FPs^0.385 * TeamSize^0.866 =0.520 * 284^0.385 * 7^0.866	24.7

Interestingly, as team size increases, the estimated amount of effort also increases. For example, if we double the team size in our example to 14 members, perhaps with the idea of getting the work done in half the time, the general formula estimates that the project would take 37.8 staff months or still close to 3 months to complete (37.8 staff months / 14 team members = 2.7 months)! The reason for this is that bigger teams are more complex to organize and execute, often leading to lower staff productivity. More informally, this is a quantitative explanation for an old project management aphorism: "Adding staff to a late project only makes it later!"

While function points themselves are a useful work estimation technique, we advise caution in using these mathematical formulas to convert function points to staff months. We say this because those formulas were created in an era when most aspects of a software application had to be coded "by hand"—for example, all aspects of authentication and security, each user interface page, creating and connecting to the database, and many other capabilities.

In contrast, in recent years, many tools and techniques have become available that can provide these features as prebuilt capabilities, greatly reducing the amount of code a developer has to write. Such tools and techniques include:

- **Software development frameworks:** Such as ASP.NET or the MEAN stack (MongoDB, ExpressJS, AngularJS, and NodeJS).
- **Open-source libraries:** That allow developers to import prebuilt software functionality directly into a software project.
- **"Low-code" development environments:** That allow developers (or even business clients) to use visual "drag-and-drop" techniques to create sophisticated applications in a fraction of the time required in traditional environments.
- **Artificial intelligence (AI) engines:** Generative pretrained transformer (GPT) engines are increasingly capable of creating sophisticated applications based on only

natural language prompts, like "Create a software application that allows me to create and maintain a list of personnel." These capabilities are based on the GPT engines "learning" to code by mining code samples from publicly available open-source code repositories.

All this means that estimating IT labor time required for a pile of work sized at 251 function points only has meaning in the context of a given team and its development tools and techniques. However, function points can still help you in estimating. For example, if you know how much time and money a project of similar size cost you in the past, you can use function points to estimate into the future. Say you previously spent $160,000 to create a 200 function point system, then (assuming the same team, tools, and process) you could reasonably estimate 251 function points would cost:

$$251 \text{ function points} / 200 \text{ function points} \times \$160{,}000 \approx \$200{,}000$$

Function points can provide an additional check on subjective approaches such as planning poker or expert judgment.

8.5.3.5 Beyond Function Points: Adjusting for Nondesign Factors with COCOMO II

When skilled estimators use function point estimates, the resulting estimates can be within 10% of each other.

But, having said that, it's also true that every software project is affected by factors beyond countable design elements. In this light, consider a systems project in terms of three general dimensions:

People: Is your IT team experienced and cohesive or new and unfamiliar with each other? Is the team in one location or many?

Product: Are you creating well-understood enhancements to a known application or developing a new, highly unfamiliar type of system?

Process: Is the team continuing to use the same software project approach they've used before, or are they transitioning to a new one?

These issues don't affect your function point count, but they can have a big impact on the amount of effort needed to complete a project. How can you adjust your function point estimates to take these issues into account?

One well-known approach is the Constructive Cost Model, version II (COCOMO II). **COCOMO II** is a quantitative model for estimating software projects developed by Barry Boehm and his colleagues at the University of Southern California (Center for Software Engineering, USC, 2000). It's regarded by some experts as the best researched and most rigorous approach to adjusting system cost estimates for people, product, and process factors (McConnell 2006, p. 66).

COCOMO II A quantitative model that can be used to estimate software effort, costs, and duration based on unadjusted function points (or lines of code) and a series of adjustment factors pertaining to the overall project environment, including elements of people, process, and product.

In a nutshell, COCOMO II uses statistical models that quantify the effects of various "effort adjustment factors" on the size of a system cost estimate. Those adjustment factors include a variety of items in the general areas of people, process, and product. Figure 8-6 provides key examples of those adjustment factors, showing their potential impact on the size of the project. For example, the first row, Analyst Capability, indicates that a project with very low capability business analysts may end up being twice the size of a project with very high capability business analysts.

Referring to Table 8-7, each adjustment factor may be rated at various levels of being low, nominal (i.e., normal), or high. For each factor, the various levels are assigned quantitative values affecting the amount of total effort estimated for the project.

For example, in Table 8-7, the "Requirements Analyst Capability" people-oriented adjustment factor is defined as the business analyst's "analysis and design ability, efficiency and thoroughness, and the ability to communicate and cooperate." This is essentially an evaluation of how well the BA can perform core SA&D duties for the project.

Not surprisingly, another people-oriented adjustment factor is "Programmer Capability (general)," defined as "the capability of the programmers as a team rather than as individuals. Major factors that should be considered in the rating are ability, efficiency and thoroughness, and the ability to communicate and cooperate."

		Potential Effect		
Factor Category	**Factor**	**Decrease**	**Increase**	**Overall Influence**
Personnel	**Analyst Capability**	-29%	42%	2.00
	Programmer Capability	-24%	34%	1.76
	Personnel Continuity	-19%	29%	1.59
	Applications Experience	-19%	22%	1.51
	Language & Tool Experience	-16%	20%	1.43
Project	**Multisite Development**	-14%	22%	1.42
	Development Schedule	0%	43%	1.43
Product	**Software Reliability**	-18%	26%	1.54
	Product Complexity	-27%	74%	2.38
	Software Reusability	-5%	24%	1.31

Figure 8-6 Key COCOMO effort adjustment factors by category showing potential effect on project size

Table 8-7 Selected COCOMO II adjustment factors, ratings, and quantitative effects

	Ratings and Quantitative Impacts						
	Very low	**Low**	**Nominal**	**High**	**Very high**	**Extra high**	**Total possible influence**
Requirements Analyst Capability	1.42	1.19	1.00	0.85	0.71	N/A	2.00
Programmer Capability (general)	1.34	1.15	1.00	0.88	0.76	N/A	1.76
Multisite Development	1.22	1.09	1.00	0.91	0.84	N/A	1.56
Required Software Reliability	0.82	0.92	1.00	1.10	1.26	N/A	1.54
Product Complexity	0.73	0.87	1.00	1.17	1.34	1.74	2.38
Use of Software Tools	1.17	1.09	1.00	.90	.78	N/A	1.50

People-oriented adjustment factors don't simply point to how skilled various teams are; they also consider how those team members are organized. For example, the factor "Multisite Development" takes into account a range of situations, from the team being fully colocated to being spread out over multiple continents.

One example of a product-oriented factor is "Required Software Reliability" (abbreviated RELY), defined as "the measure of the extent to which the software must perform its intended function over a period of time." If the effect of a software failure is only slight inconvenience, then RELY is very low. If a failure would risk human life, then RELY is very high.

Another example of a product-oriented factor is "Product Complexity," which takes into account aspects of the software such as the complexity of computations, data management, and the user interface.

Finally, an example of a process-oriented factor is "Use of Software Tools," pointing to the effects of using advanced software tools to reduce development effort.

Table 8-7 provides several key insights:

- **"Nominal" means "no impact":** First, any adjustment factor evaluated as being "nominal" has no impact on the systems estimate. For many adjustment factors, "nominal" means moderate—in the middle of the possible range of values. However, other adjustment factors have specific meanings. For example, "Multisite Development" has specific values from the "very low" to "extra high" levels: "international," "multicity and multicompany," "multicity or multicompany," "same city or metro area," "same building or complex," and "fully colocated." See the COCOMO II manual for detailed information (Center for Software Engineering, USC, 2000).
- **Understanding quantitative effects:** Second, to understand the meaning of the numbers in the tables, consider the first factor, "Requirements Analyst Capability." If a

project is initially estimated at 100 staff months of effort and, in addition, if you evaluate that the BA on a project has a very high level of capability, then, all other things being equal (or, more precisely, "nominal"), that means your estimate will be revised downward by 29% (100 staff months × 0.71 = 71 staff months). On the other hand, if you determine that the BA on a project has a very low level of capability, then that means your estimate will be revised upward by 42% (100 staff months × 1.42 = 142 months). This is a large impact, potentially affecting the size of a project by a factor of 2—that is, the high-end estimate of 142 months / low-end estimate of 71 months = 2.00. Note that only one of the example factors shown in the table has values defined for the "extra high" rating. There are other adjustment factors in the COCOMO II model that lack values for the "very low" rating levels.

- **Direction of impact varies by adjustment factor:** For example, high ratings of Requirements Analyst Capability reduce the system cost estimate, whereas high ratings of Product Complexity increase the system cost estimate.
- **"Scaling Adjustment Factors" are affected by project size:** The sample adjustment factors shown in Table 8-7 have the same impact percentage regardless of project size. For example, a very low value for multisite development, pointing to developers located in multiple countries with less than optimal communications support, will increase the software project estimate by 22%, regardless of project size. However, several adjustment factors in the COCOMO II model increase their effect on the project estimate as the project increases in size. They include:
  - **Development flexibility:** Degree to which the development team must build software exactly and rigidly as the requirements were written
  - **Team cohesion:** Degree to which all IT and business team members work together in a cooperative, aligned, and efficient manner
  - **Precedentedness:** Degree to which the application being built is familiar (precedented) or unfamiliar (unprecedented)
  - **Architecture and risk resolution:** Degree to which the project manager systematically leads the team in identifying and resolving architectural issues and project risks
  - **Process maturity:** Degree of sophistication and maturity of the software project approach used by the team

  As an example of this, take the precedentedness scaling adjustment factor. If a software application being developed is similar to several other projects developed previously, then precedentedness is high. On a project including about 2,000 function points, precedentedness can affect the overall estimate by up to 33%. However, as the project gets bigger and bigger, the effect of precedentedness also increases. For example, on a project of about 100,000 function points (about 50 times larger than the first example) the potential effect of precedentedness increases to 70%.

8.5.3.6 Effects on Calculating System Cost Estimates Using COCOMO II

From Table 8-7, you might conclude that you could calculate adjusted system cost estimates using COCOMO II via looking up the values of various adjustment factors and multiplying them by the unadjusted estimate. For example, based on unadjusted function points, suppose you generated a baseline estimate of 18.4 staff months of work. Now, further suppose you determined that the BA assigned to your project both was relatively inexperienced and had experienced difficulties in their most recent project. Based on this determination, you decided that the Requirements Analyst Capability was very low. Also, you were told that the software development team would be spread out over multiple countries and, in fact, multiple continents. Based on this distribution, you determined that the Multisite Development was very low. Taking those factors into account, you could, in fact, calculate the adjusted estimate by looking up those COCOMO II adjustment value factors and multiplying them by the baseline estimate:

18.4 Baseline Staff Months × 1.42 Requirements Analyst Capability
× 1.22 Multisite Development = 31.9 Adjusted Staff Months

However, when you introduce the five scaling adjustment factors noted in the previous subsection, the calculation becomes much more complicated because of the need to change the percentage effect of each of those five factors based on the overall size of the project.

Fortunately, online calculators exist that automate those calculations. One such calculator is provided by the University of Southern California, the university where much of the research for COCOMO II was based. The COCOMO II URL is http://softwarecost.org/tools/COCOMO/. You may wish to visit this website and play around with the various adjustment factors. If you do, to make use of the previous discussion, change the "Sizing Method" at the top of the web page from "Source Lines of Code" to "Function Points." You can then enter a number of unadjusted function points just below the Sizing Method. You may wish to choose "Java" as a programming language, which is representative of the many programming languages used today (e.g., C#, JavaScript, or Java itself). As you change adjustment factor settings, you can click the "Calculate" command button to see the effect on estimated effort in person-months.

8.5.3.7 Function Point Analysis and COCOMO II Conclusions

We offer several insights regarding function point analysis and COCOMO II.

First, we reiterate that you can't use these objective estimating techniques prior to BRUF being complete in hybrid projects or at all in agile projects using emergent requirements techniques. These techniques require a high level of up-front design detail (screens or web pages, database designs, external interfaces, and reports) not just for the highest priority user stories likely to be developed in the next sprint but for an entire project or release.

Second, function point analysis and COCOMO II highlight the ongoing tension between the desire for solid, objective estimates and the need for a degree of subjectivity. A simple scenario can illustrate this. Suppose you've designed a new e-commerce website with detailed requirements, from which you count 500 function points. Now, suppose the development team assigned to execute this project consists of a group of highly cohesive* BAs and developers who are located in the same office; are familiar with the business stakeholders; and have a proven, multiyear track record of delivering high-quality, effective software applications of this same size and type and using similar development tools. Clearly, based on COCOMO II, you can and probably should "turn the dials" to reduce the function point analysis estimate based on favorable ratings in many COCOMO II adjustment factors, thereby reducing the IT labor cost estimate.

However, now suppose a new CIO has decided to reduce the hourly costs of software developers by moving a significant amount of the development to an offshore team that costs less money per hour. Suppose this new team had only recently been hired, and its members don't know each other, don't know the types of applications and tools you use, and certainly don't know the existing IT team and business stakeholders. It's not hard to see that this change in team composition would likely significantly turn the dials in the opposite direction for most of the COCOMO II adjustment factors just mentioned, with the estimated IT labor cost also increasing significantly.

But how much, then, would you expect the estimate to increase? That depends on your subjective judgment of how much to turn the COCOMO II dials in each case. For example, perhaps you had determined that the original team had extra high team cohesion. With the introduction of the new, offshore team members, how much do you "dial back" the team cohesion rating? To very high, high, nominal, or even low or very low? A careful reading of the COCOMO II documentation describing these rating levels can provide some insight, but it's still likely that different software estimators would arrive at somewhat different, subjective conclusions.

The key takeaway is that "turning COCOMO II dials" injects subjectivity into what, with unadjusted function points, was an objective counting exercise. This may help explain why many teams

* By "cohesive," we mean the team has a great deal of experience working together; their objectives, values, and vision are aligned; and they're willing to work to accommodate each other.

today use purely subjective estimation methods, because, even when using objective estimating methods such as function points, in the end, you'll need to use a degree of subjective judgment.

8.5.4 Deeper Dive 8.4: Advanced Principles for Improving Estimates

You've learned that systems cost estimating is challenging, with underestimation being a systemic problem leading to many project failures.

That said, you've also learned to use several different estimating techniques at different stages of a project. Coupled with appropriate requirements detail via decomposition of the work, you can control the risk of inaccurate estimates as you work through cost/benefit analysis with your business clients.

In addition to these key techniques, we conclude this section with several additional principles for improving your system estimates.

8.5.4.1 Pay Special Attention to Large and Complex Projects

Large, complex projects are harder to estimate than small, simple ones. First, Planning Fallacy problems increase as projects become larger and more complex, with large projects failing three to five times more frequently than small and midsize projects (Shmueli et al., 2015). Second, large projects involve many business and IT team members and so tend to attract greater scrutiny. Finally, going over budget by X% in a large project can be seen as a much bigger problem than going X% over budget in a small project.

For example, if a $50,000 project with two developers goes 30% over budget, that extra $15,000 in cost might amount to the two developers working two extra weeks. While not nothing, this extra cost and delay are unlikely to attract too much attention in a large systems group with a multimillion-dollar systems project budget. On the other hand, if a $5,000,000 project with 30 developers goes 30% over budget, that extra $1.5 million in cost would blow a major hole in most IT departments' budgets, plus needing to keep those 30 developers on the project for several months, likely interrupting other projects. This would certainly create a lot of unwanted scrutiny.

The moral of this story is clear: Big, complex projects will especially benefit from significant up-front planning and estimation.

8.5.4.2 Don't Overcomplicate: Estimate De Facto Fixed Labor Costs as Fixed

You've seen that IT project labor costs increase as requirements become larger and more complex. But that's more true of developers than of other IT labor categories, such as BAs, testers, and operations staff.

For example, say you're estimating a hybrid project with a team of two BAs (a senior BA doubling as a project lead and a junior BA) and 10 software developers. The team's typical approach is for the BAs to create requirements and estimates for two months, then support the developers for four months during construction.

Depending on the estimates, you may need to add or subtract developers during the four months of construction, but the number of BAs probably won't change. You know up front that you'll need the two BAs for six months, so their costs are de facto fixed. As a result, you can accurately estimate the BA costs by simply multiplying their monthly costs by six months.

This principle may also work for testers, documentation specialists, and operations staff.

8.5.4.3 Identify and Attack Major Risks Early and Continuously

While you can't foresee every major project risk at the outset, you can certainly identify some of them. In that case, you should take steps to attack those risks, including their effects on your estimates.

Here are a couple of examples:

- **Proof-of-Concept (PoC) projects for technology risks:** In the WPL planning poker example, we noted that the team was unfamiliar with the scanner technology and how to integrate it. To reduce the risk of inaccurate estimates in this and similar cases, it would make sense to spend a small amount of time up front in a PoC project to determine the difficulty of integrating this technology and confirm an approach to using

this technology. A PoC can greatly increase the team's estimating accuracy and help them avoid adopting technologies that turn out to be simply unworkable.

- **Replacing team members:** Say you're replacing an experienced, colocated development team with a new, international team. Based on their inexperience with your clients and existing code base, you should expect the new team to require a significantly higher estimate to complete the same work as the old team, especially in the beginning. In this situation, you should be careful to not make commitments until you understand the new team's productivity.

8.5.5 Deeper Dive 8.5: Estimating Software Configuration Projects

We've focused the majority of this chapter on learning techniques for estimating software construction projects—that is, projects where software developers program new features using languages like JavaScript, C#, PHP, or Java. The reason for this focus is that construction has traditionally been the most frequent and challenging type of project to estimate.

However, other important categories of projects present significantly different estimating challenges. One of them, configuration of existing third-party software applications, has existed for decades. In this Deeper Dive, we explore the key issues of cost estimating these types of projects.

8.5.5.1 Third-Party Software

To begin, over the years, many common types of applications have been created many times over by software vendors. These vendors develop software applications that they sell or rent to other organizations. For example, there are many third-party software applications available to perform common functions like payroll, general ledger, accounts payable, accounts receivable, and so on. These functions are often offered together under the umbrella term *enterprise resource planning*, or ERP. But that's not all: there are packages focused on the specific needs of all sorts of firms, including hospitals, restaurants, retail websites, law firms, and many, many more.

Because of this, most organizations use third-party software for at least some of their application needs. They likely construct their own software only in areas where their firm has other needs that are highly unique.

We can tie this to the Systems Development Software Framework in Figure 8-1 as follows: A configuration project starts much like a construction project—you engage in Initial Visioning and Business Analysis. But after that, as you reach Project Planning and Implementation Approach Selection (the next row down in the diagram), you may determine that third-party application software exists that can meet your needs. You might not need to construct your own software; rather, you can use your Initial Requirements to select a third-party software application that matches your requirements at a reasonable price. Here are key cost categories you need to consider:

- **Costs of Initial Visioning and Business Analysis:** These up-front costs can be substantial, especially if the firm hires outside consultants to help them evaluate third-party software applications.
- **Costs of Software Licenses and Maintenance Costs:** Software license costs may be up-front one-time license fees or, more commonly today, ongoing license fees for using the software over time, whether hosted on the organization's own servers or via a software as a service approach. Note that third-party software today also includes open-source software, and while such software may be available to download for no cost, organizations using these packages often pay a third-party firm that focuses on supporting that open-source package.
- **Costs of Configuration:** While the prior two categories may generate substantial costs, they aren't hard to estimate. In contrast, the costs of configuring the selected software often dwarf the other two cost categories and require careful estimating. We'll explore that now.

8.5.5.2 Estimating Software Configuration Costs

The good news for estimating software configuration costs is that most such software packages have a specific list of repeatable steps you can use to plan the configuration process. As we've discussed several times, this is a bit like building a house: you know that you start with the foundation, then the framing, then exterior walls, then electrical and plumbing systems, and so on.

The bad news is that the amount of effort to complete each of these steps can vary significantly based on the complexity of the organization implementing the package. To understand why, we need to contrast the nature of software constructed by a single firm for its own use with third-party software that's designed to be used by many different firms. The key here is *flexibility*: software built for a single firm can be kept relatively simple, as it can be tailored to meet only that firm's needs. In contrast, software built to be used by many different firms must be flexible enough to support the specific needs of all those firms. Because of this need for flexibility, such third-party software typically needs to be much more complex.

We can illustrate this with an example from the field of healthcare: paying a health insurance claim, like the ones your doctor files with your insurance company after an office visit. In this example, let's say that during a visit to your doctor, you had the following two services:

- **Annual exam:** For which the insurance company would pay the doctor $100
- **Wart removal:** For which the insurance company would pay $150

The question is this: How much will the insurance company provide *you* as benefits for these services? If it doesn't pay 100% of the fees above, then you have to make up the difference to the doctor.

Keeping the example simple, let's say two different insurance companies want to use the same third-party claims system. Let's also say that their benefit plans work using the business rules in Table 8-8.

Table 8-8: Example business rules for determining healthcare benefits

Procedure	Insurance Company A	Insurance Company B
Annual Exam	• Pays 70% if doctor is in-network, not subject to deductible • Pays 50% if doctor is out-of-network, after deductible	• Pays 70% after $25 copayment if doctor is in-network, after deductible • Pays 75% if doctor is out-of-network, after deductible
Wart Removal	• Pays 70% if doctor is in-network, after deductible • No coverage if doctor is out-of-network	• Pays 90% if doctor is in-network, after deductible • Pays 50% if doctor is out-of-network, if preauthorized and after deductible, otherwise no coverage

This example includes only two types of medical services (in reality, there are thousands) and two healthcare insurance companies, yet the rules seem complex and variable, especially when comparing the two companies. How could you create a system that could flexibly support thousands of different medical services and the business rule differences between many different insurance companies using that system?

The short answer is that the third-party claims system needs to encode these complex rules in database tables, enabling it to use the correct calculation approach to determine benefits in each circumstance. This approach is called "table-driven logic" and is the problem of the software vendor to design. We won't take up this complex design problem here.

But even after the software vendor designs and programs a system with that flexibility, to implement it in a specific insurance company, you'd need to systematically collect all those business rules and encode them in the table-driven logic tables leveraging the design of that particular system. This amount of time and money is what you need to estimate for configuration.

Also, note that many large configuration projects include a degree of construction of new code to create custom facilities such as the following:

- **Interfaces to other systems**
- **User interfaces, reports, or dashboards**
- **Functionality that the third-party system lacks**

For these construction features, we can use construction estimation techniques like planning poker and others we've explored. The overall project estimate would be the sum of the configuration and construction estimates.

In complex system implementations like ERP systems, teams of more than 100 people can work for multiple years, generating costs in the tens of millions of dollars. Many companies needing this work done will hire consulting groups with deep experience in implementing a specific third-party software package. Fortunately, these groups can use their prior experience to estimate a new implementation of the package with reasonable accuracy.

8.5.6 Deeper Dive 8.6: Estimating Artificial Intelligence (AI) Projects

Along with software construction and configuration projects, there are other important categories of projects that present significantly different estimating challenges. One of those—creating artificial intelligence capabilities—is new for many organizations but is entering a period of rapid expansion.

Organizations typically implement AI capabilities for system features that can't easily be created using conventional programming techniques. Recall the property insurance example from Chapter 2, where we considered using an AI-based component to categorize the level of hail damage on roofing shingles and collect images of those shingles using drones. Neither of those functions could be readily constructed using a conventional programming language.

Without repeating AI concepts we've discussed earlier, we offer a key point regarding the creation of AI-based features: *Creating an AI-based feature is fundamentally different from creating a feature using conventional programming techniques. Specifically, for AI, we* train *the system* rather than program *it.*

- **Conventional programming:** When using a conventional programming language, developers convert a system design algorithm into executable code. For example, for the Wayback Public Library case, you can program the "calculate fine for overdue item" feature using a symbolic programming language like JavaScript or C# to implement the logic from the corresponding use case. Importantly, you can be confident that a properly coded algorithm will always generate the same, correct fine amount for a given set of inputs (e.g., days late, fine per day, cost of the item, age of the item).
- **Training AI:** In contrast, when creating a feature like "categorize hail damage using images," you're not programming the AI capability. Rather, you're training it. This training process is quite different from conventional programming: you're creating a statistical model, not implementing a conventional algorithm. As a result, a properly functioning AI model won't generate 100% consistent or accurate answers, so you have to, in effect, coach it toward an acceptable level of accuracy. This process, which we'll explore, includes a series of steps that may or may not need to be repeated and revised multiple times.

We'll explore the implications of this difference by outlining the process of developing an AI-based feature (adapted from Russell & Norvig, 2022, Chapter 18):

- **Problem formulation:** You determine what problem you need to solve: in our example, categorization of shingle images. Or, expressed as a user story, "As an insurance adjuster, I need the system to classify shingle images into damaged and undamaged categories to save time and increase consistency." Beyond this, you need to be specific: Are you including only composite shingles, or are you also including wood shingles and slate shingles? How many damage categories are you using? As you saw in the example in

Chapter 2, shingles can be damaged by many problems other than hail, and each type of damage looks different. You can add these items to a user story as acceptance criteria. Finally, what's your AI approach: supervised, unsupervised, or reinforcement learning? Here, you'd likely choose supervised learning, given that you have many shingle images that are already categorized. Finally, what level of accuracy do you need to achieve for the AI model to be valuable in your business process?

- **Data collection, assessment, and management:** You need to collect the training set of shingles images: Do you get them from one source or many? Are they all in the same image format and level of quality? How do you organize and maintain them in the AI environment?
- **Exploratory data analysis and visualization:** Once you get the images in hand, you should review them to be sure you fully understand them. In reviewing the training set images, you may need to verify that they were correctly categorized in the first place. Perhaps you want to refine the original categories—for example, separating "hail damaged shingle" and "undamaged shingle" into "clearly hail damaged," "possibly hail damaged," and "clearly undamaged." Maybe you should remove some poor-quality images, and so on. All of this may require significant human effort. Also, to ensure the robustness of the AI model, you may want to split the images into two sets: one for training the AI model and a second for checking to see if the model still works well on data that it wasn't directly trained on.
- **AI model selection:** Once the data is clean and vetted, you can turn to selecting an AI model and training it. Prior to this point, a BA could do much of the work, but now you need to involve an AI expert with deep technical understanding of AI models, especially with respect to the current problem type. In our example, the problem type is image classification using supervised learning. This AI expert needs to select an appropriate AI model, of which there are many, which are changing all the time. In our example, a "convolutional neural network (CNN)" might be a good choice (Russell and Norvig, 2022, pp. 975–978). The expert also must set numerous CNN model parameters: how many neurons, how many layers, and other items. At this point, we're beyond the scope of BA skills covered in this book. Instead, these AI skill sets are closer to (but quite different from) the technical skill sets of a conventional software developer.
- **AI model training:** Once the model is selected and parameters set, you can let the model run (potentially many times) in training mode. You're asking the system to try to categorize the images, and, at first, the system does a poor job. But after each categorization attempt, you provide feedback to the system indicating which images it correctly and incorrectly categorized. The system then uses the model's statistical learning algorithm to adjust the weights of the neural network. Over time, the system gets better at categorization. You hope to produce a model that categorizes the training data with enough accuracy to address your original problem formulation. But you can't know ahead of time how good it will get. If you don't achieve that level of accuracy, you may need to revise the model's parameters or even pick a different model, in effect starting the training process over again. Beyond that, you need to then submit the additional set of images not used in the training to verify that the model works accurately on images it wasn't trained on.
- **Establishing trust in the model:** Before you deploy the model in the real world, you need to ensure that you understand it well enough to trust it. Neural network models are famous for being "black boxes" that can't be directly interpreted, meaning the model is too complex and opaque for you to directly understand how it arrives at its answers. This is quite different from conventional programming, where you can trace your way

through the code to explain how the program generated an answer. Sometimes, you can offer an explanation like "the model classified this shingle as damaged by hail because of the presence of highly defined black circular marks." Alternatively, if the model is highly accurate, you can downplay that kind of explanation in favor of the accurate testing results.

- **Integration and deployment:** Sticking with our property insurance example, you can say that the overall solution included several different applications working together: the conventional property claims system, an AI drone system to collect the images, and the AI image classification system. For this to work, you must integrate these systems, which will likely include the programming of conventional data interfaces at the point each system passes data to another.
- **Operation and maintenance:** As with any other piece of software, you need to ensure that you have the right infrastructure and level of support to operate over time. Beyond that, you need to consider maintenance issues. For example, in our example, do you need to consider changes in the nature of shingles themselves—that is, what if the industry updates shingles to using materials that then show hail damage differently than current materials? To account for such possibilities you may need to actively monitor industry changes and how the AI model performs in the midst of those changes. This could result in the need to invest additional "staying in business" money in the system to keep it up to date.

These steps represent a highly summarized view of the process of AI development, and the industry has a long way to go to mature that process. But what does this mean for cost estimating? We can glean the following points:

- **AI features are . . . features:** To start, per our example, you saw that an AI-based capability can be defined in the same way as any other feature: as a user story, including acceptance criteria. Given that, you can add AI-based features to your product backlog list of user stories to estimate just like other features.
- **AI features include BA work:** As noted, a BA can perform several of the initial tasks. However, tasks like selection and exploratory analysis of the image data imply using skill sets that are new (or unusual) for the BA role. The costs for the work the BA performs need to be included in your overall estimate.
- **AI features need (other) experts:** When you try to estimate a user story, you'll generally ask the developer who's going to create the feature to provide key input on the estimate. AI is similar, but you need to recognize that AI experts are likely different IT team members than conventional programmers. Also, given the split in technical skill sets between conventional developers and AI experts, you'll likely need to use an overall estimation approach closer to "expert judgment" than "planning poker."
- **Avoid commitments because of the inherent uncertainty in AI estimates:** Recent research (Spurrier & Topi, 2023) finds that the project approach to developing AI features works differently than for conventional features. Specifically, AI development tends to work in a "work for a day, evaluate progress, then plan tomorrow" manner. This reflects the uncertainties of AI model development: you can't be sure if or how quickly your training data and AI model will generate the accurate results you need. It could happen quickly, it could take many revisions and a lot of time, or it might not end up working well enough at all. This results in an approach to AI development that is closer to kanban (see Chapter 15) than to sprint-based agile development. In this sense, AI development is more agile than agile development of conventional software! But this aspect dramatically decreases your ability to accurately estimate AI work. So, similar to the discussion in Deeper Dive 8.1, you need to avoid making commitments out of AI estimates.

- **Don't forget to estimate conventional development tasks needed for AI development:** You need to remember to include estimates for conventional software development tasks associated with the AI solution. We already mentioned the need for system integration interfaces in the "Integration and Deployment" part of the AI development process. But other tasks—such as obtaining training images and organizing them into a database—may best be accomplished through conventional programming capabilities. Similarly, once you implement your drone-based image acquisition solution, you'll likely need conventional programming to collect and store those images in your database.
- **AI capabilities as third-party solutions:** Speaking of drones, you need to carefully consider which AI capabilities to build and which to buy. For example, the shingle image recognition model is closely related to the property insurance company's core business and is likely not too complex, so it may be something the company can and should build itself. In contrast, consider the drone capability: creating and deploying a fleet of image-capturing drones is not closely related to the business of insurance. But it is a capability that the company could likely acquire from a third-party drone vendor and then integrate into the overall solution. The lesson here is that, over time, AI capabilities will likely include more and more generic solutions offered by AI software vendors, and you should be able to select, configure, and implement them similar to other, conventional third-party solutions.

Chapter 9

Business Benefits Estimation and Cost/Benefit Analysis

9.5 Deeper Dives: Advanced Benefits Estimation and Cost/Benefit Analysis Topics

In this section, we do deeper dives on several advanced benefits estimation and cost/benefit analysis topics that all build on and expand coverage from earlier in this chapter:

- **Deeper Dive 9.1** discusses the many ways IT can be used to increase an organization's revenues.
- **Deeper Dive 9.2** provides an in-depth treatment of ROI versus NPV approaches. It also introduces a third approach, internal rate of return (IRR).
- **Deeper Dive 9.3** Explores using generative AI to estimate business benefits and conduct the cost/benefit analysis.
- **Deeper Dive 9.4** presents an example of estimating business benefits where automation is used to reduce average labor costs and redeploy experienced staff to marketing and sales roles.

9.5.1 Deeper Dive 9.1: Using IT to Increase Revenues

The ValueForAll (VFA) and Wayback Public Library (WPL) cases we've discussed derived most of their estimated business benefits through efficiency gains (labor savings for both and reduced losses from theft for WPL). This isn't surprising. When digital computer information systems were first introduced in the 1950s and 1960s, generating labor cost savings via system automation was the key motivation. Indeed, this approach has been and still is so common that we may look at it as the classic source of computer-based business benefits.

However, using systems to generate new revenues has become an increasingly important source of business benefits since the rise of the internet beginning in the 1990s. We'll explore this practice, divided into two categories: (1) non-IT organizations that use IT to increase their revenues and (2) organizations where IT itself is the source of revenues:

- **Increasing total sales revenue of an organization's core, non-IT products:** Most organizations are not in the business of IT. Instead, they use IT to improve their performance. VFA and WPL are both good examples. But WPL's use of IT to increase fine collection revenues only hints at the importance of this category.
  - **Increasing sales volumes of an organization's core, non-IT products:** The most obvious example of this is using a retail website to enable online shopping. This is so commonplace today that we're surprised when we find an organization that *doesn't* have a retail website. But, in fact, it was highly uncommon to shop online prior to the mid-1990s, when large numbers of shoppers first began to access the internet. Third-party platforms to facilitate online shopping are common and easy to implement. So you might wonder if all these problems have already been solved.

