Chapter 6

USE CASES

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.
- Use the brief, casual, and fully dressed formats, in an essential style.
- Apply tests to identify suitable use cases.
- Relate use case analysis to iterative development.

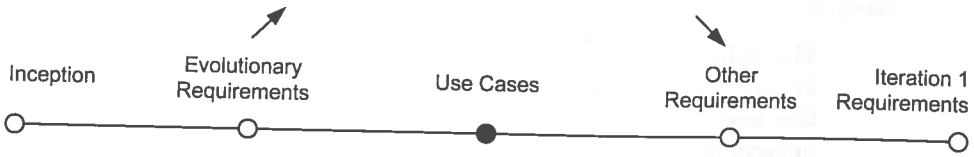
Introduction

intermediate use case topics p. 493

Use cases are text stories, widely used to discover and record requirements. They influence many aspects of a project—including OOA/D—and will be input to many subsequent artifacts in the case studies. This chapter explores basic concepts, including how to write use cases and draw a UML use case diagram. This chapter also shows the value of analysis skill over knowing UML notation; the UML use case diagram is trivial to learn, but the many guidelines to identify and write good use cases take weeks—or longer—to fully digest.

What's Next?

Having introduced requirements, this chapter explores use cases for functional requirements. The next covers other requirements in the UP, including the Supplementary Specification for non-functional requirements.



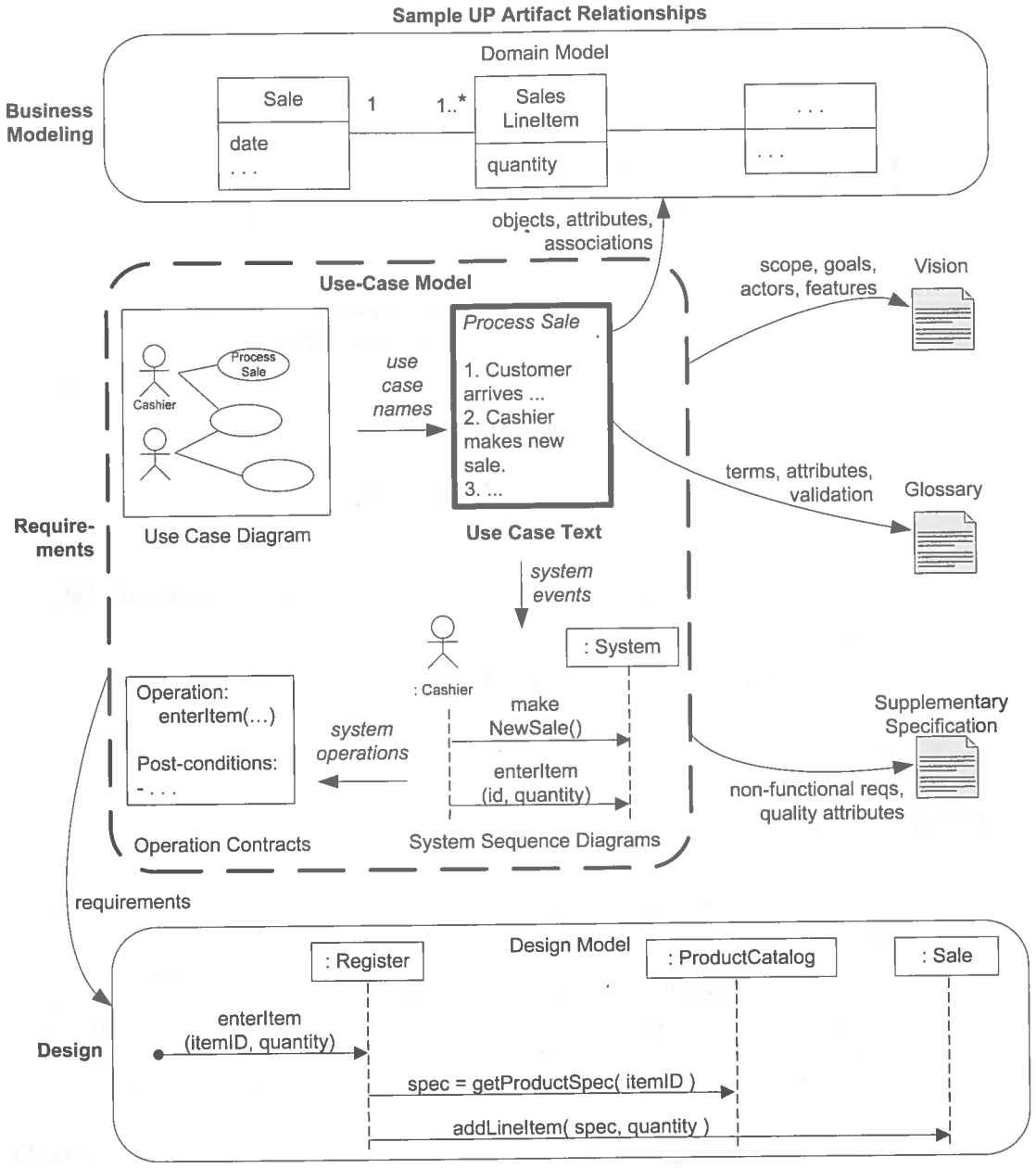


Figure 6.1 Sample UP artifact influence.

The influence of UP artifacts, with an emphasis on text use cases, is shown in Figure 6.1. High-level goals and use case diagrams are input to the creation of the use case text. The use cases can in turn influence many other analysis, design, implementation, project management, and test artifacts.

6.1 Example

Informally, use cases are *text stories* of some actor 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.

UML use case diagrams p. 89

Notice that ***use cases are not diagrams, they are text***. Focusing on secondary-value UML use case diagrams rather than the important use case text is a common mistake for use case novices.

Use cases often need to be more detailed or structured than this example, but the essence is discovering and recording functional requirements by writing stories of using a system to fulfill user goals; that is, *cases of use*.[1] It isn't supposed to be a difficult idea, although it's often difficult to discover what's needed and write it well.

