Chapter

6

USE-CASE MODEL: WRITING REQUIREMENTS IN CONTEXT

The indispensable first step to getting the things you want out of life: decide what you want.

—Ben Stein

Objectives

- Identify and write use cases.
- Relate use cases to user goals and elementary business processes.
- Use the brief, casual, and fully dressed formats, in an essential style.
- Relate use case work to iterative development.

Introduction

This is worth studying during a first read of the book because use cases are a widely used and useful mechanism to discover and record requirements (especially functional); they influence many aspects of a project, including OOA/D. It is worth both knowing about and creating use cases.

Writing use cases—stories of using a system—is an excellent technique to understand and describe requirements. This chapter explores key use case concepts and presents sample use cases for the NextGen application.

The UP defines the **Use-Case Model** within the Requirements workflow. Essentially, this is the set of all use cases; it is a model of the system's functionality and environment.

6.1 Goals and Stories

Customers and end users have goals (also known as *needs* in the UP) and want computer systems to help meet them, whether as mercantile as recording sales or as complex as estimating the flow of oil from future wells. There are several ways to capture these goals and system requirements; the better ones are simple and familiar because this makes it easier—especially for customers and end users—to contribute to their definition or evaluation. That lowers the risk of missing the mark.

Use cases are a mechanism to help keep it simple and understandable for all stakeholders. Informally, they are stories of using a system to meet goals. Here is an example *brief format* use case:

> **Process Sale**: A customer arrives at a checkout with items to purchase. The cashier uses the POS system to record each purchased item. The system presents a running total and line-item details. The customer enters payment information, which the system validates and records. The system updates inventory. The customer receives a receipt from the system and then leaves with the items.

Use cases often need to be more elaborate than this, but the essence is discovering and recording functional requirements by writing stories of using a system to help fulfill various stakeholder goals; that is, *cases of use*.[1] It isn't supposed to be a difficult idea, although it may indeed be difficult to discover or decide what is needed, and write it coherently at a useful level of detail.

Much has been written about use cases, and while worthwhile, there is always the risk among creative, thoughtful people to obscure a simple idea with layers of sophistication. It is usually possible to spot a novice use case modeler (or a serious Type A analyst) by an over-concern with secondary issues such as use case diagrams, use case relationships, use case packages, optional attributes, and so forth, rather than writing the stories. That said, a strength of the use case mechanism is the capacity to scale both up and down in terms of sophistication and formality, depending on need.

6.2 Background

The idea of use cases to describe functional requirements was introduced in 1986 by Ivar Jacobson [Jacobson92], a main contributor to the UML and UP.

1. The original term in Swedish literally translates as "usage case."

Jacobson's use case idea was seminal and widely appreciated; simplicity and utility being its chief virtues. Although many have made contributions to the subject, arguably the most influential, comprehensive, and coherent next step in defining what use cases are (or should be) and how to write them came from Alistair Cockburn, summarized in the very popular text *Writing Effective Use Cases* [Cockburn01], based on his earlier work and writings stemming from 1992 onwards. This introduction is therefore based upon and consistent with the latter work.

6.3 Use Cases and Adding Value

First, some informal definitions: an **actor** is something with behavior, such as a person (identified by role), computer system, or organization; for example, a cashier.

A **scenario** is a specific sequence of actions and interactions between actors and the system under discussion; it is also called a **use case instance**. It is one particular story of using a system, or one path through the use case; for example, the scenario of successfully purchasing items with cash, or the scenario of failing to purchase items because of a credit card transaction denial.

Informally then, a **use case** is a collection of related success and failure scenarios that describe actors using a system to support a goal. For example, here is a *casual format* use case that includes some alternate scenarios:

> **Handle Returns**
>
> *Main Success Scenario*: A customer arrives at a checkout with items to return. The cashier uses the POS system to record each returned item ...
>
> *Alternate Scenarios*:
>
> If the credit authorization is reject, inform the customer and ask for an alternate payment method.
>
> If the item identifier is not found in the system, notify the Cashier and suggest manual entry of the identifier code (perhaps it is corrupted).
>
> If the system detects failure to communicate with the external tax calcuator system, ...

An alternate, but similar definition of a use case is provided by the UP:

> A set of use-case instances, where each instance is a sequence of actions a system performs that yields an observable result of value to a particular actor [RUP].

The phrasing *"an observable result of value"* is subtle but important, because it stresses the attitude that the system behavior should emphasize providing value to the user.

> A key attitude in use case work is to focus on the question "How can using the system provide observable value to the user, or fulfill their goals?", rather than merely thinking of system requirements in terms of a "laundry list" of features or functions.

Perhaps it seems obvious to stress providing observable user value, but the software industry is littered with failed projects that did not deliver what people really needed. The feature and function list approach to capturing requirements can contribute to that negative outcome because it does not encourage the stakeholders to consider the requirements in a larger context of using the system in a scenario to achieve some observable result of value, or some goal. In contrast, use cases place features and functions in a goal-oriented context. Hence the chapter title.[2]

This is a key idea that Jacobson was trying to convey in the use case concept: do requirements work with a focus on how a system can add value.

6.4 Use Cases and Functional Requirements

Use cases are requirements; primarily they are functional requirements that indicate what the system will do. In terms of the FURPS+ requirements types, they emphasize the "F" (functional or behavioral), but can also be used for other types, especially when those other types strongly relate to a use case. In the UP—and most modern methods—use cases are the central mechanism that is recommended for their discovery and definition. Use cases define a promise or contract of how a system will behave.

To be clear: Use cases *are* requirements (although not all requirements). Some think of requirements only as "the system shall do..." function or feature lists. Not so, and a key idea of use cases is to (usually) reduce the importance or use of detailed older-style feature lists and rather, write use cases for the functional requirements. More on this point in a later section.

Use cases are text documents, not diagrams, and use case modeling is primarily an act of writing, not drawing. However, the UML defines a use case diagram to illustrate the names of use cases and actors, and their relationships.

---

2. Originally from the aptly titled *Uses Cases: Requirements in Context* [GK00] (chapter title adapted with permission of the authors).

6.5 Use Case Types and Formats

Black-Box Use Cases and System Responsibilities

Black-box use cases are the most common and recommended kind; they do not describe the internal workings of the system, its components, or design. Rather, the system is described as having *responsibilities*, which is a common unifying metaphorical theme in object-oriented thinking—software elements have responsibilities and collaborate with other elements that have responsibilities.

By defining system responsibilities with black-box use cases, it is possible to specify *what* the system must do (the functional requirements) without deciding *how* it will do it (the design). Indeed, the definition of "analysis" versus "design" is sometimes summarized as "what" versus "how." This is an important theme in good software development: During requirements analysis avoid making "how" decisions, and specify the external behavior for the system, as a black box. Later, during design, create a solution that meets the specification.

Black-box style	Not
The system records the sale.	The system writes the sale to a database. ...or (even worse) : The system generates a SQL INSERT statement for the sale...

Formality Types

Use cases are written in different formats, depending on need. In addition to the black-box versus white-box *visibility* type, there are varying degrees of *formality*:

- **brief—**terse one-paragraph summary, usually of the main success scenario. The prior *Process Sale* example was brief.
- **casual**—informal paragraph format. Multiple paragraphs that cover various scenarios. The prior *Handle Returns* example was casual.
- **fully dressed**—the most elaborate. All steps and variations are written in detail, and there are supporting sections, such as preconditions and success guarantees.

The following example is a fully dressed case for our NextGen case study:

6.6 Fully Dressed Example: Process Sale

Fully dressed use cases show more detail and are structured; they are useful in order to obtain a deep understanding of the goals, tasks, and requirements. In the NextGen POS case study, they would be created during one of the early requirements workshops in a collaboration of the system analyst, subject matter experts, and developers.

The usecases.org Format

Various format templates re available for fully dressed use cases. However, perhaps the most widely used and shared format is the template available at *www.usecases.org*. The following example illustrates this style.

Please note that this is the primary case study example of a detailed use case; it shows many common elements and issues.

Use Case UC1: Process Sale

Primary Actor: Cashier
Stakeholders and Interests:
– Cashier: Wants accurate, fast entry, and no payment errors, as cash drawer shortages are deducted from his/her salary.
– Salesperson: Wants sales commissions updated.
– Customer: Wants purchase and fast service with minimal effort. Wants proof of purchase to support returns.
– Company: Wants to accurately record transactions and satisfy customer interests. Wants to ensure that Payment Authorization Service payment receivables are recorded. Wants some fault tolerance to allow sales capture even if server components (e.g., remote credit validation) are unavailable. Wants automatic and fast update of accounting and inventory.
– Government Tax Agencies: Want to collect tax from every sale. May be multiple agencies, such as national, state, and county.
– Payment Authorization Service: Wants to receive digital authorization requests in the correct format and protocol. Wants to accurately account for their payables to the store.
Preconditions: Cashier is identified and authenticated.
Success Guarantee (postconditions): Sale is saved. Tax is correctly calculated. Accounting and Inventory are updated. Commissions recorded. Receipt is generated.