    In fact, many organizations have specialized online shopping requirements that require custom programming. For example, online streaming services constantly battle each other, striving to provide the most attractive, highly functional user experiences, including advanced content recommendation algorithms, social media features like watch parties, and so on. Another example would be an online consumer loan application function offered by a bank—requiring the ability to request, collect, and evaluate a wide range of loan application documentation from the consumer, consumer credit agencies, and so on.

  - **Charging higher prices for IT-enabled product features:** For example, car companies are beginning to charge additional fees (either one-time or ongoing subscriptions) for buyers to be able to use car features like autonomous driving, remote starting, voice controls, or even the ability to turn on heated seats.
  - **Increasing quality:** Software automation might also increase sales by increasing the quality and consistency of a firm's products and services. For example, a health insurance company might increase the accuracy and speed of its claims processing, increasing its attractiveness in the marketplace.
  - **Leveraging labor efficiencies:** You can even leverage software efficiencies to generate more revenue. For example, as mentioned in Section 9.4.2.3, you could reassign staff freed up by software automation to focus on sales and marketing. This is illustrated in an example in Deeper Dive 9.4.
- **Increasing IT revenues:** In contrast to the points above, here, we focus on firms where IT itself is the primary source of revenues.
  - **Selling or renting the software itself as a product:** Many organizations make money as third-party software vendors, creating commercial-off-the-shelf (COTS) software products that are marketed to be used by other organizations for a fee. Alternatively, a software vendor may provide support for an open-source software (OSS) system. This was described in Chapter 7. Broad-based examples that a wide range of firms could use include software supporting financials, payroll, or supply chain management, while more specialized examples include software focused on supporting industry verticals, such as hospitals, banks, and so on. Today, it's becoming increasingly common for organizations to use open-source software, where there's no license fee. However, even here, it's common for a for-profit organization to offer ongoing support to users of an open-source software project for a fee.
  - **Creating custom software for clients:** This is a software consulting model in which the organization sells custom software projects to clients. Forms of this were described in Chapter 7 including outsourcing and consulting. The goal in each project is to create a unique software application customized to meet the specific needs of a single client. Typically, the client then owns the custom software. In some respects, this resembles the prior point, in that the organization makes money by creating software to be used by other, outside organizations. The difference is that, in the prior point, the same software is a reusable product that's owned by the firm that created it and sold or leased to many different clients.
  - **Selling advertising and data revenue:** Often firms offer software "for free" that's actually paid for by advertisements that appear on the site or in the application. This is typically the case in the retail consumer space, including capabilities like social media and internet search. Alternatively (and sometimes controversially with respect to data privacy issues), user data, including items searched, ads clicked on, products bought, and location data, is sold to other organizations interested in targeting specific users meeting a certain profile.

All of these sources of benefits require different approaches to benefits estimation—in fact, far too many to cover in detail in this book. For example, entire books have been written on the subject of using social media for advertising and marketing (e.g., Macy & Thompson, 2011; Mahoney & Tang, 2016).

9.5.2 Deeper Dive 9.2: Cost/Benefit Analysis Approaches for Systems Projects: NPV versus ROI versus IRR

In both the VFA and WPL examples, we performed the cost/benefit analysis using the return on investment (ROI) approach. We used ROI because it's both practical and widely used. However, we also briefly noted a key, alternative analysis approach: net present value (NPV). NPV is not as

widely used in systems projects, even though it's considered a theoretically ideal approach. In this Deeper Dive, we contrast ROI and NPV, and in doing so tackle several key questions:

- How does cost/benefit analysis using NPV work?
- How does it differ from ROI?
- If NPV is theoretically sounder than ROI, why is ROI more widely used?

9.5.2.1 Cost/Benefit Analysis Approaches: Contrasting the NPV "Ideal" versus the ROI "Real"

Although NPV is theoretically the ideal approach (more ideal than ROI) there are several practical reasons why, for systems projects, the NPV approach may be difficult to use and may sometimes recommend unwise investment decisions.

Because of these difficulties, many organizations modify NPV in ways that correspond with the ROI approach. We begin this section by presenting the key ideas involved with NPV, including why it's theoretically ideal but in practice somewhat limited in its utility.

After that, we turn our focus to ROI.

9.5.2.2 Overview of NPV

NPV is based on the concept of the **time value of money**, which means that a dollar today is worth more than a dollar in the future. This is intuitively appealing: if you had a choice between receiving a dollar today and a dollar a year from now, you'd likely choose today. However, there's a more mathematically convincing argument for this choice. (Note: If you're already familiar with these ideas from a prior class in finance or accounting, you may choose to skip the rest of this subsection.)

Simple mathematical formulas allow you to calculate the time value of money. Specifically, suppose you receive $100 today—the present value—and can invest it to earn a 5% per year return. For example, you could invest the money in a certificate of deposit at a bank or loan it to another business so they would pay you back with the interest in a year.

Obviously, at the end of that year, your investment would be worth a future value of $105, calculated as follows:

$$\text{Present Value} \times (1 + \text{Interest Rate}) = \text{Future Value}$$

Or, for this example,

$$\$100 \times (1 + 0.05) = \$105$$

In this formula, the 5% interest rate is expressed as 0.05 because they are different ways of writing exactly the same number. Using 0.05 in the equation also is easier to understand.

If the investment is held for multiple years, then its value increases (or "compounds") each year. For example, say you make the investment three years long:

$$\underset{\text{Year 1}}{\$100 \times (1 + 0.05)} \times \underset{\text{Year 2}}{(1 + 0.05)} \times \underset{\text{Year 3}}{(1 + 0.05)} = \$115.76 \text{ (rounded)}$$

The equation shows that you increase the value of the current money for each year the money is invested. You can express this formula more generally by saying that the future value of the investment can be calculated by adding the number of years as an exponent to the interest rate:

$$\text{Present Value} \times (1 + \text{Interest Rate})^{\text{Number of Years}} = \text{Future Value}$$

Or for our example:

$$\$100 \times (1 + 0.05)^3 = \$115.76$$

A key insight is that this calculation takes a known current value of money and calculates a future value as your investment grows. To determine the value today (again, the present value) of a future amount of money (the future value), you can do the calculation in reverse—a crucial fact when using NPV for investment decisions. For example, if we're still assuming you can earn 5% per year, then what's the value of a dollar received in the future in today's terms? In other words, how do you calculate the present value of a known future value?

Remember, you *multiply* a known present value to calculate a future value. So, to reverse the process, you *divide* a known future value to calculate a present value.

To illustrate this, let's determine the current value of $100 received three years from now at 5% interest:

$$\text{Future Value} \div (1 + \text{Interest Rate})^{\text{Number of Years}} = \text{Current Value}$$

Or, for our example,

$$\$100 \div (1 + 0.05)^3 = \$86.38 \text{ (rounded)}$$

This reverse process is called *discounting* the future value to arrive at a current value. In this case, the interest rate doesn't indicate the amount of money you gain by investing now for a future payback. Instead, the **discount rate** refers to the interest rate to determine the present value of benefits received in the future.

Discount rate The interest rate used in NPV analysis to discount future cash flows to determine the present value of those cash flows.

Note that if you plug that current value of $86.38 into the original equation at 5%, you end up with a future value of $100. With this basic math in hand, you can apply these ideas to systems project investment decisions.

9.5.2.3 The Time Value of Money: Why It Matters to Systems Project Investment Decisions

You've seen that when estimating the business benefits of a systems project, you typically need to project those benefits several years into the future. You've also seen that you often need to adjust estimated business benefits over the first several years. The benefits may start small and then ramp up over time. For example, in the WPL example, when estimating the librarian labor savings resulting from patrons checking out a book using a kiosk, we assumed it would take several quarters for that kiosk usage to reach its maximum value.

However, this type of business benefits adjustment is different from NPV discounting. If we used NPV analysis for the WPL example, we'd also apply adjustments for the time value of money (again applying the principle that dollars received today are worth more to us than dollars received in the future).

Let's explore this with a simple example where the business benefits are level from the beginning: assume you've estimated that a systems project will deliver $25,000 per year of efficiency benefits (a level yearly amount starting in Year 1 and continuing thereafter) by automating certain transactions and reports.

To estimate the present value of those benefits using NPV, you need to decide how far into the future you should assume those benefits will continue. Given that software doesn't wear out with use, at least in theory you could assume those benefits continue forever, from Year 1 to Year Infinity. While Section 9.2.5.1 provided practical reasons why most organizations limit the time horizon for valuing business benefits to three to five years (for example, changing business requirements may lessen the value of the software over time), let's consider how you'd value benefits if you did project them out forever. In the field of finance, a series of benefits or payments received is called an **annuity**. If the benefits go on forever, that's called a **perpetuity**. Of course, given the time value of money, the current value of the $25,000 yearly benefit in Year 2 is less than Year 1, the value of Year 3 is less than Year 2, and so on. As you go to infinity, the value of the yearly benefit dwindles to near zero.

Annuity A fixed amount of money paid or benefit received for a series of time periods (typically, yearly).

Perpetuity An annuity in which the payments are made (or assumed to be made) forever.

Table 9-4 illustrates the current value of these $25,000 per year benefits for selected years. Note that even Year 1 is valued at less than $25,000 because we spent money to build the system in Year 0. So, if you have to wait a year beyond Year 0 to recognize Year 1 benefits, you need to discount the value even in Year 1.

Table 9-4: Current value of a $25,000 per year business benefit using a 5% discount rate

	Current Value of Selected Years with $25,000 per Year Benefits						
Yearly benefit	**Year 1**	**Year 2**	**Year 3**	**Year 10**	**Year 20**	**Year 50**	**Year 100**
$25,000	$23,810	$22,676	$21,596	$15,348	$9,422	$2,180	$190

What's the overall present value you obtain from summing this infinite series of $25,000 yearly benefits? Deriving the math is beyond the scope of this book, but the final formula for the current value of a perpetuity is surprisingly simple:

Present Value = Yearly Benefit ÷ Interest Rate

Assume that, for your firm, 5% is the right discount rate percent value to use. (We'll discuss the ins and outs of that assumption in a moment.) So, for this example

Present Value = $25,000 ÷ 0.05 = $500,000

This means the systems project, in theory, would deliver the equivalent of $500,000 in business benefits in today's money. So, should you actually proceed with the project? That depends on the estimated costs, along with some practical concerns.

9.5.2.4 NPV Decision Rule and Practical Pitfalls for Systems Projects

NPV provides a simple decision rule for making investment decisions: *if the present value of a project's business benefits exceeds costs, even by a dollar, then the project adds value and should be approved*. This difference between current costs and the discounted value of future benefits is what puts the "net" in "net present value." Using our previous example, assume the cost of the system is estimated to be $290,000. Because you'll largely spend that money to create the system now (in Year 0, meaning the present), the present value of the $290,000 cost is simply $290,000!

If you then "net" costs against benefits (subtracting the $290,000 in current costs from the $500,000 present value of the perpetuity benefits calculated earlier) you end up with a strongly positive net present value of the project of $210,000. Based on this number, the NPV decision rule would have you decide to invest in the project (unless you could find an even more strongly positive net NPV project to invest in instead).

This seems straightforward. But consider what this actually looks like over time. Table 9-5 shows the year-by-year present value of business benefits, including how they accumulate over time.

Table 9-5: Present value of business benefits year by year and accumulated

Year	Assumed Yearly Business Benefit	Current Value of Yearly Business Benefit	Accumulated Current Value of Business Benefits
1	$25,000	$23,810	$23,810
2	$25,000	$22,676	$46,485
3	$25,000	$21,596	$68,081
4	$25,000	$20,568	$88,649
5	$25,000	$19,588	$108,237
6	$25,000	$18,655	$126,892
7	$25,000	$17,767	$144,659
8	$25,000	$16,921	$161,580
9	$25,000	$16,115	$177,696
10	$25,000	$15,348	$193,043
11	$25,000	$14,617	$207,660
12	$25,000	$13,921	$221,581
13	$25,000	$13,258	$234,839
14	$25,000	$12,627	$247,466
15	$25,000	$12,025	$259,491
16	$25,000	$11,453	$270,944
17	$25,000	$10,907	$281,852
18	$25,000	$10,388	$292,240
19	$25,000	$9,893	$302,133
20	$25,000	$9,422	$311,555
Sum of years beyond 20			$188,445
Accumulated total			$500,000

Remember that the estimated project costs were $290,000. How many years of realizing benefits would it take before the project actually achieved a positive NPV? The table shows that the NPV of business benefits doesn't exceed $290,000 in costs for 18 years! Realizing that, does the decision to proceed with the project still seem reasonable? Ask yourself: If this was your own money, would you be willing to invest $290,000 now, realizing it would take that long to see a return on your investment?

Consider also the general uncertainties you face: per Chapter 8, your system cost estimate of $290,000 is likely to be quite rough. Per the planning fallacy, the most likely outcome is that the estimate might be low, with actual costs quite possibly significantly higher. Also, per the planning fallacy, you may find that your anticipated $25,000 per year business benefits may be overestimated (for example, if you find that not all of your consulting offices are willing to adopt the new system capabilities). To put this risk in concrete terms, if your system cost estimate ultimately needs to be revised upward by 50% (to

$435,000; per Chapter 8, unfortunately not an unheard of situation in systems projects), and if your yearly benefits turn out to be modestly lower at $21,000 per year (suggesting a value of $21,000 ÷ 5% = $420,000) then the project will never be worthwhile, even if the business benefits go on forever.

The inaccuracy of system costs and benefits estimates (compounded by the planning fallacy) undermines the NPV decision rule of "Do the project if the NPV is positive, even by a dollar."

Beyond that, the value of NPV may be further undercut by two additional practical concerns:

- **Uncertainty regarding the discount value to use:** In the example provided, we assumed a 5% discount rate without ever saying why that was an appropriate value. In fact, the percentage should be a value specific to each firm called the firm's **weighted average cost of capital** (WACC). The WACC is a fancy term for a firm's overall cost of raising money, by borrowing money, selling stock, or both. In short, if a firm has to raise money at 5% (as in the example), then economically it needs to make at least 5% just to pay back the investors who provided it with the money. Perhaps surprisingly, as a practical matter, many people making decisions about systems projects won't know the WACC figure they need to use, making NPV analysis impossible. For one thing, the calculation of the WACC is typically complex. But even if the WACC value is known, its usefulness may be limited because of the following point.
- **Short practical planning horizons:** NPV, in theory, considers the value of benefits for as long as the software capabilities are available. Given that software doesn't wear out, that theoretically means forever. However, just because software doesn't wear out doesn't also mean that it will, in reality, always provide business benefits. There are many reasons why many businesses don't like to assume business benefits for more than three to five years or so. Here are a few:
  - **Planning horizons limited to three to five years:** The first reason is general uncertainty regarding business strategy and practices. Put simply, many business clients will say, in effect, "I don't have a lot of confidence that we'll even be in this business or operating the business in this way for more than three years or so." Put another way, in the turbulent, high-velocity world of business today, even three years can seem like an eternity!
  - **Changing requirements over time:** Even if the firm does continue in this business, its requirements will likely change over several years. When this happens, the value of business benefits based on today's system features may decline. The firm may end up needing to make more investments to enhance the system's capabilities, investments that are not included in the current cost/benefit analysis. These investments may also be needed to keep the system's architecture up to date or to comply with new rules and regulations.
  - **Estimating errors likely dwarf NPV discount adjustments:** Finally, it's important to recognize that uncertainties regarding cost and benefit estimates generally are much greater than any short-term NPV adjustments for the time value of money. Put concretely, in the first several years of a project, the impact of discount adjustments (5% in the example and probably not higher than low double digits for most firms) will likely be swamped by the impact of estimating costs and benefits errors (50% or significantly higher).

Weighted average cost of capital (WACC) The discount rate (expressed as a percent) that represents a firm's overall cost of raising money. The WACC is the discount rate to be used in NPV cost/benefit analysis.

Because of these points, many businesses evaluating systems projects will deviate from the NPV approach as follows:

- **Three- to five-year horizon:** Many firms will only project benefits out about three years or so—rarely beyond five.
- **No discount rate applied:** Many will simply ignore the time value of money.

In doing so, such businesses are pursuing the simpler, more pragmatic investment decision approach that we explored with VFA and WPL: ROI analysis.

9.5.2.5 A Pragmatic Investment Decision Approach: Return on Investment Analysis

The previous section showed that two key sources of uncertainty cause many organizations to ignore NPV analysis in favor of a pragmatic alternative: return on investment (ROI) analysis. The first source is the lack of certainty regarding the accuracy of system costs and benefits estimates. The second source is the lack of certainty regarding the future direction of the business itself.

In general, these two sources of uncertainty cause many organizations to focus on a relatively short-term planning horizon, typically in the three- to five-year range. In turn, this short planning horizon reduces the effect of adjustments for the time value of money. As such, it becomes simpler and more straightforward to just sum the first several years of projected business benefits without adjustments for the time value of money. (As we did earlier, we'll continue to use the first three years in our examples.)

With this simplified estimate of business benefits in hand, the organization can then directly compare those benefits to the estimated system costs. Often, those costs may go beyond Year 0 costs to include ongoing costs for licenses, system administration and maintenance, and so on.

9.5.2.6 ROI Using a Cost/Benefit Analysis Grid

Considering the example described in the NPV section, using the ROI approach instead will often lead to starkly different decisions. For example, three years of unadjusted business benefits in that example are $75,000 ($25,000 per year for three years). This is far less than the estimated system cost of $290,000, which would lead us to not invest in the project.

Given this difference, let's revise the assumptions to create a more interesting example:

- **System costs:** Consisting of three estimated components:
  - **Software labor costs:** Estimated at $270,000, of which $30,000 is SA&D performed prior to the start of software development and $240,000 is the work the entire team will perform once software development has begun. (We explain in a moment why we split out costs this way.)
  - **Software licenses:** Priced at $15,000 per year on an ongoing basis. These could be items such as operating systems, database management systems, third-party reporting tools, and so on.
  - **System infrastructure:** Estimated $10,000 per year, including items such as cloud services fees, data center costs, network costs, and so on.
- **Business benefits:** Estimated at $330,000 per year starting in Year 1.

Table 9-6 presents a typical ROI **cost/benefit analysis grid** for analyzing this systems investment decision.

Cost/benefit analysis grid
A tabular approach used in ROI analysis to compare systems project costs to benefits. Each cost and benefit is classified as "one-time," "capital," or "annual." Based on those classifications, the costs and benefits are allocated in each year of the planning horizon.

Note several key points in the Table 9-6 analysis:

- **Project timing:** The system is assumed to be built in 2018 for a January 2019 go live date. So, Year 0 = 2018 and Year 1 = 2019. We look ahead into only the first three years the system is in use.
- **Cost categories:** Each cost belongs to one of the following categories. Note that these cost categories are general and can be applied to any systems project. However, the exact accounting rules for determining which category applies to any given cost depend on the country and the nature of the project; this example happens to use United States accounting rules for software created for use within the firm ("internal-use software"). Different US rules apply for software the firm intends to sell or lease externally ("external-use software").
  - **One-time costs:** Incurred in Year 0 and recognized immediately once the system goes live. Project planning and SA&D costs spent prior to software development and testing fall into this category (estimated at $30,000 in this example). These costs are recognized in the first year after the software goes live (Year 1).
  - **Capital costs:** Incurred in Year 0 but treated as assets with costs spread out ("amortized") over time. Software programming, testing, and other costs incurred once development starts are included in this category ($240,000 in this example). A typical

Table 9-6: Example return on investment cost/benefit analysis grid

			Costs (in $1000s)			Profit Impact (in $1000s)			
Cost/benefit categories	**Amortization period**	**Live date**	**One-time**	**Capital**	**Annual**	**2019**	**2020**	**2021**	**Total**
IT labor (noncapitalized)		Jan 2019	30			30			30
IT labor (capitalized)	36 months	Jan 2019		240		80	80	80	240
Licenses		Jan 2019			15	15	15	15	45
System infrastructure		Jan 2019			10	10	10	10	30
Total costs						135	105	105	345
Business benefits (positive values)					(200)	(200)	(200)	(200)	(600)
Net profit impact (positive values)						(70)	(100)	(100)	(255)

depreciation time frame for this is 36 months, or three years, so we allocated one-third of the costs (12 of 36 months, or $80,000 per year) to each of Years 1, 2, and 3.

  - **Annual costs:** Must be expended on an ongoing basis to maintain and operate the system. As such, they are shown as repeating in each of the three years in this planning horizon (in the example, annual license costs of $15,000 and annual infrastructure costs of $10,000).

- **Overall planning horizon:** The example shows both costs and benefits over three years. While different firms may use other time frames, this is not unusual and is convenient for at least two reasons. First, as discussed earlier, three years is a business planning time frame that many business leaders feel comfortable using. Second, the amortization schedule for capital costs matches this.
- **Portraying costs versus benefits:** Note that the costs are portrayed as positive numbers and the benefits as negative numbers. While this may seem counterintuitive, this approach is frequently used. In any event, the point is to distinguish costs from benefits and to net them against each other.
- **No adjustment for time value of money:** To reiterate, ROI analysis typically omits adjustments for the time value of money.

9.5.2.7 ROI Decision Rules Recognizing the Estimating Risk

Per Table 9-6, the previous example shows estimated business benefits significantly exceeding estimated costs: $600,000 versus $345,000, a seemingly "safe bet." Given the difference, should you decide to move forward with this investment? Perhaps, or perhaps not!

Again, estimates are subject to the risk of an unfortunately low degree of accuracy. For all the reasons detailed in Chapter 8, it would not be surprising for actual costs to exceed estimates by 50% to 100% or more. This could include unforeseen issues that arise after the project is started, such as the unplanned departure of existing, experienced software developers that are replaced with new, less-productive staff.

Similarly, software benefits could turn out to be lower than anticipated. For example, business users may adopt the software more slowly than expected, or business requirements may change significantly after the software is deployed, and so on.

Given these estimating risks, organizations may decide that they need an investment decision rule requiring a significantly more positive ROI ratio before approving a project. Using the ROI formula introduced earlier:

ROI Ratio = (Estimated Benefits − Estimated Costs) ÷ Estimated Costs

An organization might create a decision rule that, to approve a project, the ROI ratio must be at least 100%, to guard against cost overruns and benefits underruns. In our example:

ROI Ratio = ($600,000 – $345,000) ÷ $345,000 = 74%

Based on those figures, you wouldn't approve the project. In fact, it's not unusual for organizations, especially those with a low tolerance for risk, to require even higher ROI ratios. For example, a 100% ROI might be the lowest acceptable value, but even that might be considered a borderline case. Instead, for a high likelihood of project approval, a 200% ROI might be the threshold, as discussed earlier in this chapter.

Again, this approach helps ensure that the firm only pursues projects with a high likelihood of a positive return (or at least only a small likelihood of losing money).

9.5.2.8 One Additional Cost/Benefit Analysis Approach: IRR

Before we conclude this section, there's one additional approach to cost/benefit analysis you may encounter: the **internal rate of return (IRR)**. In a nutshell, IRR is closely related to NPV, in that it emphasizes adjustments for the time value of money.

Internal rate of return (IRR) The actual rate of return of a project. IRR is an alternative to NPV.

The key difference between IRR and NPV is this: NPV assumes a specific discount rate to calculate the present value of future business benefits. In contrast, IRR calculates the percentage return that the project is expected to create. For example, in NPV, say the WACC discount rate is 10%. You apply that 10% to discount future business benefits to see if the net present value of the benefits less costs is positive. As noted previously, the decision rule in NPV is to proceed if NPV is positive.

In contrast, in IRR you calculate the actual percentage return rate of the project. The decision rule is to proceed with the project if the calculated IRR exceeds the WACC. So, in this example, if IRR is greater than 10%, you'd proceed.

Given the similarities between NPV and IRR, the practical limitations of using NPV for systems projects also apply to IRR. Again, ROI is likely to be the better, more pragmatic choice in many situations compared to using either NPV or IRR.

Calculating IRR is beyond the scope of this book. For more information, consult a textbook in management accounting or financial management.

9.5.2.9 Bottom Line of the Bottom Line: NPV versus ROI Analysis

This section has presented two approaches (plus IRR) to determining whether a project should be approved or rejected. Each approach compares system costs estimates to business benefits estimates, and each is used by many different organizations.

Which is best? While NPV is widely seen as the soundest approach overall, practical considerations often point to the more pragmatic ROI approach. Ultimately, the choice between the two approaches will be made by business leaders.

We've presented both approaches so you'll be able to use whichever approach your organization wants to use.

9.5.3 Deeper Dive 9.3: Using Generative AI to Estimate Business Benefits and Conduct the Cost/Benefit Analysis

As we noted earlier in the book, generative AI tools don't yet have the ability to structure and run a client interview or generate meeting minutes. However, once those minutes are created, generative AI can produce a useful first draft of work products like a project vision document (see Section 2.5.7).

What about estimating business benefits and conducting the cost/benefit analysis? To find out, we fed the meeting minutes from Figure 9-2 into ChatGPT, preceded by this prompt:

The Wayback Public Library (WPL) is a public library system that is evaluating whether to create a new information system to improve its operations, which are currently 100% manual. The meeting minutes below describe the system cost and business benefits assumptions for a cost/benefit analysis. Based on those meeting minutes, generate: (1) the estimated business benefits, (2) the cost/benefit analysis ROI ratio, and (3) whether the project should be approved or not.

How did the AI system do? You can see its response in Figure 9-8:

To perform the cost/benefit analysis for the Wayback Public Library's new information system, let's break down the estimated business benefits and calculate the ROI ratio.

Estimated Business Benefits:

1. **Time Savings from Checkouts:**
   Current librarian seconds/checkout: 12 seconds Future librarian seconds/checkout (librarian checks out): 4 seconds Future librarian seconds/checkout (patron checks out): 0 seconds Checkouts per Year: 2,950,000
   Time Savings per Year = Checkouts per Year * (Current Librarian Seconds - Future Librarian Seconds) Time Savings per Year = 2,950,000 * (12 - 4) seconds = 17,700,000 seconds
   Time Savings in Hours = Time Savings per Year / 3600 seconds (1 hour) Time Savings in Hours = 17,700,000 / 3600 hours ≈ 4,916.67 hours
   Value of Time Savings = Time Savings in Hours * Librarian Salary per Hour * Value of Benefits as % of Salary Value of Time Savings = 4,916.67 hours * ($45,000 / 2080 hours per year) * 0.15 ≈ $15,837.98
2. **Reduction in Theft:**
   Total Collection Size: 405,000 Theft Rate per Year: 0.50% Average Cost per Stolen Item: $24.49 Portion "Stolen through the Door": 40%
   Number of Items Stolen through the Door = Total Collection Size * Theft Rate * Portion "Stolen through the Door" Number of Items Stolen through the Door = 405,000 * 0.005 * 0.40 ≈ 810 items
   Value of Theft Reduction = Number of Items Stolen through the Door * Average Cost per Stolen Item Value of Theft Reduction = 810 * $24.49 ≈ $19,830.90
3. **Increase in Fines Collected:**
   Percentage of Items Turned in Late: 14% Average Fine per Item per Day: $0.20 Average Days Late Once Turned In: 4.2 days Percentage of Fines Collected (Current State): 78% Percentage of Fines Collected (Future State): 85%
   Additional Fines Collected per Year = Checkouts per Year * Percentage of Items Turned in Late * Average Fine * Average Days Late * (Percentage of Fines Collected in Future - Percentage of Fines Collected in Current) Additional Fines Collected per Year = 2,950,000 * 0.14 * $0.20 * 4.2 * (0.85 - 0.78) ≈ $73,978.40

Total Estimated Business Benefits:

Total Estimated Business Benefits = Value of Time Savings + Value of Theft Reduction + Additional Fines Collected Total Estimated Business Benefits = $15,837.98 + $19,830.90 + $73,978.40 ≈ $109,647.28

Cost/Benefit Analysis ROI Ratio:

Systems Project Cost Estimate: $350,000 Total Estimated Business Benefits: $109,647.28

ROI Ratio = (Total Estimated Business Benefits / Systems Project Cost) * 100 ROI Ratio = ($109,647.28 / $350,000) * 100 ≈ 31.33%

Decision:

The decision rule for approving the project is an ROI greater than or equal to 100%. In this case, the calculated ROI ratio is approximately 31.33%, which is below the 100% threshold. Based on this analysis, the project should not be approved, as the estimated benefits don't exceed the project cost sufficiently to meet the defined ROI criteria.

Figure 9-8 ChatGPT response to request to conduct WPL benefits estimation and cost/benefit analysis

Comparing it to our own analysis in Section 9.3, you see:

- **Business benefits:** ChatGPT
  - **Increased fines collected:** ChatGPT was *accurate* (estimating about $74,000 to our $73,000, which is accurate).
  - **Reduced theft:** ChatGPT was *inaccurate* (estimating only about $20,000 to our $60,000). While ChatGPT got the one-year estimate correct here, it inexplicably failed to multiply that estimate by three to correspond to our planning horizon.
  - **Labor savings:** ChatGPT was *highly inaccurate* (estimating only about $16,000 to our $569,000). Here, ChatGPT fundamentally misinterpreted the meeting minutes. For example, it totally failed to take into account the effects of implementing the system with increasing levels of kiosk use over time. In addition, its calculation of labor savings based on librarian salaries and benefits was simply wrong. For example, ChatGPT calculated librarian cost per hour as ($45,000 / 2080 hours per year) * 0.15 = $3.25 per hour, when the correct figure is ($45,000 * (1 + 0.15)) / 2080 hours per year = $24.88 per hour.
- **Cost/Benefit Analysis:** Of course, given that ChatGPT generated grossly incorrect, low business estimates, it reached the wrong conclusion on the cost/benefit analysis, concluding that the ROI was only 31%. And note that its ROI calculation was incorrect, as well.

The moral of this story is clear: You can ask generative AI to save you time by calculating a first draft of your business benefits and cost/benefit analysis. But you have to be able to review and correct those calculations based on your own expertise!

9.5.4 Deeper Dive 9.4: Using Automation to Reduce Average Labor Costs and Redeploy Experienced Staff to Marketing and Sales Roles

In the VFA and WPL examples, we showed how system-enabled automation can generate labor cost savings by reducing the total business labor hours needed. Beyond that, though, in many cases, automation can also generate labor cost savings in a second way: by reducing the average labor cost per hour. How does this work? In short, the system can provide support to users in remaining manual tasks by improving quality and consistency. For example, the system could help users perform data quality checks, ensure all process steps are carried out, check calculations for completeness and consistency, and so on. With that future state system support, junior frontline users may not require the same level of skills and experience as in the current state. Also, senior supervisors may not need to spend as much time reviewing their employees' work.

The following case provides a comprehensive business benefits estimation example that incorporates labor cost savings from both reduction of overall staff hours and a lowering of average labor costs by increasing the proportion of the remaining work done by junior staff. It also illustrates the idea introduced in Section 9.4.2.3 of switching senior staff members whose current labor hours would be reduced or eliminated into marketing/sales roles. This switch leads to higher revenues.

> **Case:** Consider a healthcare consulting firm that each client pays to collect the client's healthcare claims data, analyze it to turn it into management information in the form of healthcare cost trends, and report that information back to the client. Suppose this consulting business process is highly manual and error prone: consulting staff manually collect client data and rekey it into a spreadsheet, then the data is copied and pasted into other spreadsheets to transform it into information, which is finally transmitted back to the client. This could be articulated: "Our consulting data analysis business process is highly manual and inconsistent, requiring too many hours of data analyst labor. We need to lower our costs and increase our quality using information technology."

The key system capabilities here might include the following:

- Creating automated extract/transform/load (ETL) capabilities to collect and store the client's claims data

- Summarizing the data into cost-trend information and visualizing key trends using a data analysis tool
- Using a secure communication tool (e.g., secure email or a secure web portal) to transmit the information back to the client

Based on these key system capabilities, we can identify the following key business benefits:

- Increased profit margins on every dollar of revenue because of a reduced number of labor hours needed to produce the consulting services
- Increased profit margins on every dollar of revenue because the firm does a higher proportion of the remaining manual work using more-junior, lower cost data analysts
- Increased revenues because senior data analysts will be freed up to help consultants sell more services to clients

Moving from initial vision to business analysis, we identify the following steps in the overall calculation process for a given client:

1. **Extract, transform, and load (ETL):** Collecting from the client the raw healthcare claims data to be analyzed, as well as checking over this data for completeness and consistency.
2. **Data aggregation and summarization:** Performing intermediate calculations to summarize the raw claims data into aggregated values that can be more easily analyzed (e.g., taking health insurance claims and summing them up by month).
3. **Analysis and projections:** Creating management information, such as conducting a cost trend analysis in the past, then projecting values into the coming years. This could include "what if" analysis manipulating assumptions such as the cost charged for a physician visit or adding healthcare services paid for by the plan.
4. **Generating a client report and conducting peer review:** Generating the report involves taking the key calculation outputs to generate a high "production values" report (e.g., well-designed graphical outputs embedded in a PowerPoint or Google Slides presentation) for client consumption, including adding commentary and recommendations. Finally, all of the previous steps in the process are carefully checked to validate that the data and results are correct (a process called "peer review" in consulting circles).