6.2 Definition: What are Actors, Scenarios, and Use Cases?

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; 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 payment denial.

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

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

> If the customer paid by credit, and the reimbursement transaction to their credit account is rejected, inform the customer and pay them with cash.
>
> 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 accounting system, ...

Now that scenarios (use case instances) are defined, an alternate, but similar definition of a use case provided by the RUP will make better sense:

> 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].

Use Cases and the Use-Case Model

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

> **Use cases are text documents, not diagrams, and use-case modeling is primarily an act of writing text, not drawing diagrams.**

UP requirements p. 101

The Use-Case Model is not the only requirement artifact in the UP. There are also the Supplementary Specification, Glossary, Vision, and Business Rules. These are all useful for requirements analysis, but secondary at this point.

use case diagram p. 89

The Use-Case Model may optionally include a UML use case diagram to show the names of use cases and actors, and their relationships. This gives a nice **context diagram** of a system and its environment. It also provides a quick way to list the use cases by name.

There is nothing object-oriented about use cases; we're not doing OO analysis when writing them. That's not a problem—use cases are broadly applicable, which increases their usefulness. That said, use cases are a key requirements input to classic OOA/D.

Motivation: Why Use Cases?

We have goals and want computers to help meet them, ranging from recording

sales to playing games to estimating the flow of oil from future wells. Clever analysts have invented *many* ways to capture goals, but the best are simple and familiar. Why? This makes it easier—especially for customers—to contribute to their definition and review. That lowers the risk of missing the mark. This may seem like an off-hand comment, but it's important. Researchers have concocted complex analysis methods that they understand, but that send your average business person into a coma! Lack of user involvement in software projects is near the top of the list of reasons for project failure [Larman03], so anything that can help keep them involved is truly desirable.

more motivation p. 92

Use cases are a good way to help keep it simple, and make it possible for domain experts or requirement donors to themselves write (or participate in writing) use cases.

Another value of use cases is that *they emphasize the user goals and perspective*; we ask the question "Who is using the system, what are their typical scenarios of use, and what are their goals?" This is a more user-centric emphasis compared to simply asking for a list of system features.

Much has been written about use cases, and though worthwhile, creative people often obscure a simple idea with layers of sophistication or over-complication. 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, and so forth, rather than a focus on the hard work of simply *writing* the text stories.

That said, a strength of use cases is the ability to scale both up and down in terms of sophistication and formality.

6.5 Definition: Are Use Cases Functional Requirements?

FURPS+ p. 56

Use cases *are* requirements, primarily functional or behavioral 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 many modern methods—use cases are the central mechanism that is recommended for their discovery and definition.

A related viewpoint is that a use case defines a *contract* of how a system will behave [Cockburn01].

To be clear: Use cases are indeed 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 old-style feature lists and rather write use cases for the functional requirements. More on this point in a later section.

Definition: What are Three Kinds of Actors?

An **actor** is anything with behavior, including the system under discussion (SuD) itself when it calls upon the services of other systems.[2] Primary and supporting actors will appear in the action steps of the use case text. Actors are roles played not only by people, but by 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.

Notation: What are Three Common Use Case Formats?

Use cases can be written in different formats and levels of formality:

mple p. 63

- **brief**—Terse one-paragraph summary, usually of the main success scenario. The prior *Process Sale* example was brief.
  - When? During early requirements analysis, to get a quick sense of subject and scope. May take only a few minutes to create.

mple p. 63

- **casual**—Informal paragraph format. Multiple paragraphs that cover various scenarios. The prior *Handle Returns* example was casual.
  - When? As above.

2. 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.

example p. 68

- **fully dressed**—All steps and variations are written in detail, and there are supporting sections, such as preconditions and success guarantees.
    - When? After many use cases have been identified and written in a brief format, then during the first requirements workshop a few (such as 10%) of the architecturally significant and high-value use cases are written in detail.

more on timing of writing use cases p. 95

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

6.8 Example: Process Sale, Fully Dressed Style

Fully dressed use cases show more detail and are structured; they dig deeper.

In iterative and evolutionary UP requirements analysis, 10% of the critical use cases would be written this way during the first requirements workshop. Then design and programming starts on the most architecturally significant use cases or scenarios from that 10% set.

Various format templates are available for detailed use cases. Probably the most widely used and shared format, since the early 1990s, is the template available on the Web at *alistair.cockburn.us*, created by Alistair Cockburn, the author of the most popular book and approach to use-case modeling. The following example illustrates this style.

First, here's the template:

Use Case Section	Comment
Use Case Name	Start with a verb.
Scope	The system under design.
Level	"user-goal" or "subfunction"
Primary Actor	Calls on the system to deliver its services.
Stakeholders and Interests	Who cares about this use case, and what do they want?
Preconditions	What must be true on start, *and* worth telling the reader?
Success Guarantee	What must be true on successful completion, *and* worth telling the reader.
Main Success Scenario	A typical, unconditional happy path scenario of success.
Extensions	Alternate scenarios of success or failure.
Special Requirements	Related non-functional requirements.

Main Success Scenario and Extensions are the two major sections

Use Case Section	Comment
Technology and Data Variations List	Varying I/O methods and data formats.
Frequency of Occurrence	Influences investigation, testing, and timing of implementation.
Miscellaneous	Such as open issues.

Here's an example, based on the template.

> *Please note that this is the book's primary case study example of a detailed use case; it shows many common elements and issues.*
>
> It probably shows much more than you ever wanted to know about a POS system! But, it's for a real POS, and shows the ability of use cases to capture complex real-world requirements, and deeply branching scenarios.

Use Case UC1: Process Sale

Scope: NextGen POS application
Level: user goal
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 easily visible display of entered items and prices. 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.
- Manager: Wants to be able to quickly perform override operations, and easily debug Cashier problems.
- 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 (or Postconditions): Sale is saved. Tax is correctly calculated. Accounting and Inventory are updated. Commissions recorded. Receipt is generated. Payment authorization approvals are recorded.

