Rekayasa Kebutuhan Perangkat Lunak

Bambang Setiawan
Departemen Sistem Informasi – ITS Surabaya @2024

MATA KULIAH	**ES234210 : Rekayasa Kebutuhan Perangkat Lunak**
	Kredit : 3 SKS
	Semester : 2

Deskripsi Matakuliah

Mata kuliah ini bertujuan untuk memberikan kemampuan kepada mahasiswa untuk dapat melakukan penggalian kebutuhan dan perancangan perangkat lunak, kemudian mendokumentasinya ke dalam dokumen spesifikasi kebutuhan perangkat lunak (SKPL) dan dokumen desain perangkat lunak (DPL).

CAPAIAN PEMBELAJARAN PRODI YANG DIDUKUNG

- Menguasai konsep & metode Perancangan Perangkat Lunak / Sistem Informasi
- Mampu menganalisis kebutuhan informasi individu, staf, unit organisasi, atau organisasi.
- Mampu merancang aplikasi SI.
- Mendengar, menyelami, mewawancarai, dan menganalisis berbagai sumber informasi.

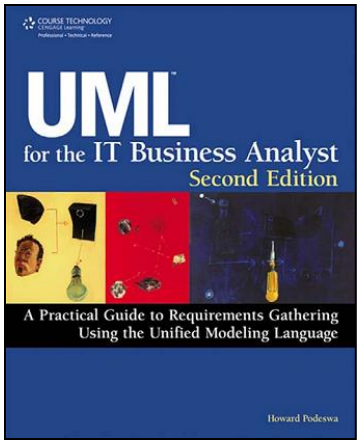
Referensi

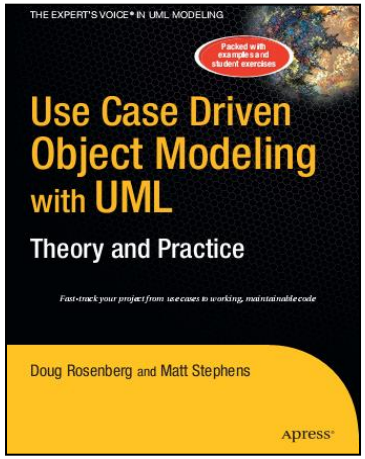
eBook

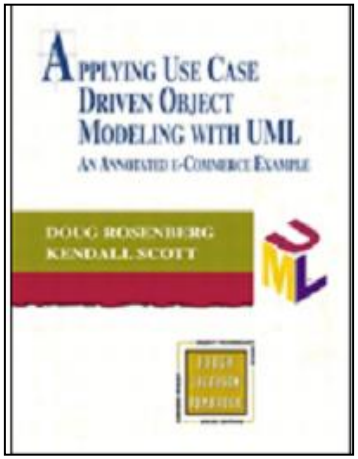
- UML for the IT Business Analyst
- Use Case Driven Object Modeling with UML
- Applying Use Case Driven Object Modeling with UML
- A Guide to the Business Analysis Body of Knowledge Version 3.0
- A Guide to the Software Engineering Body of Knowledge Version 3.0

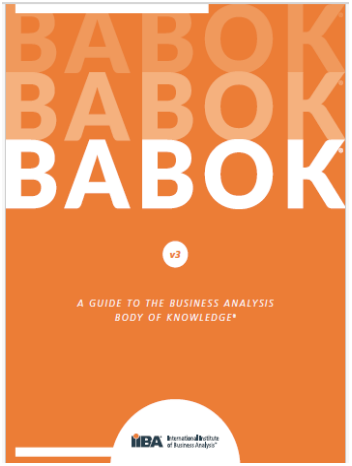
S/W Tools: **Enterprise Architecture**

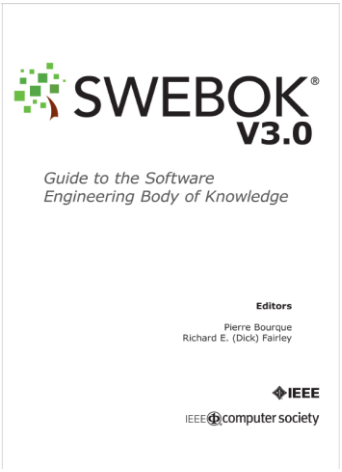
- http://www.sparxsystems.com.au/resources/tutorial_home/index.html
- http://www.sparxsystems.com





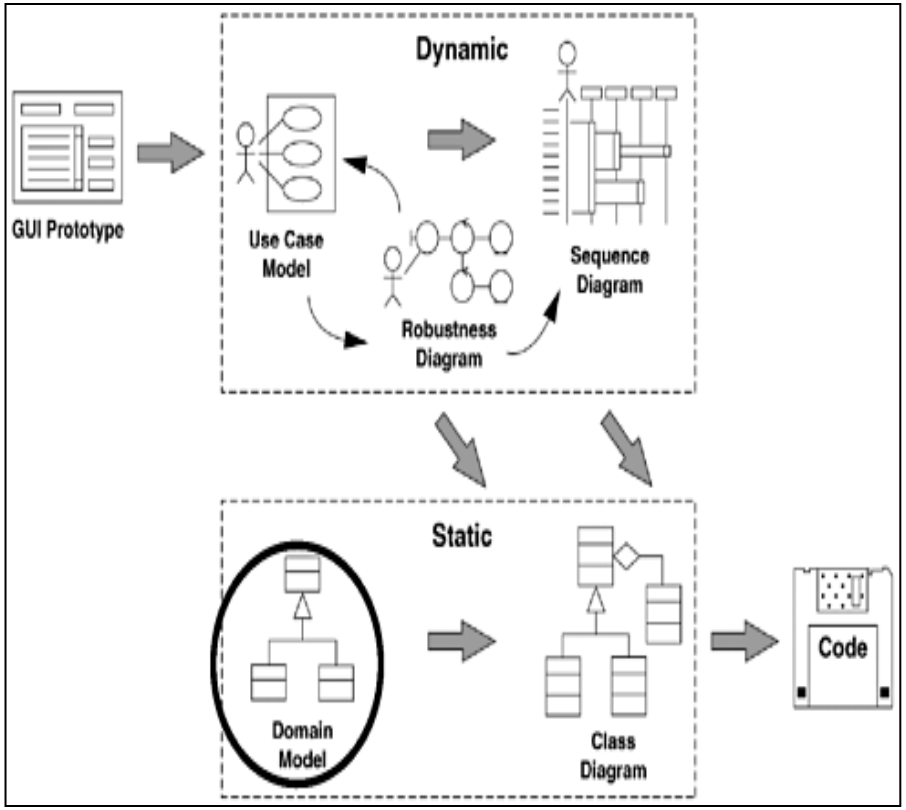




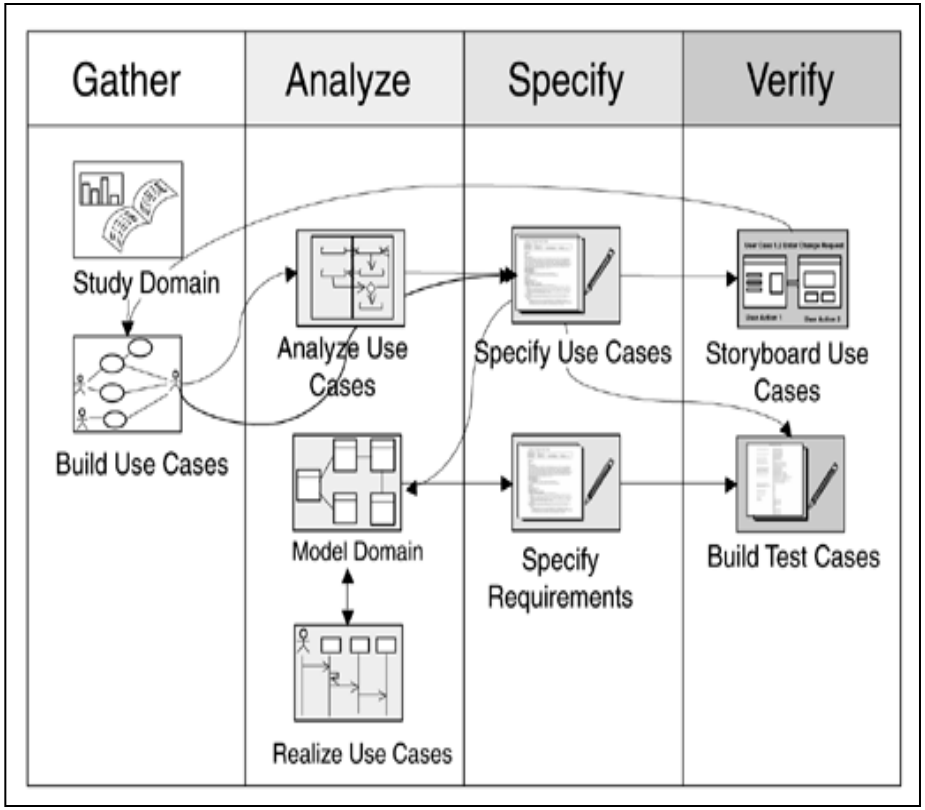


SWEBOK V3	SWEBOK V4
Introduction	Introduction
1. Software Requirements	1. Software Requirements
	2. Software Architecture
2. Software Design	3. Software Design
3. Software Construction	4. Software Construction
4. Software Testing	5. Software Testing
	6. Software Engineering Operations
5. Software Maintenance	7. Software Maintenance
6. Software Configuration Management	8. Software Configuration Management
7. Software Engineering Management	9. Software Engineering Management
8. Software Engineering Process	10. Software Engineering Process
9. Software Engineering Models and Methods	11. Software Engineering Models and Methods
10. Software Quality	12. Software Quality
	13. Software Security
11. Software Engineering Professional Practice	14. Software Engineering Professional Practice
12. Software Engineering Economics	15. Software Engineering Economics
13. Computing Foundations	16. Computing Foundations
14. Mathematical Foundations	17. Mathematical Foundations
15. Engineering Foundations	18. Engineering Foundations
Appendix A. Knowledge Area Specifications	Appendix A. Knowledge Area Specifications
Appendix B. Standards	Appendix B. Standards

Use Case Driven Object
(Iconix)



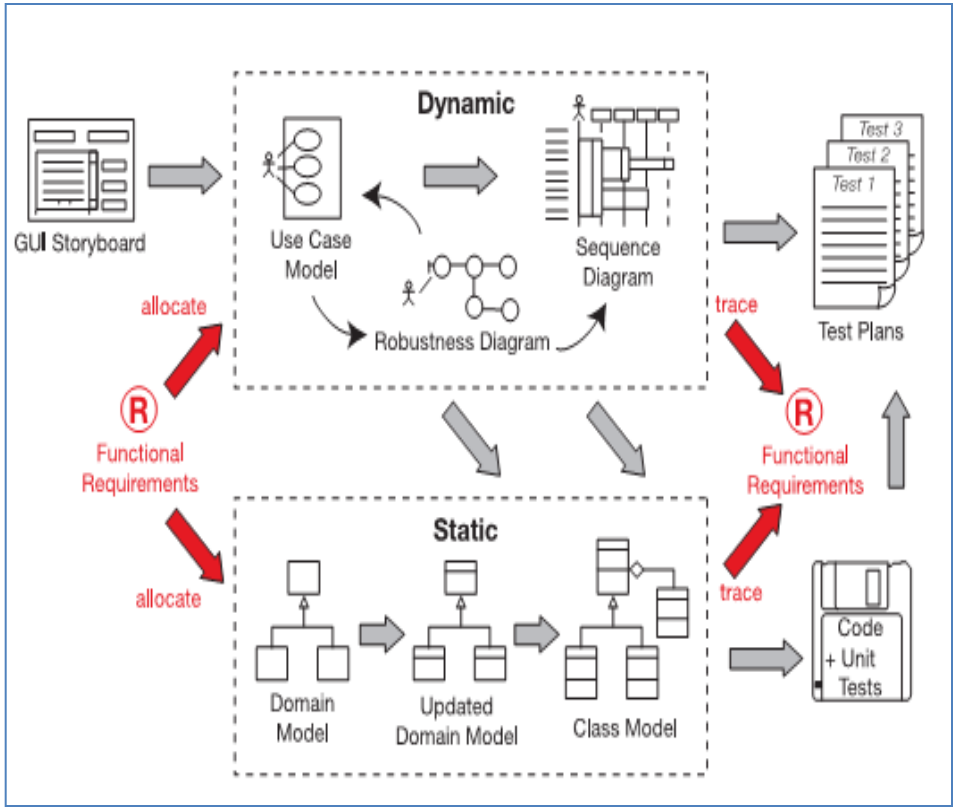
Rational Unified Process
(*Rational* Software Corporation – IBM)