Main Success Scenario (or Basic Flow):
1. Customer arrives at POS checkout with goods and/or services to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. System records sale line item and presents item description, price, and running total. Price calculated from a set of price rules.

Cashier repeats steps 3-4 until indicates done.

5. System presents total with taxes calculated.

6. Cashier tells Customer the total, and asks for payment.
7. Customer pays and System handles payment.
8. System logs completed sale and sends sale and payment information to the external Accounting system (for accounting and commissions) and Inventory system (to update inventory).
9. System presents receipt.
10. Customer leaves with receipt and goods (if any).

Extensions (or Alternative Flows):

*a. At any time, System fails:
   To support recovery and correct accounting, ensure all transaction sensitive state and events can be recovered from any step of the scenario.
   1. Cashier restarts System, logs in, and requests recovery of prior state.
   2. System reconstructs prior state.
      2a. System detects anomalies preventing recovery:
         1. System signals error to the Cashier, records the error, and enters a clean state.
         2. Cashier starts a new sale.

3a. Invalid identifier:
   1. System signals error and rejects entry.

3b. There are multiple of same item category and tracking unique item identity not important (e.g., 5 packages of veggie-burgers):
   1. Cashier can enter item category identifier and the quantity.

3-6a: Customer asks Cashier to remove an item from the purchase:
   1. Cashier enters item identifier for removal from sale.
   2. System displays updated running total.

3-6b. Customer tells Cashier to cancel sale:
   1. Cashier cancels sale on System.

3-6c. Cashier suspends the sale:
   1. System records sale so that it is available for retrieval on any POS terminal.

4a. The system generated item price is not wanted (e.g., Customer complained about something and is offered a lower price):
   1. Cashier enters override price.
   2. System presents new price.

5a. System detects failure to communicate with external tax calculation system service:
   1. System restarts the service on the POS node, and continues.
      1a. System detects that the service does not restart.
         1. System signals error.
         2. Cashier may manually calculate and enter the tax, or cancel the sale.

5b. Customer says they are eligible for a discount (e.g., employee, preferred customer):
   1. Cashier signals discount request.
   2. Cashier enters Customer identification.
   3. System presents discount total, based on discount rules.

5c. Customer says they have credit in their account, to apply to the sale:
   1. Cashier signals credit request.
   2. Cashier enters Customer identification.
   3. Systems applies credit up to price=0, and reduces remaining credit.

6a. Customer says they intended to pay by cash but don’t have enough cash:
   1a. Customer uses an alternate payment method.
   1b. Customer tells Cashier to cancel sale. Cashier cancels sale on System.

7a. Paying by cash:
   1. Cashier enters the cash amount tendered.

2. System presents the balance due, and releases the cash drawer.
3. Cashier deposits cash tendered and returns balance in cash to Customer.
4. System records the cash payment.

7b. Paying by credit:
1. Customer enters their credit account information.
2. System sends payment authorization request to an external Payment Authorization Service System, and requests payment approval.
   2a. System detects failure to collaborate with external system:
      1. System signals error to Cashier.
      2. Cashier asks Customer for alternate payment.
3. System receives payment approval and signals approval to Cashier.
   3a. System receives payment denial:
      1. System signals denial to Cashier.
      2. Cashier asks Customer for alternate payment.
4. System records the credit payment, which includes the payment approval.
5. System presents credit payment signature input mechanism.
6. Cashier asks Customer for a credit payment signature. Customer enters signature.

7c. Paying by check...
7d. Paying by debit...:
7e. Customer presents coupons:
1. Before handling payment, Cashier records each coupon and System reduces price as appropriate. System records the used coupons for accounting reasons.
   1a. Coupon entered is not for any purchased item:
      1. System signals error to Cashier.

9a. There are product rebates:
1. System presents the rebate forms and rebate receipts for each item with a rebate.

9b. Customer requests gift receipt (no prices visible):
1. Cashier requests gift receipt and System presents it.

Special Requirements:
– Touch screen UI on a large flat panel monitor. Text must be visible from 1 meter.
– Credit authorization response within 30 seconds 90% of the time.
– Somehow, we want robust recovery when access to remote services such the inventory system is failing.
– Language internationalization on the text displayed.
– Pluggable business rules to be insertable at steps 3 and 7.
– . . .

Technology and Data Variations List:
3a. Item identifier entered by bar code laser scanner (if bar code is present) or keyboard.
3b. Item identifier may be any UPC, EAN, JAN, or SKU coding scheme.
7a. Credit account information entered by card reader or keyboard.
7b. Credit payment signature captured on paper receipt. But within two years, we predict many customers will want digital signature capture.

Frequency of Occurrence: Could be nearly continuous.

Open Issues:

– What are the tax law variations?
– Explore the remote service recovery issue.
– What customization is needed for different businesses?
– Must a cashier take their cash drawer when they log out?
– Can the customer directly use the card reader, or does the cashier have to do it?

This use case is illustrative rather than exhaustive (although it is based on a real POS system's requirements). Nevertheless, there is enough detail and complication here to offer a realistic sense that a fully-dressed use case can record many requirement details. This example will serve well as a model for many use case problems.

The Two-Column Variation

Some prefer the two-column or conversational format, which emphasizes the fact that there is an interaction going on between the actors and the system. It was first proposed by Rebecca Wirfs-Brock in [Wirfs-Brock93], and is also promoted by Constantine and Lockwood to aid usability analysis and engineering [CL99]. Here is the same content using the two-column format:

Use Case UC1: Process Sale

Primary Actor: ...
... as before ...

Main Success Scenario:

Actor Intention	System Responsibility
1. Customer arrives at a POS checkout with goods and/or services to purchase.	
2. Cashier starts a new sale.	
3. Cashier enters item identifier.	4. Records each sale line item and presents item description and running total.
Cashier repeats steps 3-4 until indicates done.	5. System presents total with taxes calculated.
6. Cashier tells Customer the total, and asks for payment.	
7. Customer pays.	8. Handles payment.
	9. Logs the completed sale and sends information to the external accounting (for all accounting and commissions) and inventory systems (to update inventory). System presents receipt.
...	...

The Best Format?

There isn't one best format; some prefer the one-column style, some the two-column. Sections may be added and removed; heading names may change. None of this is particularly important; the key thing is to write the details of the main success scenario and its extensions, in some form. [Cockburn1] summarizes many usable formats.

> *Personal Practice*
>
> This is my practice, not a recommendation. For some years, I used the two-column format because of its clear visual separation in the conversation. However, I have reverted to a one-column style as it is more compact and easier to format, and the slight value of the separated conversation does not for me outweigh these benefits. I find it still simple to visually identify the different parties in the conversation (Customer, System, ...) if each party and the System responses are usually allocated to their own steps.

6.7 Explaining the Sections

Preface Elements

Many optional preface elements are possible. Only place elements at the start which are important to read before the main success scenario. Move extraneous "header" material to the end of the use case.

> **Primary Actor**: The principal actor that calls upon system services to fulfill a goal.

Important: Stakeholders and Interests List

This list is more important and practical than may appear at first glance. It suggests and bounds what the system must do. To quote:

> The [system] operates a contract between stakeholders, with the use cases detailing the behavioral parts of that contract...The use case, as the contract for behavior, captures *all and only* the behaviors related to satisfying the stakeholders' interests [Cockburn01].

This answers the question: What should be in the use case? The answer is: That which satisfies all the stakeholders' interests. In addition, by starting with the stakeholders and their interests before writing the remainder of the use case, we have a method to remind us what the more detailed responsibilities of the system should be. For example, would I have identified a responsibility for salesperson commission handling if I had not first listed the salesperson stakeholder and their interests? Hopefully eventually, but perhaps I would have missed it during the first analysis session. The stakeholder interest viewpoint provides a thorough and methodical procedure for discovering and recording all the required behaviors.

Stakeholders and Interests:
- Cashier: Wants accurate, fast entry and no payment errors, as cash drawer shortages are deducted from his/her salary.
- Salesperson: Wants sales commissions updated.
- ...

Preconditions and Success Guarantees

Preconditions state what *must always* be true before beginning a scenario in the use case. Preconditions are *not* tested within the use case; rather, they are conditions that are assumed to be true. Typically, a precondition implies a scenario of another use case that has successfully completed, such as logging in, or the more general "cashier is identified and authenticated." Note that there are conditions that must be true, but are not of practical value to write, such as "the system has power." Preconditions communicate noteworthy assumptions that the use case writer thinks readers should be alerted to.