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, Manager requests an override operation:
    1. System enters Manager-authorized mode.
    2. Manager or Cashier performs one Manager-mode operation. e.g., cash balance change, resume a suspended sale on another register, void a sale, etc.
    3. System reverts to Cashier-authorized mode.

*b. 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.

1a. Customer or Manager indicate to resume a suspended sale.
    1. Cashier performs resume operation, and enters the ID to retrieve the sale.
    2. System displays the state of the resumed sale, with subtotal.
        2a. Sale not found.
            1. System signals error to the Cashier.
            2. Cashier probably starts new sale and re-enters all items.
    3. Cashier continues with sale (probably entering more items or handling payment).

2-4a. Customer tells Cashier they have a tax-exempt status (e.g., seniors, native peoples)
    1. Cashier verifies, and then enters tax-exempt status code.
    2. System records status (which it will use during tax calculations)

3a. Invalid item ID (not found in system):
    1. System signals error and rejects entry.
    2. Cashier responds to the error:
        2a. There is a human-readable item ID (e.g., a numeric UPC):
            1. Cashier manually enters the item ID.
            2. System displays description and price.
                2a. Invalid item ID: System signals error. Cashier tries alternate method.
        2b. There is no item ID, but there is a price on the tag:
            1. Cashier asks Manager to perform an override operation.

2. Managers performs override.
3. Cashier indicates manual price entry, enters price, and requests standard taxation for this amount (because there is no product information, the tax engine can't otherwise deduce how to tax it)

2c. Cashier performs Find Product Help to obtain true item ID and price.
2d. Otherwise, Cashier asks an employee for the true item ID or price, and does either manual ID or manual price entry (see above).

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.

3c. Item requires manual category and price entry (such as flowers or cards with a price on them):
1. Cashier enters special manual category code, plus the price.

3-6a: Customer asks Cashier to remove (i.e., void) an item from the purchase:
This is only legal if the item value is less than the void limit for Cashiers, otherwise a Manager override is needed.
1. Cashier enters item identifier for removal from sale.
2. System removes item and displays updated running total.
2a. Item price exceeds void limit for Cashiers:
1. System signals error, and suggests Manager override.
2. Cashier requests Manager override, gets it, and repeats operation.

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 register.
2. System presents a "suspend receipt" that includes the line items, and a sale ID used to retrieve and resume the sale.

4a. The system supplied item price is not wanted (e.g., Customer complained about something and is offered a lower price):
1. Cashier requests approval from Manager.
2. Manager performs override operation.
3. Cashier enters manual override price.
4. 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:
1. Cashier asks for alternate payment method.
1a. 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 displays their payment for verification.
3. Cashier confirms.
   3a. Cashier cancels payment step:
      1. System reverts to "item entry" mode.
4. System sends payment authorization request to an external Payment Authorization Service System, and requests payment approval.
   4a. System detects failure to collaborate with external system:
      1. System signals error to Cashier.
      2. Cashier asks Customer for alternate payment.
5. System receives payment approval, signals approval to Cashier, and releases cash drawer (to insert signed credit payment receipt).
   5a. System receives payment denial:
      1. System signals denial to Cashier.
      2. Cashier asks Customer for alternate payment.
   5b. Timeout waiting for response.
      1. System signals timeout to Cashier.
      2. Cashier may try again, or ask Customer for alternate payment.
6. System records the credit payment, which includes the payment approval.
7. System presents credit payment signature input mechanism.
8. Cashier asks Customer for a credit payment signature. Customer enters signature.
9. If signature on paper receipt, Cashier places receipt in cash drawer and closes it.

7c. Paying by check…

7d. Paying by debit…

7e. Cashier cancels payment step:
1. System reverts to "item entry" mode.

7f. 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.

9c. Printer out of paper.
1. If System can detect the fault, will signal the problem.
2. Cashier replaces paper.
3. Cashier requests another receipt.

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:
*a. Manager override entered by swiping an override card through a card reader, or entering an authorization code via the keyboard.
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—developed with an OO design in Java). Nevertheless, there is enough detail and complexity 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.

What do the Sections Mean?

Preface Elements

Scope

The scope bounds the system (or systems) under design. Typically, a use case describes use of one software (or hardware plus software) system; in this case it is known as a **system use case**. At a broader scope, use cases can also describe how a business is used by its customers and partners. Such an enterprise-level

process description is called a **business use case** and is a good example of the wide applicability of use cases, but they aren't covered in this introductory book.

Level

EBP p. 88

see the use case "include" relationship for more on subfunction use cases p. 494

In Cockburn's system, use cases are classified as at the user-goal level or the subfunction level, among others. A **user-goal level** use case is the common kind that describe the scenarios to fulfill the goals of a primary actor to get work done; it roughly corresponds to an **elementary business process** (EBP) in business process engineering. A **subfunction-level** use case describes substeps required to support a user goal, and is usually created to factor out duplicate substeps shared by several regular use cases (to avoid duplicating common text); an example is the subfunction use case *Pay by Credit*, which could be shared by many regular use cases.

Primary Actor

The principal actor that calls upon system services to fulfill a goal.

Stakeholders and Interests List—Important!

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 (Postconditions)

First, don't bother with a precondition or success guarantee unless you are stating something non-obvious and noteworthy, to help the reader gain insight. Don't add useless noise to requirements documents.

Preconditions state what *must always* be true before a scenario is begun 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, such as logging in, that has successfully completed. Note that there are conditions that must be true, but are not worth writing, such as "the system has power." Preconditions communicate noteworthy assumptions that the 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" or "Typical Flow." It describes a 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.

Guideline

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).

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

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 important and normally comprise the majority of the text. 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.

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. For example, the customer's interest for a visible display of descriptions and prices is a usability requirement.