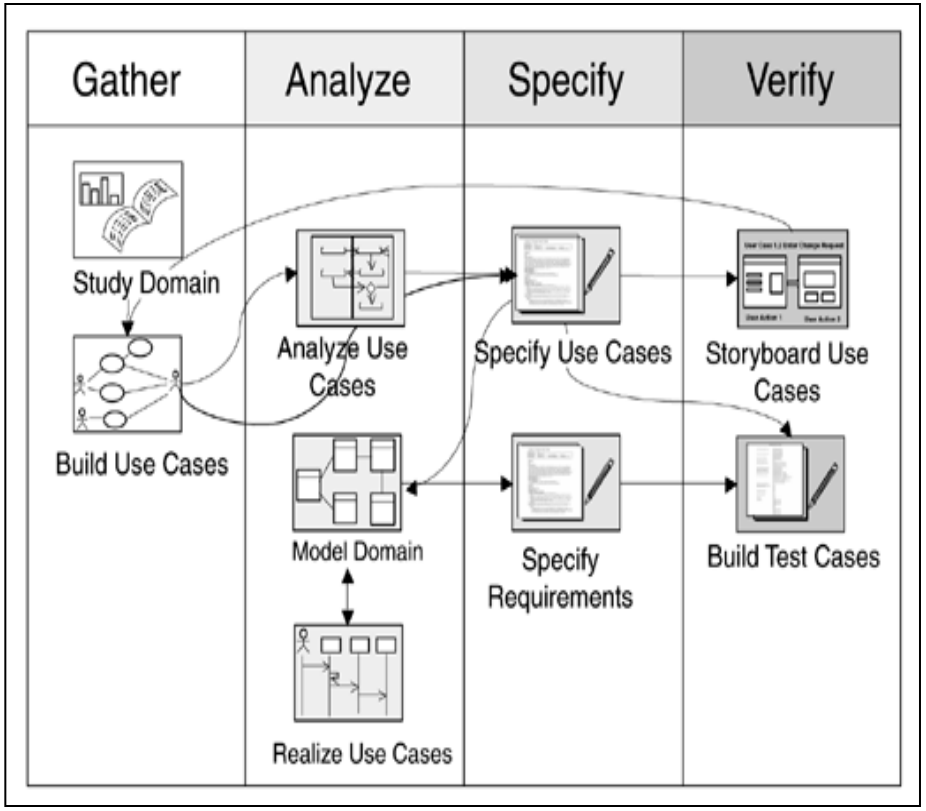
Enterprise Architect Software

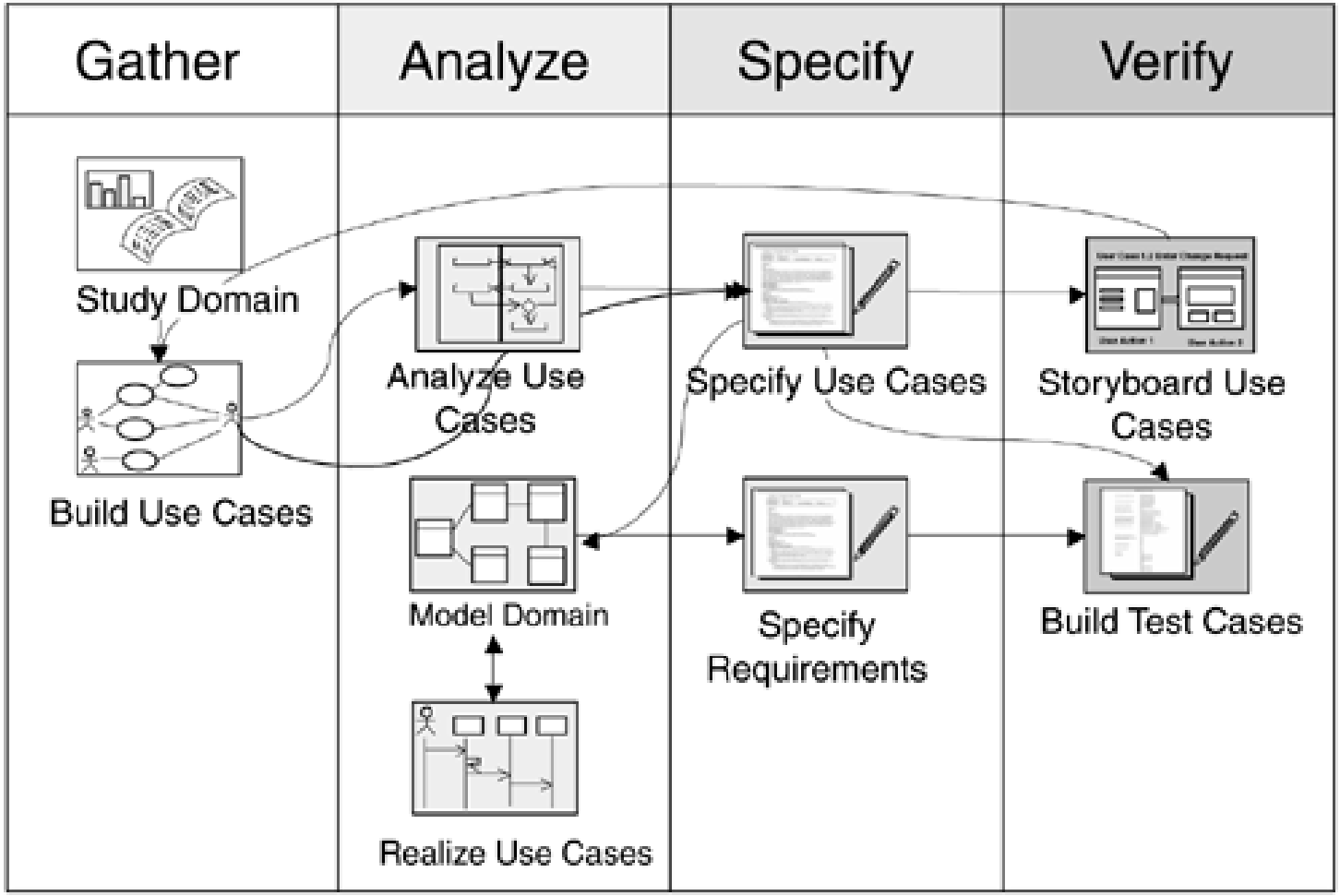
Rational Rose Software

Use Case Driven Object



Software Development with Unified Process




Gather
Analyze
Specify
Verify
Study Domain
Build Use Cases
Analyze Use Cases
Model Domain
Realize Use Cases
Specify Use Cases
Specify Requirements
Storyboard Use Cases
Build Test Cases

ICONIX Process

http://en.wikipedia.org/wiki/ICONIX

- *ICONIX* is a software development methodology that predates both the Rational Unified Process (RUP), Extreme Programming (XP), and Agile software development.
- Like RUP, the **ICONIX** process is UML Use Case driven but more lightweight than RUP.
- Unlike the XP and Agile approaches, ICONIX provides sufficient requirement and design documentation, but without analysis paralysis.
- The ICONIX Process **uses only four UML-based diagrams** in **a four-step process** that turns use case text into working code.

ICONIX Process

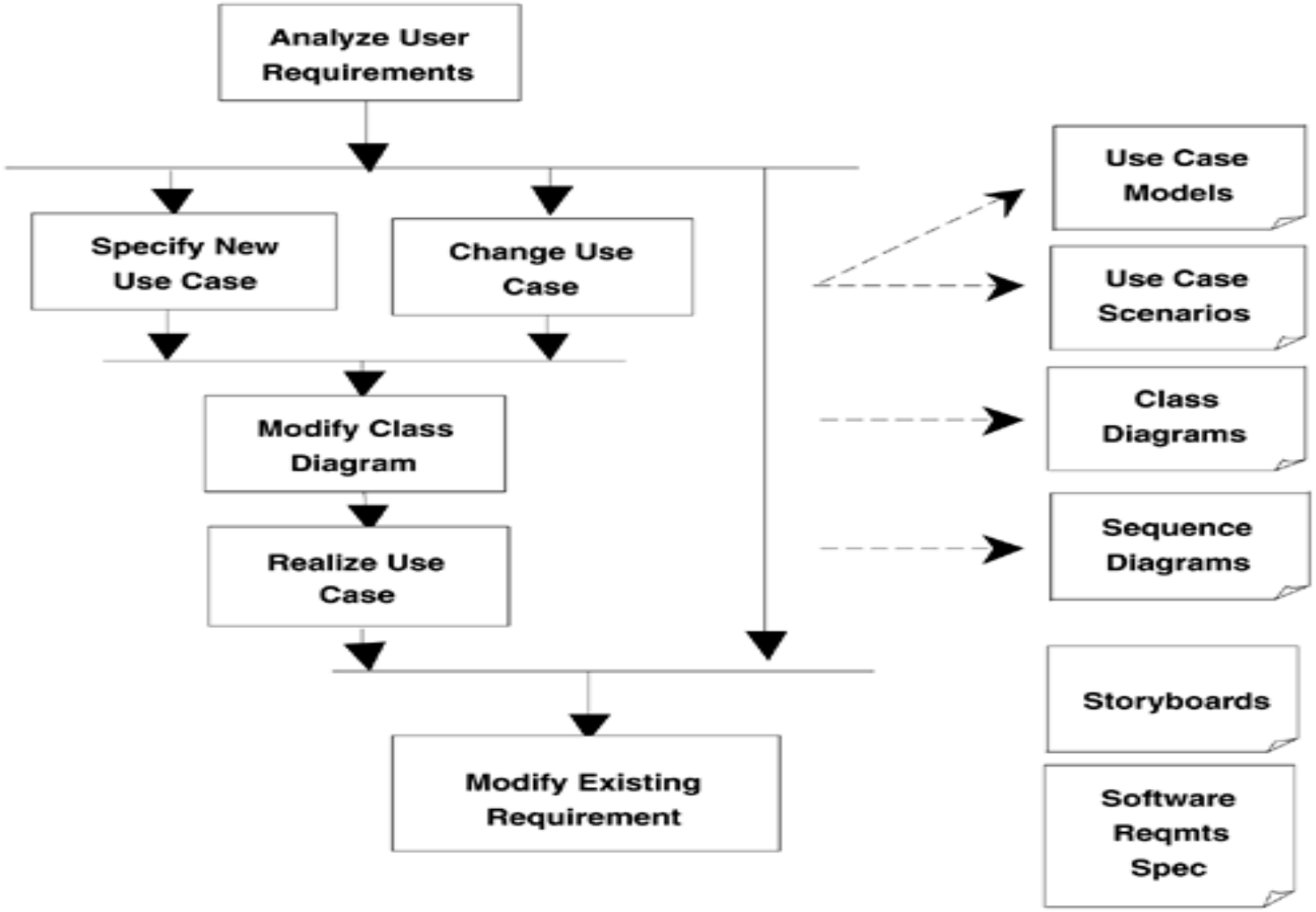
http://en.wikipedia.org/wiki/ICONIX

- A principal distinction of ICONIX is its use of robustness analysis, a method for bridging the gap between analysis and design.