Staff analysis: current state			
Employee level	**Hourly rate**	**% Effort**	**Weighted**
A: Junior	$25	20%	$5
B: Intermediate	$40	40%	$16
C: Senior	$80	40%	$32
Blended labor cost per hour			$53

Figure 9-9 Healthcare claims cost analysis business process current state average blended labor cost per hour

In this example, imagine there are three staff levels (A, B, and C) ranked by increasing level of experience, skill, and hourly costs, as portrayed in Figure 9-9. Note that the figure shows the proportion of effort needed to deliver the firm's consulting services. In the current state, with a high degree of manual activities, most of the work must be performed by Level B intermediate and Level C senior staff members, rather than by Level A junior staff members. Indeed, based on the "% effort" figures, there are twice as many both Level B and Level C staff as there are Level A staff. This results in an average cost per hour, called the **blended labor cost per hour**, of $53.

Blended labor cost per hour The average labor cost per hour for a group of business professionals producing a certain amount of work. Calculated using an average of staff labor rates weighted by the proportion of time worked by each level of staff.

An overview of this business process is presented in Figure 9-10. Note that this general portrayal of the process doesn't distinguish between the highly manual current state business process and a more highly automated future state version of the business process.

In the current state, the overall process is highly manual, requiring junior staff to visit a client's environment to download claims data, then review it and summarize it using a series of spreadsheets. Intermediate-level staff then take over to apply trend analysis, again using even more spreadsheets—and then conduct the "what if" analysis. At that point, other, more senior staff generate the report, including adding the commentary and recommendations. Finally, prior to

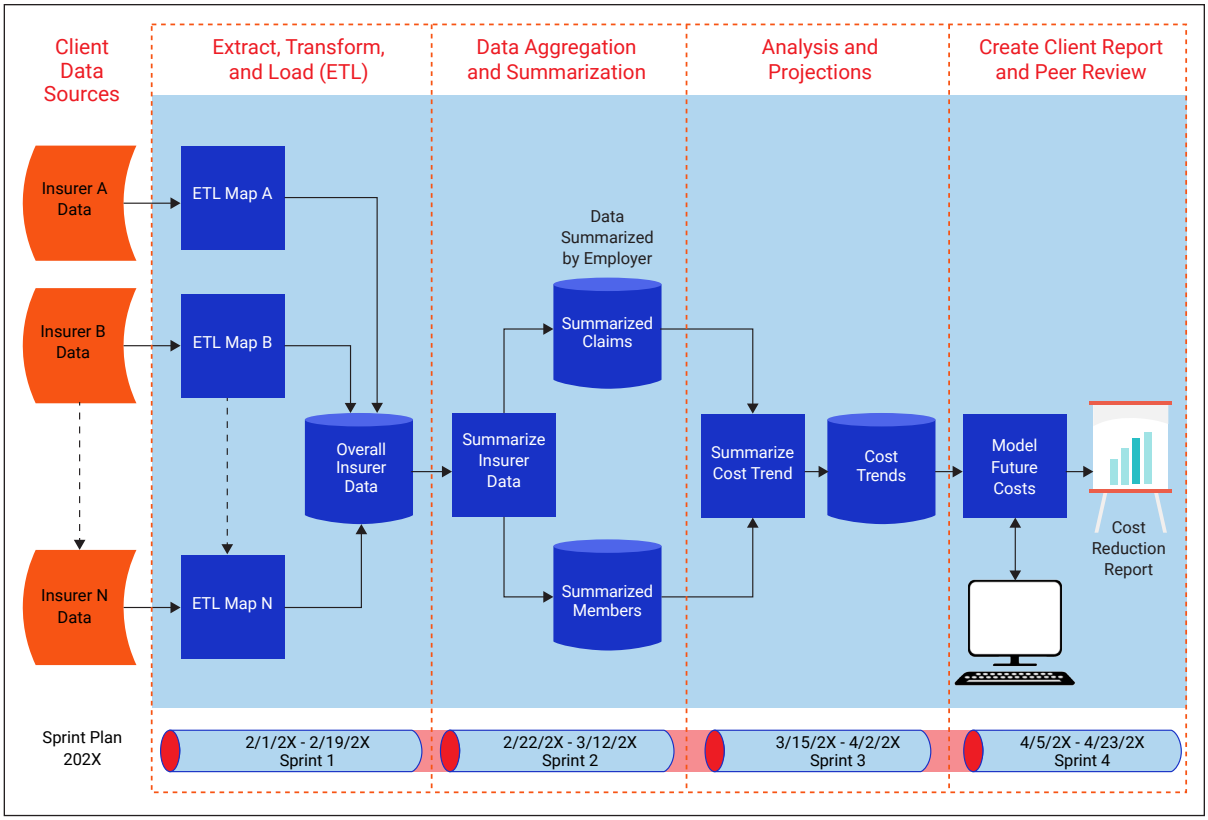


Figure 9-10 Example healthcare claims cost analysis business process

sending the report to the client, other staff not previously involved with this specific client calculation are given the data, information, and report to conduct the peer-review process. This is especially important, given the large number of manual steps in the process that could be subject to human error.

Clearly, in this example, it's not hard to imagine how implementing systems could automate many of these steps in the business process, reducing costs by reducing the labor hours expended. However, the opportunity actually is richer than that.

This process, although highly manual in the current state, can be viewed as being highly routine and repetitive. As such, it's ripe for implementing software automation. For example, a software-driven ETL process could be implemented to download and translate (or "map") the client's data to be stored in the database. At that point, another software routine could be run to check the data for consistency and completeness. It also should be possible to create a software function that would read the individual claims and summarize them, thereby eliminating the manipulation of data using spreadsheets. It's likely that some of the routine analysis and future projections work could be automated, although at this point human judgment in conducting "what if" analysis begins to be important, suggesting only a partial level of automation at this stage. Similarly, while it should be possible to automate the embedding of key calculation outputs into a presentation, it's likely that some of the free text commentary in the presentation will require human judgment and creativity.

In creating the requirements, the IT team conservatively estimates that creating and implementing all of these capabilities will cost about $350,000. As noted previously, using typical ROI evaluation criteria, the firm wishes to achieve an ROI of 200%, a ratio of business benefits to system costs of 3:1. The business benefits in this case would need to be 3 × $350,000 = $1,050,000 or better.

In creating a business benefits estimate for this project, the team realizes it wouldn't be sufficient to merely determine the amount of savings per year for a single client. Rather, they would have to determine overall savings based on the "per client" savings estimate multiplied by the number of clients the firm serves. With this in mind, the team starts by using some basic information known from the financials system, including the number of consulting offices the firm operates (5), the average number of clients each office serves (8), and the average amount each client pays per year ($20,000). Based on those figures, the team estimates that the firm was earning $800,000 per year for those services (5 offices × 8 clients per office × $20,000 per client). They know from the financials that the operation had an operating margin (profits after deducting the labor costs of the business staff) of 40%. That margin means the cost of delivering the services was 60% of the $20,000 per client revenue, or $12,000. Then, using the $53 blended labor cost per hour from Figure 9-9, they're able to derive that the team was averaging about 226 hours of effort for each client ($12,000 cost per client ÷ $53 per hour).

Turning to the benefits of the automation, the team examines each process affected by the new system capabilities and estimates they could save 60 hours of effort per client, reducing hours of effort from the current state of 226 hours to a future state of 166 hours, increasing profitability per client by $3,180. At this point, the team doesn't try to estimate any change in the proportions of staff (Levels A, B, and C) needed to deliver the services.

This analysis is summarized in Figure 9-11.

Current State	Values	Notes
Consulting offices	5	Known figure
Average clients per consulting office	8	Known figure
Annual revenue per client	$ 20,000	Known from financial statements
Operating margin %	40%	Known from financial statements
Operating costs %	60%	Derived from operating margin
Operating costs per client	$ 12,000	Computed from annual revenue per client * operating costs
Blended labor cost per hour	$ 53.00	From "Staff analysis—current state" analysis in Figure 9-9
Hours of effort per client	226	Operating labor costs per client / blended labor cost per hour
Future State		
Benefit: Estimated time savings per client	60	Based on review of manual effort impacted by new system
Revised hours of effort per client	166.4	Pre-enhancement hours – assumed time savings
Post-enhancement operating costs per client	$ 8,820	Post-enhancement hours of effort per client * post-enhancement blended labor cost per hour
Increase in profitability per client	$ 3,180	Pre-enhancement – post-enhancement operating costs per client
Increase in profitability for book of business	$ 143,100	Increase in profitability per client * 45 clients (1st year of operation)
Operating margin %	56%	Old operating margin increased by enhanced profitability

Figure 9-11 Increase in estimated per client profitability based on reduced labor hours only

Note that the firm had a plan to grow the business by adding on average one new client per year. So, in "Year 1" (the first year after the new systems capabilities are implemented), there would be 9 clients per consulting office, in "Year 2," there would be 10 clients per consulting office, and so on.

With these growth figures in mind, the team is able to project business benefits, again only based on an overall reduction in labor hours (see Figure 9-12). Note that this takes into account an improved operating margin for each client, coupled with a growing number of clients over time.

Recalling that the estimated cost of the project was $350,000, the projected business benefit of $477,000 is positive but not close to the desired ROI ratio of 3:1 ($477,000 / $350,000 = 1.36).

Based on this result, the team works to refine the estimated business benefits, this time examining the impact of the new system capabilities on the mix of staff resources needed. Given the higher level of automation—and, hence, the higher level of quality and consistency of data acquisition and analysis—the team determines that more of the remaining work could be shifted away from Level C senior staff to Level A junior staff. For example, in the future state, Level A junior

Current state: highly manual	Year 0	Year 1	Year 2	Year 3	
Consulting offices	5	5	5	5	
Clients per consulting office	8	9	10	11	
Overall clients	40	45	50	55	
Revenue	$ 800,000	$ 900,000	$ 1,000,000	$ 1,100,000	
Operating profit		$ 360,000	$ 400,000	$ 440,000	
Future state: with automation					
Consulting offices	5	5	5	5	
Clients per consulting office	8	9	10	11	
Overall clients	40	45	50	55	
Revenue	$ 800,000	$ 900,000	$ 1,000,000	$ 1,100,000	
Operating profit		$ 503,100	$ 559,000	$ 614,900	
Increased profit over 3 years		$ 143,100	$ 159,000	$ 174,900	**$ 477,000**

Figure 9-12 Increase in estimated overall profitability based on reduced labor hours only

staff could perform twice as much of the manual work: 40%, up from the current state's 20%. As portrayed in Figure 9-13, this has the impact of reducing the blended rate from $53 per hour to $42 per hour.

Modifying the per client profitability model to take this reduction in blended labor costs into account results in the revised model in Figure 9-14. Note that the increase in per client profitability has jumped from $3,180 (56% operating margin) in Figure 9-11 to an increase of $5,011 (65% operating margin) in Figure 9-14.

This approach indicates the client will save money for two reasons:

1. The task requires fewer hours of overall labor.
2. The task can be performed using a higher proportion of junior staff members, reducing the average cost per hour.

Staff analysis: current state			
Employee level	**Hourly rate**	**% Effort**	**Weighted**
A: Junior	$25	20%	$5
B: Intermediate	$40	40%	$16
C: Senior	$80	40%	$32
Blended labor cost per hour			$53

Staff analysis: future state			
Employee level	**Hourly rate**	**% Effort**	**Weighted**
A: Junior	$25	40%	$10
B: Intermediate	$40	40%	$16
C: Senior	$80	20%	$16
Blended labor cost per hour			$42

Figure 9-13 Decrease in average blended labor cost per hour based on shifting work from senior staff to junior staff

Current State	Values	Notes
Consulting offices	5	Known figure
Average clients per consulting office	8	Known figure
Annual revenue per client	$ 20,000	Known from financial statements
Operating margin %	40%	Known from financial statements
Operating costs %	60%	Derived from operating margin
Operating costs per client	$ 12,000	Computed from annual revenue per client * operating costs
Blended labor cost per hour	$ 53.00	From "Staff analysis—current state" analysis in Figure 9-9
Hours of effort per client	226	Operating labor costs per client / blended labor cost per hour
Future State		
Benefit: Estimated time savings per client	60	Based on review of manual effort impacted by new system
Revised hours of effort per client	166.4	Pre-enhancement hours – assumed time savings
Revised blended labor cost per hour	$ 42.00	
Benefit: Estimated reduced blended labor cost per hour	$ 11.00	
Post-enhancement operating costs per client	$ 6,989.43	Post-enhancement hours of effort per client * post-enhancement blended labor cost per hour
Increase in profitability per client	$ 5,010.57	Pre-enhancement – post-enhancement operating costs per client
Increase in profitability for book of business	$ 225,475	Increase in profitability per client * 45 clients (1st year of operation)
Operating margin %	65%	Old operating margin increased by enhanced profitability

Figure 9-14 Increase in estimated per client profitability based on both reduced labor hours and shifting work from senior staff to junior staff

Current state: highly manual	Year 0	Year 1	Year 2	Year 3	
Consulting offices	5	5	5	5	
Clients per consulting office	8	9	10	11	
Overall clients	40	45	50	55	
Revenue	$ 800,000	$ 900,000	$ 1,000,000	$ 1,100,000	
Operating profit		$ 360,000	$ 400,000	$ 440,000	
Future state: with automation					
Consulting offices	5	5	5	5	
Clients per consulting office	8	9	10	11	
Overall clients	40	45	50	55	
Revenue	$ 800,000	$ 900,000	$ 1,000,000	$ 1,100,000	
Operating profit		$ 585,475	$ 650,528	$ 715,581	
Increased profit over 3 years		$ 225,475	$ 250,528	$ 275,581	**$ 751,585**

Figure 9-15 Increase in estimated overall profitability based on both reduced labor hours and shifting work from senior staff to junior staff

Updating the overall profitability model with this increased per-client profitability results in Figure 9-15.

Note that the ROI ratio has improved to 2.15:1 ($751,500/$350,000), which is better but still short of the desired 3:1 ratio. With this in mind, the team turns to the issue of the senior staff. First, based on direction from business leadership, they want to reassure senior staff that they won't be laid off. Rather, leadership has directed that the senior staff be retained and given opportunities to grow. As such, the team works with sales and marketing leaders to determine if the senior staff members might be able to use their freed-up time to help the firm grow faster by taking on a new sales and marketing role. The sales and marketing leaders agree that doing so could help the firm add at least one additional client per office per year: more specifically, the firm would add two clients to each office each year, rather than only one.

Revising the model yet again, Figure 9-16 shows the final estimated benefits, this time including the impacts of the firm growing more rapidly.

Current state: highly manual	Year 0	Year 1	Year 2	Year 3	
Consulting offices	5	5	5	5	
Clients per consulting office	8	9	10	11	
Overall clients	40	45	50	55	
Revenue	$ 800,000	$ 900,000	$ 1,000,000	$ 1,100,000	
Operating profit		$ 360,000	$ 400,000	$ 440,000	
Future state: with automation					
Consulting offices	5	5	5	5	
Clients per consulting office	8	10	12	14	
Overall clients	40	50	60	70	
Revenue	$ 800,000	$ 1,000,000	$ 1,200,000	$ 1,400,000	
Operating profit		$ 650,528	$ 780,634	$ 910,740	
Increased profit over 3 years		$ 290,528	$ 380,634	$ 470,740	**$ 1,141,902**

Figure 9-16 Increase in estimated overall profitability based on reduced labor hours, shifting work from senior staff to junior staff, and repurposing senior staff to sales and marketing

With this final revision, the estimated business benefits ROI now exceeds 3:1 ($1,142,000 ÷ $350,000 = 3.26). With that result, the leadership team determines that there's a solid business case for moving forward with the project.

This example is obviously highly specific. However, it illustrates general techniques for analyzing the impacts of automation on bottom-line margins through reduced labor hours and lower blended labor costs. It also illustrates the idea that automation may be able to improve "top-line" revenues by repurposing staff freed up by automation.

Project Approach Selection

10.5 Deeper Dives: Advanced Project Approach Selection Topics

In this section, we do deeper dives on several advanced project approach selection topics that all build on and expand coverage from earlier in this chapter:

- **Deeper Dive 10.1** discusses the effects of nonfunctional requirements characteristics on project approach selection.
- **Deeper Dive 10.2** explains how to use a radar chart to comprehensively assess all project model characteristics, combining functional requirements, nonfunctional requirements, and team characteristics to determine project approach using the Home Grounds Model.
- **Deeper Dive 10.3** explores project approach selection for artificial intelligence (AI) projects.
- **Deeper Dive 10.4** explores project approach selection for data analytics projects.
- **Deeper Dive 10.5** explores project approach selection for configuration projects.

10.5.1 Deeper Dive 10.1: Effects of Nonfunctional Requirements Characteristics on Project Approach Selection

Earlier, we introduced the idea of nonfunctional requirements. Nonfunctional requirements pertain to the qualities of the system that are not specific to a business or industry. For example, let's say you determine that a system needs the following:

- Supports up to 1,000 users simultaneously, with the ability to scale to 5,000 users without major technical changes
- Averages under two-second response times under full load during normal business hours
- Needs 99.9% uptime during normal business hours
- Can communicate via standard internet interfaces called "web services" to 12 other systems
- Implements security mechanisms to protect confidential customer data, including web applications exposed to the public internet

This seems clear enough. But what industry or company are you talking about? From these points alone, there's no way to know. They could pertain to retail, manufacturing, financial services, energy, education, or many others. That's what makes these *nonfunctional* requirements: they don't speak to functionality that's specific to a particular industry or business.

In contrast to functional requirements (with their focus on logic, data, and user interface meeting the requirements of a specific business or industry), nonfunctional requirements pertain to the requirements and designs of the system's technical architecture, including infrastructure, data communications, and security. When you create these nonfunctional requirements at the outset of a project, you're engaged in **big design up-front (BDUF)** (in contrast to BRUF for functional requirements).

Big design up-front (BDUF) Creating detailed, nonfunctional requirements before software development starts.

As noted previously, while the BA needs to assess nonfunctional requirements, addressing many of them in detail may involve working with a technical specialist. This could include involving an architect, senior development lead, security specialist, data communications network engineer, and so on. As such, here, we briefly explain these items and their impacts.

As each of the following factors increases, so does the need to perform architectural planning via big BDUF:

Performance The amount of work a system must perform in terms of volumes of users or transactions. Typically includes speed and reliability metrics.

- **Performance:** This defines how much work the system must be able to effectively support. This can be expressed in terms of users simultaneously supported, average response times, transactions processed per hour, data volumes per hour, and more. Importantly, performance requirements are often stated in terms of an initial expectation and the ability to scale to higher levels without major changes in technical architecture. A relatively simple example of this is a database application built using a single-user tool such as Microsoft Access. This application might have complete and effective features for one (or a few) users. But there's no way you could scale it to, say, 100 users without having to recreate the system in a larger scale technical environment. A meatier example would be a website that can support 100 users simultaneously in a single country but needs a significantly different architecture to enable scaling to tens of thousands of users on multiple continents.

Supportability The degree to which a system needs to be easily reconfigured (maintainability) and easily updated with new features (extensibility).

- **Supportability:** This involves designing the system to ensure that changes such as reconfiguring tables of values and new software features can be implemented easily over time. The former is called *maintainability*, and the latter is called *extensibility*. The degree to which this is necessary depends on where the system fits in the overall set of applications in an organization (which is called the application portfolio). For example, consider a system that will be used far into the future and will likely need significant investments to extend its features over time. Supportability requirements would be high for such a system. On the other hand, consider a system that will be used by only a small number of users for another year, after which it will be replaced by another system. This system would have low supportability requirements.

Criticality The degree to which a system faces high costs for poor reliability, security, safety, or auditability.

- **Criticality:** This general concept points to the cost of things going wrong—things like system downtime or security breaches. Several key subfactors include:
  - **Mission criticality:** If the organization can't operate its core business without the application running, then the application is mission critical. Examples could include an airline's reservation system, a health insurer's claims processing system, or a manufacturer's logistics and supply chain management system.
  - **Sensitive/confidential data:** Customer and financial data are common examples of this kind of data. Even more critical examples can be found, for example, in patient-identifiable healthcare data.
  - **Facing the public internet:** Confidentiality concerns are magnified when the application can be accessed via the internet (as opposed to only being accessible by employees within the firm's secure intranet environment).
  - **Having impacts on human safety:** This tends to be a major concern for systems that control machines, for example, an airplane autopilot system, an autonomous driving system for a car, a power plant control system, or systems that control healthcare devices used to treat people (for example, X-rays or anesthesia). However, even some administrative systems may impact human safety, including systems that control building security sensors and systems that allocate healthcare supplies and drugs in clinics and hospitals.
  - **Being subject to regulation or audit:** Certain types of systems (for example, banking systems, some systems used in government, and so on) must comply with formal risk and legal audit requirements. This can compel the software team to use BDUF and/or BRUF simply because laws and regulations require it.

Integration The degree to which a system needs to interoperate or exchange data with other systems.

- **Integration:** This refers to the need to integrate or interface your system with other application systems. As the number of other systems increases, so does the need to plan for that interaction via BDUF. Note that this might sound a bit like the discus-

sion in Section 10.2 of the Interdependence functional requirements characteristic. In fact, there's a relationship between Interdependence and Integration: they both pertain to implementing application programming interfaces (APIs) between systems. The difference is this: Interdependence pertains to specific functionality of each specific API. It considers what specific data is exchanged, which functionality is invoked, and how those things affect your system's programming. In contrast, Integration pertains to the general architecture of your APIs; for example, do you use RESTful APIs, SOAP APIs, or some other standard API approach? And how do you integrate with a system that doesn't support your standard approach? For example, an older system may not natively support modern methods of system interaction, such as web services. You may need to use BDUF to determine how to retrofit or upgrade the system to do so.

- **Technology:** The need to introduce new technologies into your application environment increases the need for planning via BDUF. There are many examples of this. For instance, an older system using a desktop interface may need to move to a web browser interface. Or you may find that an existing technology, such as an operating system or database system, is obsolete and no longer supported, forcing you to move to a different, modern technology. You may decide to introduce a new third-party technology, such as a workflow component that will orchestrate work throughout an administrative system. This could be highly beneficial, but it also might force you to integrate it in complicated ways with many existing modules and programs. These kinds of situations cry out for BDUF. In many cases, this would include not just planning but also conducting a preliminary proof-of-concept (PoC) project. PoC projects are a great way to ensure that a new technology works both by itself and with other existing technologies. Note that new Technologies increase our IT Team New Skills needed.

Technology The degree to which a system needs to be built or enhanced using new, unproven, or not previously integrated information technologies.

10.5.2 Deeper Dive 10.2: Using Radar Charts to Comprehensively Assess Project Characteristics Using the Home Grounds Model

In previous sections, we've explored assessing project characteristics and their impact on the optimal project approach in three different categories:

- **Functional requirements characteristics:** in the Fundamentals
- **Team characteristics:** in SA&D Professional Toolkit
- **Nonfunctional requirements characteristics:** in Deeper Dive 10.1

Although all these categories and their characteristics pertain to every project, we explored each category separately because the overall model is so complex.

But now that we've covered each category individually, you can learn to assess all categories together for a single project to get an overall understanding of how to optimize your project approach.

Again, your optimized project approach choice will in all cases be a version of the hybrid approach we've described, but optimized by determining what proportion of requirements to capture up front:

- BRUF (big requirements up-front) for functional requirements
- BDUF (big design up-front) for nonfunctional requirements

10.5.2.1 Making Sense of Overall Projects Using the Home Grounds Model

In integrating this material, we'll use a framework called the **Home Grounds Model**, which was first introduced by Boehm and Turner (2004) and later extended into the model presented here (Spurrier & Topi, 2023). In short, the Home Grounds Model provides you with a complete list of characteristics defining projects that will be best supported by emphasizing agile emergent requirements versus projects that will be best supported by emphasizing hybrid BRUF.

Home Grounds Model A model that defines different sets of project characteristics, indicating where agile and plan-driven approaches are most appropriate and likely to succeed.

Table 10-2: Home Grounds Model project characteristic definitions	
Project Characteristic	**Project Characteristic Description *"The Degree to Which…"***
Functional requirements characteristics	
Number	• Project includes many new features • Features include multiple functional areas
Complexity	• Individual features are complicated • Requirements vary across different users, departments, or offices • Goals require multiple projects and/or multiple systems
Interdependence	• Existing application is difficult to update because of high coupling • New features build on each other, so they must be built in a specific order
Clarity	• Current state of business and software is clearly understood • Future state can be clearly understood up front
Stability	• Requirements change slowly over time
Nonfunctional requirements characteristics	
Performance	• Software must support large numbers of users/transactions/data • Planned need for high performance contrasts with current low performance needs
Supportability	• Software needs to easily support future extensions to functionality • Software needs to be easily maintainable
Criticality	• Software is mission critical • Software security needs to protect sensitive data • Software impacts human safety • Software is subject to legal audit or requires formal requirements
Integration	• Software must integrate or interface with many other systems
Technology	• Need to use or integrate new or unproven technologies • Technology is obsolete and needs to be updated or replaced
Team characteristics	
IT team size	• Many IT team members • Multiple IT teams to coordinate
IT team locations	• Multiple locations • Multiple time zones • Multiple native languages and cultures • Multiple organizations (e.g., internal team working with a vendor)
IT team new skills	• Many skill sets with high levels of team member specialization—especially when team currently lacks those skill sets • Team needs training on new or existing technologies • Team needs training on system development process • Team needs better cohesiveness and communication
Client team	• Many subject matter experts (SMEs), sponsors, and/or other stakeholders • Multiple areas of expertise • Multiple locations, time zones, languages, and cultures • Clients face diverse laws, regulations, and market practices • Clients value formal project planning and management

Table 10-2 presents the Home Grounds Model's key project characteristics. Echoing our earlier discussions, each characteristic is grouped into one of the three major categories:

- **Functional requirements characteristics**
- **Nonfunctional requirements characteristics**
- **Team characteristics**

Table 10-2 also lists and defines the key characteristics in each of these three categories. In general, for the characteristics listed in Table 10-2, low values for the degree to which they are true would increase the appropriateness of the agile emergent requirements. Conversely, high values would push you toward more plan-driven BRUF.

You can see this more clearly in Table 10-3. Here, we list the same categories and characteristics, but we also express the values of characteristics that define projects that best fit the "Agile Home Ground" (where you should use low BRUF and high emergent requirements) versus the "Plan-Driven Home Ground" (where you should use high BRUF and low emergent requirements).

Table 10-3: Home Grounds Model of project characteristics for agile and plan-driven (or hybrid) project approaches

Project Characteristic	Agile Home Ground	Plan-Driven Home Ground
Functional requirements characteristics		
Number	• Small number of new features • Focused on one function • Small budget	• Many new features • Focused on multiple functions • Large budget
Complexity	• Simple features • Single project • Simple data schema • Single version of requirements	• Complex features • Multiple interacting projects • Complex data schema • Many requirements variations
Interdependence	• Brand-new software application • Enhancements to a modern, well-designed existing application • User stories are independent	• Enhancements to an existing, "legacy" application that is poorly designed • User stories must be built in a specific, logical order
Clarity	• Start-up business • New product, service, or function • Responding to confusing, turbulent environment	• Current business and software well understood (or can be) • New requirements well understood (or can be)
Stability	• Requirements changing rapidly	• Requirements changing slowly
Nonfunctional requirements characteristics		
Performance	•Small number of users • Low transaction/data volume	• Many users • High transaction/data volume
Supportability	• Tactical application • Proof of concept/"throwaway code"	• Strategic application • High future investment
Criticality	• Nonessential application • Public data only • Internal access only • No safety risks • No regulations or auditability	• Mission-critical application • Protect sensitive/confidential data • Facing public internet • Impacts human safety • Subject to regulation or audit
Integration	• Software operates in isolation from other systems	• Software integrates or interfaces with many other systems • New approaches (e.g., web services)
Technology	• Continuing to use existing, proven technologies	• New tech to learn, prove, or update • Integrate with existing tech stack
Team characteristics		
IT team size	• Less than 10 team members • Single team	• Many team members • Organized into multiple teams
IT team locations	• Single location (single room) • Common language and culture • Team all from the same organization	• Multiple locations and time zones • Multiple languages and cultures • Multiple organizations
IT team need for new skills	• Strong technology skills matching project • Strong, existing project approach • Long-standing, cohesive team	• New, unfamiliar technologies and/or need for team specialization • Adopting new project approach • Multiple new team members
Client team size and diversity	• Single product owner or SME • Single department and function • Clients in single location • Single version of requirements	• Multiple product owners or SMEs • Multiple departments or functions • Multiple locations, time zones, languages, and cultures • Requirements vary significantly

10.5.2.2 Using Radar Charts to Plot Project Characteristics

The key idea of the Home Grounds Model is that agile and plan-driven characteristics are diametrically opposed. However, few software projects will exactly match either the Agile Home Ground or the Hybrid Home Ground. The clear majority will be a mixture of both.

Even if we can assess all project characteristics within the three categories, looking at this in a tabular format is challenging. Instead, to more easily make sense of these many characteristics, we use a special kind of graph called a **radar chart**. A radar chart displays a series of measurements (as in the Home Grounds Model measurements of a systems project) in the form of a graph showing these measurements plotted on axes surrounding a central point. Figure 10-12 shows an example project characteristics radar chart, plotting the characteristics of a hypothetical systems project. For each factor, we rate it using a five-point scale range: "Very Low," "Low," "Medium," "High," or "Very High." For example, in the figure, "Technology" is rated "Very Low." Conversely, "Complexity" is rated "Very High."

Radar chart A type of graph in which a series of measurements are plotted on axes surrounding a central point.

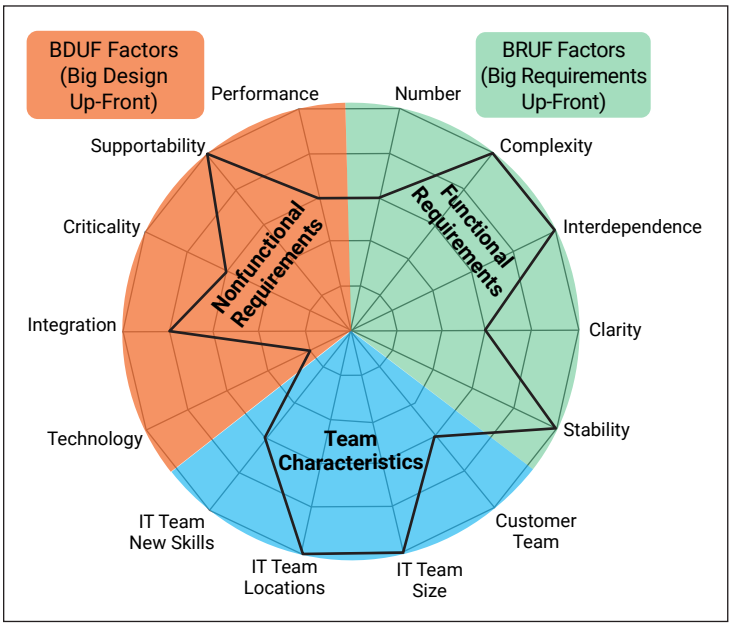


Figure 10-12 Example project characteristics radar chart

This approach allows you to visualize the relevant characteristics of a project. That chart, in turn, will guide you in deciding how agile or plan-driven your project should be.

Even so, how do you make sense of the plot? First, realize that the chart is organized so that projects aligning to the Agile Home Ground will have project characteristics measurements plotting close to the center of the chart. In contrast, projects aligning to the Plan-Driven Home Ground will have characteristics plotting near the outer edge of the chart.

Second, note that the color and pattern coding of the chart groups and highlights the specific project factors for each of the three major project characteristics categories. You group the various individual characteristics this way because each category impacts a project in a different way.

Factors pertaining to functional requirements (what the system must provide in terms of business capabilities) are grouped together in the upper right. When you plot at the outer edge of these factors, you'll need significant BRUF.

Factors pertaining to nonfunctional requirements (indicating the need for technical architecture planning) are grouped together in the upper left. When you plot at the outer edge of these factors, you'll need significant BDUF.

Finally, characteristics of the IT and business client teams are grouped together at the bottom. By "IT team," we mean the information technology professionals working on the project, which may include BA, project management, developer, tester, and other roles. Note that in the Scrum approach, these roles specifically include the Scrum master and development team.

In contrast, by "client team," we mean the individuals who the new or enhanced system serves, which include business users of the system but also other business stakeholders: project sponsors, managers, champions, and subject matter experts (SMEs). Note that in the agile Scrum approach, the primary business client is the product owner. See Chapter 11 for more details on these and other roles.