Success guarantees (or **postconditions**) state what must be true on successful completion of the use case—either the main success scenario or some alternate path. The guarantee should meet the needs of all stakeholders.

Preconditions: Cashier is identified and authenticated.
Success Guarantee (postconditions): Sale is saved. Tax is correctly calculated. Accounting and Inventory are updated. Commissions recorded. Receipt is generated.

Main Success Scenario and Steps (or Basic Flow)

This has also been called the "happy path" scenario, or the more prosaic "Basic Flow". It describes the typical success path that satisfies the interests of the stakeholders. Note that it often does *not* include any conditions or branching. Although not wrong or illegal, it is arguably more comprehensible and extendible to be very consistent and defer all conditional handling to the Extensions section.

Suggestion

Defer all conditional and branching statements to the Extensions section.

The scenario records the steps, of which there are three kinds:

1. An interaction between actors.[3]
2. A validation (usually by the system).
3. A state change by the system (for example, recording or modifying something).

Step one of a use case does not always fall into this classification, but indicates the trigger event that starts the scenario.

It is a common idiom to always capitalize the actors' names for ease of identification. Observe also the idiom that is used to indicate repetition.

Main Success Scenario:
1. Customer arrives at a POS checkout with items to purchase.
2. Cashier starts a new sale.
3. Cashier enters item identifier.
4. ...

Cashier repeats steps 3-4 until indicates done.

5. ...

Extensions (or Alternate Flows)

Extensions are very important. They indicate all the other scenarios or branches, both success and failure. Observe in the fully dressed example that the Extensions section was considerably longer and more complex than the Main Success Scenario section; this is common and to be expected. They are also known as "Alternative Flows."

In thorough use case writing, the combination of the happy path and extension scenarios should satisfy "nearly" all the interests of the stakeholders. This point is qualified, because some interests may best be captured as non-functional requirements expressed in the Supplementary Specification rather than the use cases.

Extension scenarios are branches from the main success scenario, and so can be notated with respect to it. For example, at Step 3 of the main success scenario there may be an invalid item identifier, either because it was incorrectly entered or unknown to the system. An extension is labeled "3a"; it first identifies the condition and then the response. Alternate extensions at Step 3 are labeled "3b" and so forth.

Extensions:
3a. Invalid identifier:

3. Note that the system under discussion itself should be considered an actor when it plays an actor role collaborating with other systems.

    1. System signals error and rejects entry.
3b. There are multiple of same item category and tracking unique item identity not important (e.g., 5 packages of veggie-burgers):
    1. Cashier can enter item category identifier and the quantity.

An extension has two parts: the condition and the handling.

Guideline: Write the condition as something that can be *detected* by the system or an actor. To contrast:

5a. System detects failure to communicate with external tax calculation system service:

5a. External tax calculation system not working:

The former style is preferred because this is something the system can detect; the latter is an inference.

Extension handling can be summarized in one step, or include a sequence, as in this example, which also illustrates notation to indicate that a condition can arise within a range of steps:

3-6a: Customer asks Cashier to remove an item from the purchase:
    1. Cashier enters the item identifier for removal from the sale.
    2. System displays updated running total.

At the end of extension handling, by default the scenario merges back with the main success scenario, unless the extension indicates otherwise (such as by halting the system).

Sometimes, a particular extension point is quite complex, as in the "paying by credit" extension. This can be a motivation to express the extension as a separate use case.

This extension example also demonstrates the notation to express failures within extensions.

7b. Paying by credit:
    1. Customer enters their credit account information.
    2. System requests payment validation from external Payment Authorization Service System.
        2a. System detects failure to collaborate with external system:
            1. System signals error to Cashier.
            2. Cashier asks Customer for alternate payment.
    3. ...

If it is desirable to describe an extension condition as possible during any (or at least most) steps, the labels *a, *b, ..., can be used.

*a. At any time, System crashes:
   In order to support recovery and correct accounting, ensure all transaction sensitive state and events can be recovered at any step in the scenario.
   1. Cashier restarts the System, logs in, and requests recovery of prior state.
   2. System reconstructs prior state.

Special Requirements

If a non-functional requirement, quality attribute, or constraint relates specifically to a use case, record it with the use case. These include qualities such as performance, reliability, and usability, and design constraints (often in I/O devices) that have been mandated or considered likely.

Special Requirements:
– Touch screen UI on a large flat panel monitor. Text must be visible from 1 meter.
– Credit authorization response within 30 seconds 90% of the time.
– Language internationalization on the text displayed.
– Pluggable business rules to be insertable at steps 2 and 6.
– . . .

Technology and Data Variations List

Often there are technical variations in *how* something must be done, but not what, and it is noteworthy to record this in the use case. A common example is a technical constraint imposed by a stakeholder regarding input or output technologies. For example, a stakeholder might say, "The POS system must support credit account input using a card reader and the keyboard." Note that these are examples of early design decisions or constraints; in general, it is skillful to avoid premature design decisions, but sometimes they are obvious or unavoidable, especially concerning input/output technologies.

It is also necessary to understand variations in data schemes, such as using UPCs or EANs for item identifiers, encoded in bar code symbology.

This list is the place to record such variations. It is also useful to record variations in the data that may be captured at a particular step.

Technology and Data Variations List:
3a. Item identifier entered by laser scanner or keyboard.
3b. Item identifier may be any UPC, EAN, JAN, or SKU coding scheme.
7a. Credit account information entered by card reader or keyboard.

7b. Credit payment signature captured on paper receipt. But within two years, we predict many customers will want digital signature capture.

Suggestion

This section should *not* contain multiple steps to express varying behavior for different cases. If that is necessary, say it in the Extensions section.

6.8 Goals and Scope of a Use Case

How should use cases be discovered? It is common to be unsure if something is a valid (or more practically, a useful) use case. Tasks can be grouped at many levels of granularity, from one or a few small steps, up to enterprise-level activities.

At what level and scope should use cases be expressed?

The following sections examine the simple ideas of elementary business processes and goals as a framework for identifying the use cases for an application.

Use Cases for Elementary Business Processes

Which of these is a valid use case?

- Negotiate a Supplier Contract
- Handle Returns
- Log In

An argument can be made that all of these are use cases *at different levels*, depending on the system boundary, actors, and goals. Evaluation of these candidates is presented after an introduction to elementary business processes.

Rather than asking in general, "What is a valid use case?", a more relevant question for the POS case study is: What is a useful level to express use cases for application requirements analysis?

Guideline: The EBP Use Case

For requirements analysis for a computer application, focus on use cases at the level of **elementary business processes** (EBPs).

EBP is a term from the business process engineering field[4], defined as:

> A task performed by one person in one place at one time, in response to a business event, which adds measurable business value and leaves the data in a consistent state. e.g., Approve Credit or Price Order [original source lost].

This can be taken too literally: Does a use case fail as an EBP if two people are required, or if a person has to walk around? Probably not, but the feel of the definition is about right. It's not a single small step like "delete a line item" or "print the document." Rather, the main success scenario is probably five or ten steps. It doesn't take days and multiple sessions, like "negotiate a supplier contract;" it is a task done during a single session. It is probably between a few minutes and an hour in length. As with the UP's definition, it emphasizes adding observable or measurable business value, and it comes to a resolution in which the system and data are in a stable and consistent state.

> A common use case mistake is defining many use cases at too low a level; that is, as a single step, subfunction, or subtask within an EBP.

Reasonable Violations of the EBP Guideline

Although the "base" use cases for an application should satisfy the EBP guideline, it is frequently useful to create separate "sub" use cases representing subtasks or steps within a base use case. Use cases can exist that fail the EBP test; many potentially exist at a lower level. The guideline is only used to find the dominant level of use cases in requirements analysis for an application; that is, the level to focus on for naming and writing them.

For example, a subtask or extension such as "paying by credit" may be repeated in several base use cases. It is desirable to separate this into its own use case (that does not satisfy the EBP guideline) and link it to several base use cases, to avoid duplication of the text. Chapter 25 explores the issue of use case relationships.

Use Cases and Goals

Actors have goals (or needs) and use applications to help satisfy them. Consequently, an EBP-level use case is called a **user goal**-level user case, to emphasize that it serves (or should serve) to fulfill a goal of a user of the system, or the primary actor.

---

4. EBP is similar to the term **user task** in usability engineering, although the meaning is less strict in that domain.