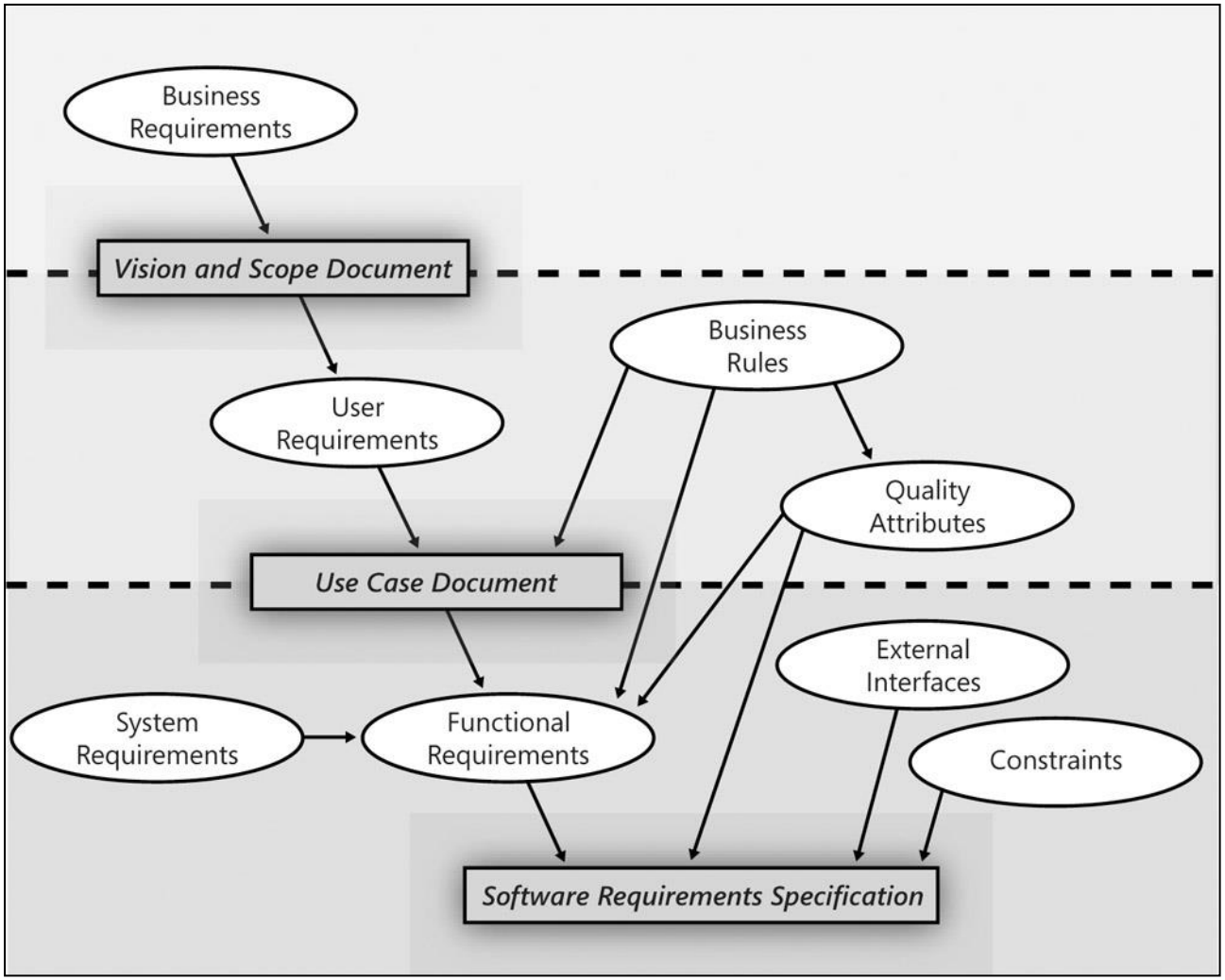
  Robustness analysis reduces the ambiguity in use case descriptions, by ensuring that they are written in the context of an accompanying domain model.

  This process makes the use cases much easier to design, test and estimate.
- The ICONIX Process is described in the book *Use Case Driven Object Modeling with UML: Theory and Practice*[1].
- Essentially, the ICONIX Process describes the core "logical" analysis and design modeling process.

  However, the process can be used without much tailoring on projects that follow different project management.


Analyze User Requirements
Specify New Use Case
Change Use Case
Modify Class Diagram
Realize Use Case
Modify Existing Requirement
Use Case Models
Use Case Scenarios
Class Diagrams
Sequence Diagrams
Storyboards
Software Reqmts Spec

Connections among several types of requirements information



Business Model

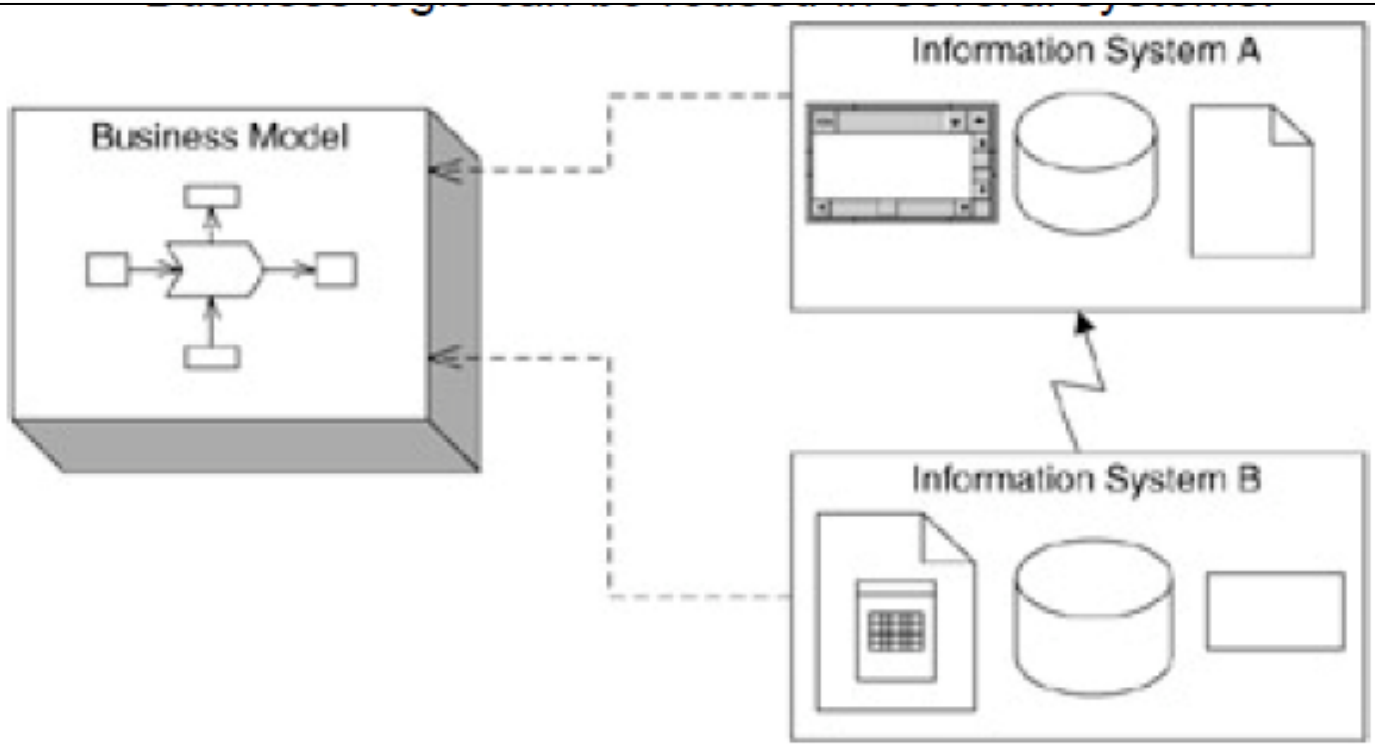


Figure 1.2: A business model can be used as the basis for defining the requirements on information systems.

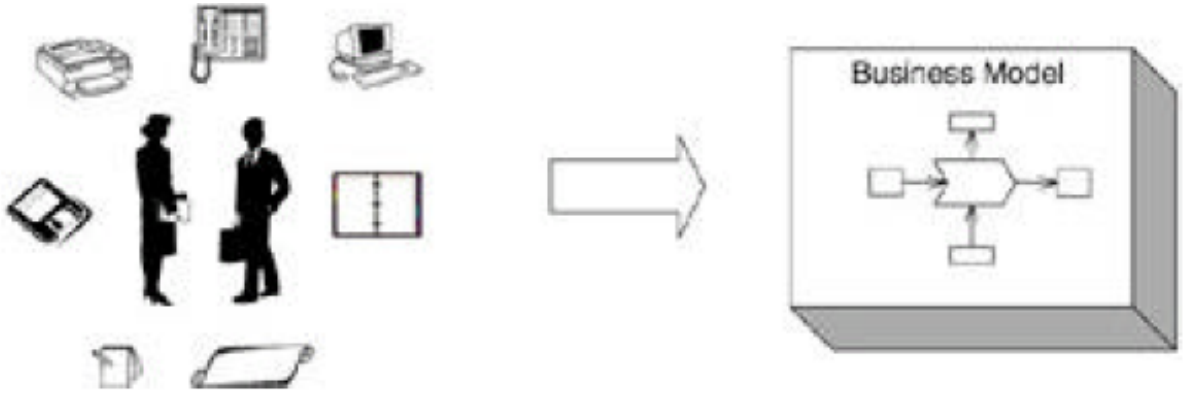
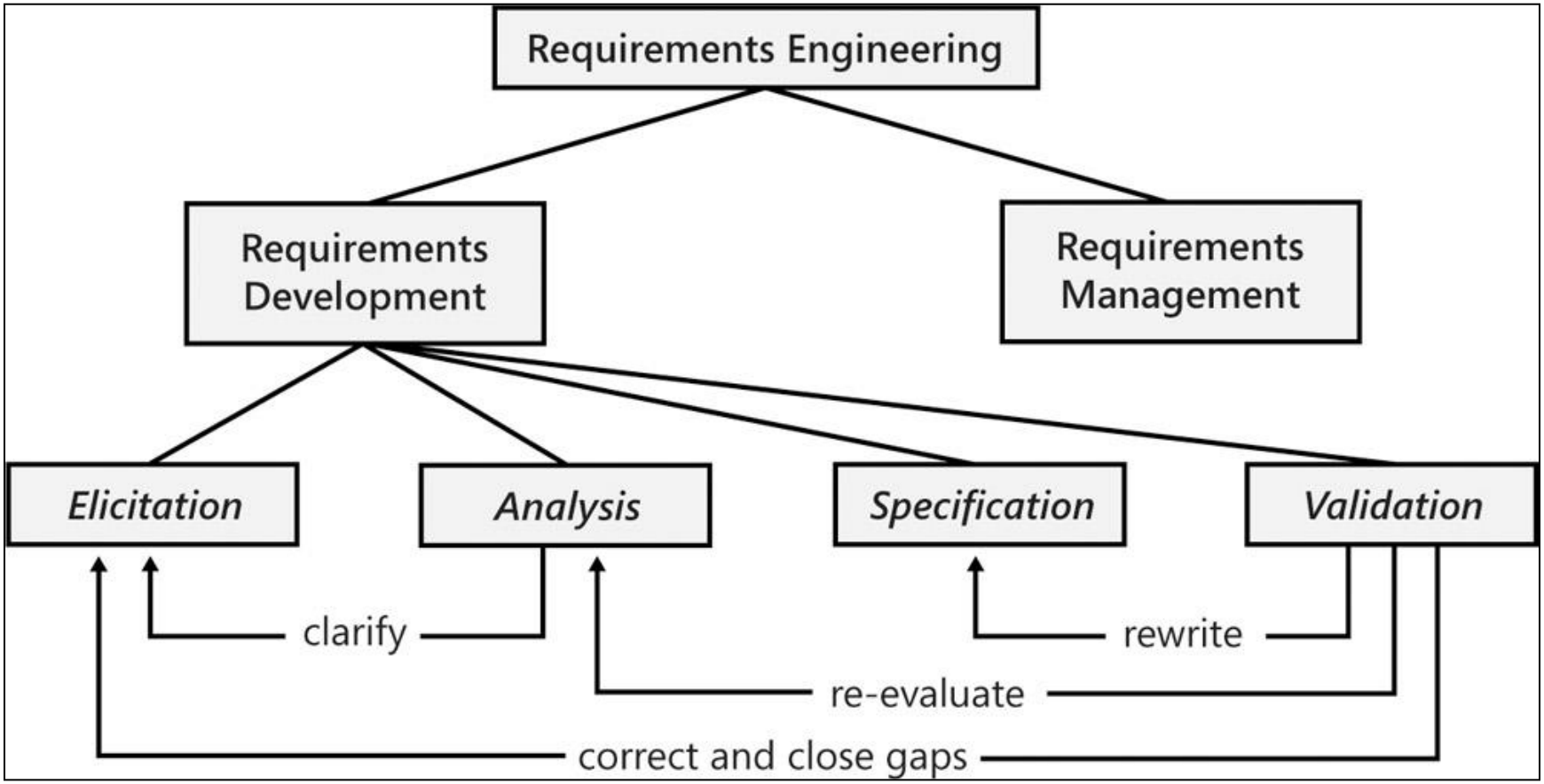