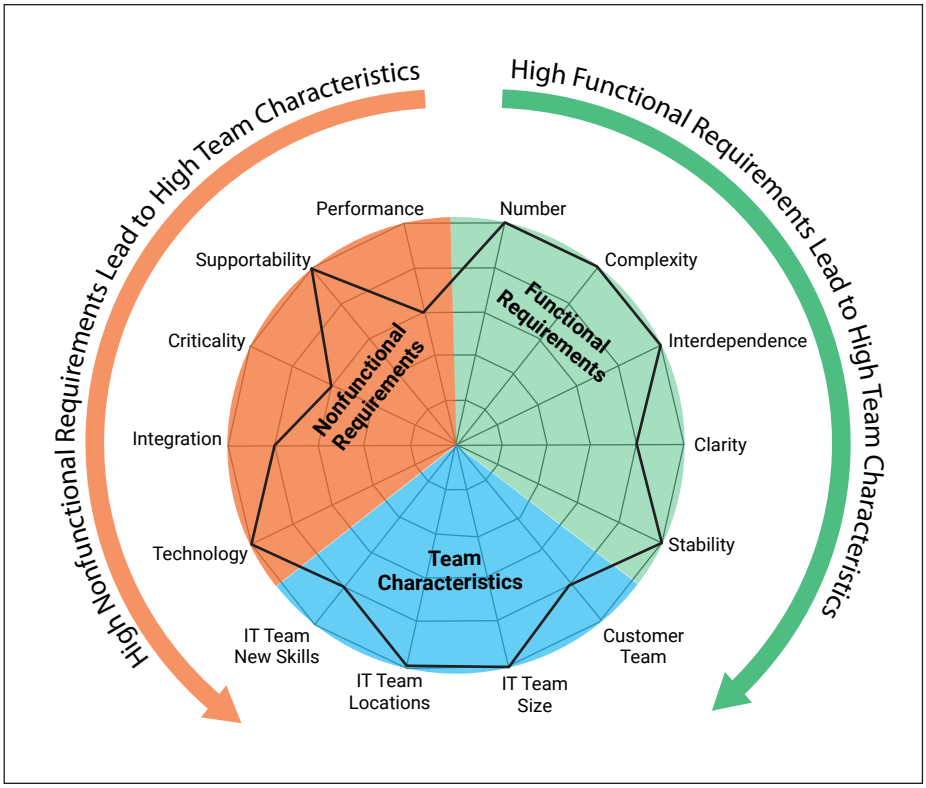


Figure 10-13 Effects on team characteristics as functional and nonfunctional requirements characteristics increase

In general, as introduced in Section 10.4, plotting at the outer edge of these team factors has two implications. First, high levels of BRUF and BDUF factors suggest high levels of team characteristics (see Figure 10-13).

For example, as the number and complexity of functional requirements increase, you'll tend to need bigger IT teams (IT Team Size), and bigger IT teams often end up being located in multiple locations (IT Team Locations). Similarly, large nonfunctional requirements often require using many specialized IT skill sets (high IT Team New Skills), which, in turn, may also increase IT Team Size and IT Team Locations.

These relationships can help determine how to create project teams of appropriate size and complexity to match functional and nonfunctional requirements. But assessing team characteristics in an existing team can help make sense of these relationships in the opposite direction. For example, if you're newly assigned to an existing team, take a look at the IT and client team characteristics: a large, highly specialized IT team suggests both high functional and high nonfunctional requirements (and, if not, then that team may need to be reorganized). Similarly, a large, diverse client team will often result in diverse requirements. Needing to support many users also tends to increase nonfunctional requirements (e.g., Performance).

10.5.2.3 Making Sense of Project Characteristics at a Glance

As noted previously, systems projects are highly complex, requiring the measurement of many factors to describe them and thereby select the best project approach. Fortunately, once these values are plotted, you can use a project characteristics radar chart to quickly characterize the project and determine the right approach to use. We illustrate this process using a series of minicase studies.

MINICASE 10.1: CREATING AN INTERNAL ANALYTICS APP TO ANALYZE PRESCRIPTION DRUG USE

Consider a situation in which a pharmacy company wants to create a data analytics application to analyze the prescription drugs it dispenses. This requires importing data from a single source: its own pharmacy administration module. Only a handful of pharmacy analysts will access the app, and they can do so within their own internal secure network, not via the public internet. You will use a database management system and data analytics tool that the company already licenses and understands. While the pharmacy analysts perform some complex analyses, the current project only needs to create the database and populate it with data. So, the number of features you need to create is small.

How do you plot this project on the radar chart? Figure 10-14 corresponds to this description. Here, you see the project plotting close to the center of the chart for nearly all factors. Only the nonfunctional factor "Criticality" reaches the medium level. This indicates the need to securely store sensitive healthcare data in the system. However, this doesn't reach a high or very high level because you don't need to expose the system for public access via the internet. The key takeaway here is that, at a glance, you can see this project is an excellent candidate for a highly agile project approach. In particular, you should minimize BRUF, and for BDUF, you should focus mostly on the Criticality factor.

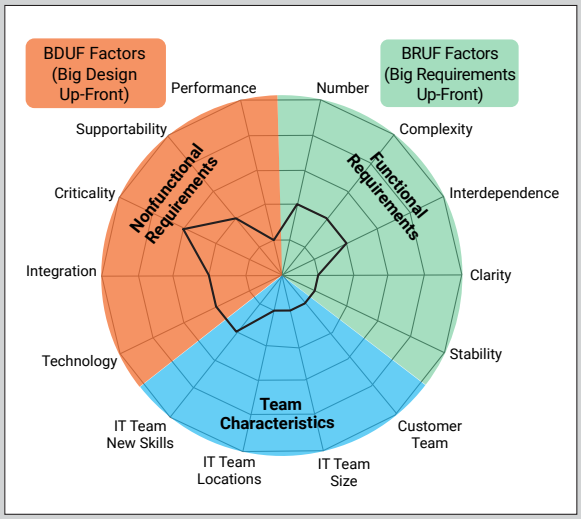


Figure 10-14 Characteristics suggesting a highly agile approach

MINICASE 10.3: MAJOR FEATURE UPDATES FOR AN INSURANCE SYSTEM

An insurer has a system it uses to process healthcare claims—for example, those sent in by a doctor's office or hospital. Processing a claim involves a series of interrelated, sequential steps: determining if the member is eligible, then checking to see if the medical service is covered, then determining the amount to pay for each service, and so on. The system performs quickly and reliably for the current 150 claims clerks who use it. The functionality is also generally satisfactory, and the requirements haven't changed much recently. However, the firm decides to buy another insurance company, which will add about 100 additional users. The acquired firm has had significant problems with its system, so the buying firm decides to retire that system and move the acquired business to its existing system. Performance testing shows that the existing system can easily expand to support the added load. However, analysis of the acquired firm's healthcare benefit plans shows that the existing system will need major new features to support additional ways to process these claims: calculating benefits, determining how much to pay healthcare providers, and so on. The acquired firm also sells life insurance, which is a functionality that will need to be added to the current system. Making this happen will require the existing client team to communicate thoroughly, with a significant number of new client team members from the acquired firm. This will make requirements discussions much broader than in the past.

How do you plot this project on the radar chart? Figure 10-16 plots this situation. The architecture and technology of the system appear to be satisfactory, so you may not need much BDUF. But you clearly need to implement a significant number of new, complex features. Given the way claims are processed in a series of related steps, interdependence is also likely to be high. Finally, if you involve both existing and new clients, you should be able to reach significant clarity about these requirements. Finally, the pace of change in these healthcare claims processing requirements has been fairly slow. Given these points, you should pursue a hybrid approach emphasizing heavy BRUF.

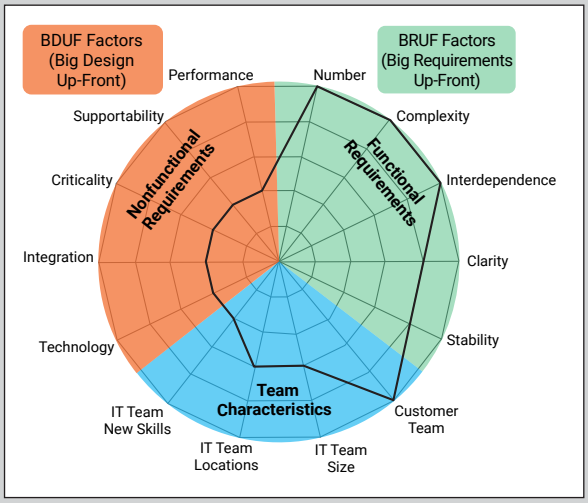


Figure 10-16 Characteristics suggesting a hybrid approach emphasizing heavy BRUF

MINICASE 10.2: A MAJOR UPDATE TO A LEGACY MANUFACTURING APP

Next, let's look at a firm that's been running its heavy equipment manufacturing business on a legacy application. Written in the 1980s in COBOL, the application consists of millions of lines of code that are poorly architected, designed, and documented. Put simply, when you try to modify the application in one functional area (like logistics), you risk inadvertently breaking functionality in another area (like assembly-line scheduling). This issue is compounded when the firm decides that, to stay competitive, it needs to create dozens of new major system features, including communicating with suppliers using web services, adding significant new types of data, revising major supply chain and scheduling algorithms, using artificial intelligence to optimize warehouse operations, and so on. In this case, the firm may very well decide that you need to drastically overhaul the underlying technology and, beyond that, add in and integrate several new technologies. This may be the only way to effectively add all this new functionality. The IT team might not be well positioned to make all this happen—for instance, the team could be too small and lack the needed skill sets. To bridge these gaps, you may need to hire IT contractors in multiple locations outside the main IT office. You may also need to involve several factory team members and outside consultants to enact the changes.

How do you plot this project on the radar chart? Figure 10-15 illustrates this situation. Here, every project factor is plotting near the outer edge of the chart, at a high or very high level. At a glance, you know you should use a hybrid approach with high levels of both BRUF for functional requirements and BDUF for architectural planning. Team factors pertaining to coordination, communication, and training also loom large.

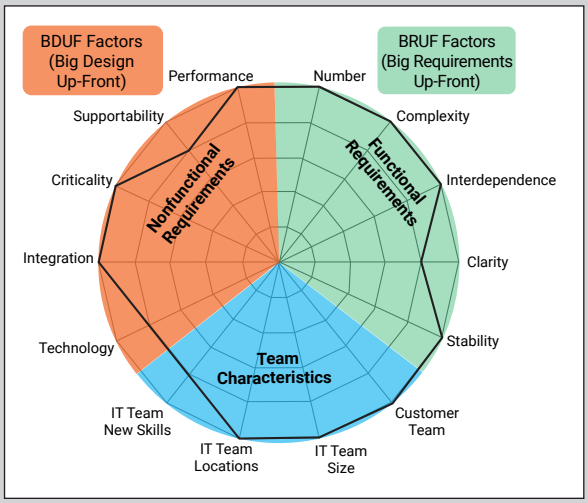


Figure 10-15 Characteristics suggesting a highly plan-driven approach

MINICASE 10.4: CREATING A NOVEL SOCIAL MEDIA PLATFORM

Finally, let's look at a start-up that wants to create a novel social media app: a dating app connecting outdoor lovers based on what they like to do (hiking, sports, picnicking) and where they like to do it (based on where they live). This functionality can leverage several existing third-party components, including geographic mapping services that can be utilized via web services. The remaining functionality is unlikely to be significantly complex, but it is not clear exactly how the app should work. The firm's leaders note that a competing firm may be pursuing a similar idea, so requirements are likely to change quickly as the firm works to one-up the competitor. The leaders also note that if this idea takes off, the number of users could skyrocket—possibly to hundreds of thousands of users. Finally, the app will acquire and store client data, including personal and payment information accessed via the public internet. The IT team consists of five developers, who are connected to a single product owner who is empowered to guide requirements and priorities.

How do you plot this project on the radar chart? Figure 10-17 shows a plot of this project: relatively low functional requirements, but with a demanding technical environment. That environment shows up alongside a need to develop IT Team New Skills. For example, the team may need to research consuming web services from a wide variety of third parties. The team also may need to work with an architect and a security specialist to be able to build a scalable, secure environment. This all points to substantial BDUF. On the other hand, for functional requirements, you can use agile, emergent requirements. For example, you can choose to prototype unclear requirements, which will likely evolve rapidly over time regardless.

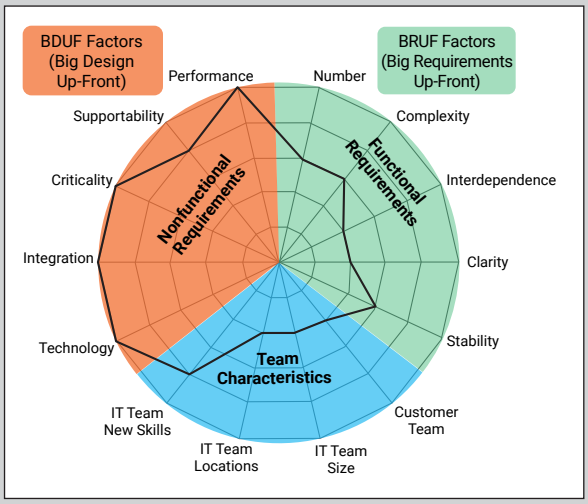


Figure 10-17 Project characteristics radar chart for a project needing high BDUF

10.5.3 Deeper Dive 10.3: Project Approach Selection for Artificial Intelligence (AI) Projects

Much of the discussion in this chapter has been implicitly focused on the conventional approach of developing (literally: programming) new features using a symbolic programming language like JavaScript, Python, or C#. Of course, today we see a major new trend: the rapid development of systems based on AI technologies. AI differs from conventional software in that, rather than being *programmed* using a symbolic language, it's *trained* using neural networks in conjunction with statistical models.

Recent research (Spurrier & Topi, 2023) suggests that the systems project approach for AI systems (or AI components of overall system solutions) differs significantly from that of conventional systems or components. As described in Chapter 8, the general process for creating AI systems or components includes, in roughly sequential order:

- **Problem formulation**
- **Data collection, assessment, and management**
- **Exploratory data analysis and visualization**
- **AI model selection**
- **AI model training**
- **Establishing trust in the model**
- **Integration and deployment**
- **Operation and maintenance**

The key point here is that the first three or four points can be pursued in a fairly plan-driven way, but after that, recent research (Spurrier & Topi, 2023) finds that the project approach to developing AI features works differently than for conventional features. Specifically, AI development tends to work in a "work for a day, evaluate progress, then plan tomorrow" manner. This process is needed because of the uncertainties of AI model development: you can't be sure if or how quickly your training data and AI model will generate the accurate results you need. It could happen quickly, it could take many revisions and a lot of time, or it might not end up working well enough at all. This results in an approach to AI development that is closer to Kanban (see Chapter 15) than to sprint-based agile development that operates in one- to four-week-long cycles. In this sense, AI development is more agile than agile development!

Given how rapidly AI-based development is growing and evolving, we can't know today exactly how the process will work in the years to come. But we can suggest that projects that contain AI components will likely need to use project approaches somewhat different (and probably more agile) from the overall projects they're embedded in.

10.5.4 Deeper Dive 10.4: Project Approach Selection for Data Analytics Projects

Somewhat like AI projects, data analytics projects (or data analytics components) tend to operate differently and in a more agile fashion than conventional, transaction processing-oriented software.

Specifically, IT teams typically do significant up-front planning for creating a data model and importing data sources. Still, then the actual use of the populated data for analytics tends to become much more ad hoc, often even without the use of agile user stories of the form "As a <user>, I want/need to <do a software function> to <achieve a business goal>." That's because the data analytics questions we ask tomorrow depend to an extent on the answers we get to today's questions. These types of projects tend to plan and execute their analytics work daily, without using fixed-cadence iterations or sprints. Here, traditional weeks-long sprints give way to an "even more organic" approach than traditional agile practices. Again, see Kanban in Chapter 15.

10.5.5 Deeper Dive 10.5: Project Approach Selection for Configuration Projects

Finally, we consider one other major alternative for implementing software features: configuration of third-party software, whether commercial off-the-shelf (COTS) products or open-source projects. Third-party software is generally highly flexible, thereby requiring complex configuration

decisions. However, because the software already exists, in comparison to software construction, the configuration of third-party software also generally involves less uncertainty and typically involves a repeatable series of implementation steps. Put plainly, the question "How do I configure this existing system for the twentieth time?" is easier to answer in a plan-driven way based on prior experience than "How do I construct new system features?" As such, configuration projects tend to be fairly plan-driven throughout, including implementation.

However, importantly, and perhaps counterintuitively, most configuration teams today do their work after initial visioning, business analysis, and project planning using sprints or similar iterative techniques. There are at least two reasons for this, both pertaining to using frequent client feedback to combat the "Yes, but" syndrome. First, as you execute configuration steps, there's a strong possibility you'll configure the software incorrectly, and subsequent implementation steps often depend on how you completed earlier implementation steps. Second, many large configuration projects include a degree of custom software development—for example, to create data interfaces to other systems, custom UI screens, reports, dashboards, and so on. By configuring and developing custom components in parallel using sprints, you again gain the advantage of aligning the configuration and construction components of the project.

Feasibility Analysis, Statement of Work, and Business Case

11.5 Deeper Dives: Advanced Topics on Project Planning and Approval

In this section, we do deeper dives on several advanced topics that build on the Fundamentals you learned earlier in the chapter:

- **Deeper Dive 11.1** builds on your understanding of risks and issues by adding two new, related concepts: assumptions and constraints. Like risks and issues, assumptions and constraints impact your ability to deliver your project. However, unlike risks and issues, assumptions and constraints are simply factors that you take to be true, not problems to address.
- **Deeper Dive 11.2** explains that change management involves the activities needed to prepare the organization to effectively implement and use the new system capabilities. While change management is typically mostly a client responsibility, it does often affect the IT team's process task responsibilities in the definition of done (DoD).
- **Deeper Dive 11.3** explores the unique risks associated with developing AI systems and components. In short, AI development is more uncertain than conventional development, leading to risks particular to the AI itself, as well as challenges in coordinating AI and conventional development work.

11.5.1 Deeper Dive 11.1: Assumptions and Constraints

Risks and issues are the most important items to identify, evaluate, and respond to during feasibility analysis. Identifying and evaluating those problems is key to determining overall project feasibility.

However, many teams will also identify and document two other kinds of items:

Assumption A current factor that you take to be true that specifically supports project success.

- **Assumptions:** An **assumption** is something you take to be true or certain that *supports* your project's success. Using the VFA case, assumptions might include:
  - *Mobile app doesn't need a tablet-specific version.* This assumption simplifies the work we'll do—we only need a version of the UI tailored to smartphones.
  - *Our product owner can spend 20 hours per week with the IT team during requirements and 10 hours per week during development.* A key principle of agile development is that the IT team can get frequent guidance and feedback from a key business client, typically a product owner. However, many clients are so busy with their "regular job" that they end up not being sufficiently available to the IT team. So this assumption supports our team's ability to develop our features with sufficient client input.

Constraint A current factor that you take to be true that limits your ability to deliver project success.

- **Constraints:** A **constraint** is something you take to be true or certain that *limits* your ability to deliver. Here are a couple of examples:
  - Again using VFA, here's an example constraint pertaining to team resources: *The backend retail system team will be busy with a major release of their own for the next month, so we won't be able to get any modifications to their APIs until after that.* Given that we likely will need API modifications, this constraint may delay our own deployment date.
  - Per Chapter 5, Section 5.2.3, a nonfunctional, project-level user story acceptance criterion may create a constraint pertaining to the functioning of the overall system: *System uptime should be at least 99.5% during normal business hours.*

You can list assumptions and constraints in a tabular format similar to what we did for risks and issues in Section 11.2. Table 11-8 lists the example assumptions and constraints just presented.

Table 11-8: Example assumptions and constraints section

ID	Description	Impact	Responsible
A1	Mobile app doesn't need a tablet-specific version	Simplifies UI development work	Sivaraman
A2	Product owner can spend 20 hours per week with IT team during requirements and 10 hours per week during development	Key to getting requirements and working software correct	Sarker
C1	Backend retail system team unavailable to make API mods for a month	May delay some development until Sprint 3	Frye

Here are two key points that apply to both assumptions and constraints:

- **There's nothing to do about them:** While assumptions and constraints both affect your ability to deliver the project, you simply assume they're true. This is unlike a risk (where you need to take mitigation action) or an issue (where you need to take action to solve this current problem).
- **Need to reevaluate project if they turn out not to be true:** For example, consider if during requirements analysis we learn that our product owner is being transferred, and no replacement for them has been identified. This violates assumption A2 in Table 11-8 and affects our ability to deliver the project as planned. This would likely turn into an issue like "Need to get a replacement product owner ASAP." Conversely, of course, if we find out that the retail team can make an API developer available to us now, then that could revise constraint C2, enabling us to accelerate the project.

11.5.2 Deeper Dive 11.2: Change Management for Systems Projects

As noted in Section 11.1.1, implementing new software features changes how an organization operates. As such, the organization needs to plan and execute specific **change management** actions to prepare itself for those changes.

Change management Series of tasks that must be completed for the organization to be able to successfully deploy a new or enhanced software system.

In Section 11.2.1, we focused on the importance of a DoD in defining the IT team's process tasks in general and coordinating with change management tasks in particular. But change management, including especially the client's tasks, is a big topic that's crucial to your project's overall success. Put plainly, many projects have delivered excellent software only to fail because the organization hadn't sufficiently prepared to implement the new system.

In this Deeper Dive, we explore two key change management dimensions mentioned earlier: policies and procedures (P&Ps) and user training. Elsewhere, we tackle two other key change management issues: user acceptance testing (in Chapter 5) and data preparation (in Chapter 16).

11.5.2.1 Change Management: Policies and Procedures

A new or updated system will typically change the way users work. Per our ValueForAll and Wayback Public Library examples, this can include automating tasks; providing new options for reporting, calculations, and workflows; reducing the need for manual checking of reports; and on and on. The key here is that most organizations adopt standard approaches—called policies and procedures, or P&Ps—for performing their work. P&Ps help ensure that an organization's products and services are delivered with consistent quality and efficiency. They also provide a sort of reference manual that trained users can employ to determine how to handle unusual situations, understand the meaning of a specific field, or remind themselves how to perform a certain infrequently performed task.

As the system processes change, so must the P&Ps. Updating the content of P&Ps normally falls to the business. However, IT needs to ensure that the P&Ps accurately correspond with the planned system changes. And while P&Ps may exist as stand-alone documents outside the system, they are increasingly integrated into the system itself as online help. When this is the case, IT may need to help the business implement those changes.

11.5.2.2 Change Management: Training Manuals

While P&Ps are the "reference manual," they often don't work well for training new staff members in using the system. P&Ps may not even work well for experienced users trying to learn how to use major new software features. The need here is training manuals: step-by-step tutorials that walk a user through typical system functions. As with P&Ps, they may be stand-alone documents or integrated into the system itself. It should go without saying that, as P&Ps are updated, training manuals should be updated too. Also, as with P&Ps, if training manuals are integrated into the system, IT may need to help.

Training manuals today are often deployed in the form of online tutorials, such as what you might find on YouTube or in similar environments.

11.5.3 Deeper Dive 11.3: Unique Risks of AI Projects

Consistent with other deeper dives we made on AI, these projects present unique risks and issues, both with regard to themselves and when they're part of an overall solution combining AI and conventional (programmed) software components. Let's consider AI-specific risks and issues in light of the property insurance company solution introduced in Chapter 2, Deeper Dive 2.6, and, in particular, the AI system component focused on categorizing shingles into "damaged by hail," "damaged by other (non-hail) causes," "worn out from age," "undamaged," and so on.

Let's start with AI-specific risks and issues that apply to an AI system (or component, as in our example) considered alone. For our example, risks and issues could include the following (for each one, we tie it to the corresponding feasibility dimension):

- **Unclear performance requirements (Technical—Functional Requirements dimension):** We've noted that AI systems are trained, not programmed, and that through that training we hope to achieve adequate performance. But what, exactly, do we mean by "adequate?" In our shingle categorization example, is 90% accuracy sufficient? 95%? 99%? 99.9%? Defining this up front is critically important because without it our team won't know if the AI component meets the need. Note, too, that this is critical for determining economic feasibility, and, in particular, estimating business benefits. While our approach has a human expert in the loop to check on the AI shingle categorizations, as accuracy declines, that level of human effort will increase, reducing business benefits. This underscores the need to clearly specify the required level of performance up front.
- **Required performance may be unattainable or take much more effort than expected (Economic dimension):** Even if you have a clear performance requirement, you can't know how long it'll take to train the model to achieve that performance, or even if that level of performance is attainable with your initial AI model and training. You might have to change the data or the model or both. This points to a risk that you can't count on your cost estimates for AI to be accurate. In fact, if the AI model turns out not to be sufficient even with substantial rework, you might find you have to substantially revise or even toss your overall solution design. All this points to increasing economic feasibility uncertainties when using AI.
- **Inability to source and integrate team members with AI development skill sets (Technical—Staffing and Skill Sets dimension):** The skill sets needed to develop an AI system or component are quite different from those needed for traditional programming. As a result, you'll likely need to find new team members and integrate them into your overall team. This kind of change is always risky. Given the shortage of AI-capable developers, you also run the risk that you won't even be able to find these skill sets.
- **Inability to explain and trust the model (Technical—System Architecture and Legal):** As we've noted before, AI systems often suffer from being "black boxes," where you can't clearly understand or explain how the system is reaching its answers. You can't

even count on them generating the same answer two times in a row, especially when the training data set is updated frequently. Although you sometimes can infer how the system does its work, you may have to fall back on being able to say that you can trust the system because of its level of accuracy. But if the system then fails in ways that create high costs for customers, you may find yourself with economic losses and even lawsuits from those customers.

- **Maintaining the model over time (Scheduling/Operational—Deployment):** As noted in the previous point, AI models may pull in new training data sets on an ongoing basis. For example, the property insurance company might update the shingle training data set as new types of shingles are introduced or in response to climate change–driven changes in shingle damage. The point here is that changing the training data set will also change the performance of the AI model. If we don't have ongoing oversight of these changes baked into our system operations and maintenance, we run the risk of the model's performance deteriorating without anyone being aware of it.

Beyond these items, when the AI system is a component within a broader solution, as is the case in the property insurance example, you face the risk that your overall project will run into problems:

- **Inability to coordinate work between AI and conventional software teams (Technical—Project Management):** As noted, AI system development is a more uncertain process than conventional software development. Rather than designing and creating conventional code, where you have clear expectations as to how it will function and can plan your development work in sprints, with AI you often evaluate the model's training progress at the end of each day and then plan work for the next day. As we'll cover in Chapter 15, because of this uncertainty, AI system development is difficult to plan in sprints, even one week into the future. As a result, it's often difficult to coordinate work between AI and conventional teams.

Over the coming years, the IT field will learn much more about how best to conduct AI projects and projects combining AI and conventional software components. But in the meantime, you need to be especially careful when assessing the feasibility of these kinds of projects.

Up-Front Project and Release Planning

12.5 Deeper Dives: Up-Front Project Management and Release Planning

In this section, we do deeper dives into project and release planning that build on and expand coverage from earlier in this chapter:

- **Deeper Dive 12.1** expands on the reasons traditional project management techniques tend to not work well for systems projects. Specifically, you'll learn how traditional project management techniques focus on managing replication risk, whereas in systems projects, the key issue to manage is design risk.
- **Deeper Dive 12.2** discusses issues pertaining to release planning for systems projects focused on the configuration of third-party software. You'll learn that, even though these projects are more predictable than systems projects focused on development of new software, most teams today still plan that development using sprints.

12.5.1 Deeper Dive 12.1: Why Traditional Project Management Fails for Systems Project—Replication Risk versus Design Risk

In Section 12.2, we discussed up-front project planning using Gantt-lite charts. This included a brief introduction to and contrast with traditional Gantt charts. You learned that traditional Gantt charts work well for high-predictability projects, like building multiple copies of a house from a blueprint, but not so well for systems projects, where the need to create and then develop a new design significantly reduces predictability.

Still, that leaves you with the need to somehow plan and structure the front end of the project—that is, all the activities in the Systems Development Process Framework that precede software development. You learned that Gantt-lite charts retain features of traditional Gantt charts that are useful in systems projects while dispensing with other features that assume a level of predictability and precision that are unrealistic in that context.

In this Deeper Dive, we explore these issues in more detail, including a specific example using the ValueForAll (VFA) case that illustrates the difficulties of using traditional Gantt charts in a systems project.

12.5.1.1 Systems Projects: Managing High Unpredictability

The concept of using Gantt charts to plan and manage a project like constructing a house seems reasonably simple and straightforward. Indeed, in the past, many systems projects have been (and sometimes still are) managed in this way. For that reason, if no other, you should familiarize yourself with Gantt charts.

However, in the age of options, a consensus has emerged that using classic project management techniques and tools to manage systems projects seldom works very well. The question is, why? Systems projects are, well, projects! There must be a reason why classic project management tools like traditional Gantt charts don't work well for systems projects. It all has to do with predictability, which is often high when building a house but usually low when building a system. Consider a house builder who's constructing houses in a suburban neighborhood. Suppose this builder offers six house plans, all using the same basic blueprints, materials, and construction techniques. Their construction crews are experienced, having built dozens of these kinds of houses before. Individual houses built might vary in their details: specific cabinets installed, color of paint, wood flooring versus carpeting, and so on. But the vast majority of one house built using blueprint #1 will be the same as the next house built using blueprint #1.

Given this consistency, the builder, over time, can create accurate measurements of how much time and effort each construction task requires. By using these measurements in project plans for creating new houses using these same blueprints, the builder can accurately predict how the project should go. They can also quickly realize when something is amiss—for example, a three-day task still being incomplete after four days.

These are the attributes of a high-predictability project. What's more, the risk being managed is not about the *design* of the house. That's already specified in detail in the existing blueprints. Rather, the risk being managed is creating—really *replicating*—the same house design over and over. This illustrates the idea of **replication risk**.

Replication risk The risk that a product based on a well-understood design can't be created again with sufficient quality, cost, and timing.

Many kinds of projects focus on replication: building an airplane, assembling a car, or any task that replicates a fixed design. In fact, as you make more and more of these copies, you begin to transition away from project management into operations management. For example, a restaurant makes (replicates) a very high number of copies of menu items for its customers—individual servings of food using blueprints in the form of recipes. They make so many copies that they cease thinking about each of them as individual projects.

12.5.1.2 Systems Development as a Low-Predictability Project Managing Design Risk

Let's contrast the example of house construction with systems projects. In systems, you don't really contend with replication risk; you can make as many copies of data or software as you want. Each copy will be perfect and cost almost nothing to produce. To put it another way, if you have a software application that meets your needs, you don't need a systems project to create it—you just need to acquire a server and copy the software onto it!

Instead, systems projects focus on creating something unique and new. Again, this is more like creating a new blueprint, prior to constructing a building from that blueprint. Rather than focusing on replication risk, here, you focus more on **design risk**. This is the risk that your creation is incomplete (i.e., doesn't provide the features you need) or incorrect (i.e., contains design flaws or defects).

Design risk The risk that new or enhanced product features will be incomplete or incorrect. This includes what the system must do and how the system must do it.

This is a less certain enterprise than house building. It involves making sense of a complex problem domain, including the current state and future state of data, logic, and how users will interact with them. This is the exact summary of requirements analysis and design topics covered in this book!

Now, imagine what it would mean to try to plan out the business analysis process in detail. Figure 12-9 is a traditional Gantt chart showing some of the task details for business analysis of

ID	*Task Name*	*Start*	*Finish*	*Time*
35	**Analysis: Customer Account Management (CAM)**	1/27/2025	2/3/2025	6d
36	Meet with CAM SMEs	1/27/2025	1/27/2025	1d
37	Draft requirements 1	1/28/2025	1/29/2025	2d
38	Draft requirements 2	1/31/2025	1/31/2025	1d
39	Review/approve	2/3/2025	2/3/2025	1d
40	**Analysis: Self-Register (SR)**	**1/30/2025**	**2/6/2025**	**6d**
41	Meet with SR SMEs	1/30/2025	1/30/2025	1d
42	Draft requirements	2/4/2025	2/5/2025	2d
43	Review/approve	2/6/2025	2/6/2025	1d

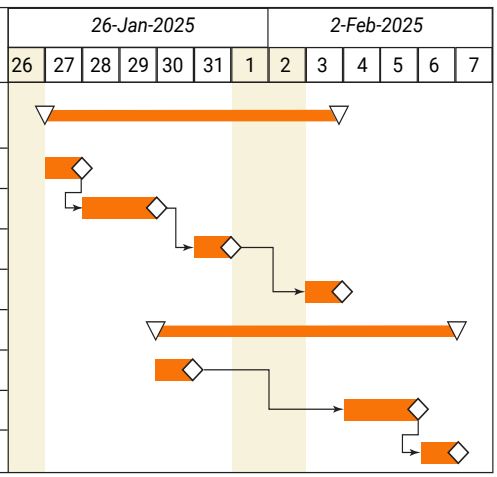


Figure 12-9 Classic Gantt chart for VFA project showing a small subset of business analysis tasks and dependencies

the VFA project. We show supertasks for two of the epics: "Analysis: Customer Account Management (CAM)" (Task 35) and "Analysis: Self-Register (SR)" (Task 40). (This is only a fragment of the overall Gantt chart, but we zoom in on these selected tasks here for the sake of illustration.) Now we try to plan tasks using the traditional Gantt chart. Starting with CAM, we think we can meet with those subject matter experts (SMEs) on January 27. We estimate that this might take one day. In fact, we probably can't meet with them for more than a few hours because they have regular operations to manage as well. In truth, we really don't know how complicated and time-consuming this will be. It could take an hour, or it could take multiple days. Still, we try our best to plan ahead.

Next, we plan the task of writing the requirements. Again, we don't know how complicated they'll turn out to be, but we estimate three days. But then things get even more complicated. It turns out that the SR SMEs in Task 41 can only meet with us on Thursday, January 30. This forces us to split Task 37 into two pieces, fitting that writing process around Task 41. Again, all of these tasks are uncertain in terms of their respective durations because we can't really understand their complexity until we're in the middle of the task.