And it leads to a recommended procedure:

1. Find the user goals.
2. Define a use case for each.

This is slight shift in emphasis for the use case modeler. Rather than asking "What are the use cases?", one starts by asking: "What are your goals?" In fact, the name of a use case for a user goal should reflect its name, to emphasize this viewpoint—Goal: capture or process a sale; use case: *Process Sale*.

Note that because of this symmetry, the EBP guideline can be equally applied to decide if a goal or a use case is at a suitable level.

Thus, here is a key idea regarding investigating user goals vs. investigating use cases:

> Imagine we are together in a requirements workshop. We could ask either:
>
> - "What do you do?" (roughly a use case-oriented question) or,
> - "What are your goals?"
>
> Answers to the first question are more likely to reflect current solutions and procedures, and the complications associated with them.
>
> Answers to the second question, especially combined with an investigation to move higher up the goal hierarchy ("what is the goal of that goal?") open up the vision for new and improved solutions, focus on adding business value, and get to the heart of what the stakeholders want from the system under discussion.

Example: Applying the EBP Guideline

As the system analyst responsible for the NextGen system requirements discovery, you are investigating user goals. The conversation goes like this: During a requirements workshop:

System analyst: "What are some of your goals in the context of using a POS system?"
Cashier: "One, to quickly log in. Also, to capture sales."
System analyst: "What do you think is the higher level goal motivating logging in?"
Cashier: "I'm trying to identify myself to the system, so it can validate that I'm allowed to use the system for sales capture and other tasks."
System analyst: "Higher than that?"
Cashier: "To prevent theft, data corruption, and display of private company information."

Note the analyst's strategy of searching up the goal hierarchy to find higher level user goals that still satisfy the EBP guideline, to get at the real intent behind the action, and also to understand the context of the goals.

"Prevent theft, ..." is higher than a user goal; it may be called an enterprise goal, and is not an EBP. Therefore, although it can inspire new ways of thinking about the problem and solutions (such as eliminating POS systems and cashiers completely), we will set it aside for now.

Lowering the goal level to "identify myself and be validated" appears closer to the user goal level. But is it at the EBP level? It does not add observable or measurable business value. If the CEO asked, "What did you do today?" and you said "I logged in 20 times!", she would not be impressed. Consequently, this is a secondary goal, always in the service of doing something useful, and is not an EBP or user goal. By contrast, "capture a sale" does fit the criteria of being an EBP or user goal.

As another example, in some stores there is a process called "cashing in", in which a cashier inserts their own cash drawer tray into the terminal, logs in, and tells the system how much cash is in drawer. *Cashing In* is an EBP-level (or user goal level) use case; the log in step, rather than being a EBP-level use case, is a subfunction goal in support of the goal of cashing in.

Subfunction Goals and Use Cases

Although "identify myself and be validated" (or "log in") has been eliminated as a user goal, it is a goal at a lower level, called a **subfunction goal**—subgoals that support a user goal. Use cases should only occasionally be written for these subfunction goals, although it is a common problem that use case experts observe when asked to evaluate and improve (usually simplify) a set of use cases.

It is not illegal to write use cases for subfunction goals, but it is not always helpful, as it adds complexity to a use case model; there can be hundreds of subfunction goals—or subfunction use cases—for a system.

Important point: The number and granularity of use cases influences the time and difficulty to understand, maintain, and manage the requirements.

The most common, valid motivation to express a subfunction goal as a use case is when the subfunction is repeated in or is a precondition for multiple user goal-level use cases. This in fact is probably true of "identify myself and be validated," which is a precondition of most, if not all, other user goal-level use cases.

Consequently, it may be written as the use case *Authenticate User.*

Goals and Use Cases Can be Composite

Goals are usually composite, from the level of an enterprise ("be profitable"), to many supporting intermediate goals while using applications ("sales are captured"), to supporting subfunction goals within applications ("input is valid").

Similarly, use cases can be written at different levels to satisfy these goals, and can be composed of lower level use cases.

> These varying goal and use case levels are a common source of confusion in identifying the appropriate level of use cases for an application. The EBP guideline provides guidance to filter out excessive low-level use cases.

6.9 Finding Primary Actors, Goals, and Use Cases

Use cases are defined to satisfy the user goals of the primary actors. Hence, the basic procedure is:

1. Choose the system boundary. Is it just a software application, the hardware and application as a unit, that plus a person using it, or an entire organization?
2. Identify the primary actors—those that have user goals fulfilled through using services of the system.
3. For each, identify their user goals. Raise them to the highest user goal level that satisfies the EBP guideline.
4. Define use cases that satisfy user goals; name them according to their goal. Usually, user goal-level use cases will be one-to-one with user goals, but there is at least one exception, as will be examined.

Step 1: Choosing the System Boundary

For this case study, the POS system itself is the system under design; everything outside of it is outside the system boundary, including the cashier, payment authorization service, and so on.

If it is not clear, defining the boundary of the system under design can be clarified by defining what is outside—the external primary and supporting actors. Once the external actors are identified, the boundary becomes clearer. For example, is the complete responsibility for payment authorization within the system boundary? No, there is an external payment authorization service actor.

Steps 2 and 3: Finding Primary Actors and Goals

It is artificial to strictly linearize the identification of primary actors before user goals; in a requirements workshop, people brainstorm and generate a mixture of both. Sometimes, goals reveal the actors, or vice versa.

Guideline: Emphasize brainstorming the primary actors first, as this sets up the framework for further investigation.

Reminder Questions to Find Actors and Goals

In addition to obvious primary actors and user goals, the following questions help identify others that may be missed:

Who starts and stops the system?	Who does system administration?
Who does user and security management?	Is "time" an actor because the system does something in response to a time event?
Is there a monitoring process that restarts the system if it fails?	Who evaluates system activity or performance?
How are software updates handled? Push or pull update?	Who evalutes logs? Are they remotely retrieved?

Primary and Supporting Actors

Recall that primary actors have user goals fulfilled through using services of the system. They call upon the system to help them. This is in contrast to *supporting actors*, which provide services to the system under design. For now, the focus is on finding the primary actors, not the supporting ones.

Recall also that primary actors can be—among other things—other computer systems, such as "watchdog" software processes.

> *Suggestion*
>
> Be suspicious if no primary actors are external computer systems.

The Actor-Goal List

Record the primary actors and their user goals in an actor-goal list. In terms of UP artifacts it should be a section in the Vision artifact (which is described in the next chapter).

For example:

Actor	Goal	Actor	Goal
Cashier	process sales process rentals handle returns cash in cash out . . .	System Administrator	add users modify users delete users manage security manage system tables . . .
Manager	start up shut down . . .	Sales Activity System[a]	analyze sales and performance data
. . .	. . .	. . .	. . .

a. The Sales Activity System is a remote application that will frequently request sales data from each POS node in the network.

Project Planning Dimension

In practice, this list has additional columns for priority, effort, and risk; this is briefly covered in Chapter 36.

The Messy Reality

This list looks neat, but the reality of its creation is anything but. Lots of brainstorming and thrashing about in a requirements workshop goes on. Consider the earlier example that illustrated applying the EBP rule to the "log in" goal. During the workshop while creating this list the cashier may offer "log in" as one of the user goals. The system analyst digs deeper and raises the level of the goal beyond the low-level mechanism of logging in (the cashier was probably thinking of using a dialog box on a GUI) up to the level of "identify and authenticate user." Yet, the analyst then realizes it does not pass the EBP guideline, and discards it as a user goal. Of course, the reality is even somewhat different than this because an experienced analyst has a set of heuristics from past experience or study, one of which is "user authentication is seldom an EBP," and so is likely to have filtered this out quickly.

Primary Actor and User Goals Depend on System Boundary

Why is the cashier, and not the customer, the primary actor in the use case *Process Sale*? Why doesn't the customer appear in the actor-goal list?

The answer depends on the system boundary of the system under design, as illustrated in Figure 6.1. If viewing the enterprise or checkout service as an aggregate system, the customer *is* a primary actor, with the goal of getting goods or services and leaving. However, from the viewpoint of just the POS system (which is the choice of system boundary for this case study), it services the goal of the cashier (and the store) to process the customer's sale.