Figure 1.1: A business model is a simplified view of a business.

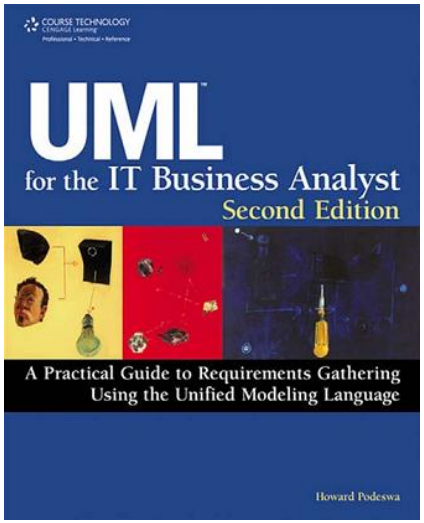
A *business model* is an abstraction of how a business functions. Its details differ according to the perspective of the person creating the model, each of whom will naturally have a slightly different viewpoint of the goals and visions of the business, including its efficiency and the various elements that are acting in concert within the business. This is normal, and the business model will not completely resolve these differences. What the business model will do is provide a simplified view of the business structure that will act as the basis for communication, improvements, or innovations, and define the information systems requirements that are necessary to support the business. It isn't necessary for a business model to capture an absolute picture of the business or to describe every business detail.

Subcomponents of requirements engineering



B.O.O.M.

- ***Business Object Oriented Modeling*** (B.O.O.M.), merupakan prosedur untuk memunculkan, menganalisis, mendokumentasikan, dan persyaratan pengujian menggunakan teknik berorientasi objek (*a procedure for eliciting, analyzing, documenting, and testing requirements using object-oriented and complementary techniques*).
- Tahapan B.O.O.M.(*The B.O.O.M. Steps*):
    1. *Initiation*
    2. *Discovery*
    3. *Construction*
    4. *Final Verification and Validation* (V&V)
    5. *CloseOut*



The B.O.O.M. Steps

(UML FOR THE IT BUSINESS ANALYST, 2nd EDITION , *pg. 33*)

1. ***Initiation***: Membuat kasus bisnis untuk suatu proyek. Kegiatan berdasarkan pada pengalaman pengguna dan pada konsep rancangan arsitektur bukti dari konsep. Upaya prototyping selama fase Inisiasi harus berdasar risiko dan terbatas untuk memperoleh keyakinan bahwa solusi adalah mungkin.

   Make the business case for the project. Work also begins on the user experience and on drafts of architectural proof of concepts. The prototyping effort during the Initiation phase should be risk-driven and limited to gaining confidence that a solution is possible.

The B.O.O.M. Steps

(UML FOR THE IT BUSINESS ANALYST, 2nd EDITION , *The B.O.O.M. Steps, pg. 33*)

2. ***Discovery*****:** investigasi perilaku mengarah ke pemahaman tentang perilaku solusi yang diinginkan. (Pada iteratif proyek, analisis kebutuhan puncak selama fase ini tetapi tidak pernah hilang sama sekali). Selama fase ini, bukti dari konsep arsitektur juga dibangun.

   Conduct investigation leading to an understanding of the solution's desired behavior. (On iterative projects, requirements analysis peaks during this phase but never disappears entirely.) During this phase, architectural proofs of concept are also constructed.

The B.O.O.M. Steps

(UML FOR THE IT BUSINESS ANALYST, 2nd EDITION , *The B.O.O.M. Steps, pg. 33*)

3. ***Construction*****:** Melengkapi analisis dan desain, kode, mengintegrasikan, dan menguji perangkat lunak. (Pada iteratif proyek, kegiatan ini dilakukan untuk setiap iterasi dalam fase Desain dan pelaksanaan coding dilakukan pada semua tahap, teetapi puncak pelaksanaannya ada pada fase ini).

Complete the analysis and design, code, integrate, and test the soft-ware. (On iterative projects, these activities are performed for each iteration within the phase. Design and coding appear in all phases, but peak during this phase.)

The B.O.O.M. Steps

(UML FOR THE IT BUSINESS ANALYST, 2nd EDITION , *The B.O.O.M. Steps, pg. 34*)

4. ***Final Verification and Validation* (V&V):** Melakukan Final Testing sebelum produk atau jasa yang dialihkan ke dalam proses produksi. (Meskipun Final Testing dilakukan pada fase ini, kegiatan pengujian dapat dilakukan pada semua tahapanSDLC-misalnya, sebelum disain atau sebagai pengganti untuk itu).

   Perform final testing before the product or service is transitioned into production. (While final testing occurs in this phase, testing activities may occur throughout the SDLC—for example, before design or as a replacement for it.)

The B.O.O.M. Steps

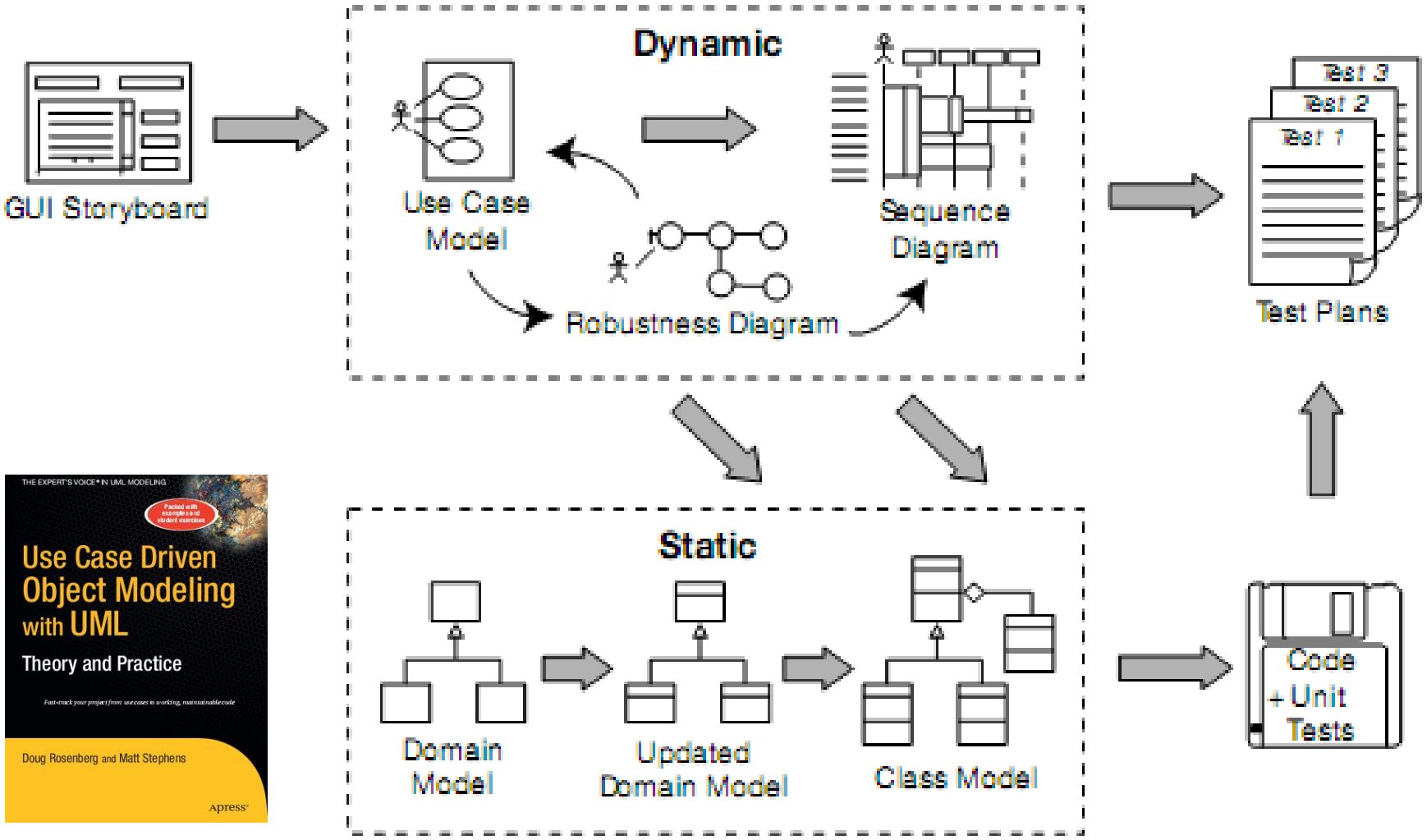
(UML FOR THE IT BUSINESS ANALYST, 2nd EDITION , *The B.O.O.M. Steps, pg. 33*)

5. ***CloseOut*****:** Mengelola dan mengkoordinasikan penyebaran ke dalam proses produksi dan menutup proyek TI.

Manage and coordinate deployment into production and close the IT project.

Iconix (Usecase Driven Object)Process

(Use Case Driven Object Modeling with UML: Theory and Practice, pg. 1)



1. REQUIREMENTS

a. **Functional requirements**: Define what the system should be capable of doing. Depending on how your project is organized, either you'll be involved in creating the functional requirements or the requirements will be "handed down from on high" by a customer or a team of business analysts.

b. **Domain modeling**: Understand the problem space in unambiguous terms.

c. **Behavioral requirements**: Define how the user and the system will interact (i.e., write the first-draft use cases). We recommend that you start with a GUI prototype (storyboarding the GUI) and identify all the use cases you're going to implement, or at least come up with a first-pass list of use cases, which you would reasonably expect to change as you explore the requirements in more depth.

d. **Milestone 1**:

Requirements Review:

Make sure that the use case text matches the customer's expectations. Note that you might review the use cases in small batches, just prior to designing them.