Let's get even more realistic. We carve out time on January 30 to meet with the SR SMEs, but at the last minute, something comes up, and they say they can't meet for another week. (They can't tell us an exact date yet!)

At this point, you should begin to realize that planning a systems project is a pretty unpredictable affair! If you really try to use detailed Gantt charts to plan your work in detail, with specific dates and mathematical precision, you'll find yourself constantly planning and replanning (and replanning and replanning). Pretty soon, the whole planning effort tends to collapse. Rather than helping you get your work done, you may find the (re)planning work really just gets in the way of accomplishing the substance of your project. This is one key reason why advocates of the agile approach argue that plan-driven techniques are a waste of time. The idea of continually revisiting and revising a Gantt chart, rather than focusing on creating working software, can be frustrating, to say the least (Figure 12-10).

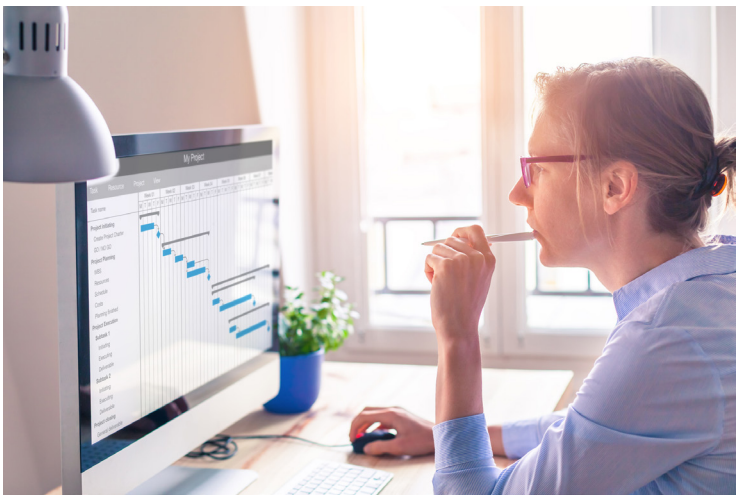


Figure 12-10 A BA replanning (and replanning) project tasks in a Gantt chart, rather than focusing on creating working software (iStock.com/Kwangmoozaa)

So far, we've limited our discussion to up-front project tasks conducted prior to starting development. But the problem is just as bad, if not worse, during software development. Specifically, because of the clear superiority of iterative, sprint-based development, you can't—and, in fact, shouldn't—try to plan out software development tasks in detail. Instead, you need to create flexible, agile release plans for development and testing. We covered agile release planning earlier in this chapter. We'll cover flexibly managing and updating those plans in Chapter 15, where we explore how to lead development sprints.

12.5.2 Deeper Dive 12.2: Release Planning for Configuration Projects

This Deeper Dive builds on Deeper Dive 10.5, where you considered systems projects focused on configuring third-party software.

Both the VFA and Wayback Public Library cases focus on software construction, not configuration of third-party software. But what about projects that primarily focus on selecting and configuring third-party software? This includes using different pre-development tasks, especially ones focused on soliciting information and bids from third-party software vendors. We covered these topics in Chapter 7.

Beyond that, though, you need to consider how you plan and execute development work once you've chosen the third-party software. We previously noted in Deeper Dive 10.5 that third-party software, whether a commercial off-the-shelf (COTS) product or an open-source project, is generally highly flexible and so requires a series of complex configuration decisions. However, in comparison to software construction, the configuration of third-party software also involves less uncertainty because it uses a repeatable series of configuration steps. Put plainly, the question "How do you configure this existing system for the twentieth time?" is easier to answer based on prior experience than "How do you design new system features that you'll construct with new code?" As such, tying into the discussion above in this chapter, configuration projects fall between the high predictability of building a copy of a house from an existing blueprint and the low predictability of creating and constructing a brand-new system design.

Still, even with a repeatable series of steps, most configuration teams today do their development work using sprints. There are two key reasons for this. First, as you execute configuration steps, there's a strong possibility you'll configure the software incorrectly, and subsequent implementation steps often depend on how you completed earlier implementation steps. By using sprints, you can gather frequent client feedback, helping you make course corrections, create a solid foundation for later implementation steps, and avoid the "Yes, but" syndrome. Second, many large configuration projects include a significant level of custom software development. The team members doing this development work will likely also be working in sprints. So, by doing configuration and development tasks in parallel using parallel sprints, you can ensure that the two teams stay in sync with each other.

System Architecture

13.5 Deeper Dives: Advanced Topics on Architecture

In this section, we do deeper dives on advanced topics that build on and expand coverage from earlier in this chapter:

- **Deeper Dive 13.1** introduces and provides guidance in the use of the Zachman Framework, a widely used mechanism for classifying and organizing enterprise architecture documents.
- **Deeper Dive 13.2** discusses additional, alternative approaches to specifying and structuring organizational architecture models. It presents Kotusev's Enterprise Architecture Model as an interesting alternative to TOGAF and the Zachman Framework.
- **Deeper Dive 13.3** describes layered models as a frequently used mechanism for analyzing and structuring data network technologies, focusing on the seven-layer OSI model and the four-layer TCP/IP model as the most popular examples.
- **Deeper Dive 13.4** discusses key types of organizational network elements.

13.5.1 Deeper Dive 13.1: The Zachman Framework

Zachman Framework A well-known mechanism for organizing enterprise architecture artifacts and the taxonomy of such artifacts.

The oldest enterprise architecture tool is the Zachman Framework for Enterprise Architectures (Zachman, 1987; Sowa & Zachman, 1992), otherwise known as the **Zachman Framework**. Some analysts (such as Sessions, 2007) consider the Zachman Framework to be more a taxonomy than a framework because it's primarily a way to classify and organize enterprise architecture documents. Whatever label it might have, the Zachman Framework has been highly influential in guiding IS/IT professionals' thinking regarding enterprise architecture and, in general, recognizing the importance of high-level architectural thinking.

The main structuring tool of the Zachman Framework is a six-by-five matrix in which the two dimensions are the IT team actor role and perspective (such as planner, owner, designer, builder, and subcontractor) and descriptive focus area of the overall architectural model (data, function, network, people, time, and motivation; Figure 13-15 shows the framework with sample artifact descriptions). These two dimensions form 30 cells, and the idea is that each architectural artifact (model/document) has a focus that allows it to be classified into one of the cells. Both dimensions of the matrix capture important ways of thinking regarding the process of modeling systems in an organizational context.

The *actor role perspective* (vertical) dimension points out the importance of analyzing and documenting a system at multiple levels of abstraction and reminds us that observing a target from multiple perspectives provides valuable new information that will easily be lost if the perspectives are condensed together. In systems development, a business leader who is considering a new approach to doing business enabled by information systems capabilities should look at the system with very different questions in mind compared to a software developer whose responsibility it is to, say, optimize an algorithm to improve the quality of the recommendations the system makes and decrease any delays until those recommendations are available. Both perspectives are very important, and, in general, the communication between different levels of abstraction must work well. The actor roles in the Zachman Framework are as follows:

- *Planner* refers to a high-level decision-maker (e.g., executive or investor) who wants to capture an abstract model regarding the overall scope of the system.

Model: Perspective:	Data (What)	Function (How)	Network (Where)	People (Who)	Time (When)	Motivation (Why)
Scope *Planner*	List of things important to the business	List of business processes	List of business locations	List of key user groups	General model of timing of business events	Organizational vision and mission
Enterprise Model *Owner*	EER Domain model	Activity diagram	High-level conceptual model of the network	List of user roles (personas)	Volume of key transactions by hour	High-level objectives and strategies to achieve them
System Model *Designer*	Relational model	Use case; system sequence diagram	High-level architectural design of the network	Detailed persona and deliverable descriptions	Model for managing simultaneous transactions	Specific business goals and rules for achieving them
Technology Model *Builder*	SQL implementation	Design sequence diagram; design class diagram	Detailed network design diagram	Technical description of persona UX expectations	Management of timing requirements between transactions	System module outcomes and processing rules
Component Model *Subcontractor*	Segment of SQL implementation	Design sequence diagram with algorithm specifications	Detailed design of a network component	Detailed technical description of UX expectations of a persona	Implementation of timing at a single-module level	Module-level outcomes and processing rules

Figure 13-15 Structure of the Zachman Framework (Sowa & Zachman, 1992; content specifics generated by the authors of this textbook)

- *Owner* also refers to a role interested in the effect on the business, but, in this case, internally, from the perspective of somebody who "will have to live with [the system] in the daily routines of business" (Sowa & Zachman, 1992, p. 592).
- *Designer* refers to systems/business analyst types of roles: professionals whose focus is on the specification of system functionality and the data it maintains.
- *Builder* is a role whose perspective is driven by the need to construct a functioning system.
- *Subcontractors* build parts of the system based on detailed specifications.

The *focus area of the architectural model* (horizontal) dimension essentially provides a list of categories that need to be considered in separate systems development projects and in the organization's development work in general:

- *Data* (what) includes questions regarding the concepts that are important for the enterprise and their relationships.
- *Function* (how) focuses on the processes with which the enterprise achieves its goals.
- *Network* (where) describes the locations in which the enterprise operates (both physically and virtually).
- *People* (who) addresses the organizational roles and units that are contextually relevant.
- *Time* (when) covers the documents that address the timing of business events and their dependencies.
- *Motivation* (why) describes the reasons why system actions are necessary.

To summarize, the primary use of the Zachman Framework is to categorize work on enterprise architecture, help identify gaps in coverage, ensure consistency between different viewpoints, and support the analysis of existing architectural work. From the beginning, Zachman has emphasized the importance of the role of the framework in ensuring that organizational IT solutions are aligned with business needs.

13.5.2 Deeper Dive 13.2: Alternative Perspectives on Enterprise Architecture

Not all experts agree regarding the value of and need for comprehensive architecture models. For example, Sessions (2007, p. 2) gives the following recommendation: "For many enterprises, none of these [existing] methodologies will, so, be a complete solution. For such organizations, [we propose] another approach, one that might be called a blended methodology. Choose bits and pieces from each of these methodologies and modify and merge them according to the specific needs of your organization." In many ways, this recommendation is fully consistent with the intelligent modeling philosophy we advocate for throughout this book. In the same way as with individual modeling artifacts within a systems development project, in architectural modeling, the organization should consider which perspectives on modeling truly add value and focus on those. The models themselves are useless unless they will lead to better systems outcomes over time.

Some go even further. Kotusev (2017) directly criticizes the leading enterprise architecture approaches and suggests that none of the comprehensive methodologies are widely used in organizations. He proposes an alternative that also fits well with the philosophy underlying this book. His views on enterprise architecture (including its components) are also important to consider: he recognizes that organizational practices vary significantly and that different organizations see value in different models. He doesn't question the value of the time spent on considering architectural-level questions. Instead, he believes, based on his analysis of enterprise architecture development practices, that organizational needs vary significantly. He suggests that companies would be better off focusing on six categories of EA artifacts, divided into rules, structures, and change requirements as follows:

- *Considerations* cover *global conceptual rules* that are important for business and so also relevant for IT. They help the organization achieve a general agreement regarding its general values, goals, and direction. The most important considerations documents are descriptions of essential *principles* that cover the entire organization.
- *Standards* are the *IT-focused rules* that are needed to achieve technical consistency. The most important standard types are *technology reference models* (rules and recommendations regarding the selection of specific technologies) and *guidelines* (narrow technical guidance regarding specific issues).
- *Visions* are various types of business-focused *structures*, such as business activity models, process maps, and other such documents that help ensure compatibility between business needs and IT capabilities. The most important visions documents are *business capability models* (describing business capabilities in a hierarchical structure) and *road maps* (long-term business-focused plans for IT investments).
- *Landscapes* are high-level *structural* descriptions of an organization's technical IT capabilities, including those describing IT asset life cycles. The most important documents in this category are *landscape diagrams*, which show the connections between various components (such as applications, data stores, platforms, and so on) of the enterprise-level IT landscape.
- *Outlines* include various business-focused *change* documents, which analyze the overall business value of IT initiatives. The most important outlines documents are *solution overviews*, which provide brief business-focused overviews of IT projects.
- *Designs* are, in turn, technically focused *change* models that help implement the business change described in the outlines documents. The design documents are typically diagrams of data, applications, and infrastructure at a high level of abstraction, often using modeling approaches described in Chapters 2, 7, and 14.

Considerations, visions, and outlines serve as interfaces between business and IT, whereas standards, landscapes, and designs are internal IT tools mostly invisible to business.

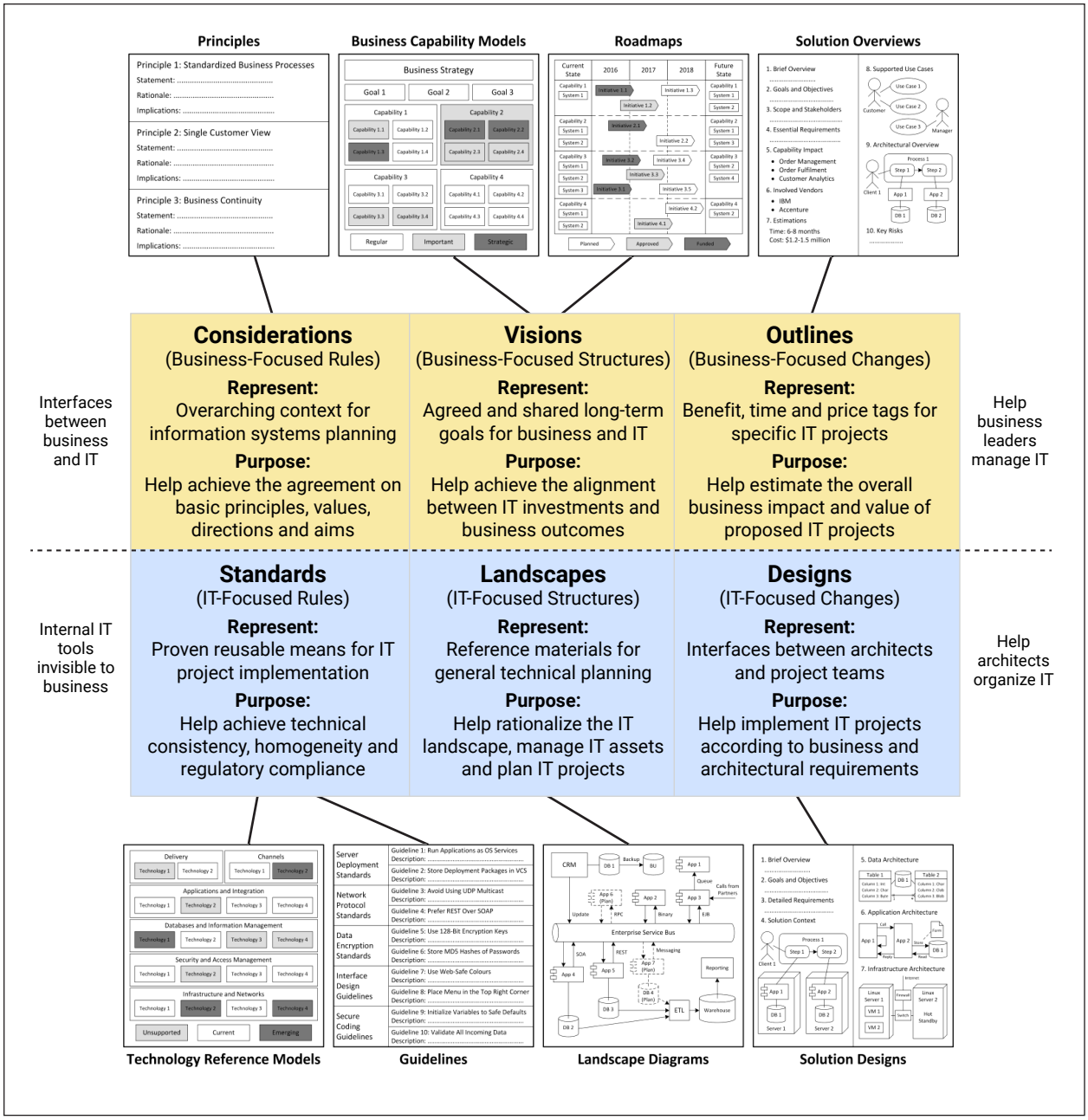


Figure 13-16 Kotusev's Enterprise Architecture Model (Kotusev, 2017)

Kotusev's model is carefully considered, based on systematic research, and forms a promising foundation for future work in this area. With that in mind, we've included its key characteristics in Figure 13-16.

13.5.3 Deeper Dive 13.3: Layered Models for Networking

A key term for understanding data communication technologies is a **protocol**. Protocols are foundational rules for communication that specify how two or more computing agents (applications or devices) should act in communication with each other. Protocols provide the other party information

Protocol Foundational rules of communication between two or more computing agents (applications or devices).

about how a currently communicating party is behaving and will behave and listen to others. As you'll see, plenty of protocols are used for multiple different purposes.

It's typical to assess network technologies by organizing them into layers. The two most common models are the seven-layer OSI model and the four-layer TCP/IP model. Fortunately, Layers 2, 3, 4, and 7 in the OSI model align with Layers 1 through 4 of the TCP/IP model. Let's look at all seven layers:

- Layer 1 is the *physical layer*, which specifies what media is used for carrying the signals from one networked device to another, how the signals are encoded on the physical medium and encoded by the receiving device, how the signals are actually transmitted on the medium, and what the physical topology of the network looks like. It's formally not part of the TCP/IP model.
- Layer 2 is the *data link layer* (OSI) or network interface layer (TCP/IP, Layer 1), which governs the use of the physical layer in a specific context. It has two sublayers, logical link control (LLC) and media access control (MAC). The latter term might be familiar from the term *MAC address*, a unique address assigned to a hardware device that identifies it in the context of a specific network. In general, the MAC layer controls the use of the physical layer medium and prevents clashes. The LLC layer specifies how devices on a network establish logical links through which they can communicate with each other within a network.
- Layer 3 is the *network layer* (OSI) or internet layer (TCP/IP, Layer 2), which specifies rules that allow communication across network boundaries. This does, however, require that all devices that communicate via connected networks share the same addressing space. By far the most common set of rules for the network layer is the Internet Protocol, or IP. The most important role of the network layer is to move packets of data from one device at a specific IP address to another at a different IP address (often over dozens of interconnected networks).
- Layer 4 is the *transport layer* (OSI and TCP/IP, Layer 3). While the network layer focuses on moving packets from one device to another, the transport layer establishes end-to-end connections between applications at the connected devices (such as a web browser on a laptop and a web server to which the user wants to connect). It uses the services of the lower layers to achieve this task. Many rule sets (protocols) at this layer provide mechanisms for reliability and flow control. The most important transport layer protocol is called Transmission Control Protocol (TCP)
- Layer 5 is the *session layer* (OSI), and, as the name states, it specifies ways for connected devices to create sessions for structured communication.
- Layer 6 is the *presentation layer* (OSI), and its task is to provide meaningful translations between different device types (e.g., between languages, encryption schemes, or character systems).
- Layer 7, or the *application layer* (OSI and TCP/IP, Layer 4), is the layer at which the users of the network services reside. This layer also comes with some familiar terms, such as the Hypertext Transfer Protocol (HTTP) for the web, Simple Mail Transfer Protocol (SMTP) for email, and File Transfer Protocol (FTP) for file transfer. As you can see, these elements are not your typical end-user applications; they're protocols that user-oriented applications can incorporate and use for communication between each other.

In the next section, you'll see how layered models and protocols can be used to understand real-world networks. Figure 13-17 illustrates how a wireless client uses the services of various layers on several network devices to reach a server destination; the details of the diagram are explained in the next section.

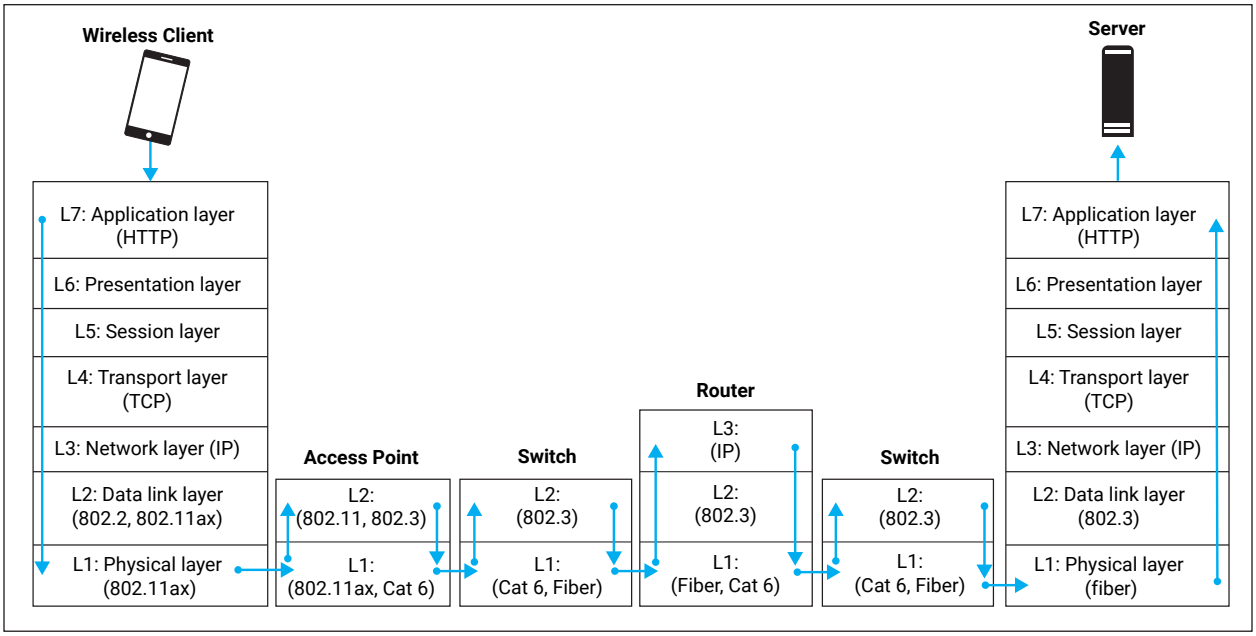


Figure 13-17 Schematic representation of a layered model

13.5.4 Deeper Dive 13.4: Network Types

Building on Deeper Dive 13.3, we'll explore in this section the most important elements of organizational networks—critically important infrastructure components without which no modern organization can operate.

The simplest context for data communication is likely to be a single **local area network** (LAN). LANs can range broadly in size and complexity, from small home networks to large corporate campuses, but many of the principles remain the same. A LAN is a network that's geographically located within a limited area and typically serves a known individual or organization (or set of organizations). Computing devices can connect to a LAN in two primary ways: through a wired connection or wirelessly.

Local area network A computing network located within a limited geographic area.

For decades, the primary technology family used to provide *wired network connectivity* has been a standard called Ethernet (IEEE 802.3). In a typical Ethernet LAN, computing devices with wired connectivity (servers, workstations, desktops, and sometimes laptops) are connected to a networking device called a **switch** at the *access layer*. Access layer switches can, in turn, be connected using *distribution layer* switches, and in large networks, those can be linked using *core layer* switches (see Figure 13-5 for an example of access and core layer switches). At each layer, the switches need to be faster than at the lower layers to avoid forming a bottleneck. Ethernet specifies Layers 1 and 2: both the options for the physical medium and its use and the rules that connected devices will use within a single network. Switches operate within one IP network.

Switch A Layer 2 network device that provides connectivity to clients, servers, or other network devices within one IP network.

Another widely used set of technologies combining Layer 1 and Layer 2 is another IEEE 802 family—this time IEEE 802.11 (or Wi-Fi). By far, this is the most widely used LAN standard for *wireless connectivity*. The nominal speed of Wi-Fi networks has been gradually increasing, starting from 802.11b's 11 Mbps (millions of bits per second) and going up to the latest widely used standard's (802.11ax or Wi-Fi 6) 9.61 Gbps (billions of bits per second; about 900 times faster than 802.11b). The next standard (802.11be or Wi-Fi 7) will soon be formally adopted, and compatible devices exist already, promising capacity up to 46 Gbps (4,000 times more than 802.11b about 20 years ago). The increased speed and reliability of wireless networks are creating very significant new opportunities, not only for streaming your latest 8K movie without any

interruptions but, more importantly, for enabling entirely new technology use categories (such as IoT) requiring that thousands and thousands of devices share significant amounts of data with each other.

Router A Layer 3 network device that connects IP networks and routes traffic between them.

When two separate LAN networks need to be connected (while still keeping them separate), we need Layer 3 capabilities, which are provided by devices called **routers**. Routers provide IP addressing and routing services and allow packets to cross network boundaries. By far the best-known use of routers is to provide connectivity to the public internet, but it's not the only one. There are many examples where multiple private networks are connected through routing.

Campus network A collection of local area networks on an organization's campus.

Campus networks are private networks that link local area networks of the same organization to each other—connecting, for example, buildings or floors of buildings. Campus networks typically involve the use of routers at their core, but they can still be hierarchical. Connecting links between component networks are typically wired.

Wide-area network (WAN) A network that connects other networks over long distances.

Metropolitan-area networks (MANs) and **wide-area networks (WANs)** are networks that provide connectivity across longer distances and organizational boundaries. The use of wide-area networks (which we'll use as a common term for both) is essential for providing connectivity for modern distributed organizations. Many different types of organizations need WAN connectivity, such as:

- A professional service firm that has local offices in dozens of cities
- A manufacturing firm with headquarters, manufacturing facilities, and many sales offices
- A multicampus university or a hospital system
- A technology firm with dozens of research and development facilities in several countries and manufacturing in another set of countries

From the perspective of a distributed enterprise, the purpose of a WAN is to connect all locations of the organization with a seamless shared network that enables connectivity between all units of the organization. The two primary current options for creating a WAN are as follows:

- Option 1: Renting private network capacity from a network service provider that provides various guarantees regarding the quality of the network service using service-level agreements (SLAs). A commonly used technology for implementing a WAN is called MPLS (multiprotocol label switching). The benefits of this option are vendor-guaranteed security and SLA-based capacity commitments; however, these benefits come at a high cost. According to Gartner, a hub and spokes MPLS network architecture is still the most common WAN model. (Figure 13-5 included connectivity between the headquarters and the branch offices using a hub and spokes model—the headquarters as the hub and the branch offices as spokes.)
- Option 2: Using **virtual private networks (VPNs)** that are implemented using the public internet. The idea here is quite straightforward: the company pays an established monthly fee to an internet service provider at each of its locations and uses the connectivity across the public internet to connect its locations using VPN technology. The downside of this option for corporate use is that using the public internet as the connecting technology doesn't provide any guarantees regarding the quality of service. Although this option is cheaper, it might not be acceptable for mission-critical services.

Virtual private network (VPN) A software-enabled private communication channel implemented over a public network.

It's not surprising that an approach combining the benefits of these foundational models has been introduced. It's called a **software-defined wide-area network (SD-WAN)**, and it's a WAN model that not only recognizes the need to integrate MPLS and internet-based WANs, but also acknowledges the increasingly wide use of cloud-based services, which we'll discuss in more detail later in this chapter. SD-WAN services integrate private WAN capacity with the public internet, allocating traffic intelligently, depending on the needs of the application. Cisco, a leading network equipment provider, outlines the benefits of SD-WAN as improved application experience, integrated and improved security, an optimized approach for the cloud, and simplified operations.

Software-defined wide-area network (SD-WAN) A network that combines private (MPLS) and public (internet) networks into a seamless network service.

There's one more network type we haven't discussed yet: **mobile networks** based on cellular technologies (frequently labeled with terms such as 3G, 4G, 4G LTE, and 5G, referring to technology generations, with later generations providing higher speeds). These are the access technologies that, in their newest forms, provide theoretical speeds of up to 10 Gbps, and actual speeds of several Gbps in practical applications. Widely available 100–150 Mbps speeds are sufficient for many users and uses. They enable truly ubiquitous connectivity that's not dependent on any specific local area network or its connections to the public internet.

Mobile network Public network services that provide client connectivity using cellular technologies, such as 4G, LTE, or 5G.

Mobile networks are provided by wireless operators that sell data communication capacity to companies and private consumers. Every reader of this book has likely used such capacity. Mobile network operators are often structured so that individual countries serve as their base service areas, but in many parts of the world, much broader access to data connectivity services has become available at a very reasonable cost (for example, free roaming is now mandated within the European Union). In addition to smartphones, many other ways to connect to the public internet exist. They, in turn, provide access to a variety of IT resources. Other mobile devices, such as tablets and sometimes also laptops, frequently accept connectivity-enabling SIM cards. Connected mobile devices frequently provide Wi-Fi access via a mobile hotspot.

14.5 Deeper Dives: Advanced Topics in Technical Design and Implementation

In this section, we do deeper dives into advanced topics that build on and expand coverage from earlier in this chapter:

- **Deeper Dive 14.1** describes normalization as a mechanism for organizing a group of related data items into a set of well-structured tables and relationships.
- **Deeper Dive 14.2** outlines the physical database design actions that are required before a relational data model can be implemented with a database management system.
- **Deeper Dive 14.3** discusses the use of relational databases in the context of applications based on the object-oriented model.
- **Deeper Dive 14.4** demonstrates the use of UML sequence diagrams for modeling an application's internal design structure.
- **Deeper Dive 14.5** describes how a design sequence diagram is used to identify behaviors that classes in object-oriented design require.
- **Deeper Dive 14.6** discusses multiple commonly used design patterns and frameworks.

14.5.1 Deeper Dive 14.1: Normalization

When you use the process described in Section 14.2.1.2 to convert a conceptual data model into a relational data model, you'll end up with a relational model that's well structured and avoids structural problems that could compromise the integrity of the relational database. Put plainly, the process will automatically lead to a good result.

But is there a formal way to evaluate the quality of an existing relational logical model? What if you don't have a conceptual data model but, instead, just a set of data items that need to be structured into a set of tables?

Normalization A formal process for organizing a group of related data items into well-structured relations.

Fortunately, there's a good solution: the process of **normalization** specifies a mechanism that can be used to verify the quality of the existing tables or to organize a group of related data items into a set of well-structured tables.

Functional dependency A dependency relationship between two data items that exists between X and Y if there's exactly one value of Y for any value of X.

Normalization requires that you first understand specific relationships between the data items, called **functional dependencies**. Any attribute Y is functionally dependent on an attribute X if, and only if, there's exactly one value of Y for any value of X. For example, motor vehicles have unique vehicle identification numbers or VINs. For each value of VIN, there's exactly one value of the vehicle's make, model, model year, manufacturing completion date, original sale date, and so on. So, make, model, model year, and so on are all functionally dependent on a specific VIN. In the same way, most countries have a national personal identification number that is unique for an individual (e.g., the US Social Security number or the UK National Insurance number). For such a number, there's only one (current) first name, middle initial, last name, birthdate, primary address, and so on, so the attributes are functionally dependent on the identification number. These dependencies are notated as follows:

VIN → Make, Model, ModelYear, ManufCompletionDate, OrigSalesDate

IDNumber → FirstName, MiddleInitial, LastName, Birthdate, PrimaryAddress

Functional dependencies are neither good nor bad—they're just inherent characteristics of the attributes that describe how the attribute values are related.

When you explore a set of data items where the items are related to each other, you can typically

determine how they're functionally dependent on each other. For example, if we explore our retail store context, we might find the following (significantly simplified) set of data items:

ProductID, ProductName, ProductCost, ProductPrice, SaleNbr, SaleDate, SaleTime, SaleLineItemProductID, SaleLineItemSaleNbr, SaleLineItemPrice, SaleLineItemQty, SaleCashierID, CashierName

A closer review of this set would reveal the following functional dependencies:

ProductID → ProductName, ProductCost, ProductPrice

SaleNbr → SaleDate, SaleTime, SaleCashierID

SaleCashierID → CashierName

SaleLineItemProductID, SaleLineItemSaleNbr → SaleLineItemPrice, SaleLineItemQty

Based on these functional dependencies, we can determine a way to structure this set of attributes into logical tables. This process is called normalization, and it will lead us from a messy set of data items into a well-defined set of tables through so-called **normal forms**: first (1NF), second (2NF), and third (3NF) normal forms. (There are others, but, as a practical matter, we care only about the first three.)

Normal form One of the steps in the process toward a well-structured set of relations using normalization.

Achieving the first normal form requires the data items to be organized into a table. What does that take? At a minimum, it requires three things: (1) the set of data items needs to have a name; (2) the set of data items needs to have a primary key that uniquely identifies each row of values; and (3) there need to be no rows on which a column includes multiple values.

In this case, we need to give the attribute set a name and select a primary key. What do these data items actually convey? Fundamentally, each row in this data set is a sales line item, and they can be separated from each other by the joint value of product ID and sale number. So we have the following first normal form table:

SALELINEITEM(ProductID, ProductName, ProductCost, ProductPrice, SaleNbr, SaleDate, SaleTime, SaleLineItemProductID, SaleLineItemSaleNbr, SaleLineItemPrice, SaleLineItemQty, SaleCashierID, CashierName)

But this isn't a very good way to store the data. For example, we wouldn't be able to add a product to this table without also using the product in a sale (because the primary key needs both product ID and sale number). Also, we'd be repeating basic product information many, many times because every sale line item includes full product data. This happens because this data set has something called **partial functional dependencies**, meaning not all attributes are dependent on the entire key. Let's explore further. Looking at this table, we'll soon find out that only SaleLineItemPrice and SaleLineItemQty are dependent on the entire two-column primary key. The others are not, which leads us to structural problems.