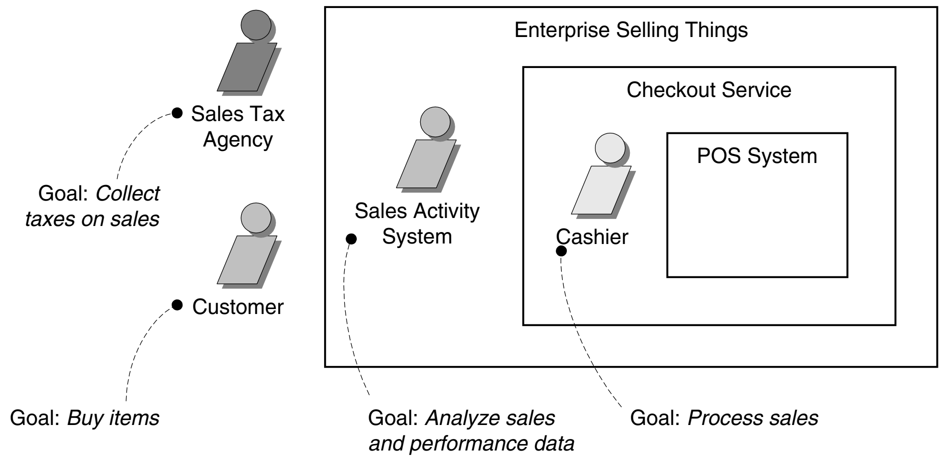

Figure 6.1 Primary actors and goals at different system boundaries.

Actors and Goals via Event Analysis

Another approach to aid in finding actors, goals, and use cases is to identify external events. What are they, where from, and why? Often, a group of events belong to the same EBP-level goal or use case. For example:

External Event	From Actor	Goal
enter sale line item	Cashier	process a sale
enter payment	Cashier or Customer	process a sale
. . .		

Step 4: Define Use Cases

In general, define one EBP-level use case for each user goal. Name the use case similar to the user goal—for example, Goal: process a sale; Use Case: *Process Sale*.

Also, name use cases starting with a verb.

A common exception to one use case per goal is to collapse CRUD (create, retrieve, update, delete) separate goals into one CRUD use case, idiomatically called *Manage <X>*. For example, the goals "edit user," "delete user," and so forth are all satisfied by the *Manage Users* use case.

"Define use cases" has several levels of effort, ranging from a few minutes to simply record names, up to weeks to write fully dressed versions. The later UP process section of this chapter puts this work—when and how much—in the context of iterative development and the UP.

6.10 Congratulations: Use Cases Have Been Written, and Are Imperfect

The Need for Communication and Participation

The NextGen POS team is writing use cases in multiple requirements workshops over a series of short development iterations, incrementally adding to the set, and refining and adapting based on feedback. Subject matter experts, cashiers, and programmers actively participate in the writing process. There are no intermediaries between the cashiers, other users, and the developers; rather, there is direct communication.

Good, but not good enough. Written requirement specifications give the illusion of correctness; they are not. The use cases and other requirements still will not be correct—guaranteed. They will lack critical information and contain wrong statements. The solution is not the "waterfall" process attitude of trying harder to record requirements perfect and complete at the start, although of course we do the best we can in the time available. But it will never be enough.

A different approach is required. A large part of this is iterative development, but something else is needed: *ongoing personal communication*. Continual—daily—close participation and communication between the developers and someone who understands the domain and can make requirement decisions. Someone the programmers can walk up to in a matter of seconds and get clarification, whenever a question arises. For example, the Extreme Programming practices [Beck00] contain an excellent recommendation: *User full-time on the project, in the project room*.

6.11 Write Use Cases in an Essential UI-Free Style

New and Improved! The Case for Fingerprinting

Investigating and asking about goals rather than tasks and procedures encourages a focus on the essence of the requirements—the intent behind them. For example, during a requirements workshop, the cashier may say one of his goals is to "log in." The cashier was probably thinking of a GUI, dialog box, user ID, and password. This is a mechanism to achieve a goal, rather than the goal itself. By investigating up the goal hierarchy ("What is the goal of that goal?"), the system analyst arrives at a mechanism-independent goal: "identify myself and get authenticated," or an even higher goal: "prevent theft ...".

This discovery process can open up the vision to new and improved solutions. For example, keyboards and mice with biometric readers, usually for a fingerprint, are now common and inexpensive. If the goal is "identification and authentication" why not make it easy and fast, using a biometric reader on the keyboard? But properly answering that question involves some usability analysis work as well, such as knowing the typical users' profiles. Are their fingers covered in grease? Do they have fingers?

Essential Style Writing

This idea has been summarized in various use case guidelines as "keep the user interface out; focus on intent" [Cockburn01]. Its motivation and notation has been most fully explored by Larry Constantine in the context of creating better user interfaces (UIs) and doing usability engineering [Constantine94, CL99]. Constantine calls the writing style **essential** when it avoids UI details and focuses on the real user intent.[5]

In an essential writing style, the narrative is expressed at the level of the user's *intentions* and system's *responsibilities* rather than their concrete actions. They remain free of technology and mechanism details, especially those related to the UI.

Write use cases in an essential style; keep the user interface out and focus on actor intent.

All the previous example use cases in this chapter, such as *Process Sale*, were written in an essential style.

---

5. The term comes from "essential models" in *Essential Systems Analysis* [MP84].

Note that the dictionary defines *goal* as a synonym for intention [MW89], illustrating the connection between the *essential* style idea of Constantine and the goal-oriented viewpoint previously stressed in this chapter. Indeed, many actor *intention* steps in an essential use case can also be be characterized as subfunction *goals*.

Contrasting Examples

Essential Style

Assume that the *Manage Users* use case requires identification and authentication. The Constantine-inspired essential style uses the two-column format. However, it can be written in one column.

. . .

Actor Intention	System Responsibility
1. Administrator identifies self.	2. Authenticates identity.
3. . . .	

In the one-column format this is shown as:

. . .
1. Administrator identifies self.
2. System authenticates identity.
3. . . .

The design solution to these intentions and responsibilities is wide open: biometric readers, graphical user interfaces (GUIs), and so forth.

Concrete Style—Avoid During Early Requirements Work

In contrast, there is a **concrete use case** style. In this style, user interface decisions are embedded in the use case text. The text may even show window screen shots, discuss window navigation, GUI widget manipulation and so forth. For example:

. . .
1. Adminstrator enters ID and password in dialog box (see Picture 3).
2. System authenticates Adminstrator.
3. System displays the “edit users” window (see Picture 4).
4. . . .

These concrete use cases may be useful as an aid to concrete or detailed GUI design work during a later step, but they are not suitable during the early requirements analysis work. During early requirements work, “keep the user interface out—focus on intent.”

6.12 Actors

An actor is anything with behavior, including the system under discussion (SuD) itself when it calls upon the services of other systems.[6] Primary and supporting actors will appear in the action steps of the use case text. Actors are not only roles played by people, but organizations, software, and machines. There are three kinds of external actors in relation to the SuD:

- **Primary actor—**has user goals fulfilled through using services of the SuD. For example, the cashier.
    - Why identify? To find user goals, which drive the use cases.
- **Supporting actor—**provides a service (for example, information) to the SuD. The automated payment authorization service is an example. Often a computer system, but could be an organization or person.
    - Why identify? To clarify external interfaces and protocols.
- **Offstage actor—**has an interest in the behavior of the use case, but is not primary or supporting; for example, a government tax agency.
    - Why identify? To ensure that *all* necessary interests are identified and satisfied. Offstage actor interests are sometimes subtle or easy to miss unless these actors are explicitly named.

6.13 Use Case Diagrams

The UML provides use case diagram notation to illustrate the names of use cases and actors, and the relationships between them.

> Use case diagrams and use case relationships are secondary in use case work. Use cases are text documents. Doing use case work means to write text.

A common sign of a novice (or academic) use case modeler is a preoccupation with use case diagrams and use case relationships, rather than writing text. World-class use case experts such as Anderson, Fowler, Cockburn, among others, downplay use case diagrams and use case relationships, and instead focus on writing.

6. This was a refinement and improvement to alternate definitions of actors, including those in early versions of the UML and UP [Cockburn97]. Older definitions inconsistently excluded the SuD as an actor, even when it called upon services of other systems. All entities may play multiple *roles*, including the SuD.

With that as a caveat, a simple use case diagram provides a succinct visual context diagram for the system, illustrating the external actors and how they use the system.

Suggestion
Draw a simple use case diagram in conjunction with an actor-goal list.

A use case diagram is an excellent picture of the system context; it makes a good **context diagram**, that is, showing the boundary of a system, what lies outside of it, and how it gets used. It serves as a communication tool that summarizes the behavior of a system and its actors. A sample *partial* use case context diagram for the NextGen system is shown in Figure 6.2.