Then in each iteration (i.e., for a small batch of use cases), you do the following.

2. ANALYSIS/PRELIMINARY DESIGN

a. **Robustness analysis**: Draw a robustness diagram (an "object picture" of the steps in a use case), rewriting the use case text as you go.

b. **Update the domain model** while you're writing the use case and drawing the robustness diagram. Here you will discover missing classes, correct ambiguities, and add attributes to the domain objects (e.g., identify that a Bookobject has a Title, Author, Synopsis, etc.).

c. **Name all the logical software functions** (controllers) needed to make the use case work.

d. **Rewrite the first draft use cases**.

3. Milestone 2: Preliminary Design Review (PDR)

4. DETAILED DESIGN

a. **Sequence diagramming** : Draw a sequence diagram (one sequence diagram per use case) to show in detail how you're going to implement the use case. The primary function of sequence diagramming is to allocate behavior to your classes.
b. **Update the domain model:** while you're drawing the sequence diagram, and add operations to the domain objects. By this stage, the domain objects are really domain classes, or entities, and the domain model should be fast becoming a static model , or class diagram—a crucial part of your detailed design.
c. **Clean up the static model**.

5. Milestone 3: Critical Design Review (CDR)

6. IMPLEMENTATION

a. **Coding/unit testing** : Write the code and the unit tests. (Or, depending on your preferences, write the unit tests and then the code.)

b. **Integration and scenario testing** : Base the integration tests on the use cases, so that you're testing both the basic course and the alternate courses.

c. **Perform a Code Review and Model Update** to prepare for the next round of development work.

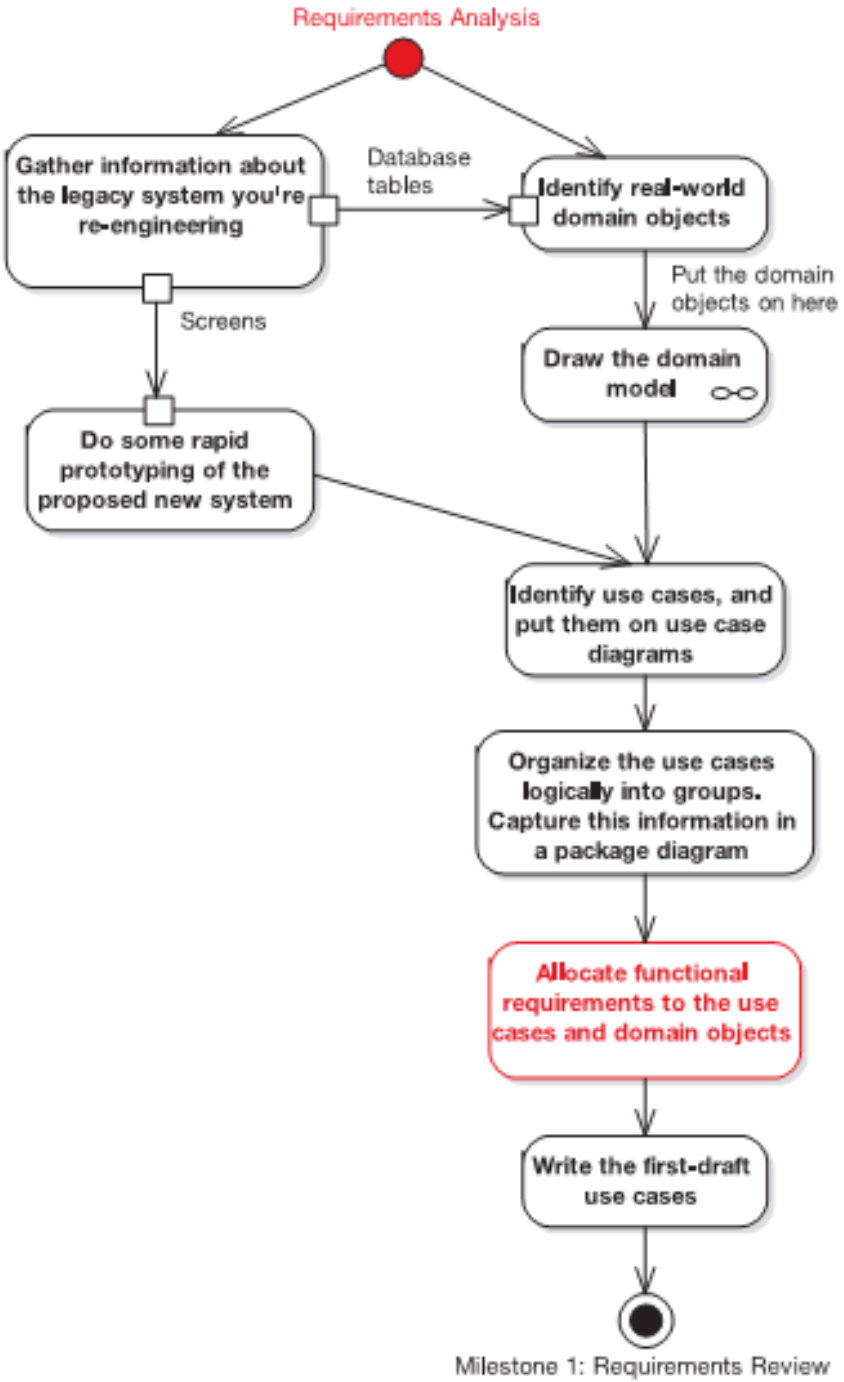
Top 10 Requirements Gathering Guidelines

The principles discussed in this chapter can be summed up as a list of guidelines. Our top 10 list follows.

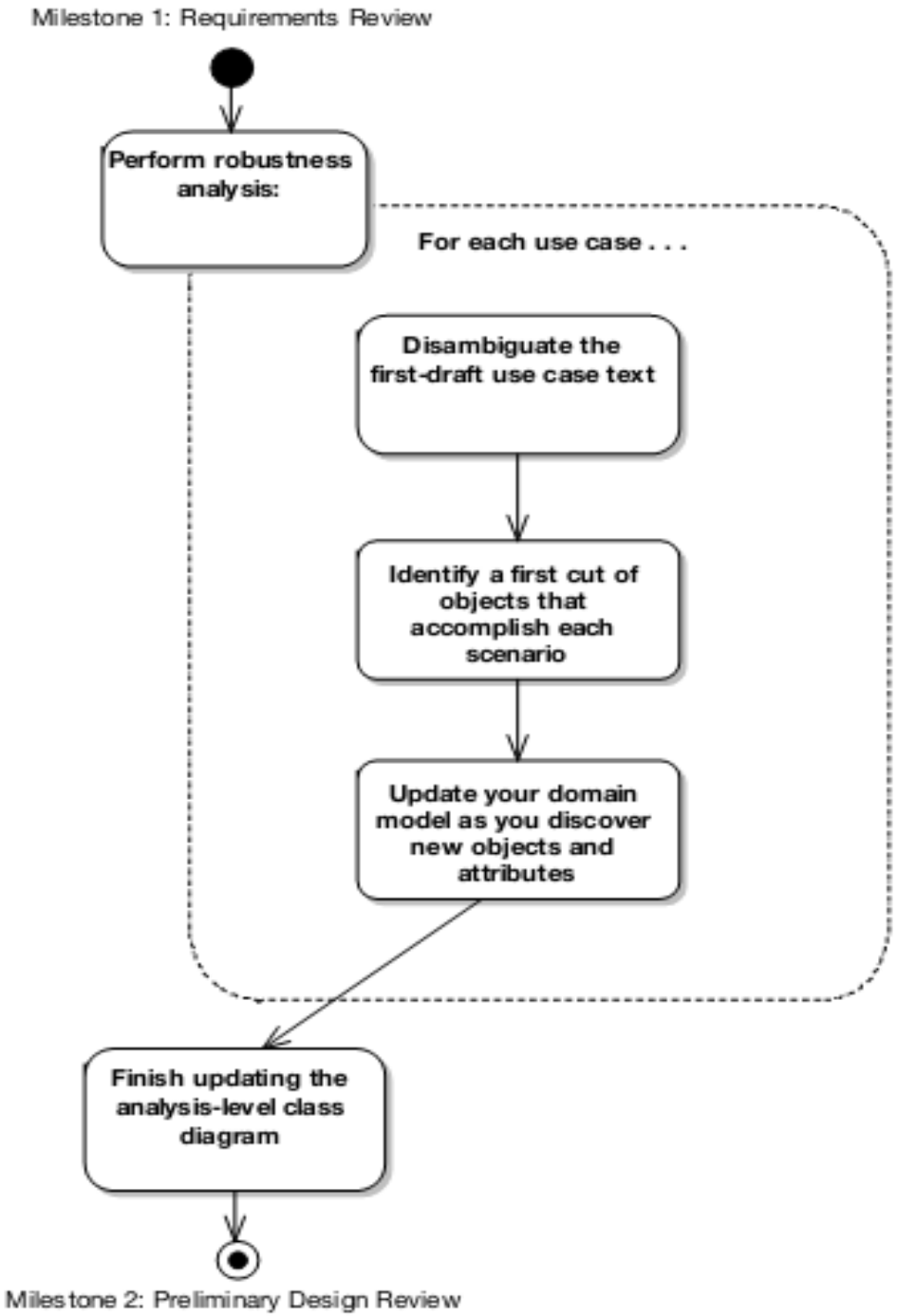
10. Use a modeling tool that supports linkage and traceability between requirements and use cases.
9. Link requirements to use cases by dragging and dropping.
8. Avoid **dysfunctional requirements** by separating functional details from your behavioral specification.
7. Write at least one test case for each requirement.
6. Treat requirements as first-class citizens in the model.
5. Distinguish between different types of requirements.
4. Avoid the "big monolithic document" syndrome.
3. Create estimates from the use case scenarios, not from the functional requirements.
2. Don't be afraid of examples when writing functional requirements.
1. Don't make your requirements a technical fashion statement.

Let's look at each of these items in more detail.