Partial functional dependency A state of a relation in which all attributes are not functionally dependent on the entire primary key.

So we need to remove the partial functional dependencies. The only way to do so is to split the table into smaller ones. We keep one called SALELINEITEM, but in it we include only the attributes that are dependent on the primary key. So we get:

SALELINEITEM(SaleLineItemProductID, SaleLineItemSaleNbr, SaleLineItemPrice, SaleLineItemQty)

How about the other attributes? We can't lose them, so we need to create additional tables in which all attributes are either directly or indirectly dependent on the entire primary key. We can achieve this by creating separate PRODUCT and SALE tables:

PRODUCT(ProductID, ProductName, ProductCost, ProductPrice)

SALE(SaleNbr, SaleDate, SaleTime, SaleCashierID, CashierName)

The three tables, SALELINEITEM, PRODUCT, and SALE are in much better shape than the original SALELINEITEM. We say that a set of tables that doesn't have partial functional dependencies is in the second normal form or 2NF.

These tables still aren't good enough. The problem is within SALE: it has one attribute that's not *directly* dependent on its primary key. What could that be?

It's CashierName. Within SALE, SaleCashierID is dependent on SaleNbr and CashierName is dependent on SaleCashierID or SaleNbr → SaleCashierID → CashierName. This is so-called **transitive dependency**, and it's also a source of problems: the SALE table would have a lot of replicated data because the cashier's name is repeated every time they complete a sale. Furthermore, we couldn't add a cashier without having at least one sale that had this specific cashier included. The transitive dependencies have to be removed to get the data to the third normal form or 3NF. In this case, we need to split the SALE table into two, as follows:

Transitive dependency A state of a relation in which an attribute of a relation is indirectly functionally dependent on the primary key.

SALE(SaleNbr, SaleDate, SaleTime, SaleCashierID)

CASHIER(SaleCashierID, CashierName)

The other two tables stay the same:

PRODUCT(ProductID, ProductName, ProductCost, ProductPrice)

SALELINEITEM(SaleLineItemProductID, SaleLineItemSaleNbr, SaleLineItemPrice, SaleLineItemQty)

These four tables are now in the third normal form, which is typically considered to be good enough for most practical purposes.

The so-called normalization process is fundamentally the same for each data item set:

1. Determine the functional dependencies between the data items.
2. Organize the data set into a table that satisfies the minimum characteristics of a table (1NF).
3. Remove partial functional dependencies by dividing the data items into smaller tables within which each data item is directly or indirectly dependent on the entire primary key (2NF).
4. Remove transitive dependencies by dividing the data items into smaller tables within which each data item is directly dependent on the entire primary key (3NF).

By analyzing the dependencies between the data items in tables created by converting from an EERD, you will find out that these tables are already in 3NF.

Anomaly A structural problem with a database design.

A 3NF database design is free of three major types of problems (or **anomalies**) associated with non-normalized design: (1) *insertion anomaly*, or the inability to add an instance of an entity without adding instance(s) of another entity at the same time; (2) *deletion anomaly*, or the risk of deleting the only instance of another entity; and (3) *modification anomaly*, or the need to modify multiple copies of the same entity instance because of unplanned replication.

In addition to the specification of the logical structure of the tables, including attributes, primary keys, and foreign keys, logical design will include a specification of a domain for each of the columns and a set of constraints for the columns that need them. In this context, the term *domain* has a specific meaning that differs from its use elsewhere in this book. Here, domain refers to the specification of all allowed values for a set of attributes. In some cases, it's sufficient to describe the domain by specifying a data type, but in many others, the domain includes additional constraints, such as a range of values or an enumerated list of values. A domain of a DateofSale column could be specified simply by stating that it consists of all valid dates, which in implementation could be enforced with the appropriate date data type. If we wanted to also store a DayofWeekOfSale column, we could specify the domain as an enumerated list of "Sunday, Monday, Tuesday, Wednesday, Thursday, Friday, Saturday" (or any appropriate abbreviations).

14.5.2 Deeper Dive 14.2: From a Logical Model to Physical Database Design and Database Implementation

A third normal form (3NF) relational database design is not yet an implemented database. Instead, it only describes the tables, their data items and primary keys, data item domains, and the foreign

keys that associate the tables with each other. The database implementation will include a variety of additional aspects of the database, such as the following:

- Detailed technical specifications of the column data types, which typically can be derived from the domain specifications using the data types of the chosen DBMS and the mechanisms it makes available for specifying constraints. For example, for the DateofSale column, we could simply choose a DATE or TIMESTAMP data type if we were using Oracle; in MySQL, the options would be DATETIME or TIMESTAMP. If we had a column capturing water temperature (WaterTemp), it could have an Oracle data type NUMBER(5,2) (assuming we want to store two decimal points) with a CHECK statement CHECK (WaterTemp > 0 and WaterTemp < 100), assuming conditions where the boiling point is 100°C.
- Specifications for the SQL (CREATE TABLE) statements that implement the database.
- Specification of the **indices** (plural of **index**), which are additional directory structures that will speed up data retrieval, often in a very significant way that's essential for ensuring system nonfunctional (performance) requirements are met.
- Possible denormalization of the database to improve its performance by reducing the number of joins. (The star schema discussion earlier provided examples of denormalization.) This is a topic that specialized books on database management cover extensively. We'll simply recognize its existence here and not explore it further.
- Other detailed technical specifications of the database implementation that are highly dependent on the chosen database management system and are typically taken care of by database administrators.

Index A data structure that provides quick access to data in a database table based on a key value or key values.

Many practical design challenges need to be addressed in the context of database design and implementation. In this section, we'll briefly discuss two of them as examples of cases that industry practitioners face in most projects:

- Deleting records from a table
- Primary key data types

14.5.2.1 Deleting Records from a Table

The first design challenge deals with how to delete data regarding a specific record (or row or instance) of a table from a relational database. This superficially sounds like a nonissue, and, technically, deleting a row from a relational database is very simple: DELETE FROM <table name> WHERE <PrimaryKeyColumn> = <desired value> will do the trick, *physically deleting* the record from the database. How could this be challenging? In most cases, physically deleting a row from a database will lead to several follow-up issues, such as:

1. **Referential integrity:** What should we do with the rows that reference the deleted row with a foreign key? For example, if Wayback Public Library (WPL) removes a lost book from the ITEMCOPY table, should all the loan events associated with that book be deleted too? The referential integrity rule requires that for each foreign key reference, there's a corresponding primary key in the referred table. If the row with that primary key is deleted, the referential integrity is violated.
2. **Completeness of data for analytics:** How should we deal with the analytics processes that would need information regarding the deleted instance? If a very active library patron moves away, no longer uses the services of the library, and, so, needs to be removed as an active patron, shouldn't their activity while at WPL still be considered in historical data?
3. **Auditing:** How can we maintain a complete audit trail if certain rows are permanently deleted from the database? What if a patron to be removed had unpaid fines? Shouldn't all information related to those loans be saved?

Because of these issues, many organizations don't physically delete rows from a database. Instead, they mark them as inactive (using Boolean Active columns, as we did in the database solution developed in Section 14.3.2). This kind of deletion is called a *soft deletion* or *logical deletion*. Here, you aren't removing the data from the database—instead, you're keeping the data but marking (or "flagging") it as being inactive. Soft or logical deletes allow an organization to maintain a historical record of deleted data and ensure continued referential integrity, which enables complete data for analytical reporting and a full audit trail. However, soft or logical deletes do introduce their own challenges, largely because of the need to adjust programming logic to deal differently with the active and nonactive instances. For example, if we logically delete the Item Definition *War and Peace*, does that mean we should disallow patrons from checking out the associated copies of that title?

14.5.2.2 Primary Key Data Types

Another design issue to consider is the data type of the primary key (PK) columns. In general, PKs should not be associated with any real-world meaning. Instead, the system should create its own internal identifier for each record; so-called nonintelligent keys should be used so there's no reason to ever have to change a PK value. But what primary key value should you assign to each record?

Traditionally, many organizations have sequentially assigned simple four-byte integers as primary keys, typically assigning the first record in a table PK = 1, the second record PK = 2, and so on. But in recent years, a key alternative to integer PKs has emerged: GUIDs (globally unique identifiers, sometimes also called universally unique identifiers, or UUIDs). GUIDs are 128-bit (16-byte) randomly generated identifiers. Here's an example of a GUID:

F4AB02B7-9D55-483D-9081-CC4E3851E851

Unlike integer PKs, which typically are assigned in sequential order (1, 2, 3, 4, and so on), sequential GUIDs look random. For example, using GUIDs, the first four records in a table might have the following PKs:

395cb624-c074-4869-9aaf-1c38ed6b2c53

47582875-0590-4581-aac5-56746e360f3d

0ac4e03c-554b-4094-a5ad-fd9b84da2351

f79399f2-e458-4ad5-ac73-f1ce4318493e

The number of unique GUIDs is astronomically high, and when generated randomly, duplication of GUIDs is very unlikely. For example, if you generated 1 billion GUIDs per second a year, you'd only have a 50/50 chance of creating a duplicate.

Compared to integer PKs, GUIDs come with several pros and cons:

- **PRO Easy to merge different tables (no PK collisions):** Say you wanted to merge the same type of data (same table structure) from two different databases. Using integer PKs, each table would have a record with PK = 1, another record with PK = 2, and so on. So merging that data would create a whole series of duplicate PKs (collisions), forcing you to renumber many of the records. In contrast, using GUIDs, you could just merge that data with no concern about PK value collisions.
- **PRO Can generate unique identifiers from multiple systems:** For example, say you create a mobile app that creates and stores a new record on the mobile device, but then needs to merge that record into a central database. A bit like the prior point, when using integer PKs, the PK created on the phone would likely collide with the PK of an existing record in the central database. Again, using GUIDs, you could expect all these values to be unique, which would avoid collisions.
- **PRO Doesn't inadvertently disclose insights about data:** If you use sequentially assigned integer PKs, you can determine that, say, record 234 was created earlier than record 235 and all higher PK values. For example, if a university uses integer PKs for

student ID numbers, it may be possible to determine when a given student first registered with the university simply based on the student ID value. GUIDs avoid this problem.

- **CON Slower performance:** Each GUID is four times bigger than a regular four-byte integer, typically resulting in somewhat slower performance.
- **CON Challenging to identify and select a specific record for debugging:** For example, say a cashier at VFA experiences a bug when finalizing a sales record. Using integer PKs, that sales record might have an identifier of 930495837, which is easy to share with an IT operations or development team member. In contrast, using GUID PKs, that same sales record might have an identifier of 0aa97fb6-4453-4360-920b-977cbe47d831, which is much harder to read and communicate.

The moral of this story is that both integer and GUID PKs are still often used in real-world systems, so you need to be familiar with both approaches.

In concluding this Deeper Dive, note that this is the only stage of the database implementation process that is fully dependent on the database management system (DBMS), and not only the DBMS but also its version. In-depth understanding of the DBMS used to implement the database is essential for the success of this stage.

14.5.3 Deeper Dive 14.3: Integrating Relational Data Management with Object-Oriented Development

One of the challenges in modern software development is that the data management technologies used as the foundation of transactional systems are still frequently relational, but the environments used for construction are primarily object-oriented. This causes a discrepancy that is sometimes called the "object-relational mismatch." Let's look at a quick example of what we mean by this. In the student data example introduced earlier, say a developer is coding in Java, which is an object-oriented language. In Java, the developer has created a class "Student." For each student, this class includes not just basic Student data, such as Last Name, First Name, Phone, and Email, but also an array containing data about that student's course section Registrations. Any given student could have any number of registrations, ranging from none to many dozens. As such, a Student object couldn't be stored directly into a relational database. The Student table can't have a variable number of Registration columns because that would violate the 1NF rule. Instead, in a relational database, you'd store the basic Student data in one table and the Registration data in another table. Most likely, the Student and Registration tables would be linked by a one-to-many relationship, with the StudentID field included in the Registration table as a foreign key.

This example illustrates the general problem of mapping object-oriented data in a computer's memory so it can be stored in (or retrieved from) a relational database. This storage mapping scheme is called **object-relational mapping (ORM)**. It's not quite clear why object-oriented database management systems never took off the same way object-oriented programming languages did. But OO DBMSs are so rarely used that you may never see one being used in the real world.

Object-relational mapping (ORM) A mechanism for creating a link between object-oriented construction and relational data management technologies.

There are many ways to address this technical challenge, and it's not possible for us to discuss even one of them at a detailed level in the context of this chapter. This is one of the areas that the technical design process has to address when relational technologies are used with object-oriented construction. Fortunately, the technical solutions are mature. Most commercial and open-source software development libraries and frameworks come with built-in ORM components. If you, as a BA, are also active in development, you need to understand how to use your environment's ORM component.

We conclude by noting that NoSQL databases, such as MongoDB, introduced in the prior section, avoid many of these mapping issues. In MongoDB, data is stored as documents, which closely resembles storing data as objects. For example, JavaScript has a built-in *stringify* function that transforms a JavaScript object into a JSON document, which can then be directly stored in a

MongoDB database. When bringing the data back to JavaScript from MongoDB, the JavaScript *parse* function will perform the conversion in the opposite direction. (Note that, internally, MongoDB uses a binary format called BSON, but, externally, MongoDB can interact in JSON.) Given how simple this is, why doesn't everybody use MongoDB or some other document database instead of the still dominant relational databases? One reason definitely is that relational databases have over decades become so widely used that moving away from them would require a monumental effort, including, for example, transition from relational SQL to other query and manipulation languages. Beyond that, relational databases provide strong built-in support for enforcing referential integrity, which is very important when dealing with large volumes of transactional data. Also, for many purposes, MongoDB provides too much flexibility—we may not want it to be too easy to add nonstandard attributes to our database.

14.5.4 Deeper Dive 14.4: Designing Internal System Structure Using the Sequence Diagram

The system sequence diagrams discussed in Section 14.2.3.2 provided you with a good example of one important use of sequence diagrams. Remember, however, that an SSD still doesn't cover the internal structure of the system of interest; it just provides additional details described in a more formal way regarding how the system is expected to communicate with its external environment, including users and other systems. The most common use of the sequence diagram notation is to model the *internal* structure of a system, particularly from the perspective of how responsibilities are allocated to objects in different classes. For this purpose, you need to be aware of the following additional sequence diagram elements and more complex uses of the structures that you learned about earlier (see Figure 14-11).

1. An *execution specification* (informally often called *activation*) specifies a part of the lifeline during which a specific agent (object, actor, etc.) is currently active.
2. A message that is sent from an activation to itself specifies recursion—that is, an object calling itself.

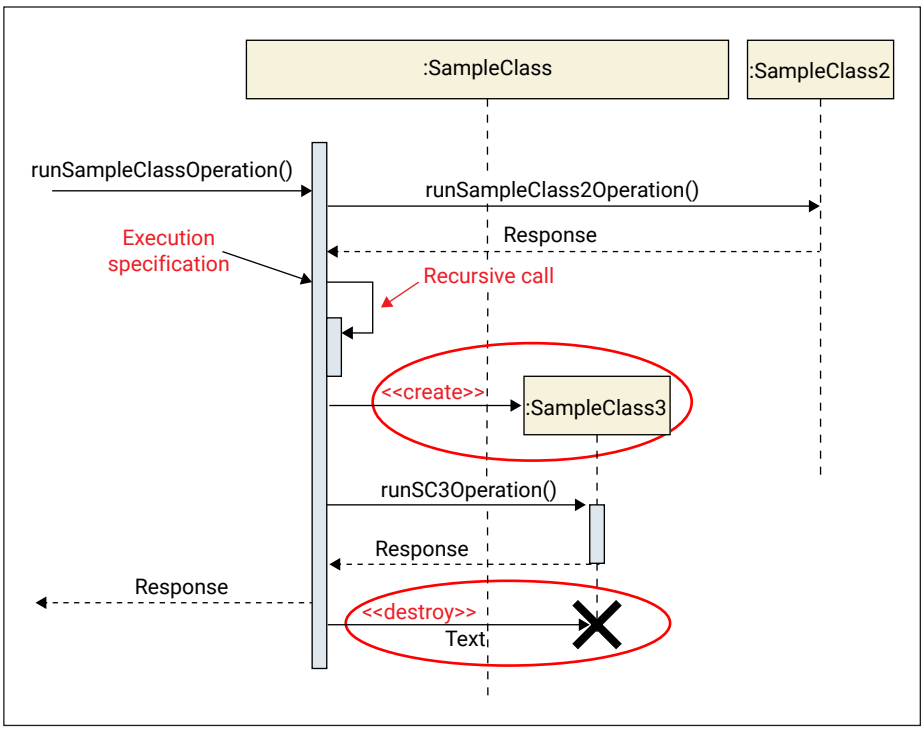


Figure 14-11 Additional structural elements of the UML sequence diagram

3. An object can *create an instance* of another object.
4. An object can *be destructed* when it's no longer needed.

14.5.4.1 Layered Application Architecture

A very typical way to design an internal structure of a system capability is to divide it into three categories:

- **Presentation Layer:** User interface classes that implement the mechanisms through which users interact with the system.
- **Business Logic Layer:** The orchestration of the communication between the Presentation and Data Layers and implementation of the internal logic of the communication between the Business Logic Layer and Data Layer classes.
- **Data Layer:** Primarily the classes that model the domain concepts, those included in the conceptual data model.

We'll demonstrate the Layered Application Architecture model in a simplified form in which we assume that the Presentation Layer captures the system events (messages from the user) and sends them to the Business Logic Layer. The Business Logic Layer orchestrates the division of the responsibilities between the Data Layer objects. Once the Business Logic Layer has generated the required response to the request coming from the Presentation Layer (using the capabilities of the Data Layer objects), it sends the response back to the Presentation Layer, which, in turn, makes it available to the user. We won't model the internal structure of the Presentation Layer, which is often quite complex and requires in-depth understanding of the technical environment that implements the characteristics of the Presentation Layer. In addition, the Presentation Layer is typically implemented with a technology different from the one used for the Business Logic Layer and the Data Layer. The Presentation Layer (essentially front-end) development is currently frequently done using JavaScript/TypeScript or a framework based on one of those languages. These presentation layer tools offer the developer a rich variety of reusable UI components for the Presentation Layer.

The solution could be architected so the Presentation Layer is implemented essentially assuming that it forms a direct channel to the user, only having responsibility for the most fundamental verification logic. In practice, modern front-end development environments provide good tools for the Presentation Layer. The Business Logic Layer and the Data Layer are, in turn, implemented using a back-end framework. So, in this example, we're abstracting the Presentation Layer and assuming it will provide the necessary data for the user–system interaction.

Figures 14-12 and 14-13 present two separate **design sequence diagrams (DSDs)** for a couple of the system events identified earlier in Figure 14-8. These design sequence diagrams still don't capture all the details at a level that would allow automatic conversion into programming code. This is not the intent either. The key point of this type of use of sequence diagrams for internal design is to help in the process of allocating responsibilities to different objects and ensuring the correctness of the internal logic.

Design sequence diagram (DSD) The use of the UML sequence diagram for modeling the internal structure of the system.

Let's explore the essential elements of each of these DSDs. Consider first Figure 14-12, which deals with adding a single product:

- The context is that we've initiated a sale, including having already created the object *thisSale*. This sequence specifies the process of adding a product to the sale.
- The system event *addProduct(upcCode)* sends the Business Logic Layer a request to retrieve the label and price of the product, the UPC code of which was initially sent to the Business Logic Layer as a parameter. The UPC code is a result of a scanning activity.
- The Business Logic Layer, in turn, sends a message *getProductDetailsForSale(upcCode)* to an internal object *productCollection* to retrieve the details of the scanned product. The *productCollection* object includes all products and can navigate to any individual product object based on the UPC code. Alternatively, the implementation of *productCollection* is able to do a direct database query based on the UPC code.
- *productCollection* returns the label and price of the product to the Business Logic Layer.

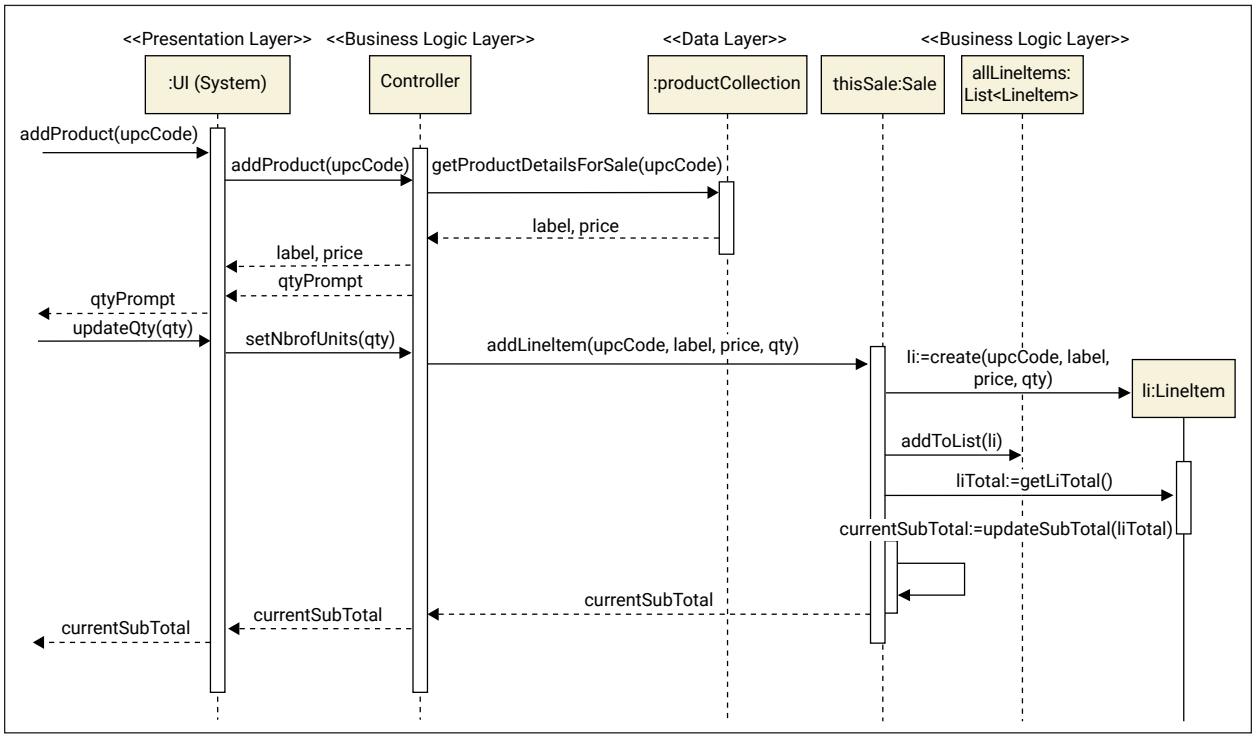


Figure 14-12 Design sequence diagram for system event addProduct(upcCode)

- The Business Logic Layer returns the label and price to the Presentation Layer, which presents them to the user and continues with a prompt for the quantity (defaulting to 1) *qtyPrompt*.
- The UI/user communicates with the Presentation Layer, which, in turn, asks the Business Logic Layer to set the number of required product units to a specific number (qty) with the system event *setNbrofUnits(qty)*.
- To respond to *setNbrofUnits(qty)*, the Business Logic Layer sends a request *addLineItem(upcCode, label, price, qty)* to the *thisSale* object to add a new line item with parameters upcCode, label, price, and quantity. For *thisSale* to achieve this, it needs to first create a new instance of *LineItem* called *li* and then add this new object to a collection of line items (*allLineItems*) that it owns. *thisSale* also requests a line item total from *li*, which calculates it by multiplying quantity and price and returns it to *thisSale*, which, in turn, adds the line item total to its *currentSubTotal* attribute. thisSale returns the value of *currentSubTotal* as a response to *addLineItem*. The Business Logic Layer returns *currentSubTotal* to the user/UI (through the Presentation Layer), together with a prompt for scanning a new item or completing the sale.

Figure 14-13 is about completing the sale, covering part of the internal structure (with the rest left out to avoid making the model overly complex):

- If the Presentation Layer sends the *completeSale* request to the Business Logic Layer, the latter submits a *completeSale* request to the *thisSale* object. *thisSale* then asks the collection *allLineItems* for the sum of all line item totals, stores the sum in its own *saleTotal* attribute, and returns *saleTotal* to the Business Logic Layer. The Business Logic Layer returns *saleTotal* to the Presentation Layer, which returns *saleTotal* to the user/UI, together with a prompt for the payment method.

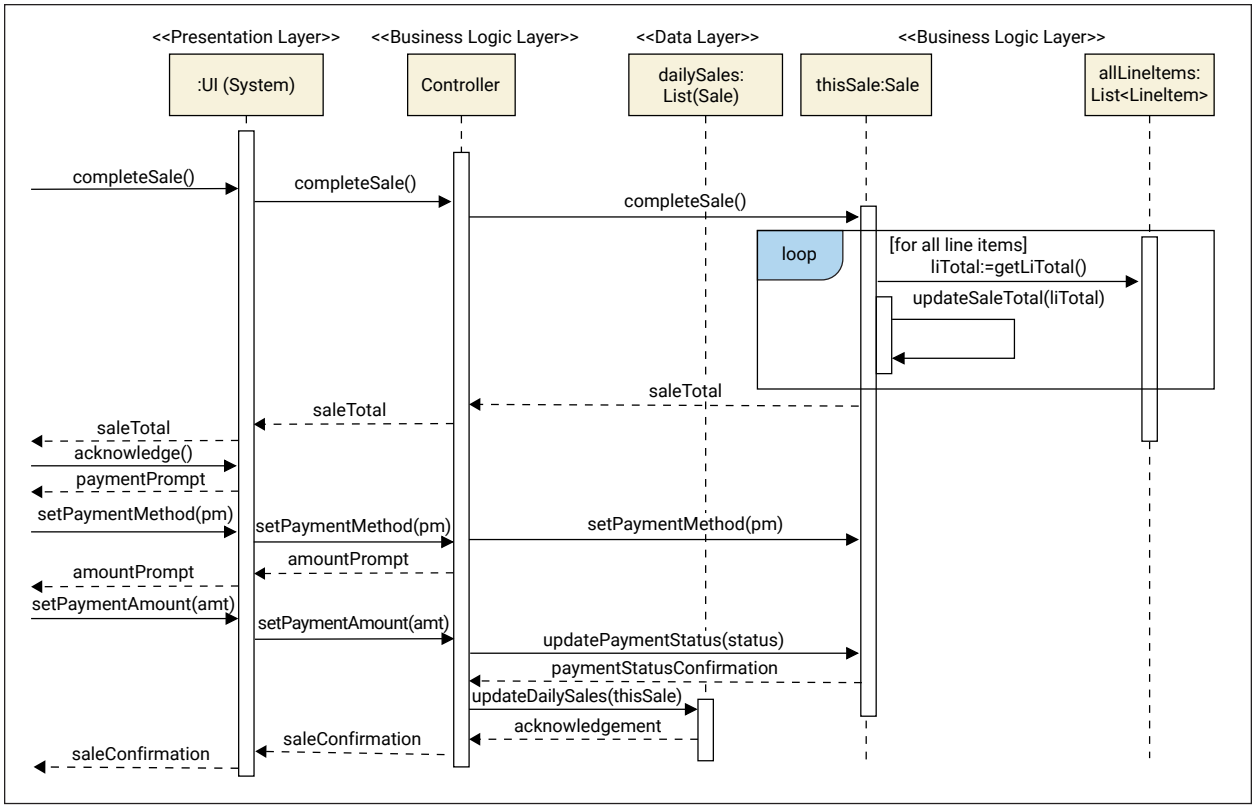


Figure 14-13 Design sequence diagram for system event completeSale()

- The user/UI selects the payment method. Let's assume Cash is the selected method. If this is the case, the Business Logic Layer sends a *setPaymentMethod("Cash")* message to *thisSale* and responds to the user/UI by prompting for the cash payment amount with *amountPrompt*.
- The user/UI enters the cash payment amount, which generates the *setPaymentAmount(amt)* message to the Business Logic Layer, which returns the amount of change to the user/UI and sends an *updatePayment Status("paid")* message to *thisSale*, assuming amt is greater than or equal to *saleTotal*.
- Now *thisSale* has been fully completed, with *paymentStatusConfirmation*, and the Business Logic Layer can add it to its *dailySales* collection (and write it to the Sales and LineItems tables in the database) with *udateDailySales(thisSale)*.

Is it necessary to create technical specifications at this level of detail for all system events? No, it isn't. Again, it's important to follow the intelligent modeling principle we discussed earlier in the book. Often designers find it helpful to draft design sequence diagrams by hand, not to create a permanent document but to clarify their own thinking. The project context and approach also matter, per Chapter 10. For example, if you have a new team of developers in multiple locations, it may well be that the developer you assign this story may need this level of guidance and formal communication to understand what class or classes they need to build and how those classes fit in with classes built by other developers. On the other hand, if you have an experienced team of developers in a single location who are knowledgeable about the business problem and able to communicate continuously with the product owner and each other, that would probably be overkill.

14.5.5 Deeper Dive 14.5: Identifying Class Behaviors Based on the Sequence Diagram

Where did the Business Logic and Data Layer classes in Figures 14-12 and 14-13 come from? The short answer is that they came from the conceptual data model: This business problem domain clearly involves data about products, sales, sales line items, and so on. It's a key principle of object-oriented design that program classes are based on domain entities (which are, in practice, frequently documented with a conceptual data model). Importantly, these classes include both the data and logic associated with the domain entities. As such, most of the program logic will be implemented as methods located in one of these domain entity classes.

The most important reason for modeling the assignment of responsibilities to objects using the UML sequence diagram (or any other similar technique) is that it will allow you to identify the behaviors (essentially, the methods) that should be assigned to each class when coding them. This is one of the essential purposes of technical design. Using sequence diagrams (or communication diagrams) to specify interaction sequences is an excellent way to determine which behaviors should be assigned to which classes in object-oriented design.

Let's see what we learned from the behaviors assigned to key classes in the examples provided previously:

- The Sale class needs to be able to respond to the following requests:
  - addLineItem(upcCode, label, price, qty), returning currentSubTotal
  - completeSale(), returning saleTotal
  - updatePaymentStatus(status), returning acknowledgement of the status of the update
  - updateSaleTotal(liTotal), returning the current subtotal
- The productCollection class needs to be able to respond to the following request:
  - getProductDetailsForSale(upcCode), returning label and price
- The LineItem class needs to be able to respond to the following request:
  - getLiTotal(), returning the total value of the lineItem

These new behaviors are included in a draft design class diagram, available in Figure 14-14.

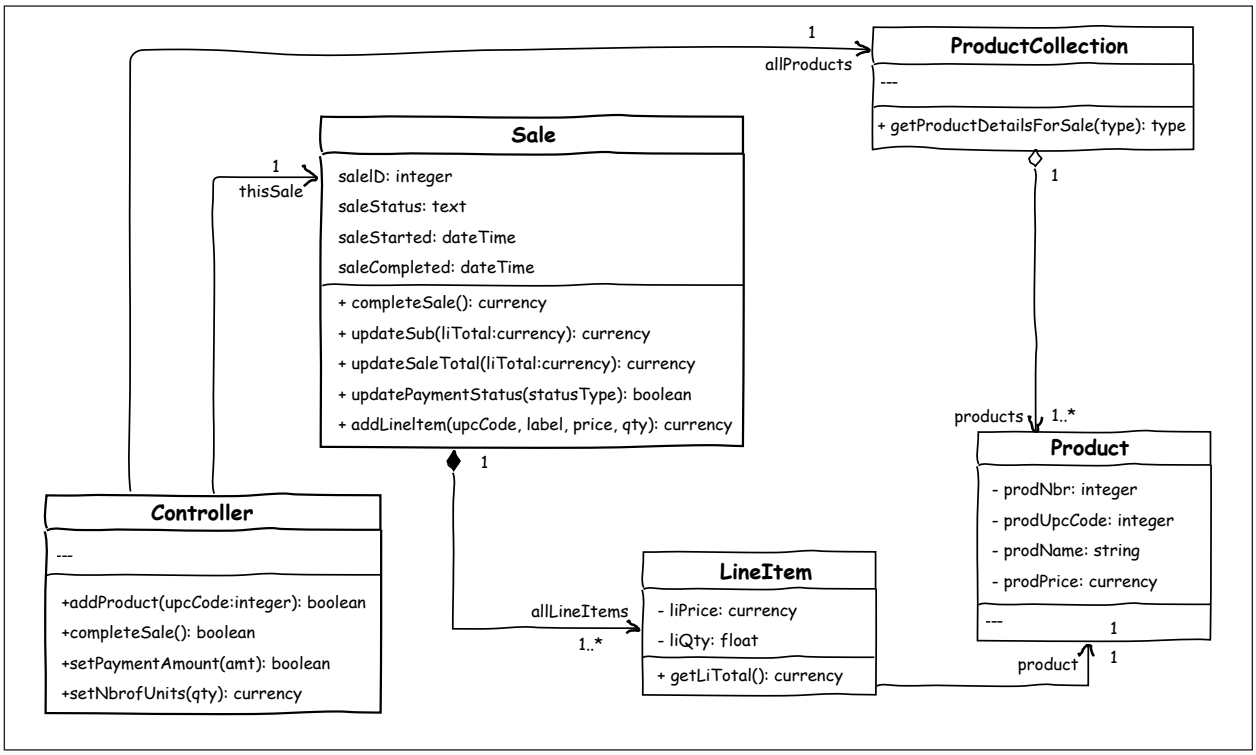


Figure 14-14 UML class diagram representing the internal design of a point-of-sale (POS) system (inspired by Larman, 2004)

When determining in what class a method should be placed, the class that contains all (or most) of the data the method needs is typically the right choice. There may be situations where a method doesn't fit into a domain entity. For example, suppose we need to summarize Sale objects into a SalesSummary entity, with each SalesSummary object or row consisting of the total number and value of sales for a specific product, month, and store. We might do this, for example, as part of creating a data warehouse using a star schema for analyzing our sales trends. In this case, we need a method (let's call it summarizeSales()) to read the Sale objects, summarize them, and then create and store the SalesSummary objects. Where should that method go? It can't go inside the Sale class, because each Sale object is read *by* the summarizeSales() method. Similarly, it can't go inside the SalesSummary class, because the method *creates* those objects. The answer is that this isolated chunk of logic needs to go into its own "logic only" class (we could call it SummarizeSales) that's not part of the conceptual data model, and hence not a part of the database schema.