Extension scenarios are branches from the main success scenario, and so can be notated with respect to its steps 1...N. 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:
   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: When possible, 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 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.

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.

Performing Another Use Case Scenario

Sometimes, a use case branches to perform another use case scenario. For example, the story *Find Product Help* (to show product details, such as description, price, a picture or video, and so on) is a distinct use case that is sometimes performed while within *Process Sale* (usually when the item ID can't be found). In

Cockburn notation, performing this second use case is shown with underlining, as this example shows:

> 3a. Invalid item ID (not found in system):
> 1. System signals error and rejects entry.
> 2. Cashier responds to the error:
> 2a. ...
> 2c. Cashier performs Find Product Help to obtain true item ID and price.

Assuming, as usual, that the use cases are written with a hyperlinking tool, then clicking on this underlined use case name will display its text.

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.

Recording these with the use case is classic UP advice, and a reasonable location when *first* writing the use case. However, many practitioners find it useful to ultimately move and consolidate all non-functional requirements in the Supplementary Specification, for content management, comprehension, and readability, because these requirements usually have to be considered as a whole during architectural analysis.

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.

Congratulations: Use Cases are Written and Wrong (!)

The NextGen POS team is writing a few use cases in multiple short requirements workshops, in parallel with a series of short timeboxed development iterations that involve production-quality programming and testing. The team is incrementally adding to the use case set, and refining and adapting based on feedback from early programming, tests, and demos. Subject matter experts, cashiers, and developers actively participate in requirements analysis.

That's a good evolutionary analysis process—rather than the waterfall—but a dose of "requirements realism" is still needed. Written specifications and other models give the *illusion* of correctness, but models lie (unintentionally). Only code and tests reveals the truth of what's really wanted and works.

The use cases, UML diagrams, and so forth won't be perfect—guaranteed. They will lack critical information and contain wrong statements. The solution is not the waterfall attitude of trying to record specifications near-perfect and complete at the start—although of course we do the best we can in the time available, and should learn and apply great requirements practices. But it will never be enough.

This isn't a call to rush to coding without any analysis or modeling. There is a middle way, between the waterfall and ad hoc programming: iterative and evolutionary development. In this approach the use cases and other models are incrementally refined, verified, and clarified through early programming and testing.

You know you're on the wrong path if the team tries to write in detail all or most of the use cases before beginning the first development iteration—or the opposite.

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.

.10 Notation: Are There Other Formats? A Two-Column Variation

Some prefer the two-column or conversational format, which emphasizes the interaction 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 Action (or 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. 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. [Cockburn01] 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 visually 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.

.11 Guideline: Write in an Essential UI-Free Style

New and Improved! The Case for Fingerprinting

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 *root-goal* 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. 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 more 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.[4]

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.

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

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

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

Contrasting Examples

Essential Style

Assume that the *Manage Users* use case requires identification and authentication:

> ...
> 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 Administrator.
> 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 Guideline: Write Terse Use Cases

Do you like to read lots of requirements? I didn't think so. So, write terse use cases. Delete "noise" words. Even small changes add up, such as "System authenticates..." rather than "*The* System authenticates..."

6.13 Guideline: Write Black-Box Use Cases

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, one can specify *what* the system must do (the behavior or 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...

.14 Guideline: Take an Actor and Actor-Goal Perspective

Here's the RUP use case definition, from the use case founder Ivar Jacobson:

> 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.*

The phrase *"an observable result of value to a particular actor"* is a subtle but important concept that Jacobson considers critical, because it stresses two attitudes during requirements analysis:

- Write requirements focusing on the users or actors of a system, asking about their goals and typical situations.
- Focus on understanding what the actor considers a valuable result.

Perhaps it seems obvious to stress providing observable user value and focusing on users' typical goals, but the software industry is littered with failed projects that did not deliver what people really needed. The old feature and function list approach to capturing requirements can contribute to that negative outcome because it did not encourage asking who is using the product, and what provides value.

nction lists p. 92

.15 Guideline: How to Find Use Cases

Use cases are defined to satisfy the 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 goals fulfilled through using services of the system.
3. Identify the goals for each primary actor.
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.

Of course, in iterative and evolutionary development, not all goals or use cases will be fully or correctly identified near the start. It's an evolving discovery.

Step 1: Choose 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 the definition of the boundary of the system under design is not clear, it can be clarified by further definition of 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: Find 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: Brainstorm the primary actors first, as this sets up the framework for further investigation.

Are There Questions to Help Find Actors and Goals?

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

- Who starts and stops the system?
- Who does user and security management?
- Who does system administration?
- 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 evaluates logs? Are they remotely retrieved?
In addition to *human* primary actors, are there any external software or robotic systems that call upon services of the system?	Who gets notified when there are errors or failures?

How to Organize the Actors and Goals?

There are at least two approaches:

use case diagrams p. 89

1. As you discover the results, draw them in a use case diagram, naming the goals as use cases.
2. Write an actor-goal list first, review and refine it, and then draw the use case diagram.

If you create an actor-goal list, then in terms of UP artifacts it may be a section in the Vision artifact.

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	analyze sales and performance data
. . .	. . .	. . .	. . .

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

Why Ask About Actor Goals Rather Than Use Cases?

Actors have goals and use applications to help satisfy them. The viewpoint of use case modeling is to find these actors and their goals, and create solutions that produce a result of value. This is slight shift in emphasis for the use case

modeler. Rather than asking "What are the tasks?", one starts by asking: "Who uses the system and what are their 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*.

Thus, here is a key idea regarding investigating requirements and use cases:

> Imagine we are together in a requirements workshop. We could ask either:
>
> - "What do you do?" (roughly a task-oriented question) or,
> - "What are your goals whose results have measurable value?"
>
> Prefer the second question.

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 root 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.

Is the Cashier or Customer the Primary Actor?

Why is the cashier, and not the customer, a primary actor in the use case *Process Sale*?