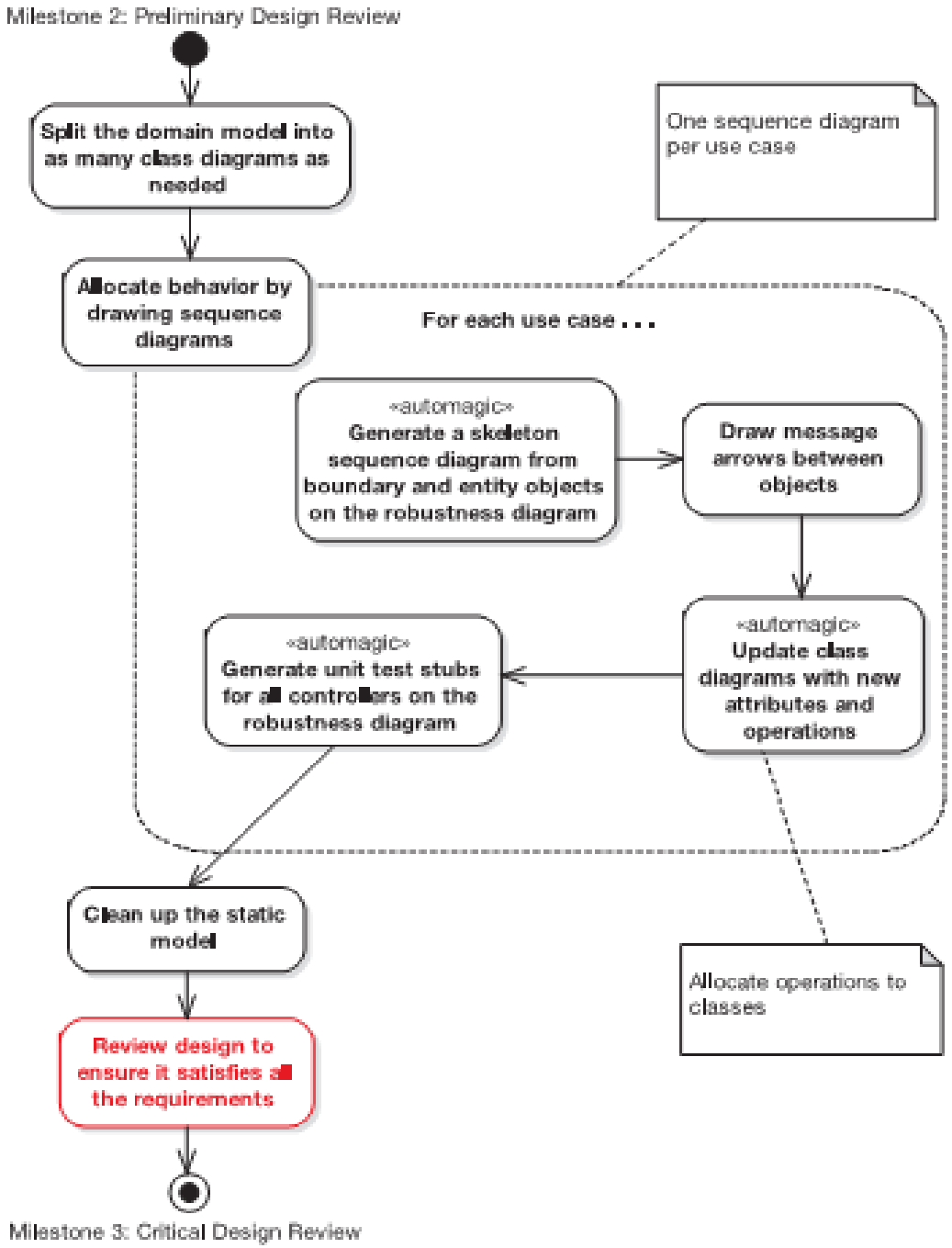
Requirements Analysis



Analysis/preliminary design



Detail Design



Implementation

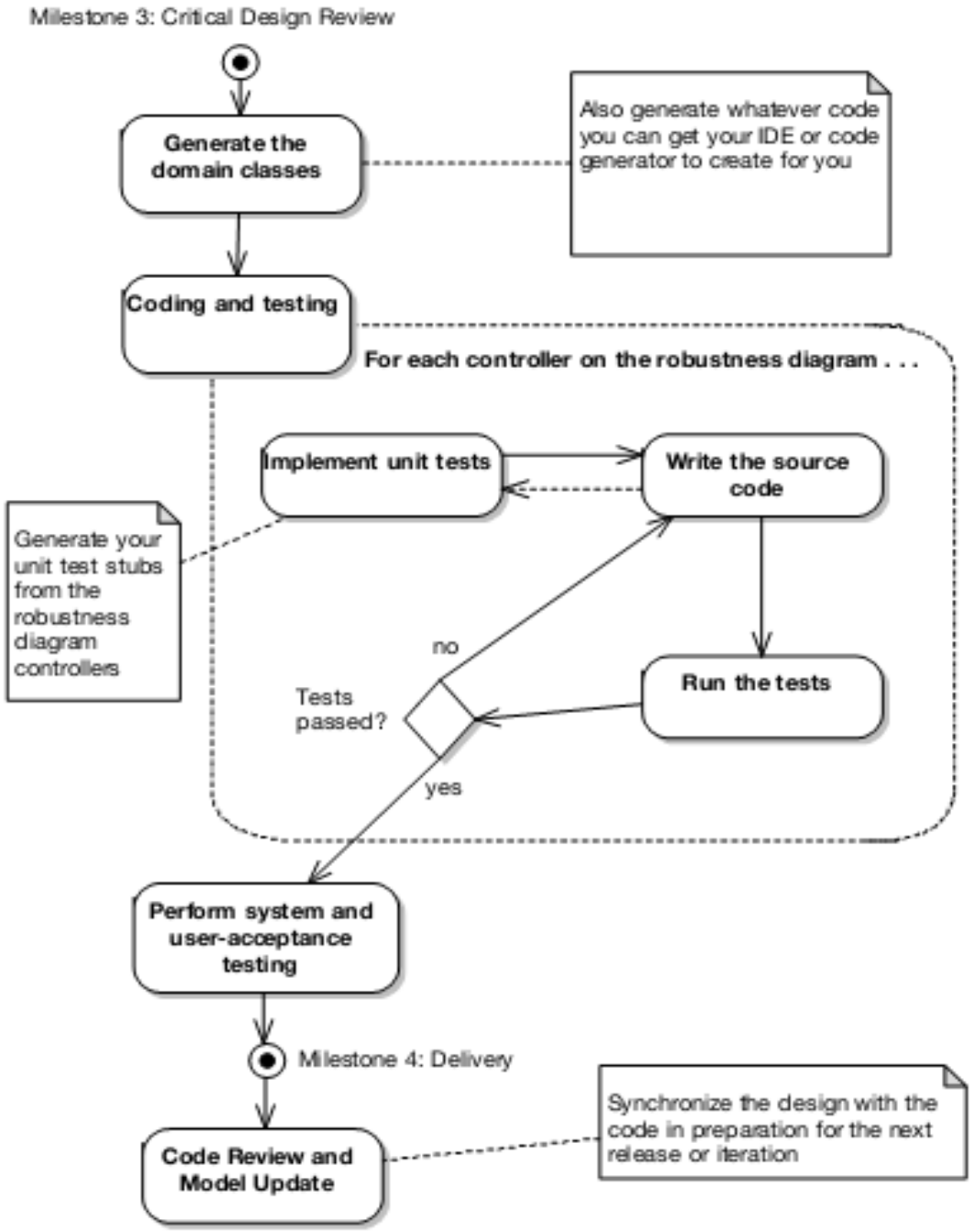
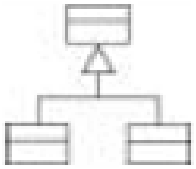
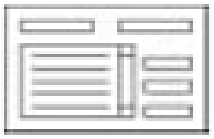


Figure 1-8. Requirements Analysis

- Identify your real-world domain object and the generalization and aggregation relationships among those objects. Start drawing a high-level class diagram.

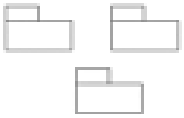


- If it's feasible, do some rapid prototyping of the proposed system. Or gather whatever substantive information you have about the legacy system you are reengineering.

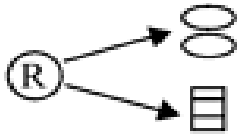


- Identify your use cases, using use case diagrams.

- Organize the use cases into groups. Capture this organization in a package diagram.



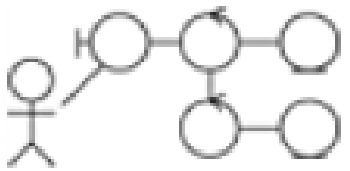
- Allocate functional requirements to the use cases and domain objects at this stage.



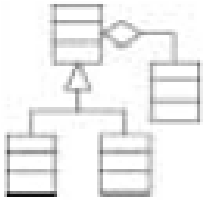
Milestone 1: Requirements Review

Figure 1-9. Analysis and Preliminary Design

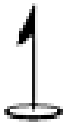
- Write descriptions of the use cases—basic courses of action that represent the "mainstream" and alternative courses for less-frequently traveled paths and error conditions.



- Perform robustness analysis. For each use case:
  - Identify a first cut of objects that accomplish the stated scenario. Use the UML Objectory stereotypes.
  - Update your domain-model class diagram with new objects and attributes as you discover them.



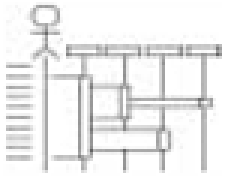
- Finish updating the class diagram so that it reflects the completion of the analysis phase of the project.



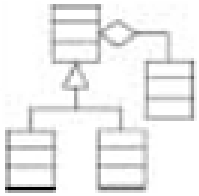
Milestone 2: Preliminary Design Review (PDR)

Figure 1-10. Design

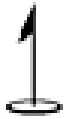
- Allocate behavior. For each use case:
  - Identify the messages that need to be passed between objects, the objects, and the associated methods to be invoked. Draw a sequence diagram with use case text running down the left side and design information on the right. Continue to update the class diagram with attributes and operations as you find them.



  - If you wish, show, on a collaboration diagram, the key transactions between objects.
  - If you wish, use a state diagram to show the real-time behavior.



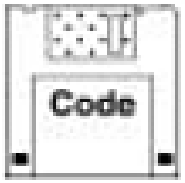
- Finish the static model by adding detailed design information (for instance, visibility values and patterns).
- Verify with your team that your design satisfies all the requirements you've identified.



Milestone 3: Critical Design Review (CDR)

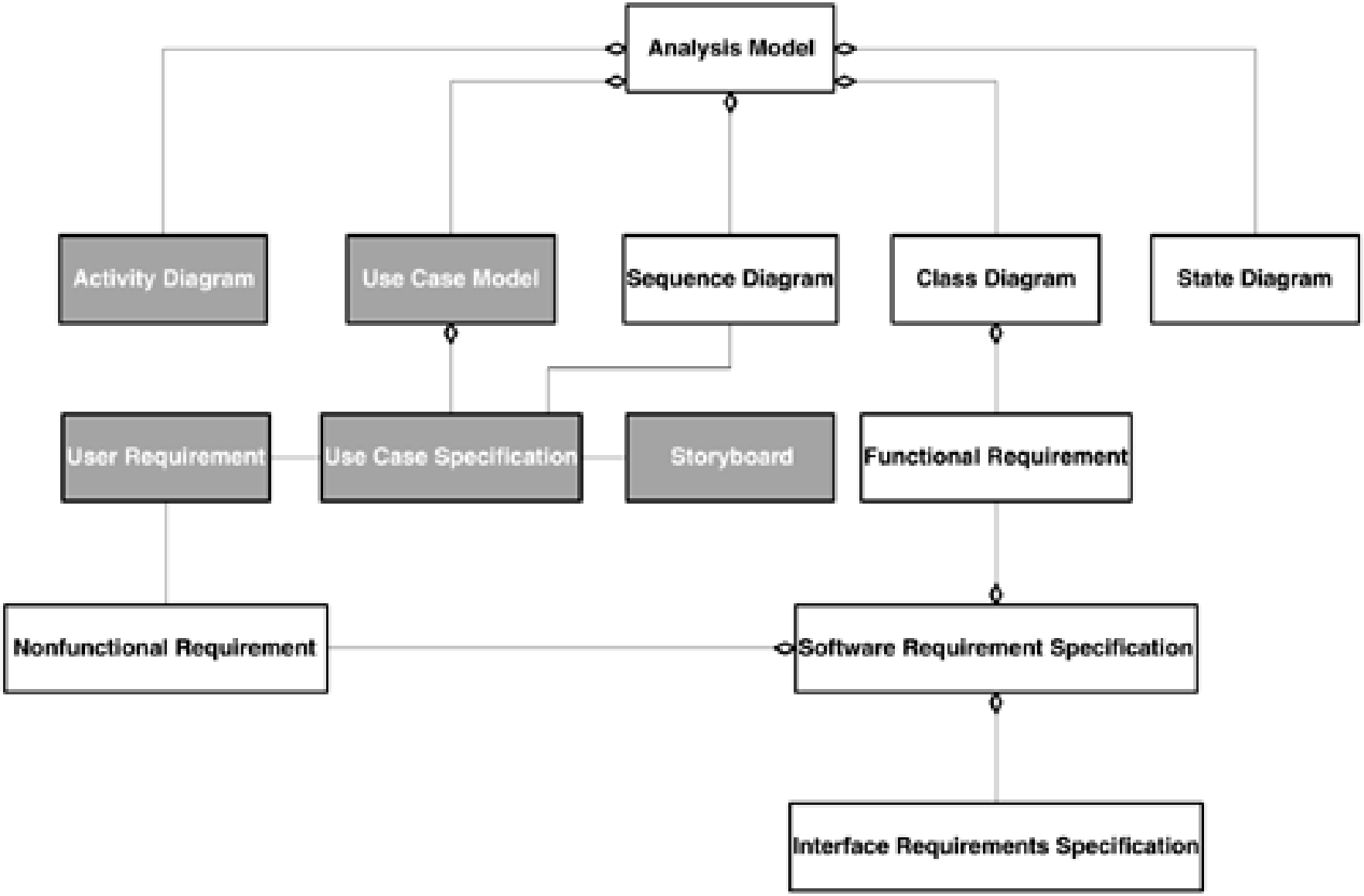
Figure 1-11. Implementation

- As needed, produce diagrams, such as deployment and component diagrams, that will help you with the implementation phase.
- Write/generate the code.
- Perform unit and integration testing.
- Perform system and user-acceptance testing, using the use cases as black-box test cases for the latter.