Obviously, these method names and parameter specifications are not yet sufficient information for generating the final code the compiler will understand. In practice, they specify the method's signature. Each method's signature consists of a set of attributes that the programming language's compiler can use to identify the method. These attributes include the name of the method, the number of parameters passed to the method when it is invoked, the data type of each parameter, and the order of the parameters. The actual internal logic of the method that uses the parameters still has to be written separately, but often it's relatively straightforward. If that's not the case, designers can often leverage the low-level logic from a use case narrative (Chapter 5). In the absence of use cases, they can also write separate notes in, for example, structured English or pseudocode to provide additional guidance to those writing the code.

The design approach specified here has to be adapted for the purposes of the technical environment for which you're designing an application. It's not going to work as such without consideration of the details of the programming environment. It does, however, form a foundation for one approach to thinking about the allocation of behaviors and responsibilities to various classes in object-oriented development.

14.5.6 Deeper Dive 14.6: Design Patterns and Frameworks

14.5.6.1 Introduction

Many design problems share underlying characteristics, even if the application logic derived from the functional requirements differs. This has led to the creation of shared **design patterns** and frameworks, which help designers avoid reinventing the wheel by providing guidance for addressing common design problems. The most useful patterns apply to multiple technical implementation environments.

Design pattern A general solution for a commonly occurring software design or development problem.

An exhaustive review of design patterns is beyond the scope of this chapter. Instead, we'll provide examples of several useful patterns that highlight shared design principles. We'll first discuss three widely cited patterns called MVC, MVP, and MVVM, which are most frequently applied for user interface development. We then cover two frameworks, LAMP and MEAN, that help in organizing the entire technology set required for web-based or mobile applications.

14.5.6.2 MVC, MVP, and MVVM Patterns

Some of the most widely used patterns are Model-View-Controller (MVC), Model-View-Presenter (MVP), and Model-View-ViewModel (MVVM). Of these three, MVC is the original one (developed already in the 1970s for an early version of the Smalltalk programming language), and the other two are essentially modifications of MVC. In all of them, the core principle is the separation of concerns that we briefly discussed at the beginning of this chapter: encouraging designers to separate the visible characteristics of the presentation of data and users' interaction with it from the business logic associated with the changes to the data and the storage of the data. Originally, all of these patterns were primarily focused on structuring and organizing the users' interaction with the system. As we'll discuss later, particularly the MVC pattern has been adapted for many other uses.

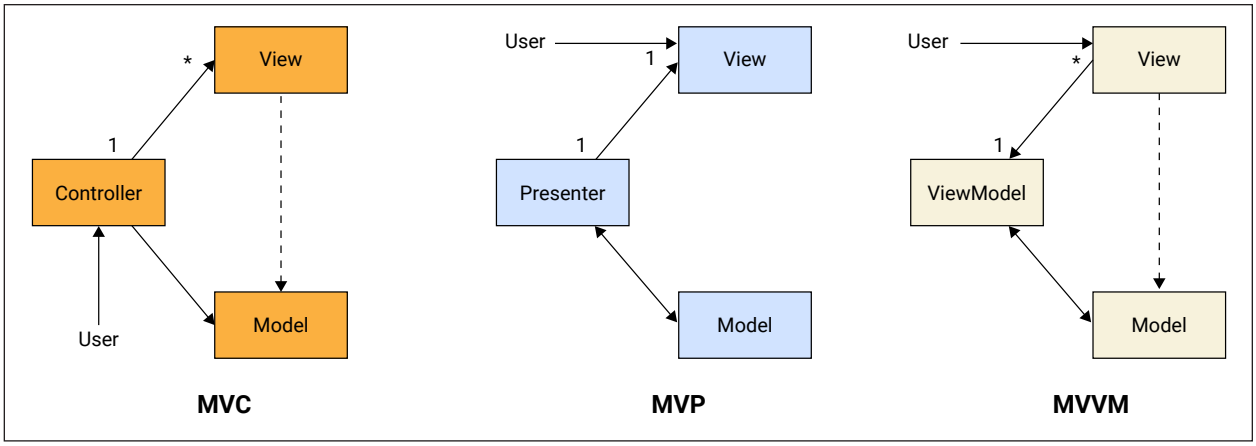


Figure 14-15 MVC, MVP, and MVVM frameworks

As is clear based on Figure 14-15, all three patterns include the same two components: Model (which represents the data of the application) and View (which enables the presentation of the data to the user in a specific way).

MVC

In MVC, the final component is called the Controller. In the original MVC, both View and Controller were components of user interface implementation. In practice, even the Model referred to data as it was seen and used in the context of the system user interface. In the original MVC, the user communicated with the Controller, which, in turn, used the services of the Model to enact the user's request. As necessary, the View modified itself based on the changes in the Model (but the Model had no visibility of the View).

The MVC model has been leveraged in many other contexts for a variety of different purposes. For example, for the Layered Application Architecture model discussed in Deeper Dive 14.4, following the approach described in Larman (2004), we could have named the Presentation Layer as View, the Business Logic Layer as Controller, and the Data Layer as Model (recalling that here the Controller is the coordinating class that receives requests from the UI and initiates action in other classes). This modifies the original MVC approach, but the key purpose still stays the same: enabling the separation of concerns between the user interface, the data, and the core logic of the application.

MVP

The MVP model replaces the Controller with Presenter, and the user interacts directly with the View. There's no direct communication between the View and Model in MVP; all communication is mediated by the Presenter, which essentially provides the View with all of its logic.

MVVM

The newest pattern is MVVM, adding ViewModel as the third element. The user interacts with the View, which, in turn, passes the requests to ViewModel, which is a specialized representation of a Model for the purposes of a specific View. The ViewModel specifies a View's behavior and state. One of the specific features of MVVM is that the pattern requires binding between View and ViewModel so that changes in ViewModel will be automatically reflected in the View. In this pattern, the components are highly independent so that ViewModel is not aware of the View(s) bound to it and the View(s) are not aware of the Model.

Presenting examples of these patterns would require more coding experience than most of the users of this textbook will have. Still, it's useful for you to at least be aware of the three patterns, their main components, and the main reason underlying their existence: the implementation of the separation of concerns principle. Just remember that there's no one commonly accepted interpretation

of any one of these popular patterns: you can be sure about the interpretation of the meaning of a particular pattern only once it's implemented in a specific set of development tools. None of the real, working implementations are simple to recreate. For example, Josh Smith from Microsoft published an article in 2009 titled "Patterns—WPF Apps with the Model-View-ViewModel Design Pattern," which describes how the MVVM pattern works in the context for which it was originally created (Smith, 2009). If printed, this excellent example would take about 30 printed pages (including hundreds of lines of code). We won't attempt to replicate this here.

The MVVM pattern has informed several of the modern web development frameworks. For example, both Vue.js and Angular have architectural characteristics that resemble MVVM, particularly its two-way data binding. In practice, this means that changes in the Model will automatically be reflected in the View and changes in the View in the Model. This magic is made possible by the ViewModel component of MVVM.

14.5.6.3 Full-Stack Development Frameworks: Two Examples

Full-stack development frameworks are commonly used combinations of tools that developers use to improve the quality and efficiency of their development work. The "stack" (often called a "tech stack") in this case refers to a set of technology options that serve different purposes but are well organized to work together toward a common goal.

You may have heard of the LAMP stack, which has been around for a long time. It refers to the following four components:

- **L**inux as the operating system for the application server (which handles the business logic)
- **A**pache as the web server (which processes HTTP requests, arriving, for example, from a user's web browser)
- **M**ySQL (or **M**ariaDB) as the database management system
- **P**HP (or Python) as the (scripting) programming language

LAMP is popular, in part, because all four components are open source.

The LAMP stack (or its WAMP Windows equivalent, where Windows Server runs in place of Linux) has been used to develop countless web-based applications. Note that this stack doesn't specify a particular specialized technology for the client level; it typically would consist of a combination of HTML, CSS, and JavaScript. As Figure 14-16 shows, the client-side technologies issue requests to the web server (Apache), which, in turn, identifies the situations when the services of an application server are needed (developed typically in PHP or Python). When persistent (long-term) data storage is needed, the application server will use the data storage services of the database (e.g., MySQL). As you can see, the division between the Presentation Layer, Business Logic Layer, and Data Layer applies here directly.

A more modern stack, MEAN, allows the use of the same programming language (JavaScript, or its more modern version, TypeScript) at all levels (IBM, n.d.). It consists of the following elements:

- **M**ongoDB, introduced previously, is one of the most popular NoSQL databases (a so-called document database that uses JSON).
- **E**xpress.js is a server-side framework that makes it faster for developers to build applications. In turn, it uses the services of Node.js.
- **A**ngularJS is a client-side web development framework based on the TypeScript scripting language (a superset of JavaScript) that we mentioned in the context of MVVM.
- **N**ode.js is a so-called JavaScript run-time environment, which makes JavaScript also available on the server side.

Figure 14-16 also includes a graphical representation of the MEAN framework. Again, the familiar Presentation Layer, Business Logic Layer, and Data Layer structure appears: AngularJS at the Presentation Layer, Express.js and Node.js at the Business Logic Layer, and MongoDB at the Data Layer.

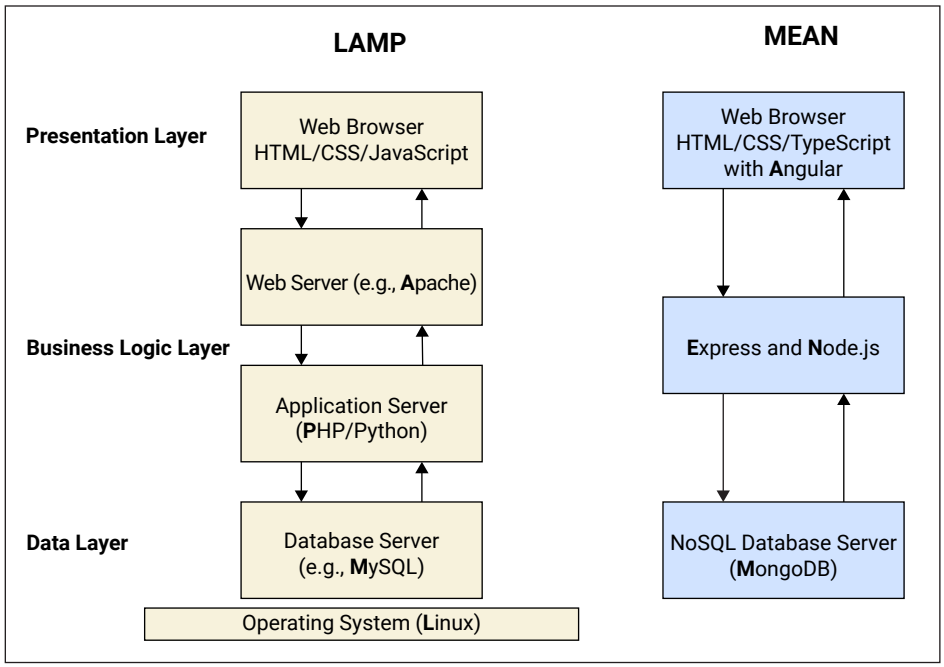


Figure 14-16 LAMP and MEAN stacks

As you can see, the components of these different stack models are not fully aligned with each other: LAMP includes the web server (Apache) and operating system (Linux) components that are not specified in the MEAN stack, whereas the MEAN stack includes two separate components (Express.js and Node.js) for server-side development. In addition, it's important to emphasize that the division between the Presentation/Business Logic/Data Layers is not as clear cut as the model in Figure 14-16 suggests: depending on the design approach, particularly Business Logic can be incorporated at various levels (including the Data Layer).

There are many other tech stacks, often resulting from replacing one tool in an existing stack. For example, the MERN stack is the MEAN stack, except AngularJS is replaced by a web development library called React.

14.5.6.4 GRASP Patterns

In his book *Applying UML and Patterns*, Craig Larman (2004) introduced nine so-called GRASP (general responsibility assignment software patterns) principles, which collectively capture many central software design principles. They provide useful guidance for detailed design decisions, regardless of the technology environment in which the development takes place. Detailed coverage of them is beyond this book, but we'll share brief descriptions of six of them here (adapted from Larman, 2004):

- *Low coupling* is one of the most widely used design principles. It says that independent elements should be as loosely connected with other elements as possible. All interconnections that do exist should take place through well-defined interfaces. Low coupling is essentially the same as the separation of concerns principle discussed earlier in this chapter. It's the basis for the benefits of modern application architectures such as service-oriented architecture (SOA) and microservices, as described in Chapter 13.
- *High cohesion* complements low coupling and is also widely used. It says that the responsibilities within a design element should be closely related and fit well together: a design element should have one or very few purposes for its existence, and all components of the elements should contribute toward the same purpose.

- The *information expert* pattern states that when deciding which class should get a specific responsibility (like a method), the designer should assign that responsibility to the class that contains the data needed to implement the responsibility. This corresponds to the previous section on using sequence diagrams to assign methods to classes.
- The *controller* design pattern is a core element of the MVC pattern discussed previously. It says that the fulfillment of system events (user actions or requests) in an object-oriented design needs to be coordinated by a non–user-interface object. These objects are called *controllers*, which, in the MVC pattern, aren't part of either the View or the Model. As noted previously, the benefits of this pattern include the ability to create multiple user interfaces (web browser, mobile web browser, mobile application, and so on) that can share a common business logic and data layer.
- The *pure fabrication* pattern says that you often can benefit from creating design structures (classes) that don't correspond to entities in the conceptual data model, especially if they help you achieve other design goals, such as low coupling and high cohesion. For example, the SummarizeSales class described previously is a pure fabrication not corresponding to a conceptual data model entity. Similarly, presentation classes (including reusable controls such as text boxes, labels, radio buttons, sliders, and so on) are fabrications. Finally, certain common system functions (for example, object-relational mapping) are best "outsourced" to specialized database classes that can be reused in any business context.
- The final design principle is *protected variations*, which says that you should protect a class from variations or instability in another class. For example, in the summarizing sales example discussed earlier, we may need to calculate sales tax. Calculating sales tax can vary for every nation, state, province, county, and municipality. So the Sale class needs a SalesTax class that includes methods to do each of those calculations. But we don't want the Sale class to have to change every time we add a new SalesTax calculation. To avoid this, we should create a single, stable CalculateSalesTax() interface in the SalesTax class that the Sale class can invoke. Using an approach called *polymorphism*, the Sale class can get the SalesTax class to use the correct sales tax calculation through that single interface.

14.5.6.5 Gang of Four Patterns

No discussion on design patterns is complete without at least mentioning a seminal computer science book on patterns by the so-called Gang of Four (Gamma et al., 1994). The name of the book is *Design Patterns: Elements of Reusable Object-Oriented Software*, and it's still widely used as a source by those who want to learn more about complex design patterns. In this final section on patterns, we briefly describe only a few of them, but studying all of them is valuable.

- The *Singleton* pattern is used for creating a class that can have only one single instance (object). Several other well-known patterns use Singleton in their implementation. For example, consider an application that handles user support request tickets. This application may include workflow functionality that routes each ticket to the customer service representative best qualified to respond to it. Here, you may create a Workflow class that tracks and routes the tickets. It's likely that you should create only a single object of this Workflow class because multiple copies might cause chaos, as each copy attempts to route the same tickets simultaneously.
- The *strategy* pattern is used in situations when it's advantageous to delay the decision regarding which particular version of programming code (most often, a specific algorithm) to use until run time. For example, in the same summarizing sales example, suppose we need to add functionality to handle how ordered items are selected from a warehouse. In addition, suppose a user can select from two different approaches to shipping: "Ship each item as quickly as possible" or "Group all items into a single shipment."

Here, we may design these classes using inheritance: implement a general "ShipApproach" superclass, with subclasses "ShipQuickly," "ShipTogether," and so on.

- The *observer* pattern creates a dependency among objects so that when the state of one object changes (e.g., variable value changes), all of its dependent objects are notified (e.g., by calling one of their methods). For example, this is often used in subscription situations, such as subscribers to a blog receiving a notification when a new blog entry is published.
- The *state* pattern allows objects to behave differently depending on their internal state. For example, in the summarizing sales example, consider the possible inventory states of a product when a user wants to order it: (1) in-stock with enough inventory to fill the order, (2) in-stock but without enough inventory to fill the order, (3) out-of-stock but reordered, and (4) out-of-stock and not reordered. Without going into detail, we could create subclasses that implement different behaviors for different states. For example, in State 3 (out-of-stock but reordered), the system could ask if the user wants to place the order for delivery when the reorder is fulfilled.

The key idea underlying patterns is that they serve as model solutions to frequently occurring programming problems. Instead of having to solve a complex problem from scratch, a pattern provides developers with guidelines for creating high-quality, flexible, and time-tested solutions.

Chapter 15

Leading Iterative Systems Development

15.5 Deeper Dives: Advanced Iterative Development Topics

In this section, we'll do deeper dives on several iterative development topics:

- **Deeper Dive 15.1** explains the use of Kanban boards to manage AI and analytics projects.
- **Deeper Dive 15.2** explains product backlog items (PBIs) and their relationship to user stories in iterative development.
- **Deeper Dive 15.3** discusses how to create a burndown chart of engineering tasks when the original user stories and acceptance criteria are estimated in story points.
- **Deeper Dive 15.4** explores how to assess your team's velocity in a single sprint and over several sprints.
- **Deeper Dive 15.5** explores the problem of technical debt and its impacts on sprint velocity.

15.5.1 Deeper Dive 15.1: Using Kanban Boards to Manage AI and Data Analytics Projects

In Section 15.4.1, we explored agile task boards as a supplemental tool (in addition to burndown charts) for monitoring sprint-based development. We concluded that section by noting that agile task boards are visually similar to but different from **Kanban boards**.

Kanban board A column-based format for portraying the status of work items. Visually similar to an agile task board, but used in the Kanban approach to software work.

Kanban is another agile project methodology, like Scrum or Extreme Programming (XP). If all of these methodologies are agile, what makes Kanban different from Scrum and XP? The key difference is that Scrum and XP are typically used in projects where you can plan your work ahead at least a week into the future—in other words, where you can create and execute a sprint plan.

In contrast, Kanban focuses on managing work that fundamentally can't be planned—not even a week into the future. In IT systems, Kanban has traditionally been most closely associated with managing ad hoc system issues, or "tickets," for an IT service desk. Here, tickets could include things like defects/bugs, change requests, requests to add peripherals, and so on. If you think about it, you'll realize that an IT service desk team can't plan out much of their work ahead of time. They just have to respond to whatever incoming messages and calls they receive that day.

But even if an IT service desk group can't plan its work, it *can* track and manage it using a Kanban board. Figure 15-22 shows an example of a Kanban board, with tickets that could correspond to the defects, change requests, and so on, we've discussed. As noted earlier, a Kanban board looks a lot like an agile task board. You can even add columns as needed. (In Figure 15-22, we've included a Testing column, which we also could have done in the agile task board example in Figure 15-19.)

But what does managing a stream of tickets to an IT service desk have to do with software development? It turns out that some types of systems development projects involve such high levels of uncertainty that they're also hard to plan more than a day or so in the future. These can include both AI and data analytics projects:

- **Artificial intelligence:** As introduced in Section 8.5.6, AI projects often involve tasks that are hard to estimate and may have to be repeated and revised many times. For example, in training an AI model, you can't know up front how much training the model will require or how well it will ultimately perform. If a data set and training model fail in the end to meet your performance goals, you may have to throw out that approach and go back to the drawing board. This uncertainty means you must plan this work day-by-day, depending on how each day's training goes.

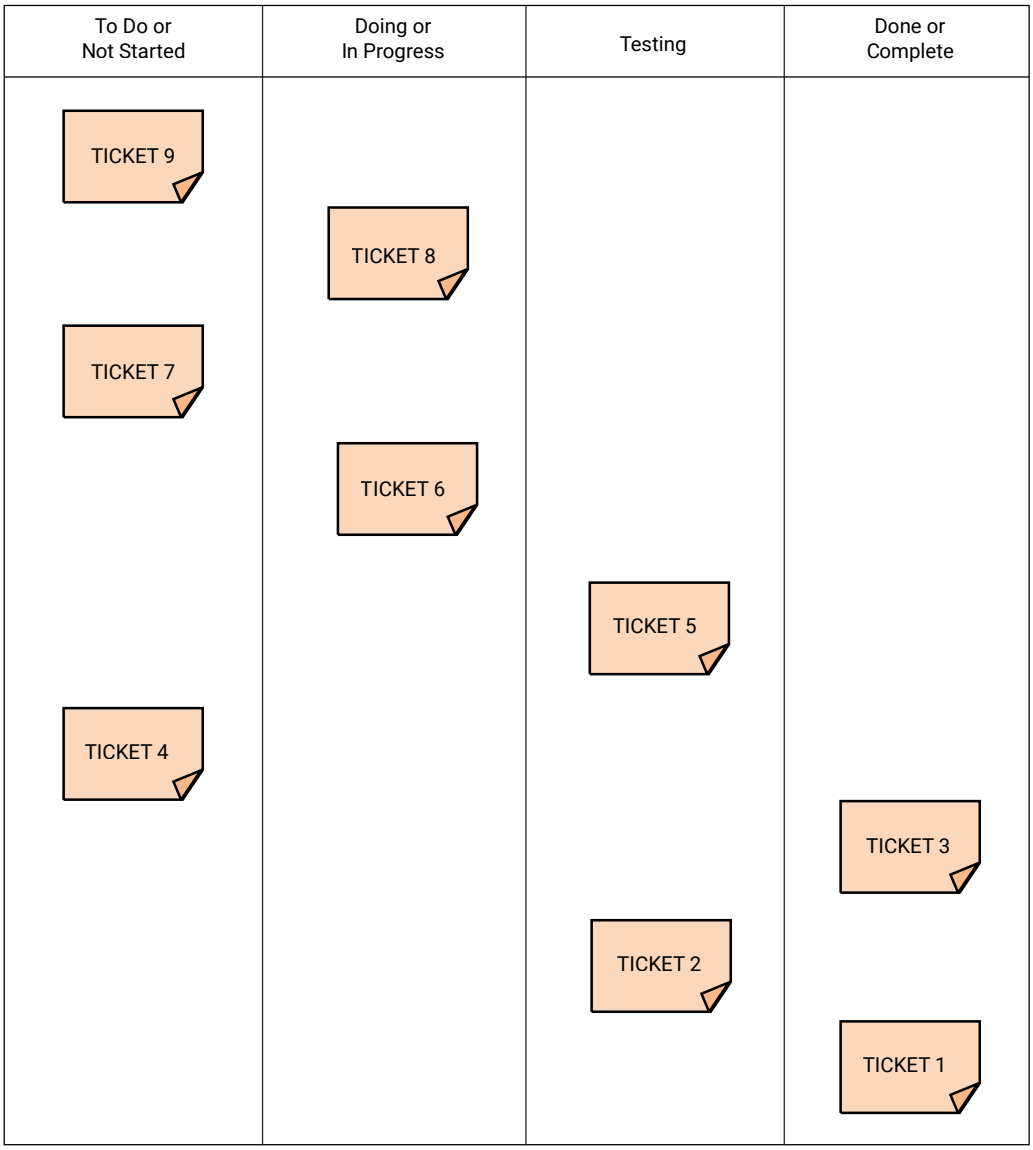


Figure 15-22 Kanban board for managing tickets or nontraditional development projects, such as AI and data analytics

- **Data analytics:** Creating data analytics solutions also involves considerable day-to-day uncertainty, especially after you've set up the data warehouse and started generating reports. Consider a situation in which a large amount of data from many data sources has been organized in a data warehouse. Once you start generating reports, the data analytics questions you ask tomorrow may depend on the answers you get today. Again, this pushes you toward a day-to-day management approach.

So the key point is that IT teams working on these kinds of projects often use a "work for a day, evaluate progress, then plan tomorrow" method. This is a perfect situation for using a Kanban board.

In conclusion, using Kanban to manage AI and analytics software projects one day at a time results in a software development approach that, in a sense, is even more agile than sprint-based development.

15.5.2 Deep Dive 15.2: Product Backlog Items (PBIs) versus User Stories

Throughout this book, we've expressed system features as user stories. However, in product and sprint backlogs, you may occasionally encounter a different term that might seem to mean roughly the same thing: **product backlog items**, or **PBIs**. Confusingly, user stories and PBIs aren't exactly the same thing, but they *are* overlapping concepts.

Product backlog item (PBI) An item of work from the product backlog that can be assigned to a sprint. A PBI may be a feature (same as a user story) but may also be a defect to fix, a technical task, a change request, or a knowledge/skill set acquisition task.

By way of explanation, "user story" is an Extreme Programming (XP) term that has come to be widely used in all sorts of project methodologies. In contrast, "PBI" originated in the Scrum methodology, but in many Scrum teams it has been replaced by "user story."

Here's the relationship: all user stories are PBIs, but not all PBIs are user stories. PBIs include a broader range of work items than user stories They can be (Rubin, 2013, p. 100) of the following types:

- **Features:** Typically expressed as user stories
- **Defects/bugs:** To fix.
- **Technical tasks:** For example, "stand-up new servers" or "migrate production data into the test environment"
- **Change requests:** For example, "update the list of authors prepopulated into the Author table"
- **Knowledge or skill set acquisition:** For example, "train team members to develop using the React presentation layer JavaScript library"

Why is this distinction even an issue? It's because some agile experts hold that your team should only work on *features*, items that add business value to your clients in a clear and direct way. They argue against including items from the other four PBI types, concerned that those items will sidetrack you from delivering value to clients.

We disagree with that "purist" agile perspective: all of these PBI item types represent real work that will consume our team's time. As such, our view is that all of them can—and, if they're important to the overall project, should—be included in product backlogs, sprint plans, and burndown charts.

15.5.3 Deeper Dive 15.3: Creating Burndown Charts When the User Stories and Acceptance Criteria Were Estimated Using Story Points

In Chapter 8, Section 8.4.2, we explained that user stories and acceptance criteria can be estimated using story points, rather than ideal days or hours. As a quick reminder of that discussion, here are some key facts about story points:

- Story points are unitless, relative measures of the size of a user story or PBI. In other words, five story points indicates a certain amount of work but doesn't directly tie to hours or days of effort.
- Story points are nonstandard. So the meaning of, say, five story points for one team will be different than for another team.
- Still, a given team can use story points for general estimating because five points means something to them (perhaps "five points means a small to medium size item"), and an item sized at 10 points will be twice as big as a 5-point item.
- A team may also know that they generally can complete, say, 25 story points in a given sprint.

Still, when it's time to assign user stories or PBIs to sprints, you have to get specific *in terms of time* (hours or days of effort) about whether those items will fit into the sprint's capacity. This generally occurs when the team translates user stories or PBIs into engineering tasks (Rubin, 2013, pp. 345–346).

In the examples where we've estimated user stories and acceptance criteria using ideal hours, we hoped and expected that engineering task hours would sum up to approximately the same number of hours (which, in our examples, they did).

In contrast, when using story points, you can't explicitly check for the number of engineering task hours corresponding to the (no time unit) story points. But, if you've established that you can complete about 25 story points per sprint, you can hope and expect that the corresponding engineering hours for 25 story points' worth of items will also fit into the sprint. If that's *not* true, especially for multiple sprints in a row, then you need to revisit your use of story points for estimating.

To summarize, here are the key points:

- In sprint planning and management, you measure sprint capacity in terms of hours or days, and you estimate and assign engineering tasks to sprints in terms of hours or days.
- This is true regardless of whether the user stories, acceptance criteria, or PBIs were estimated in time units or story points.

15.5.4 Deeper Dive 15.4: Assessing the IT Team's Velocity

Velocity Expresses the work value of items of work that can be or are completed in a sprint (or other unit of time). "Work value" may be expressed as ideal days/hours or story points.

Throughout this chapter, in using burndown charts, we've focused on evaluating whether your team is on schedule, behind schedule, or ahead of schedule. That is, without saying so explicitly, we've been trying to assess your team's **velocity**: how quickly you can get work done and how well that corresponds to your estimates. More concisely, how many work units (in days, hours, or story points) can you complete in a sprint?

It's critically important for you to know how fast you accomplish work, for both completing the current sprint and planning future sprints. Can you complete stories during sprints about as fast as you planned? If so, you may be fundamentally on track to deliver the project successfully. However, if you find your velocity is consistently and significantly slower than originally planned, then your original estimates will turn out to be too low. If your estimates are generally too low, then that could blow up your project's budget, timing, and scope. Such issues can easily lead to outright project failure.

15.5.4.1 Velocity as a Measure of Stories and Acceptance Criteria Completed

In sprint-based development, you generally measure velocity via the work value of sprint backlog items completed in the sprint: user stories/PBIs and acceptance criteria that you finish. This is conceptually simple to evaluate via a burndown chart at the end of a sprint. For example, in Figure 15-3, we estimated that we could accomplish 100 hours of work across the two user stories and selected acceptance criteria. And, in Figure 15-8, we showed a sprint that fully accomplished all that work. In that case, velocity was simply 100 hours per sprint, the same as capacity.

Of course, you'd like to exceed your planned velocity. For example, in Figure 15-9, we finished all of the work, with a couple of team members finishing early. What if those team members were able to complete, say, an additional eight-hour task that hadn't originally been planned for the sprint? In that case, we'd be happy to report that our sprint velocity was 108 hours: the original 100 hours plus 8 more.

This seems simple, but things get more complicated when you finish behind schedule. For example, examine Figure 15-10. Here, our team finished the sprint well behind schedule. But clearly, we didn't do nothing. What's the velocity here? You might see three possible approaches:

1. **Difference between planned hours and hours remaining:** We planned 100 hours and ended up with 43 hours remaining. So is the velocity 100 – 43 = **57 hours**?
2. **Value of engineering tasks completed:** We finished three engineering tasks: CAM2 Task4, CAM3 Task1, and CAM3 Task2. They were estimated at a total of 40 hours. So is our velocity **40 hours**?
3. **Agile principle: Value of user stories/acceptance criteria completed:** Agile development principles argue that you should only count user stories and acceptance criteria that you *complete* in a sprint (Rubin, 2013, p. 133; Cohn, 2006, p. 117). Per Figure 15-3, we had two chunks of features defined for this sprint: (1) CAM2 base user story in combination with AC1 and AC2 and (2) CAM3 base user story. For us to have

			Sprint Day				
Story or Task	Dev	Start	Day 1	Day 2	Day 3	Day 4	Day 5
CAM2 Task1	Jesse	16	7	0	0	4	4
CAM2 Task2	Jesse	10	10	7	0	0	3
CAM2 Task3	Jesse	8	8	8	8	8	8
CAM2 Task4	Lee	12	10	8	10	12	14
CAM2 Task5	Lee	8	8	8	8	8	8
CAM2 Task6	Lee	6	6	6	6	6	6
CAM2 Task7	Lee	6	6	6	6	6	6
CAM3 Task1	Shakti	16	13	10	7	3	1
CAM3 Task2	Shakti	12	12	12	12	12	12
CAM3 Task3	Shakti	6	6	6	6	6	6
Estimated Hours Remaining		100	86	71	63	65	68
Ideal Trend Line		100	80	60	40	20	0

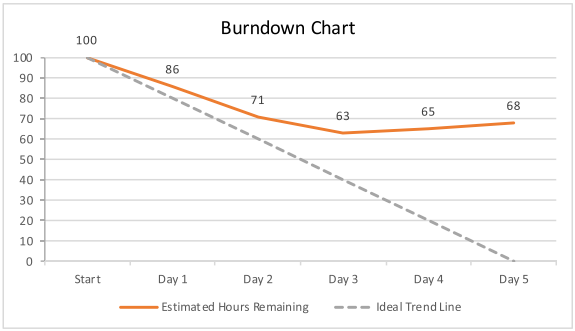


Figure 15-23 VFA Sprint 1 burndown chart, showing team not completing any engineering tasks

completed CAM2 with AC1 and AC2, we'd have to have completed *all seven* corresponding engineering tasks, which we didn't! Similarly, to have completed CAM3, we'd have to have completed all three of its engineering tasks, which we also didn't. So . . . *using agile principles*, our velocity for this sprint is **zero hours**!