The answer depends on the system boundary of the system under design, and who we are primarily designing the system for, as illustrated in Figure 6.2. If the enterprise or checkout service is viewed 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), the system services the goal of a trained cashier (and the store) to process the customer's sale. This assumes a traditional checkout environment with a cashier, although there are an increasing number of self-checkout POS systems in operation for direct use by customers.

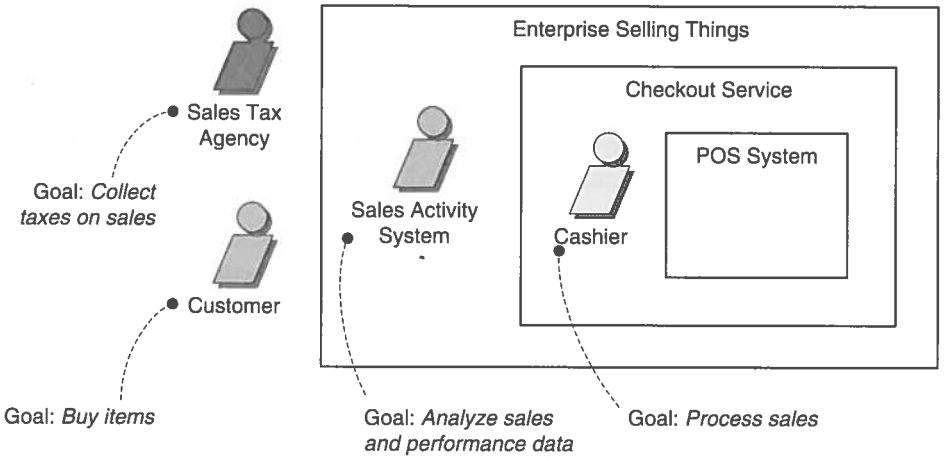


Figure 6.2 Primary actors and goals at different system boundaries.

The customer *is* an actor, but in the context of the NextGen POS, not a primary actor; rather, the cashier is the primary actor because the system is being designed to primarily serve the trained cashier's "power user" goals (to quickly process a sale, look up prices, etc.). The system does not have a UI and functionality that could equally be used by the customer or cashier. Rather, it is optimized to meet the needs and training of a cashier. A customer in front of the POS terminal wouldn't know how to use it effectively. In other words, it was designed for the cashier, not the customer, and so the cashier is not just a proxy for the customer.

On the other hand, consider a ticket-buying website that is identical for a customer to use directly or a phone agent to use, when a customer calls in. In this case, the agent is simply a proxy for the customer—the system is not designed to especially meet the unique goals of the agent. Then, showing the customer rather than the phone agent as the primary actor is correct.

Other Ways to Find Actors and Goals? 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 use case. For example:

External Event	From Actor	Goal/Use Case
enter sale line item	Cashier	process a sale

External Event	From Actor	Goal/Use Case
enter payment	Cashier or Customer	process a sale
. . .		

Step 4: Define Use Cases

In general, define one 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*.

> Start the name of use cases 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.

6.16 Guideline: What Tests Can Help Find Useful Use Cases?

Which of these is a valid use case?

- Negotiate a Supplier Contract
- Handle Returns
- Log In
- Move Piece on Game Board

An argument can be made that all of these are use cases *at different levels*, depending on the system boundary, actors, and goals.

But rather than asking in general, "What is a valid use case?", a more practical question is: "What is a useful level to express use cases for application requirements analysis?" There are several rules of thumb, including:

- The Boss Test
- The EBP Test
- The Size Test

The Boss Test

Your boss asks, "What have you been doing all day?" You reply: "Logging in!" Is your boss happy?

If not, the use case fails the Boss Test, which implies it is not strongly related to achieving results of measurable value. It may be a use case at some low goal level, but not the desirable level of focus for requirements analysis.

That doesn't mean to always ignore boss-test-failing use cases. User authentication may fail the boss test, but may be important and difficult.

The EBP Test

An **Elementary Business Process** (**EBP**) is a term from the business process engineering field,[5] 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].

Focus on use cases that reflect EBPs.

The EBP Test is similar to the Boss Test, especially in terms of the measurable business value qualification.

The definition 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.

The Size Test

A use case is very seldom a single action or step; rather, a use case typically contains many steps, and in the fully dressed format will often require 3–10 pages of text. A common mistake in use case modeling is to define just a single step within a series of related steps as a use case by itself, such as defining a use case called *Enter an Item ID*. You can see a hint of the error by its small size—the use case name will wrongly suggest just one step within a larger series of steps, and if you imagine the length of its fully dressed text, it would be extremely short.

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

Example: Applying the Tests

- Negotiate a Supplier Contract
  - Much broader and longer than an EBP. Could be modeled as a *business* use case, rather than a system use case.
- Handle Returns
  - OK with the boss. Seems like an EBP. Size is good.
- Log In
  - Boss not happy if this is all you do all day!
- Move Piece on Game Board
  - Single step—fails the size test.

Reasonable Violations of the Tests

Although the majority of use cases identified and analyzed for an application should satisfy the tests, exceptions are common.

see the use case "include" relationship for more on linking subfunction use cases p. 494

It is sometimes useful to write separate subfunction-level use cases representing subtasks or steps within a regular EBP-level use case. For example, a subtask or extension such as "paying by credit" may be repeated in several base use cases. If so, it is desirable to separate this into its own use case, even though it does not really satisfy the EBP and size tests, and link it to several base use cases, to avoid duplication of the text.

Authenticate User may not pass the Boss test, but be complex enough to warrant careful analysis, such as for a "single sign-on" feature.

6.17 Applying UML: Use Case Diagrams

The UML provides use case diagram notation to illustrate the names of use cases and actors, and the relationships between them (see Figure 6.3).[6]

> 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.

6. "Cash In" is the act of a cashier arriving with a drawer insert with cash, logging in, and recording the cash amount in the drawer insert.

World-class use case experts such as Fowler and Cockburn, among others, downplay use case diagrams and use case relationships, and instead focus on writing. 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.