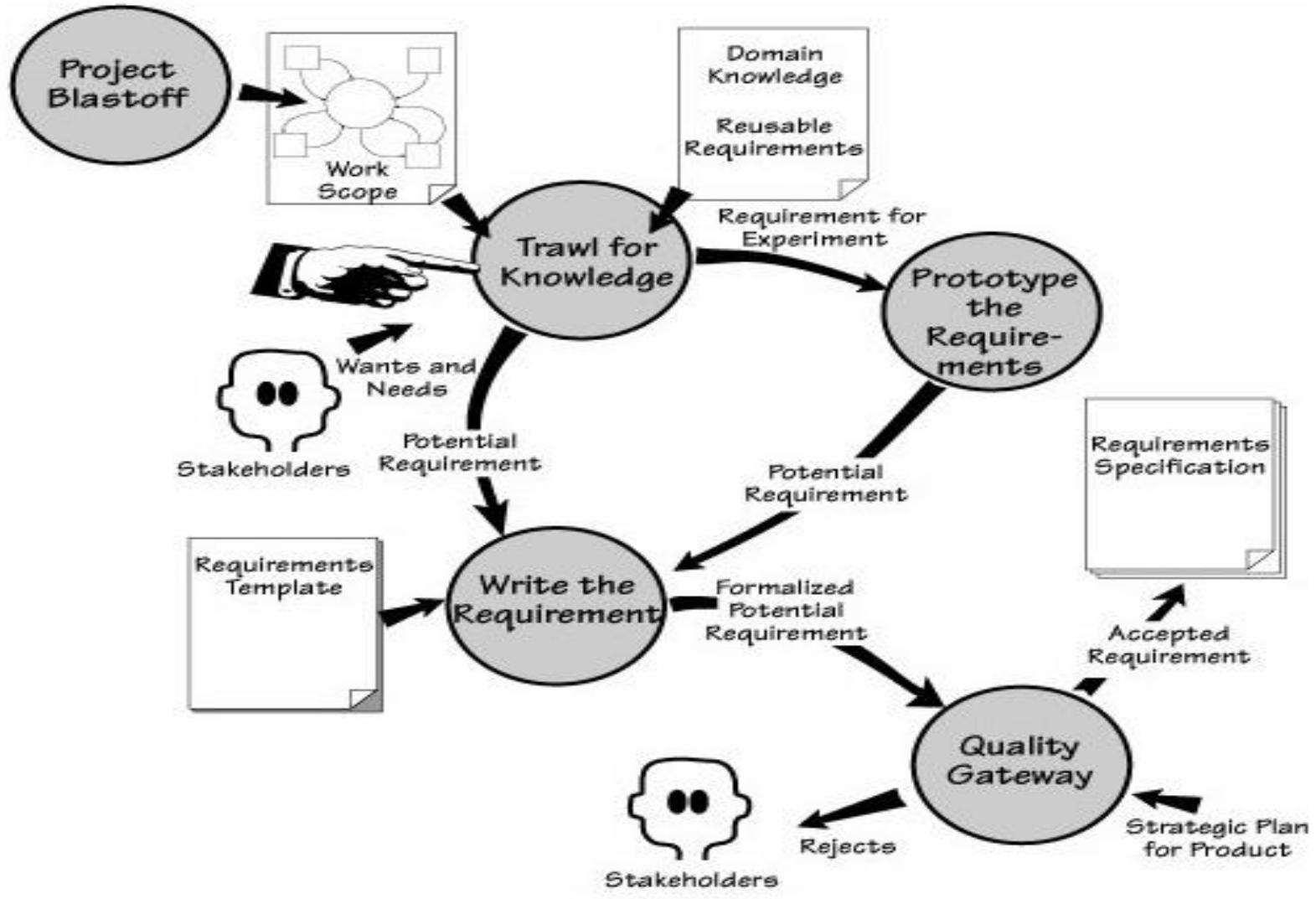


Milestone 4: Delivery

Software Requirements Using the Unified Process



Trawling for Requirements



Six Use Case Categorization Schemes

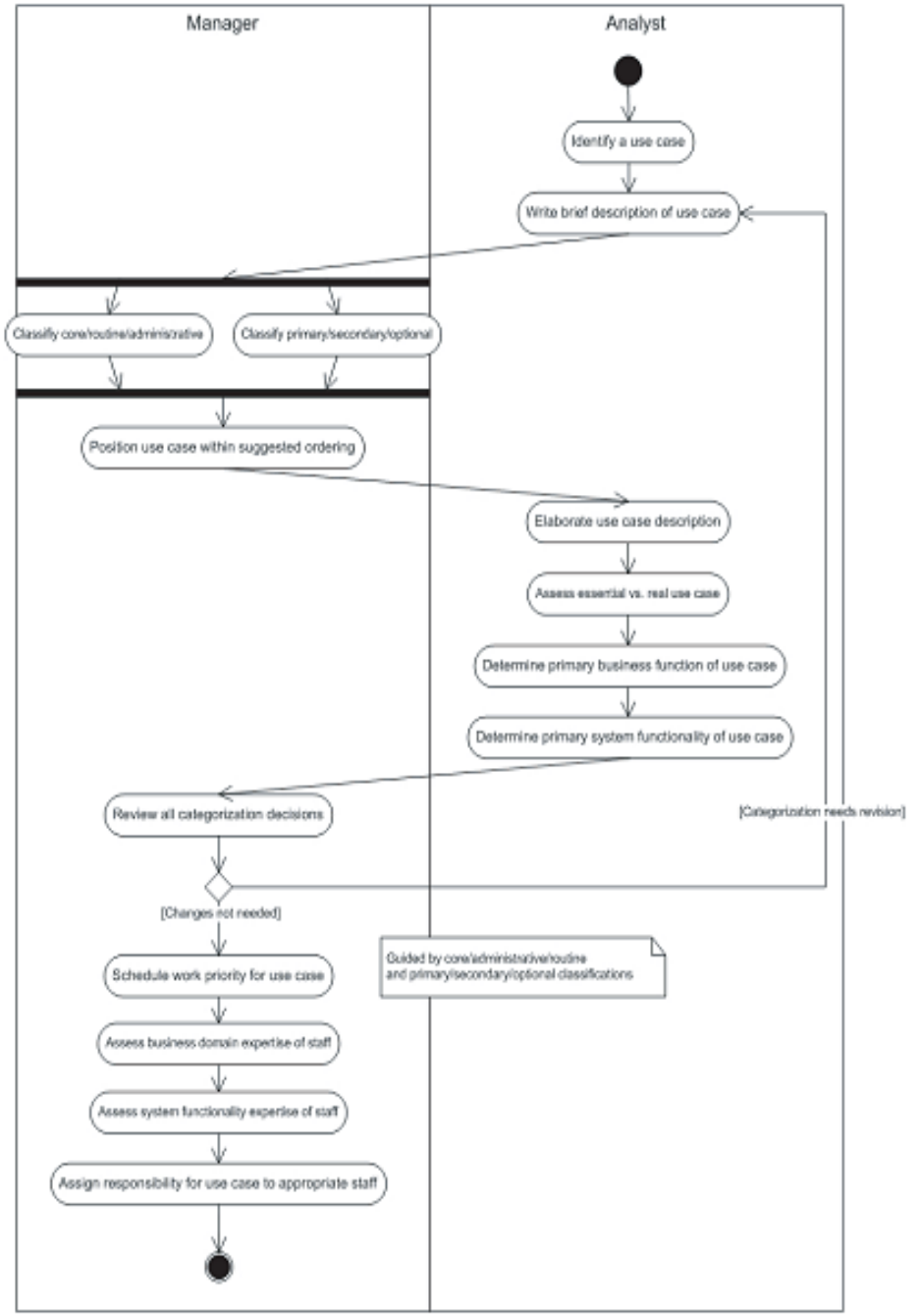
Use Case Categorization Schemes	Definition	Why scheme is valuable
Business use cases vs. system use cases	**Business use cases** reflect business processes and include both manual and system operations. Business use cases are used for business process reengineering. **System use cases** reflect only use of a system, and are used in system requirements analysis.	Business use cases illustrate how computer system fits within business processes, and how it interacts with manual processes. System use cases show how the computer system is used.
Essential use cases vs. concrete use cases	**Essential use cases** are abstract and do not rely on implementation details. **Concrete use cases** consider interaction details such as user interface elements.	Essential use case descriptions survive changes in technology. Concrete use cases show how the technology will be used.
Primary, secondary, and optional use cases	Use cases are prioritized to describe their importance to the project, and to indicate the ideal order for design and implementation.	Project resources are always constrained. Maximizing value requires that the most important use cases be designed and implemented first.

Organizing and Managing Use Cases
by James L. Goldman and Il-Yeol Song
jlmg@drexel.edu song@drexel.edu
College of Information Science and Technology
Drexel University, Philadelphia, PA 19104