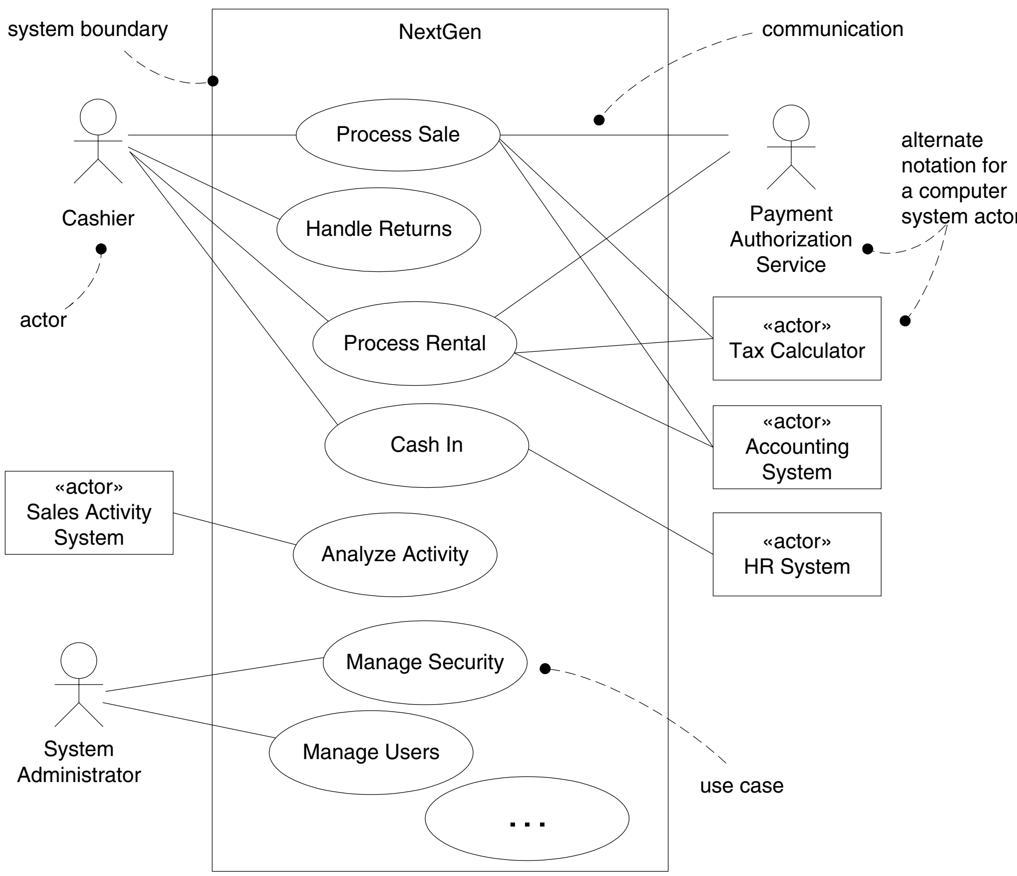

Figure 6.2 Partial use case context diagram.

Some prefer to highlight external computer system actors with an alternate notation, as illustrated in Figure 6.3.

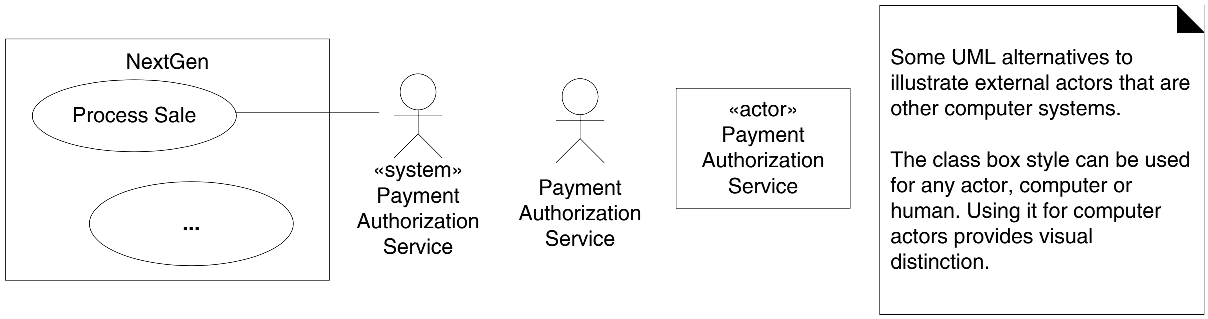

Figure 6.3 Alternate actor notation.

Diagramming Suggestions

Figure 6.4 offers some diagram advice.

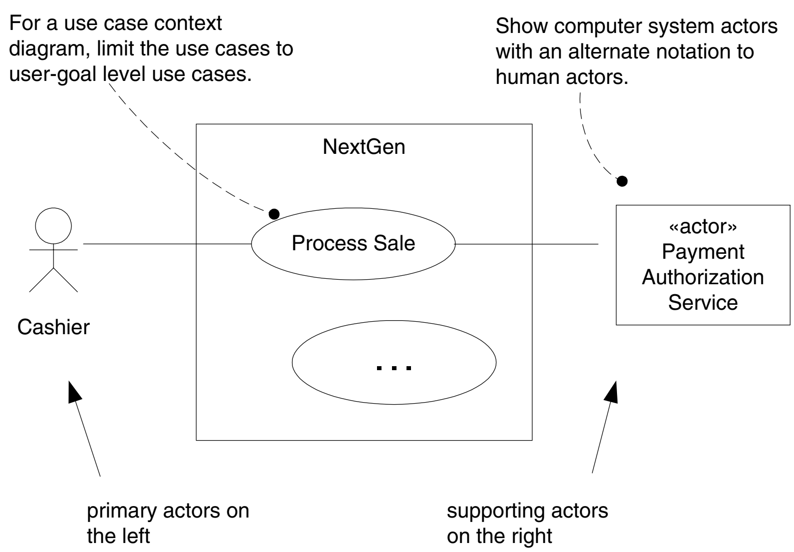

Figure 6.4 Notation suggestions.

A Caution on Over-Diagramming

To reiterate, the important use case work is to write text, not diagram or focus on use case relationships. If an organization is spending many hours (or worse, days) working on a use case diagram and discussing use case relationships, rather than first focussing on writing text, relative effort has been misplaced.

6.14 Requirements in Context and Low-Level Feature Lists

As implied by the title of the book *Uses Cases: Requirements in Context* [GK00], a key motivation of the use case idea is the consideration and organization of requirements in the context of the goals and scenarios of using a system. That's a good thing—it improves cohesion and comprehension. However, use cases are not the only necessary requirements artifact. Some non-functional requirements, domain rules and context, and other hard-to-place elements are better captured in the Supplementary Specification, which is described in the next chapter.

One idea behind use cases is to replace detailed, low-level feature lists (which were common in traditional requirements methods) with use cases (with some exceptions). These lists tended to look as follows, usually grouped into functional areas:

ID	Feature
FEAT1.9	The system shall accept entry of item identifiers.
...	...
FEAT2.4	The system shall log credit payments to the accounts receivable system.

Such detailed lists of low-level features are somewhat usable. However, the complete list is not a half-page; more likely it is dozens or a hundred pages. This leads to some drawbacks, which use cases help address. These include:

- Long, detailed function lists do not relate the requirements in a cohesive context; the different functions and features increasingly appear like a disjointed "laundry list" of items. In contrast, use cases place the requirements in the context of the stories and goals of using the system.
- If both use case and detailed feature lists are used, there is duplication. More work, more volume to write and read, more consistency and synchronization problems.

Suggestion

Strive to replace detailed, low-level feature lists with use cases.

High-Level System Feature Lists Are Acceptable

It is common and useful to summarize system functionality with a terse, high-level feature list called system features in a Vision document. In contrast to 100 pages of low-level, detailed features, a system features list tends to include only a few dozen items. The list provides a very succinct summary of system functionality, independent of the use case view. For example:

Summary of System Features

- sales capture
- payment authorization (credit, debit, check)
- system administration for users, security, code and constants tables, and so on
- automatic offline sales processing when external components fail
- real-time transactions, based on industry standards, with third-party systems, including inventory, accounting, human resources, tax calculators, and payment authorization services
- definition and execution of customized "pluggable" business rules at fixed, common points in the processing scenarios
- . . .

This is explored in the next chapter.

When Are Detailed Feature Lists Appropriate?

Sometimes use cases do not really fit; some applications call out for a feature-driven viewpoint. For example, application servers, database products, and other middleware or back-end systems need to be primarily considered and evolved in terms of *features* ("We need XML support in the next release"). Use cases are not a natural fit for these applications or the way they need to evolve in terms of market forces.

6.15 Use Cases Are Not Object-Oriented

There is nothing object-oriented about use cases; one is not doing object-oriented analysis if writing use cases. This is not a defect, but a point of clarification. Indeed, use cases are a broadly applicable requirements analysis tool that can be applied to non-object-oriented projects, which increases their usefulness as a requirements method.

However, as will be explored, use cases are a pivotal input into classic OOA/D activities.