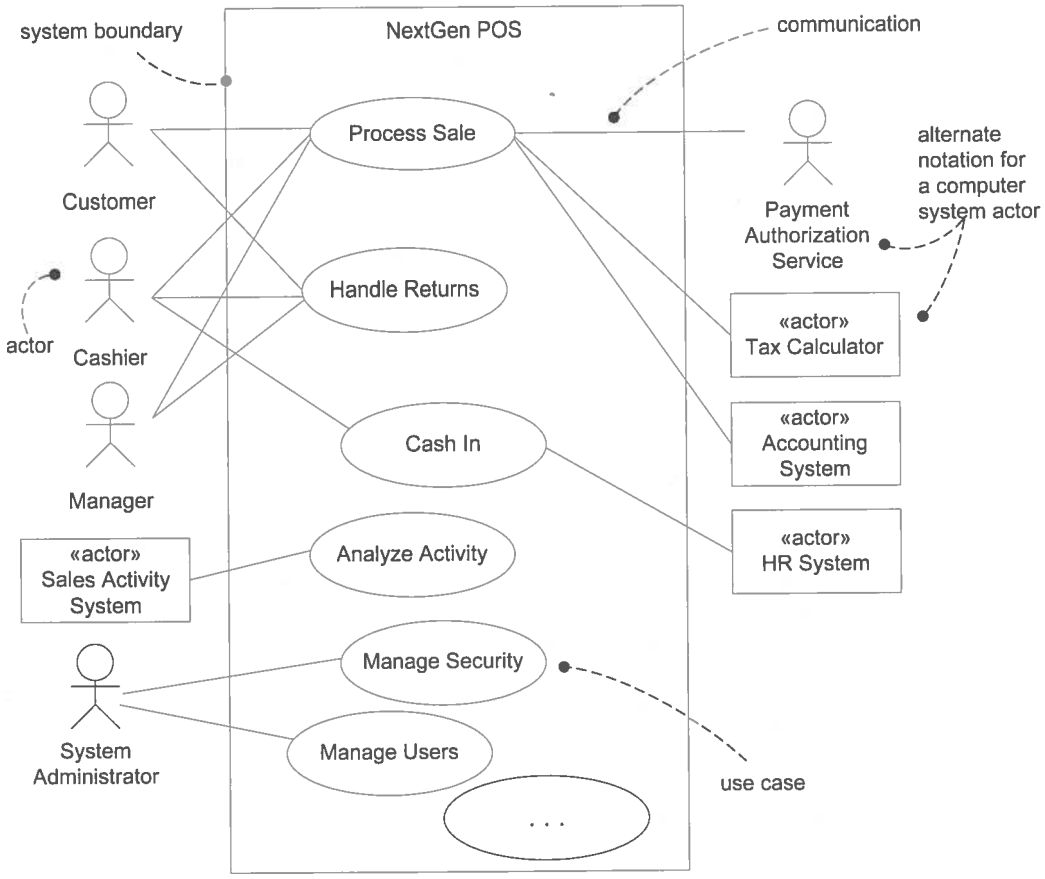


Figure 6.3 Partial use case context diagram.

Guideline

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 6.3.

Guideline: Diagramming

Figure 6.4 offers diagram advice. Notice the actor box with the symbol «actor». This style is used for UML **keywords** and **stereotypes**, and includes guillemet symbols—special *single*-character brackets («actor», not <<actor>>) most widely known by their use in French typography to indicate a quote.

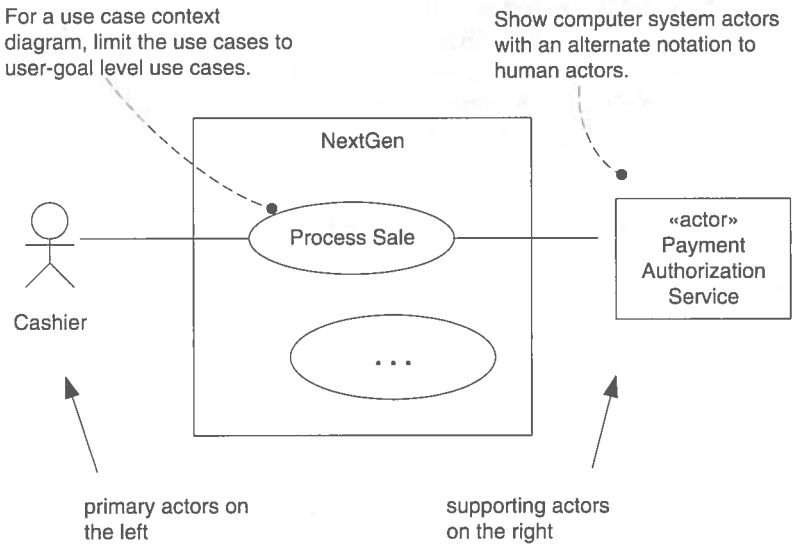


Figure 6.4 Notation suggestions.

To clarify, some prefer to highlight external computer system actors with an alternate notation, as illustrated in Figure 6.5.

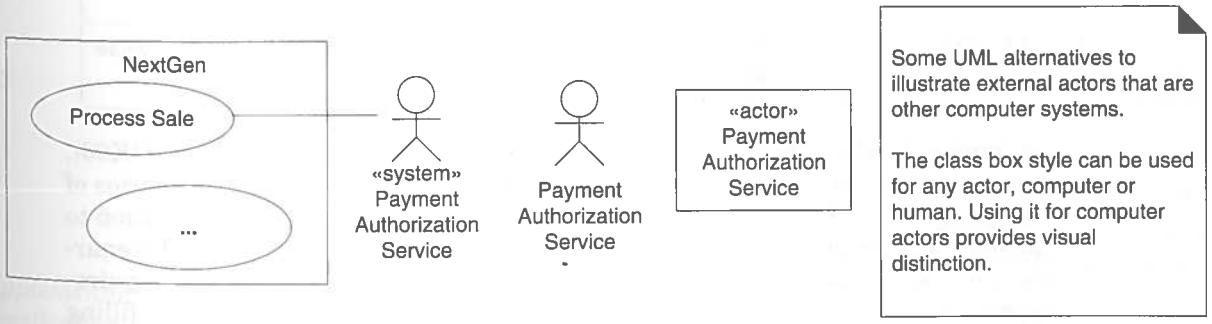


Figure 6.5 Alternate actor notation.