To reiterate, the third approach is the standard agile development approach, so, based on that, we ended up with a zero velocity!

You may or may not decide to agree with this. For example, consider another VFA Sprint 1 example shown in Figure 15-23. This is based on Figure 15-10, but assuming that Lee encounters major problems on their first task (CAM2 Task4) and Shakti doesn't quite finish their first task either. So, in this version, the team doesn't complete a single engineering task.

Here's the point: in both Figure 15-10 and Figure 15-23, using agile principles we'd measure the team as having zero velocity because in neither situation did we finish an entire user story/AC combination. But, if you were leading the team, which situation would you rather find yourself in? Probably the one in Figure 15-10, because at least there we accomplished two things:

- Finished with 43 hours of work remaining (versus 68 hours in Figure 15-23)
- Finished several engineering tasks worth 40 hours (versus none in Figure 15-23)

So you may object to the agile approach. How do we make sense of this? The answer is to assess average team velocity over several sprints.

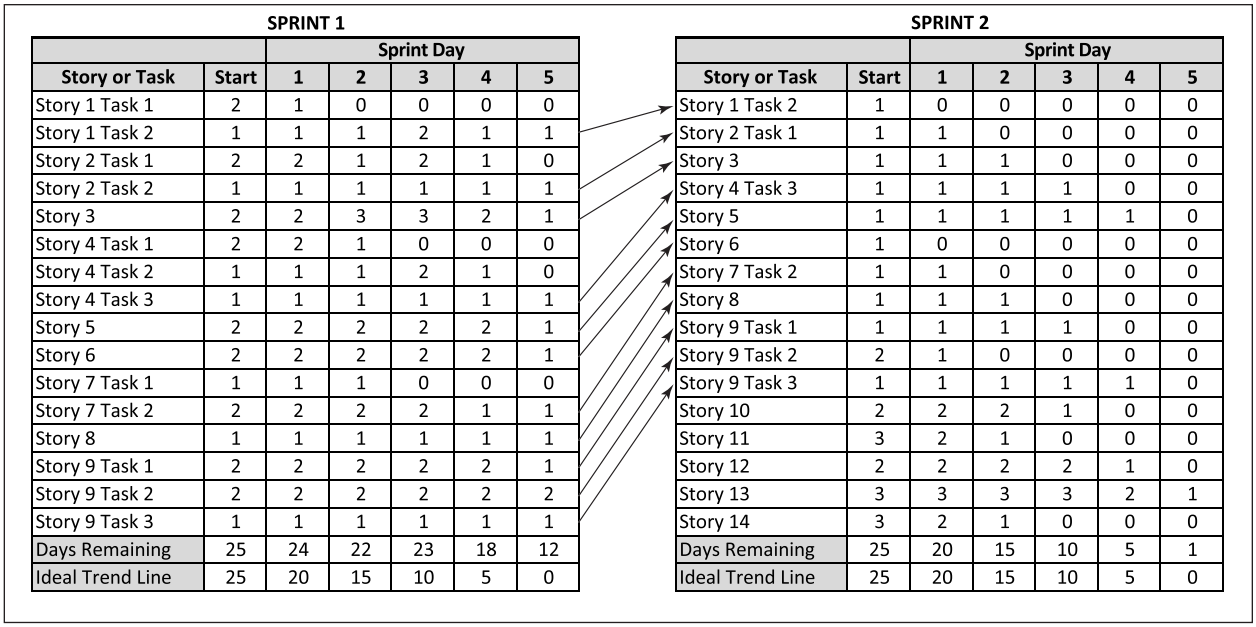
15.5.4.2 Calculating Average Velocity over Multiple Sprints

The examples we just covered illustrate that measuring velocity over only a single sprint can generate highly misleading velocity numbers. Instead, you should measure *velocity averaged over several consecutive sprints*.

To illustrate this, consider Figure 15-24. This shows tabular burndown chart data for one team over two consecutive sprints. In this example, the stories and tasks are estimated in ideal days, rather than the ideal hours shown in the VFA examples above. The capacity here is 25 days of work. For the sake of simplicity, this only shows user stories (not acceptance criteria within user stories).

As you review the example, consider the following:

- **Sprint 1 velocity = 0:** In Sprint 1, velocity using standard agile principles was zero, because, while the team did complete some engineering tasks, they didn't complete any user story in its entirety.
- **Sprint 2 planning incorporates remaining Sprint 1 work:** Again, for the sake of simplicity, we assume that all incomplete Sprint 1 tasks were pushed to Sprint 2. In each case, the estimated work corresponds to what was left at the end of Sprint 1. The team then included 13 days' worth of additional stories (starting with Story 10) to round out



SPRINT 1

		Sprint Day				
Story or Task	Start	1	2	3	4	5
Story 1 Task 1	2	1	0	0	0	0
Story 1 Task 2	1	1	1	2	1	1
Story 2 Task 1	2	2	1	2	1	0
Story 2 Task 2	1	1	1	1	1	1
Story 3	2	2	3	3	2	1
Story 4 Task 1	2	2	1	0	0	0
Story 4 Task 2	1	1	1	2	1	0
Story 4 Task 3	1	1	1	1	1	1
Story 5	2	2	2	2	2	1
Story 6	2	2	2	2	2	1
Story 7 Task 1	1	1	1	0	0	0
Story 7 Task 2	2	2	2	2	1	1
Story 8	1	1	1	1	1	1
Story 9 Task 1	2	2	2	2	2	1
Story 9 Task 2	2	2	2	2	2	2
Story 9 Task 3	1	1	1	1	1	1
Days Remaining	25	24	22	23	18	12
Ideal Trend Line	25	20	15	10	5	0

SPRINT 2

		Sprint Day				
Story or Task	Start	1	2	3	4	5
Story 1 Task 2	1	0	0	0	0	0
Story 2 Task 1	1	1	0	0	0	0
Story 3	1	1	1	0	0	0
Story 4 Task 3	1	1	1	1	0	0
Story 5	1	1	1	1	1	0
Story 6	1	0	0	0	0	0
Story 7 Task 2	1	1	0	0	0	0
Story 8	1	1	1	0	0	0
Story 9 Task 1	1	1	1	1	0	0
Story 9 Task 2	2	1	0	0	0	0
Story 9 Task 3	1	1	1	1	1	0
Story 10	2	2	2	1	0	0
Story 11	3	2	1	0	0	0
Story 12	2	2	2	2	1	0
Story 13	3	3	3	3	2	1
Story 14	3	2	1	0	0	0
Days Remaining	25	20	15	10	5	1
Ideal Trend Line	25	20	15	10	5	0

Figure 15-24 Assessing sprint velocity using multiple sprints

Sprint 2 to its 25-day capacity. (Alternatively, we may have had to push other stories already planned for Sprint 2 to a later sprint.)

- **Average velocity over Sprints 1 and 2 = 17.5 days:** By the end of Sprint 2, we completed all of the stories (Stories 1 through 9) originally planned for Sprint 1—worth 25 days. We also completed new Sprint 2 Stories 10, 11, 12, and 14, collectively worth 10 days. So, the average velocity of the team across the two sprints was 25 days plus 10 days divided by 2 sprints equals 17.5 days.

This 17.5-day average velocity is quite different from the zero days per sprint velocity measured at the end of Sprint 1! The key insight to draw from this is that sprint velocity should only be measured over multiple sprints averaged together. Ideally, you should average over at least the last three sprints.

15.5.5 Deeper Dive 15.5: Effects of Technical Debt on Velocity

Technical debt Problems in an existing application that accumulate over time. Typically results from cutting corners on design, architecture, coding, and testing. Generally creates extra work for developers that reduces the amount of work on new features that a team can do.

So far in this chapter, we've focused on failing to achieve expected velocity because of your IT team's issues (things like underestimating features, lacking the right technical skills, and so on).

But behind-schedule sprint performance may arise from another, outside factor: **technical debt** in an existing system that you're working to enhance with new features. What is technical debt? It's a term that refers to problems in the existing application and technical environment that have accumulated over time because of poor architecture, poor application design, inadequate testing and other reasons (Figure 15-25).

Technical debt includes problems in the existing application code (defects, bad design, inadequate architecture, and the like) that the team may discover and then need to work to correct during a sprint. This is called "paying on the technical debt." Examples might include:

- A developer needs to stop working on Story 2 to fix a high-severity bug that they discovered in the existing code.
- A security test may detect a potential flaw in the existing application, requiring the team to divert their efforts to first plug that security flaw.
- A developer may try to reuse some existing functionality in the application, only to find old, poorly designed code that must be refactored before it can be extended.

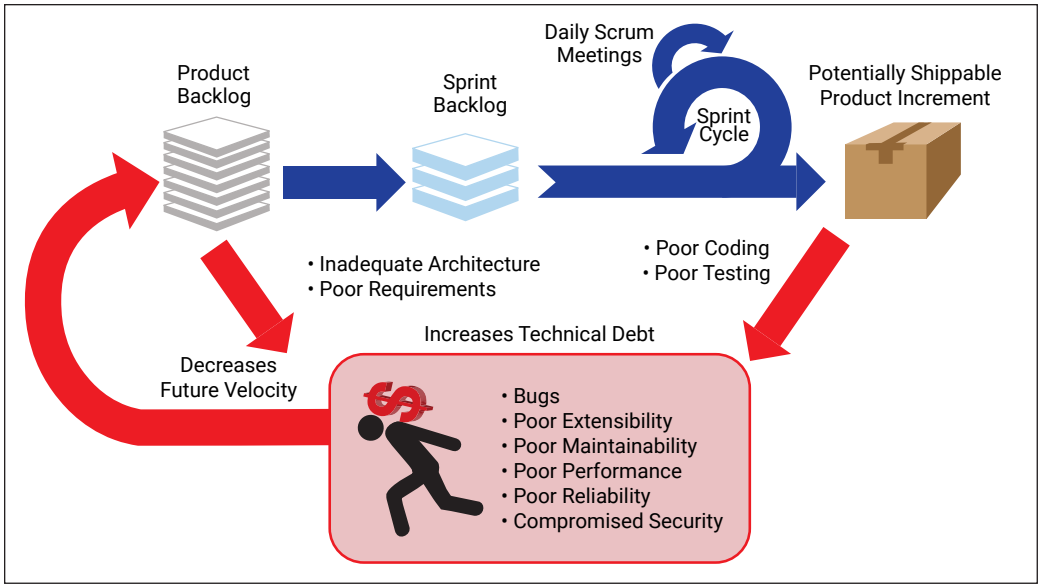


Figure 15-25 Sources and effects of technical debt (Icon courtesy of Creative Commons, https://ccsearch.creativecommons.org/photos/761174d5-bf6f-41f0-a411-bb5da88615a7)

Importantly, these items likely wouldn't have been included in your original estimates, engineering tasks, or sprint plan. Put plainly, they represent additional work that you didn't plan or budget for.

Any technical debt that you didn't plan for will reduce your velocity and your ability to meet overall project goals. So even if your team creates accurate estimates for creating new features, unanticipated technical debt can slow your velocity in each sprint. Technical debt robs you of time you could have spent on creating new features. This effort, in which you spend time and money to deal with existing technical debt issues, is a bit like having to pay interest on a monetary loan; hence the name: technical debt.

While you may suffer from needing to *pay* on the technical debt, you also need to avoid *adding* to the technical debt. As shown in Figure 15-25, this can quickly become a vicious cycle: First, you may already be operating at reduced velocity because of existing technical debt. Second, this may cause you to cut corners on new features in an effort to stay on schedule, while also not creating robust, flexible designs; not doing adequate testing; and so on. Cutting corners will *add* to the existing technical debt and compound your problems going forward.

If you're caught in this situation, you need to recognize it and speak frankly with leadership. You may need to pause the project and spend additional money and time to first fix the underlying problems in the existing software (that is, to *pay down* the principal of the technical debt) rather than just address technical debt problems as you encounter them.

In some cases, technical debt may be so great that the application may need to be scrapped. In that case, it may be better to do a total rewrite of the system or replace it with a commercial-off-the-shelf software product. In less severe situations, the team will ideally budget "technical debt reduction" stories that can be included in sprint planning and the overall project budget.

Chapter 16

Deployment

16.5 Deeper Dives: DevOps Deployment in an Age of Options

If you consider traditional deployment alternatives like direct deployment, parallel deployment, pilot deployment, and phased deployment, you'll realize that you lived in an "age of options" even before DevOps. The advent of DevOps techniques has exploded the range of possibilities. With DevOps, the number of new techniques to learn about and implement is broad and challenging. Although it's potentially revolutionary, you need to think carefully about how to move to DevOps, especially with regard to the frequent production releases involved in DevOps continuous deployment.

In this section's Deeper Dives, we consider several topics affecting the move to DevOps:

- **Deeper Dive 16.1** explores how changes in the nature of IT projects led to DevOps becoming a realistic option for many organizations. DevOps is a recent development, and a major reason for that is that in decades past the nature of IT projects was incompatible with the DevOps approach.
- **Deeper Dive 16.2** takes an in-depth look at how to move from traditional deployments to continuous integration, then continuous delivery, and, finally, continuous deployment.
- **Deeper Dive 16.3** explores the many techniques that teams use to successfully implement DevOps continuous deployment.

16.5.1 Deeper Dive 16.1: How the Evolution of IT Projects Made DevOps Possible

Agile development is a relatively new concept. It didn't start gaining traction until the late 1990s and early 2000s. By that time, business software systems had already existed for several decades.

But DevOps is even newer: some of the key ideas didn't emerge publicly until 2008 and 2009 (Kim et al., 2016). As such, many teams today still don't use DevOps or, if they do, don't take DevOps to its logical conclusion—updating changes into production very rapidly, perhaps even daily.

Why is that the case? Why has DevOps just started to gain traction now? At least two factors are involved. First, DevOps leverages agile development concepts, and it has taken awhile for agile development to become a major feature of development projects.

Second, DevOps, like all agile concepts, works best in small, relatively simple situations. However, unlike agile development, DevOps is an idea that operates closer to the overall project release level. Until recently, many systems projects were simply too big and complex to be agile in that way.

This point was made by Adrian Cockcroft, a leader at innovative technology companies such as eBay, Netflix, and Amazon Web Services. In an influential presentation in 2013, Cockcroft explained that systems projects, in general, have changed dramatically over the past several decades. In particular, they've tended to become smaller, faster, and, so, less risky. Table 16-1 shows how systems projects have evolved since before the 1980s to now (as cited in Kim et al., 2016).

In the mainframe era, projects were massive, often costing many millions of dollars and lasting for years on end. This was the era of "pure" plan-driven (traditional SDLC, or "waterfall") development. Because of the waterfall approach, huge amounts of software were produced by developers before customers saw any demonstration of it. Not surprisingly, this led to major misunderstandings, with customers in effect saying, "Yes, that's what I asked for, but that's not (or no longer) what I really

Table 16-1: Changing characteristics of projects and technologies by computing eras			
Computing Era	**Mainframe (through 1980s)**	**Client/Server (1990s)**	**Web/Cloud (2000–present)**
Project:			
• **Duration**	1 to 5 years	3 months to 1 year	2 weeks to 3 months
• **Budget**	$1M to $10M	$100K to $10M	$10K to $1M
• **Bet level**	Entire company	Product line or division	Product feature
• **Cost of failure**	• Bankruptcy • Bought out	• Revenue hit • Stock price decline • CIO job loss	• Team leader(s) reputation
Technologies:			
• **Language**	COBOL	C++	Java, PHP, Ruby, C#
• **DBMS**	DB2	Oracle	MySQL and NoSQL
• **Infra-structure**	Mainframe	Client-server	Virtual machines and cloud
adapted from Cockcroft, 2013, and Kim et al., 2016			

need," or, more briefly, the "Yes, but" syndrome that we introduced in Chapter 2. With IT projects betting this much time and money, project failure might mean failure of the entire company!

Things began to change in the 1990s, when the move from mainframes to servers broke up applications into smaller chunks. That, combined with better tools, led to smaller and shorter projects. Agile approaches, and especially agile construction, started to emerge, leading to somewhat faster deliveries that were also more frequently on target.

As we headed into the 2000s to the present, the agile movement picked up steam, with project scope continuing to fall into smaller and smaller chunks. Smaller, shorter projects were inherently less risky.

To quickly summarize, these changes are partially an outcome of the general maturing of information technologies, including both hardware and software architectures. However, such changes are more recently a result of the influence of agile development practices themselves. Overall, this trend to smaller, less risky projects helps explain why organizations can and need to speed up the delivery of new functionality. So, specifically with respect to deployments, you can see how industry trends have helped create an environment where DevOps could become a realistic possibility.

16.5.2 Deeper Dive 16.2: The Technical Path to DevOps

16.5.2.1 The Starting Point: Agile Development with Traditional Deployment

In particular, let's focus on the "Traditional Approach" column of Figure 16-7. More specifically, let's consider this column in the context of a team that's already "agile," at least in the sense that it constructs changes using construction sprints, which we've described as now highly common practice across the industry.

You've seen that agile development uses sprints to deliver frequent feature updates, typically every one to four weeks. Ideally, each sprint delivers complete features, rather than just pieces of features that won't work because they're incomplete. By delivering code frequently, we keep changes small, making testing less onerous. As such, we show Development in that first "Traditional Approach" column as being orange, meaning agile and, so, compatible with DevOps.

Why can't you deliver those frequent development changes into production using the traditional approach? Because the agility we just described in the previous paragraph is limited to the first box in the process: Development. Once those features are developed and unit tested by developers, they then need to be merged and integrated into a test/QA environment so you can conduct integration testing, regression testing, system testing, and so on (per Chapter 5). In the traditional approach, deciding what to update and when is a manual process. So features may sit around for several days, waiting to be promoted to test/QA. This same story is repeated for the UAT and

production environments. In general, manual merge/integration processes introduce significant process inconsistencies, errors, and delays. When testing does happen, it's highly manual, and feedback to the developers takes a long time.

Given this situation, you might think that the first step in moving toward DevOps would be to automate the merge/integration process and begin introducing testing automation. And you'd be right!

16.5.2.2 The First Step Toward DevOps: Continuous Integration

Continuous integration (CI) Automatic merging and integration of code updates into an environment with significant integration and regression testing.

The first step toward DevOps is **continuous integration (CI)**. Here, special utility software merges and integrates all updates from development to the QA/test environments automatically every day. This is enabled using automated integration testing (Do the changes work together?) and regression testing (Do the changes break any existing functionality?). Automatic, repeatable code merging and integration is a function that many teams have already accomplished. In addition, more and more teams are beginning to use automated testing tools. Figure 16-7 shows these initial changes in the second column.

However, in CI, updates to UAT typically are still manual and infrequent. Users may see new software demonstrated by developers during sprint reviews. However, they often have to wait to test the new code directly themselves in staging/UAT. In part, this is because staging/UAT environments are kept closer to a "production-like" state than test/QA environments. More specifically, staging/UAT will typically be fairly frequently refreshed with data from the production environment. This makes it a more realistic test environment. Also, test/QA tends to be less stable than staging/UAT, simply because it's the place where IT makes frequent changes to the code while finalizing the changes. In addition, CI may not fully implement automated system testing (per Chapter 5, system testing determines if the new features meet the requirements) in test/QA and generally won't implement it at all in staging/UAT. As a result, you still need to do a significant amount of manual system testing of each new feature.

16.5.2.3 Continuous Delivery: Routinely Deploying Tested Code into "Production-Like" Environments

Continuous delivery (CDE) The extension of continuous integration to include automatic merging and integration of code into production-like environments.

The next step in the DevOps journey is **continuous delivery (CDE)**. Here, you deploy code automatically and frequently into production-like testing environments. In this process, the terminology and, to an extent, the environments begin to change. Rather than development/test-QA/staging-UAT/production "environments," many teams will speak of development **branches**. An example of this is shown in Figure 16-9. Here, the horizontal lines represent evolving lines of code. Each line roughly corresponds to the old term "environment."

Branch A modern term roughly equivalent to the older term "environment." Refers to a complete copy of the code that may be updated with new, tracked versions over time.

Trunk or mainline branch A branch that contains all unit-tested code changes checked in by developers. Used for integration and regression testing.

The main development environment is now called the **trunk or mainline branch**. It contains all checked-in code changes and is used for integration and regression testing. Rather than checking

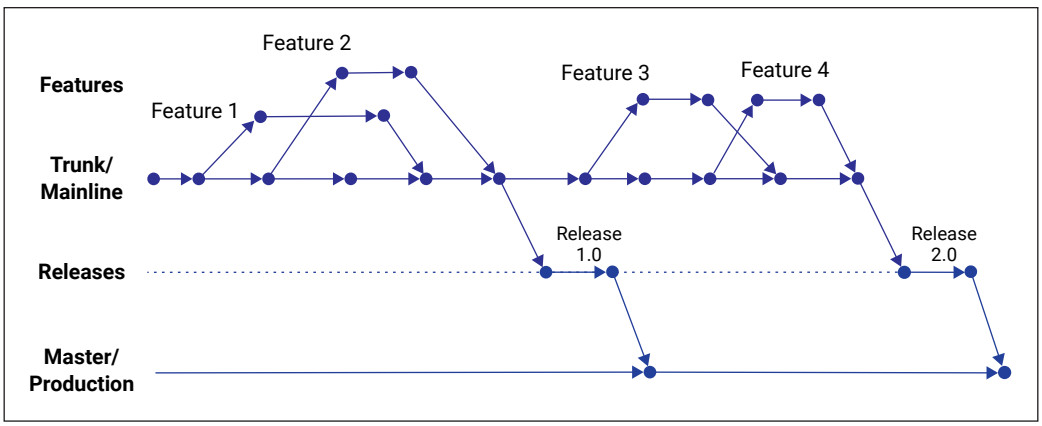


Figure 16-9 Example branching diagram for a continuous delivery environment

out files from a development environment, developers will create temporary copies of the entire code base to create changes. Each such **feature branch** should exist for only a few days, while the developer creates a small feature (read: user story) and unit tests it. As soon as that's done, the feature is merged into the trunk/mainline, at which point all the automated integration and regression tests are run. If the feature passes, then the code is accepted. If it fails, it's rolled back immediately. This feature branch approach prevents developers from doing too much work apart from each other. As a result, it helps minimize breakage as the code is merged.

Feature branch A branch created for a short amount of time to create system changes for a feature. The branch ends after the new feature is checked into the trunk/ mainline branch.

In this way, the trunk/mainline is kept in a state that allows it to update frequently and automatically to a production-like **release branch**, which roughly corresponds to the traditional staging/UAT environment. Here, automated system and user acceptance tests are run to ensure the code can be released to the **master (or production) branch**, which corresponds to the production environment.

Release branch A branch that contains code undergoing final testing before updating the master/ production branch.

Master/production branch A branch containing the code running in production.

CDE is really an extension of automated testing that goes beyond the automated integration and regression testing seen in CI by also including user acceptance testing. CDE often will allow you to promote changes to production more frequently than with traditional deployment approaches. However, though you theoretically could then push those changes to production in an automated manner, with CDE you stop short of that. Updates to production are still a manual process. The level of risk is still just too great.

To update production in a highly frequent and fully automated fashion, you need to take the next step, which is a big one: continuous deployment.

16.5.2.4 Continuous Deployment: Releasing Features at the Speed of Agile

With the final step in the DevOps journey (**continuous deployment**, or **CD**) you begin to deploy very frequently. This often (but not always) happens on a fully automated basis every time code is checked in to the trunk/mainline branch and passes a wide range of automated tests. Revisiting Figure 16-9, in many cases, this results in the merging of integration and user acceptance testing environments into a single trunk/mainline branch. This is a logical extension of the coordination and leveraging of IT system testing and business user acceptance testing described in Chapter 5.

Continuous deployment (CD) Extension of continuous delivery to include automated, frequent deployments to the production environment.

Put bluntly, once a developer checks in code and that code passes a series of automated tests, it can be deployed immediately into production.

The idea of continuous deployment is shown in Figure 16-1, which shows a production release at the end of every sprint. In a situation in which CD has been fully implemented, Figure 16-1 may understate the case: changes may be implemented every day!

For anyone who has deployed major changes into production, this sounds risky to the point of being scary, maybe even crazy! How can you make this happen without risking disastrous consequences?

The general answer is that you need to implement several types of advanced techniques that help reduce deployment risk. However, this statement doesn't do the needed changes justice. In many cases, getting to CD requires a transformational reinvention of the development process. In some cases, it requires a transformation of the application itself.

16.5.2.5 The Paradox of DevOps: Agile Deployment Takes a Lot of Planning!

We've described DevOps as extending agile development concepts to deployment. However, with respect to planning, there seems to be a qualitative difference between agile development and agile deployment.

More specifically, when we've talked about moving from plan-driven development to agile development, we've often discussed doing less up-front planning. In agile development, we tend to emphasize a more informal, highly responsive way of working, where adjustments to requirements can be done on-the-fly.

In contrast, DevOps at the CD level requires even more planning than traditional deployments, including new, advanced techniques for requirements and deployment. We'll explore those advanced techniques in Deeper Dive 16.3.

16.5.3 Deeper Dive 16.3: De-Risking DevOps with Advanced Requirements and Deployment Techniques

As we've discussed requirements analysis throughout this book, we've focused on several key requirements dimensions. They include functional design factors such as the business process, data, logic, and user interface. They also include nonfunctional factors such as reliability, maintainability, scalability, and security.

However, we haven't had to worry too much about the sequence of deployment. Our product backlog and prioritization (e.g., using the MoSCoW model) has served to define a minimum (MVP) and maximum scope that, as we execute sprints, converge on a single set of features. This feature set implies a set of components (database entities, classes, pages, and more) that we generally can deploy and "turn on" together during a traditional deployment.

Contrast this with DevOps, especially with continuous deployment, where you need to be able to deploy feature components immediately to production, but in some cases in a piecemeal fashion and with the ability to selectively turn on components (e.g., feature flags and toggles) and/or replace old versions with new versions (e.g., versioning and evolutionary databases).

These factors represent a new requirements dimension: deployment requirements. This is a new area that the BA will need to address in conjunction with developers, especially during technical designs. We don't have the space to cover all these techniques, but we can outline them here.

We described that journey at a high level in Deeper Dive 16.2. By reviewing Figure 16-7, you can see that each stage builds on the improvements from the prior stages. The improvements at the CI level set the stage for the CDE level, and the CDE level sets the stage for the full CD level. These improvements generally focus on capabilities such as automated, repeatable code updates; automated testing; rearchitecting the application; and ways of deploying large updates to production in smaller, safer ways.

Table 16-2 summarizes those key changes at each stage of DevOps (Gruver & Mouser, 2015; Kim et al., 2016).

Table 16-2: DevOps increasing capabilities, progressing beyond traditional development to CI then CDE then CD

DevOps Level			Capability	Specific Techniques
CI	CDE	CD		
✓	✓	✓	Automated deployments	Automated build tools On-demand environment/branch creation Production-like code and data everywhere Version control for all environments/branches Automated integration tests
✓	✓	✓	Automated integration tests	Integration tests Regression tests
	✓	✓	Automated functionality tests	System tests User acceptance tests Performed in a "production-like" environment
		✓	Rearchitect application	Paid-down technical debt Refactor from monolithic to microservices
		✓	Automated security tests	Static analysis Dynamic analysis Penetration testing Dependency scanning Source code integrity and signing
		✓	Enable piecemeal feature deployment	Versioning Feature flags/toggles Evolutionary databases
		✓	Promote safer deployments	Dark launches with special users Blue-green deployments Canary releases Cluster immune system

based on Gruver and Mouser, 2015 and Kim et al., 2016

We don't have space to fully cover all of these techniques, but we can provide sketch descriptions of them:

- **Reducing the likelihood of deploying bad updates to production:** In CI, you've already implemented automated tests for integration and regression. Moving from CI to CDE, you add automated systems and user acceptance tests. However, even these tests will typically not be enough to move to CD. Other techniques needed include the following:
  - **Automated security tests:** We obviously live in a world where cybersecurity is a paramount concern. To implement CD, you need to ensure that a whole series of security tests are executed automatically for every production deployment. This includes a wide range of automated tests (which typically are implemented using packaged software):
    - **Static analysis:** This examines the source code to look for bad coding practices, "back doors" that enable users (or developers) to circumvent regular security, and other malignant code.
    - **Dynamic analysis:** This tests the security of the code while it's executing. It includes the idea of simulating attacks using "penetration testing."
    - **Dependency scanning:** This looks at third-party software components and utilities that the application may depend on. The idea is to ensure that the latest versions of these components are used and that they contain no known vulnerabilities.
  - **Paying down technical debt:** As described in Chapter 15, an application that is full of bugs and bad design has a high level of technical debt. High technical debt increases the likelihood that deploying changes will create new problems. As such, before moving to CD, it's a good idea to spend time fixing the application by "paying down" technical debt.
  - **Rearchitecting the application:** This is related to but distinct from the previous bullet. It involves having to break a monolithic application into smaller chunks, such as that seen in a microservices architecture approach (per Chapter 13). By rearchitecting an application into microservices, you turn one big, complicated software project into several smaller, simpler projects (one for each microservice). This reduces the likelihood of a change breaking other code. It also tends to restrict the impact of defects when they do find their way into production to a single microservice.
- **Reducing the impact of deploying bad updates to production:** Even if you use all of the techniques we've listed, you can never be entirely certain that you won't deploy problem code into production. In addition, for larger features, you may not be able to deploy the entire feature at once. Instead, you may end up having to deploy pieces or components of a feature corresponding to different engineering tasks over several deployments. Here, you should do two things: (1) not "turn on" the feature until all the pieces are in place and (2) not break any existing code in the meantime. Doing so requires using several advanced techniques:
  - **Feature flags or toggles:** This means coding changes with logic that allows them to be deployed to production without actually being turned on. Instead, you create requirements that construct features that can be turned on or off via a flag (e.g., setting a system configuration file value to "Feature1On == false," with the code checking for "Feature1On == true" before running each component). Sometimes, these flags are called "toggles." If you use them, at the time of deployment, you can deploy the feature without immediately impacting users. This allows you to do:
    - **Dark launches:** With the feature deployed but turned off, regular users won't see it. But testers (sometimes called "special users") can verify that the feature is working in Production before turning it on.

    - **Piecemeal deployment:** By deploying features that are turned off, you can deploy several components for one feature (or several related complete features) into production over time. When all the pieces are in place, you can test that they all work together before turning them on.
- **Versioning:** This is somewhat similar in concept to feature flags. In versioning, if you have an existing class that's working correctly, you don't directly replace that class in production. Instead, you keep the old class and deploy the new version beside it. Once you verify that the new version works, you can switch over to it and remove the old version.
- **Evolutionary database:** This is essentially the same concept as versioning applied to database entities. Instead of modifying a table's existing columns and other properties, you add a new version of the table with the schema changes (e.g., adding a new column rather than modifying an old column).
- **Safe deployment patterns:** You can use several approaches to deploy features to (or turn them on in) production and recover gracefully if they don't work. They include:
    - **Blue-green deployment:** In this pattern, you have two different Production environments: blue and green. Only one of them is active at any given time. If the blue environment is active, you deploy changes to the green one. You can then do testing there. Once you're confident, you switch from the blue environment to the green. If you still encounter problems, you simply switch back to the blue.
    - **Canary deployment:** In this pattern, you cluster users into groups of servers. For example, on a retail website, all the internal employees could be on server cluster 1, with outside customers on server cluster 2. You can first deploy the updates to cluster 1, where your employees act as canaries in the coal mine. If they "die," you can roll back the changes before they impact your customers. If they "live," you can implement to the remaining servers supporting the outside customers.
    - **Cluster immune deployment:** This is an extension of the canary deployment approach. Here, you implement system monitoring based on known performance statistics. For example, on a retail website, you may know that, on average, customers view five products per visit and place two of them into their carts. If you deploy a release and those statistics fall by a predetermined amount, the system would infer that something is wrong and automatically roll back to the old version.

For additional, more detailed information, see the resources listed at the end of this chapter (especially Kim et al., 2016, and Gruver & Mouser, 2015).

To briefly summarize, here are several observations: First, these techniques are diverse, complex, and far-reaching. This is especially true as you move to full, continuous deployment. In the topics we've covered, we've only hinted at the full richness and complexity of the techniques.

Second, these DevOps techniques are not totally foreign or unrelated to the "traditional deployment" approaches described earlier. For example, rearchitecting an application to microservices and then deploying features specific to each microservice is an improved form of phased deployment. Similarly, consider using feature flags/toggles/evolutionary databases to enable dark launches tested by a few special users, or canary deployments where you turn on the changes for, say, internal employees first (before external customers). In each of these cases, rolling out the changes to a small subset of users could be considered an advanced form of pilot deployment.

Finally, note that some of these techniques are simply good ideas, albeit significant investments, for any team. In particular, automated testing represents a big commitment: you need to select a testing tool, train your team to use it, and then work to create and maintain a bank of robust, reusable tests in the areas of integration, system, regression, and (ultimately) user acceptance. As noted previously, this can also require the automation of sophisticated security testing. In years past, the level of this commitment has kept many teams from moving to automated testing. But with the dawn of DevOps, automated testing is increasingly an idea whose time has come.