6.16 Use Cases Within the UP

Use cases are vital and central to the UP, which encourages **use-case driven development**. This implies:

- Requirements are primarily recorded in use cases (the Use-Case Model); other requirements techniques (such as functions lists) are secondary, if used at all.
- Use cases are an important part of iterative planning. The work of an iteration is—in part—defined by choosing some use case scenarios, or entire use cases. And use cases are a key input to estimation.
- **Use-case realizations** drive the design. That is, the team designs collaborating objects and subsystems in order to perform or realize the use cases.
- Use cases often influence the organization of user manuals.

The UP distinguishes between system and business use cases. **System use cases** are what have been examined in this chapter, such as *Process Sale*. They are created in the Requirements workflow, and are part of the Use-Case Model.

Business use cases are less commonly written. If done, they are created in the Business Modeling workflow as part of a large-scale business process reengineering effort, or to help understand the context of a new system in the business. They describe a sequence of actions of a business as a whole to fulfill a goal of a **business actor** (an actor in the business environment, such as a customer or supplier). For example, in a restaurant, one business use case is *Serve a Meal*.

Use Cases and Requirements Specification Across the Iterations

This section reiterates a critical idea in the UP and iterative development: The timing and level of effort of requirements specification across the iterations. Table 6.1 presents a sample (not a recipe) which communicates the UP strategy of how requirements are developed.

Note that a technical team starts building the production core of the system when only perhaps 10% of the requirements are detailed, and in fact, there is a deliberate delay in continuing with concerted requirements work until near the end of the first elaboration iteration.

This is the key difference in iterative development to a waterfall process: Production-quality development of the core of a system starts quickly, long before all the requirements are known.

Observe that near the end of the first iteration of elaboration, there is a second requirements workshop, during which perhaps 30% of the use cases are written in detail. This staggered requirements analysis benefits from the feedback of having built a little of the core software. The feedback includes user evaluation, testing, and improved “knowing what we don’t know.” That is, the act of building software rapidly surfaces important assumptions and questions that need clarification.

Workflow	Artifact	Comments and Level of Requirements Effort				
		Incep 1 week	Elab 1 4 weeks	Elab 2 4 weeks	Elab 3 3 weeks	Elab 4 3 weeks
Requirements	Use-Case Model	2-day requirements workshop. Most use cases identified by name, and summarized in a short paragraph. Only 10% written in detail.	Near the end of this iteration, host a 2-day requirements workshop. Obtain insight and feedback from the implementation work, then complete 30% of the use cases in detail.	Near the end of this iteration, host a 2-day requirements workshop. Obtain insight and feedback from the implementation work, then complete 50% of the use cases in detail.	Repeat, complete 70% of all use cases in detail.	Repeat with the goal of 80-90% of the use cases clarified and written in detail. Only a small portion of these have been built in elaboration; the remainder are done in construction.
Design	Design Model	none	design for a small set of high-risk architecturally significant requiremnts	repeat	repeat	Repeat. The high risk and architecturally significant aspects should now be stabilized.
Implementation	Implementation Model (code, etc.)	none	implement these	Repeat. 5% of the final system is built.	Repeat. 10% of the final system is built.	Repeat. 15% of the final system is built.
Project Management	SW Development Plan	Very vague estimate of total effort.	Estimate starts to take shape.	a little better...	a little better...	Overall project duration, major milestones, effort, and cost estimates can now be rationally committed to.

Table 6.1 Sample requirements effort across the early iterations; this is not a recipe.

Timing of UP Artifact Creation

Table 6.2 illustrates some UP artifacts, and an example of their start and refinement schedule. The Use-Case Model is started in inception, with perhaps only 10% of the use cases written in any detail. The majority are incrementally written over the iterations of the elaboration phase, so that by the end of elaboration, a large body of detailed use cases and other requirements (in the Supplementary Specification) are written, providing a realistic basis for estimation through to the end of the project.

Workflow	Artifact Iteration→	Incep. I1	Elab. E1..En	Const. C1..Cn	Trans. T1..T2
Business Modeling	Domain Model		s		
Requirements	***Use-Case Model***	s	r		
	Vision	s	r		
	Supplementary Specifications	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		
	Data Model		s	r	
Implementation	Implementation Model		s	r	r
Project Management	SW Development Plan	s	r	r	r
Testing	Test Model		s	r	
Environment	Development Case	s	r		

Table 6.2 Sample UP artifacts and timing. s - start; r - refine

Use Cases within Inception

The following discussion expands on the information in Table 6.1.

Not all use cases are written in their fully dressed format during the inception phase. Rather, suppose there is a two-day requirements workshop during the early NextGen investigation. The earlier part of the day is spent identifying goals and stakeholders, and speculating what is in and out of scope of the project. An actor-goal-use case table is written and displayed with the computer projector. A use case context diagram is started. After a few hours, perhaps 20 user goals (and thus, user goal level use cases) are identified, including *Process Sale*, *Handle Returns*, and so on. Most of the interesting, complex, or risky use cases are written in brief format; each averaging around two minutes to write. The team starts to form a high-level picture of the system's functionality.

After this, 10% to 20% of the use cases that represent core complex functions, or which are especially risky in some dimension, are rewritten in a fully dressed format; the team investigates a little deeper to better comprehend the magnitude, complexities, and hidden demons of the project, through a small sample of interesting use cases. Perhaps this means two use cases: *Process Sale* and *Handle Returns*.

A requirements management tool that integrates with a word processor is used for the writing, and the work is displayed via a projector while the team collaborates on the analysis and writing. The *Stakeholders and Interests* lists are written for these use cases, to discover more subtle (and perhaps costly) functional and key non-function requirements—or system qualities—such as for reliability or throughput.

The analysis goal is not to exhaustively complete the use cases, but spend a few hours to obtain some insight.

The project sponsor needs to decide if the project is worth significant investigation (that is, the elaboration phase). The inception work is not meant to do that investigation, but to obtain low-fidelity (and admittedly error-prone) insights regarding scope, risk, effort, technical feasibility, and business case, in order to decide to move forward, where to start if they do, or if to stop.

Perhaps the NextGen project inception step lasts five days. The combination of the two day requirements workshop and its brief use case analysis, and other investigation during the week, lead to the decision to continue on to an elaboration step for the system.

Use Cases within Elaboration

The following discussion expands on the information in Table 6.1.

This is a phase of multiple timeboxed iterations (for example, four iterations) in which risky, high-value, or architecturally significant parts of the system are incrementally built, and the "majority" of requirements identified and clarified. The feedback from the concrete steps of programming influences and informs the team's understanding of the requirements, which are iteratively and adaptively refined. Perhaps there is a two-day requirements workshop in each iteration—four workshops. However, not all use cases are investigated in each workshop. They are prioritized; early workshops focus on a subset of the most important use cases.

Each subsequent short workshop is a time to adapt and refine the vision of the core requirements, which will be unstable in early iterations, and stabilizing in later ones. Thus, there is an iterative interplay between requirements discovery, and building parts of the software.

During each requirements workshop, the user goals and use case list are refined. More of the use cases are written, and rewritten, in their fully dressed format. By the end of elaboration, "80-90%" of the use cases are written in detail. For the POS system with 20 user goal level use cases, 15 or more of the most complex and risky should be investigated, written, and rewritten in a fully dressed format.

Note that elaboration involves programming parts of the system. At the end of this step, the NextGen team should not only have a better definition of the use cases, but some quality executable software.

Use Cases within Construction

The construction step is composed of timeboxed iterations (for example, 20 iterations of two weeks each) that focus on completing the system, once the risky and core unstable issues have settled down in elaboration. There will still be some minor use case writing and perhaps requirements workshops, but much less so than in elaboration. By this step, the majority of core functional and non-functional requirements should have iteratively and adaptively stabilized. That does not mean to imply requirements are frozen or investigation finished, but the degree of change is much lower.

6.17 Case Study: Use Cases in the NextGen Inception Phase

As described in the previous section, not all use cases are written in their fully dressed form during inception. The Use-Case Model at this phase of the case study could be detailed as follows:

Fully Dressed	Causual	Brief
Process Sale Handle Returns	Process Rental Analyze Sales Activity Manage Security ...	Cash In Cash Out Manage Users Start Up Shut Down Manage System Tables ...

6.18 Further Readings

The most popular use case guide, translated into several languages, is *Writing Effective Use Cases* [Cockburn01].[7] This has emerged with good reason as the most widely read and followed use case book and is therefore recommended as a primary reference. This introductory chapter is consequently based on and consistent with its content. Suggestion: Do not be put off the book by the author's use of icons for different use case levels, or the early emphasis on levels and use case taxonomy. The icons are optional and minor. And although the discussion of levels and goals may at first seem a diversion to those new to use cases, those who have worked with them for some time appreciate that the level and scope of use cases are key practical issues, because their misunderstanding is a common source of complication in use case modeling.