Guideline: Downplay Diagramming, Keep it Short and Simple

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 focusing on writing text, effort has been misplaced.

8 Applying UML: Activity Diagrams

activity rams p. 477

The UML includes a diagram useful to visualize workflows and business processes: activity diagrams. Because use cases involve process and workflow analysis, these can be a useful alternative or adjunct to writing the use case text, especially for *business* use cases that describe complex workflows involving many parties and concurrent actions.

9 Motivation: Other Benefits of Use Cases? Requirements in Context

vation p. 64

A motivation for use cases is focusing on who the key actors are, their goals, and common tasks. Plus, in essence, use cases are a simple, widely-understood form (a story or scenario form).

Another motivation is to replace detailed, low-level function lists (which were common in 1970s traditional requirements methods) with use cases. These lists tended to look as follows:

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.

As implied by the title of the book *Uses Cases: Requirements in Context* [GK00], use cases organize a set of requirements *in the context of* the typical scenarios of using a system. That's a good thing—it improves cohesion and comprehension to consider and group requirements by the common thread of user-oriented scenarios (i.e., use cases). In a recent air traffic control system project: the requirements were originally written in the old-fashioned function list format, filling volumes of incomprehensible, unrelated specifications. A new leadership team analyzed and reorganized the massive requirements primarily by use cases. This provided a unifying and understandable way to pull the requirements together—into stories of requirements in context of use.

Supplementary Specification p. 104

To reiterate, however, use cases are not the only necessary requirements artifact. Non-functional requirements, report layouts, domain rules, and other hard-to-place elements are better captured in the UP Supplementary Specification.

High-Level System Feature Lists Are Acceptable

Vision p. 109

Although detailed function lists are undesirable, a terse, high-level feature list, called *system features,* added to a Vision document can usefully summarize system functionality. In contrast to 50 pages of low-level features, a system features list includes only a few dozen items. It provides a succinct summary of 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
- ...

When Are Detailed Feature Lists Appropriate Rather than Use Cases?

Sometimes use cases do not really fit; some applications cry 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 Web Services 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.

6.20 Example: Monopoly Game

The only significant use case in the Monopoly software system is *Play Monopoly Game*—even if it doesn't pass the Boss Test! Since the game is run as a computer simulation simply watched by one person, we might say that person is an observer, not a player.

This case study will show that use cases aren't always best for behavioral requirements. Trying to capture all the game rules in the use case format is awkward and unnatural. Where do the game rules belong? First, more generally, they are **domain rules** (sometimes called business rules). In the UP, domain rules can be part of the Supplementary Specification (SS). In the SS "domain rules" section there would probably be a reference to either the official paper booklet of rules, or to a website describing them. In addition, there may be a pointer to these rules from the use case text, as shown below.

Supplementary Specification p. 104

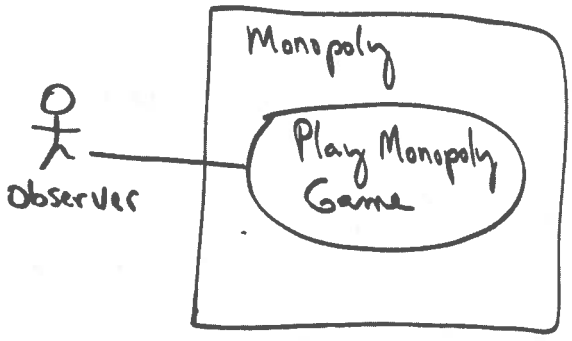


Figure 6.6 Use case diagram ("context diagram") for Monopoly system.

The text for this use case is very different than the NextGen POS problem, as it is a simple simulation, and the many possible (simulated) player actions are captured in the domain rules, rather than the Extensions section.

Use Case UC1: Play Monopoly Game

Scope: Monopoly application
Level: user goal
Primary Actor: Observer
Stakeholders and Interests:
– Observer: Wants to easily observe the output of the game simulation.

Main Success Scenario:
1. Observer requests new game initialization, enters number of players.
2. Observer starts play.

3. System displays game trace for next player move (see domain rules, and "game trace" in glossary for trace details).

Repeat step 3 until a winner or Observer cancels.

Extensions:
*a. At any time, System fails:
(To support recovery, System logs after each completed move)
1. Observer restarts System.
2. System detects prior failure, reconstructs state, and prompts to continue.
3. Observer chooses to continue (from last completed player turn).

Special Requirements:
– Provide both graphical and text trace modes.

6.21 Process: How to Work With Use Cases in Iterative Methods?

Use cases are central to the UP and many other iterative methods. The UP encourages **use-case driven development**. This implies:

- Functional 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.
- Functional or system testing corresponds to the scenarios of use cases.
- UI "wizards" or shortcuts may be created for the most common scenarios of important use cases to ease common tasks.

How to Evolve Use Cases and Other Specifications Across the Iterations?

This section reiterates a key idea in evolutionary iterative development: The timing and level of effort of specifications across the iterations. Table 6.1 presents a sample (not a recipe) that 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, the team deliberately delays in continuing with deep requirements work until near the end of the first elaboration iteration.

This is a key difference between iterative development and 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." The act of building software rapidly surfaces assumptions and questions that need clarification.

In the UP, use case writing is encouraged in a requirements workshop. Figure 6.7 offers suggestions on the time and space for doing this work.

Discipline	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. Pick 10% from the high-level list to analyze and write in detail. This 10% will be the most architecturally important, risky, and high-business value.	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 requirements.	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.

When Should Various UP Artifact (Including Use Cases) be Created?

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 architecturally significant 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.

Discipline	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 Specification	s	r		
	Glossary	s	r		
Design	Design Model		s	r	
	SW Architecture Document		s		

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

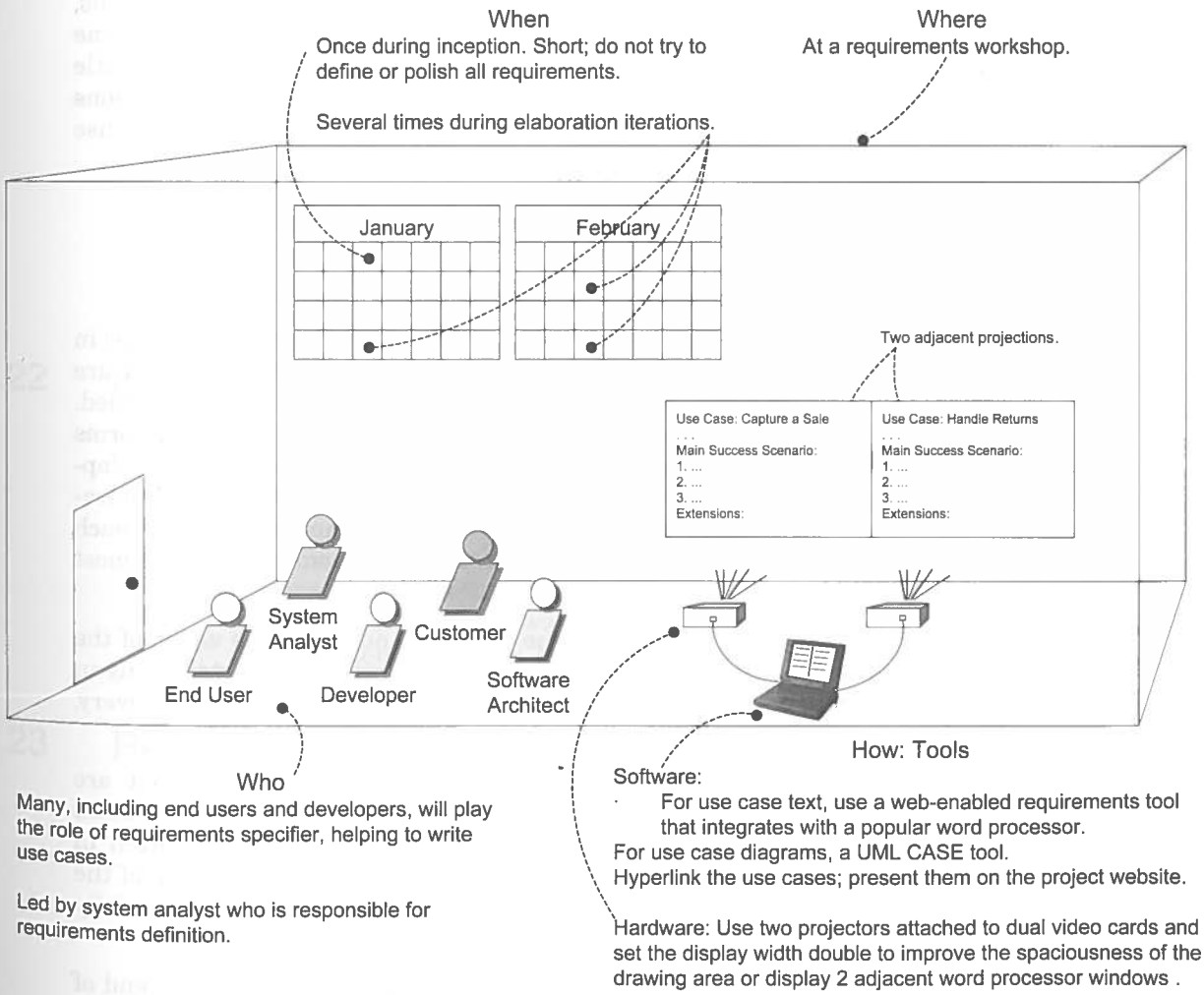


Figure 6.7 Process and setting context for writing use cases.

How to Write Use Cases in 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 use cases are identified *by name*, 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, require building the core architecture, or that 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 deep investigation of a small sample of influential use cases. Perhaps this means two use cases: *Process Sale* and *Handle Returns*.

How to Write Use Cases in 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.

How to Write Use Cases in Construction?

The construction phase 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 may still be some minor use case writing and perhaps requirements workshops, but much less so than in elaboration.

Case Study: Use Cases in the NextGen Inception Phase

As described in the previous sections, 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	Casual	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 ...

6.22 History

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. Jacobson's use case idea was seminal and widely appreciated. Although many have made contributions to the subject, arguably the most influential and coherent next step in defining what use cases are and how to write them came from Alistair Cockburn (who was trained by Jacobson), based on his earlier work and writings stemming from 1992 onwards [e.g., Cockburn01].

6.23 Recommended Resources

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.

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

Patterns for Effective Use Cases by Adolph and Bramble in some ways picks up where *Writing* leaves off, covering many useful tips—in pattern format—related to the process of creating excellent use cases (team organization, methodology, editing), and how to better structure and write them (patterns for judging and improving their content and organization).

Use cases are usually best written with a partner during a requirements workshop. An excellent guide to the art of running a workshop is *Requirements by Collaboration: Workshops for Defining Needs* by Ellen Gottesdiener.

Use Case Modeling by Bittner and Spence is another quality resource by two experienced modelers who also understand iterative and evolutionary development and the RUP, and present use case analysis in that context.

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

Use Cases: Requirements in Context by Kulak and Guiney is also worthwhile. It emphasizes the important viewpoint—as the title states—that use cases are not just another requirements artifact, but are the central vehicle that drives requirements work.