"Structuring Use Cases with Goals" [Cockburn97] is the most widely cited paper on use cases, available online at *www.usecases.org*.

Use Cases: Requirements in Context [GK00] is another useful text. It emphasizes the important viewpoint—as the title states—that use cases are not just

7. Note that Cockburn rhymes with *slow burn*.

another requirements artifact, but that they are the central vehicle that drives requirements work and information.

Another worthwhile read is *Applying Use Cases: A Practical Guide* [SW98], written by an experienced use case teacher and practioner that understand and communicate how to apply use cases in an iterative lifeycle.

6.19 UP Artifacts and Process Context

As illustrated in Figure 6.5, use cases influence many UP artifacts.

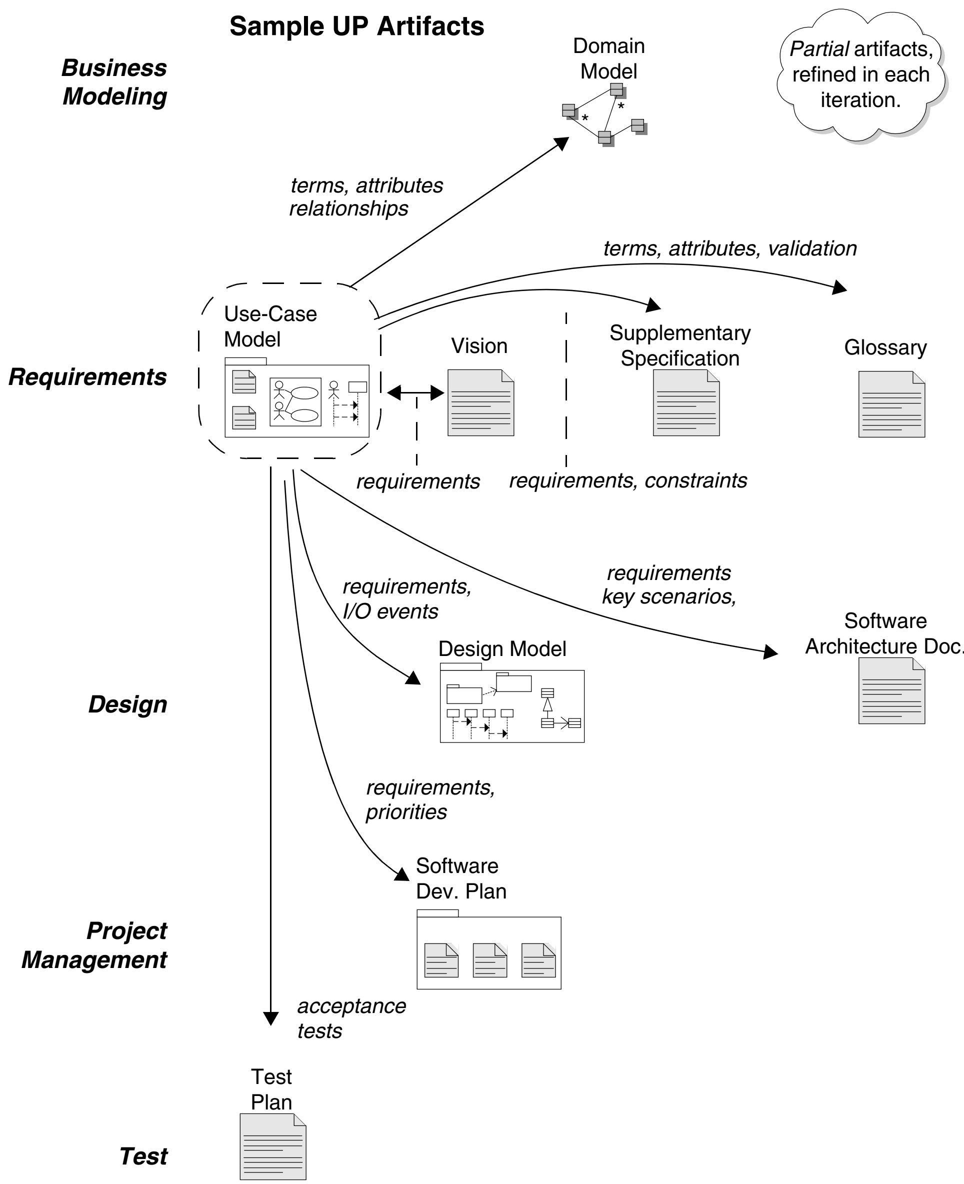

Figure 6.5 Sample UP artifact influence.

In the UP, use case work is a requirements discipline activity which could be initiated during a requirements workshop. Figure 6.6 offers suggestions on the time and space for doing this work.

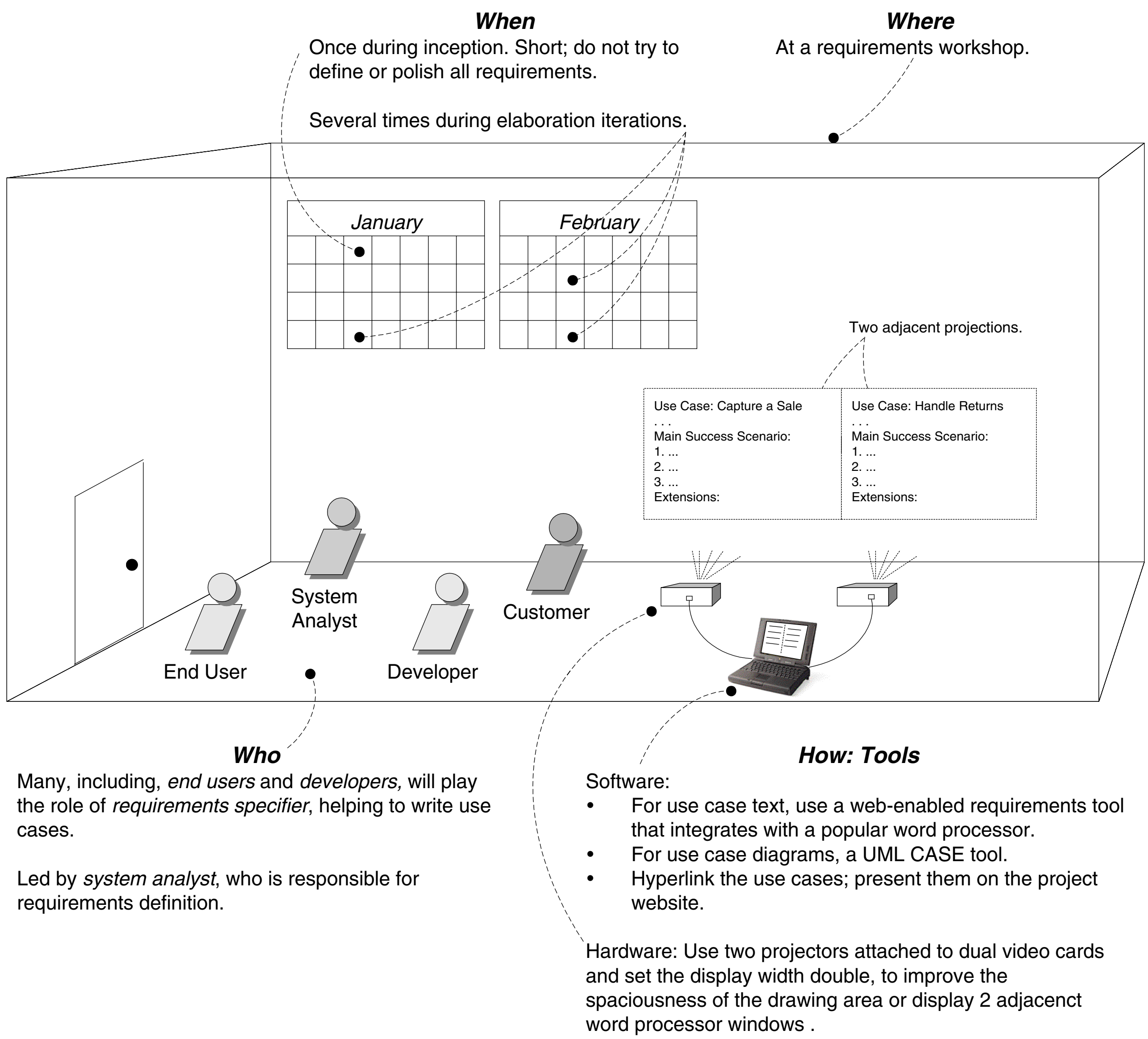

Figure 6.6 Process and setting context.