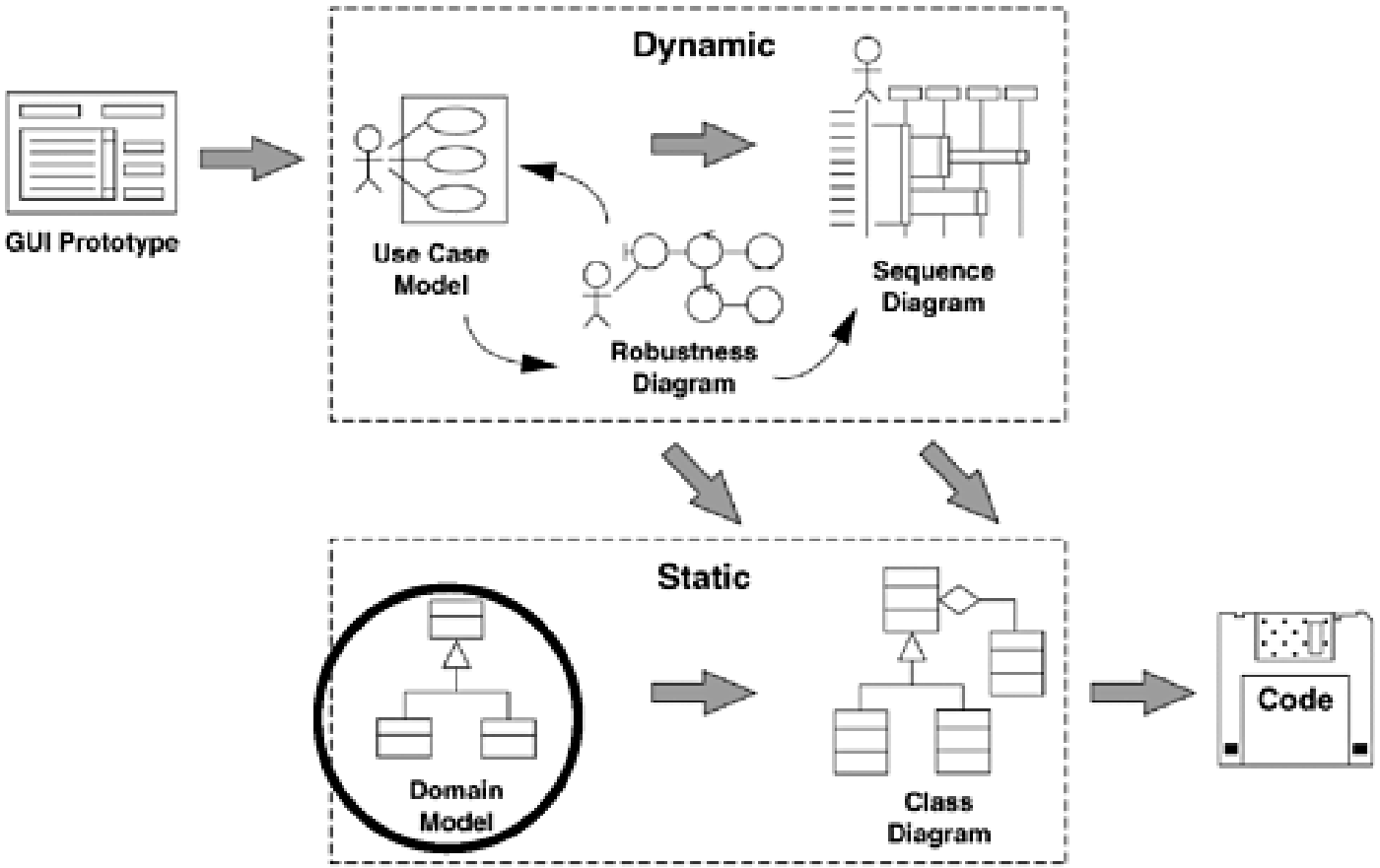
Six Use Case Categorization Schemes

Core, routine, and administrative use cases	**Core use cases** reflect business goals. **Routine use cases** represent incidental, but necessary, system functionality. **Administrative use cases** are operations that ensure continuing healthy operation of the system.	Designating core use cases promotes the proper allocation of resources, avoiding unintentional over allocation of resources to routine and administrative functionality.
Business functions	Categorizing use cases by their dominant **business function** (accounting, shipping, customer service, sales, etc.) application.	End users understand business activities in terms of business functions: the business function categorization helps them understand the use case collection.
System functionality *New Scheme*	**System functionality** summarizes the typical mix of the computing operations being performed in the course of enacting the use case.	Analysts, designers, and programmers each have their own particular skills. Helps match appropriate use cases to staff with right skills & experience.

Procedure for Applying Categorization Schemes



Domain Model



Domain Model

Domain modeling is **the task of building a project glossary**, or **a dictionary of terms used in your project** (e.g., an Internet bookstore project would include domain objects such as Book, Customer, Order, and Order Item).

Its purpose is to make sure everyone on the project understands the problem space in unambiguous terms.

The domain model for a project defines the scope and forms the foundation on which to build your use cases.

The domain model also provides a common vocabulary to enable clear communication among members of a project team.

Expect early versions of your domain model to be wrong; as you explore each use case, you'll "firm up" the domain model as you go.

- **The domain model is a live, collaborative artifact. It is refined and updated throughout the project, so that it always reflects the current understanding of the problem space.**

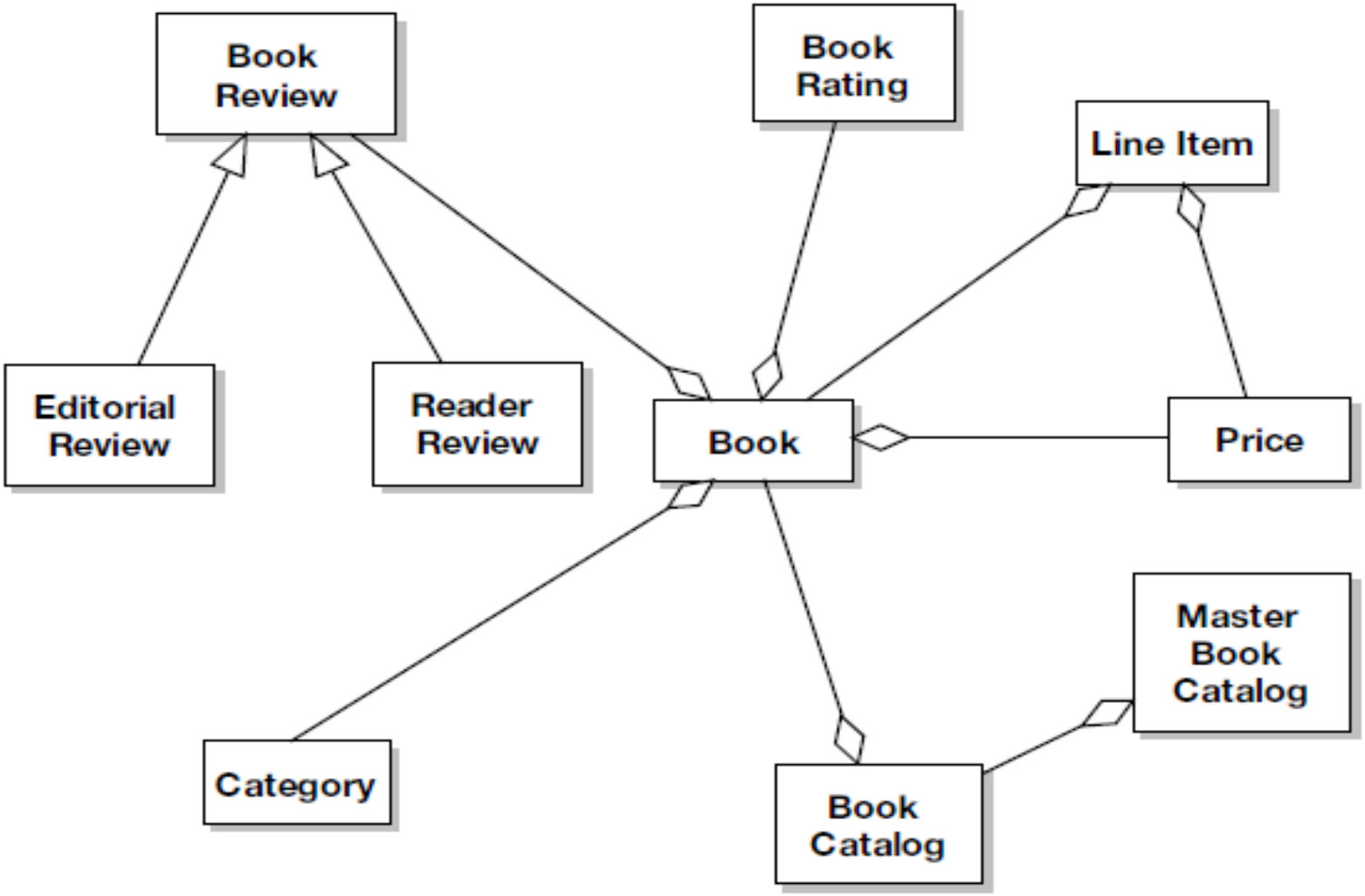


Figure 2-1. *Example of a domain model diagram*

The Key Elements of Domain Modeling

The best sources of domain classes are **likely to be the high-level problem statement, lower-level requirements, and expert knowledge of the problem space**.

To get started on the road to discovery, lay out as many relevant statements from these sources (and others, such as marketing literature) as you can find, and then circle or highlight all the nouns and noun phrases.

Chances are that you will find a large majority of the important domain objects (classes) this way.

After refining the lists as work progresses, this is what tends to happen:

· **Nouns and noun phrases** **become objects and attributes**.

· **Verbs and verb phrases** **become operations and associations**.

· **Possessive phrases** **indicate that nouns should be attributes rather than objects**.

The next step is to sift through your list of candidate classes and eliminate the items that are unnecessary (because they're redundant or irrelevant) or incorrect (because they're too vague, they represent things or concepts outside the scope of the model, or they represent actions even though they're phrased as nouns).

➔ **Use Case Driven Object Modeling with UML: Theory and Practice**

Chapter 2 Domain Model pg.30

With that in mind, let's go through the high-level requirements for the Internet Bookstore and extract some domain classes from them.

1. The bookstore will be web based initially, but it must have a sufficiently flexible architecture that alternative front-ends may be developed (Swing/applets, web services, etc.).
2. The bookstore must be able to sell books, with orders accepted over the Internet.
3. The user must be able to add books into an online shopping cart, prior to checkout.
   a. Similarly, the user must be able to remove items from the shopping cart.
4. The user must be able to maintain wish lists of books that he or she wants to purchase later.
5. The user must be able to cancel orders before they've shipped.
6. The user must be able to pay by credit card or purchase order.
7. It must be possible for the user to return books.
8. The bookstore must be embeddable into associate partners' websites using mini-catalogs, which are derived from an overall master catalog stored in a central database.
   a. The mini-catalogs must be defined in XML, as they will be transferred between this and (later to be defined) external systems.
   b. The shipping fulfillment system shall be carried out via Amazon Web Services.
9. The user must be able to create a customer account, so that the system remembers the user's details (name, address, credit card details) at login.
   a. The system shall maintain a list of accounts in its central database.
   b. When a user logs in, his or her password must always be matched against the passwords in the master account list.
10. The user must be able to search for books by various search methods—title, author, keyword, or category—and then view the books' details.
11. It must be possible for the user to post reviews of favorite books; the review comments should appear on the book details screen. The review should include a customer rating (1–5), which is usually shown along with the book title in book lists.

top 10 domain modeling guidelines

10. Focus on real-world (problem domain) objects.
9. Use generalization (is-a) and aggregation (has-a) relationships to show how the objects relate to each other.
8. Limit your initial domain modeling efforts to a couple of hours.
7. Organize your classes around key abstractions in the problem domain.
6. Don't mistake your domain model for a data model.
5. Don't confuse an object (which represents a single instance) with a database table (which contains a collection of things).
4. Use the domain model as a project glossary.
3. Do your initial domain model before you write your use cases, to avoid name ambiguity.
2. Don't expect your final class diagrams to precisely match your domain model, but there should be some resemblance between them.
1. Don't put screens and other GUI-specific classes on your domain model.

10. Focus on Real-World Objects

When creating a domain model, **be sure to focus on real-world objects within the problem domain**. Try to organize your software architecture around what the real world looks like. The real world tends to change less frequently than software requirements.

CLASS NOTATION

Figure 2-2 shows two different types of class notation. On a full-blown detailed class diagram, you'd use the version on the left, with attributes and operations. However, during the initial domain modeling effort, it's too early to allocate these parts of a class. It's better to use the simpler notation shown on the right. This version only shows the domain class's name.

Figure 2-2. *Class notations*

9. Use Generalization (Is-a) and Aggregation (Has-a) Relationships

Over time, you'll flesh out your domain model with new domain classes, as and when you identify them. You'll **also notice relationships (or *associations) between them*—** *for example,* a Book Review belongs to a Book, and a Purchase Order and Credit Card are two of a kind, as they're both Payment Types.

The first relationship **(*Book Review belongs to a Book)*** ***is called aggregation (has-a,*** because **a Book has a Book Review)**.

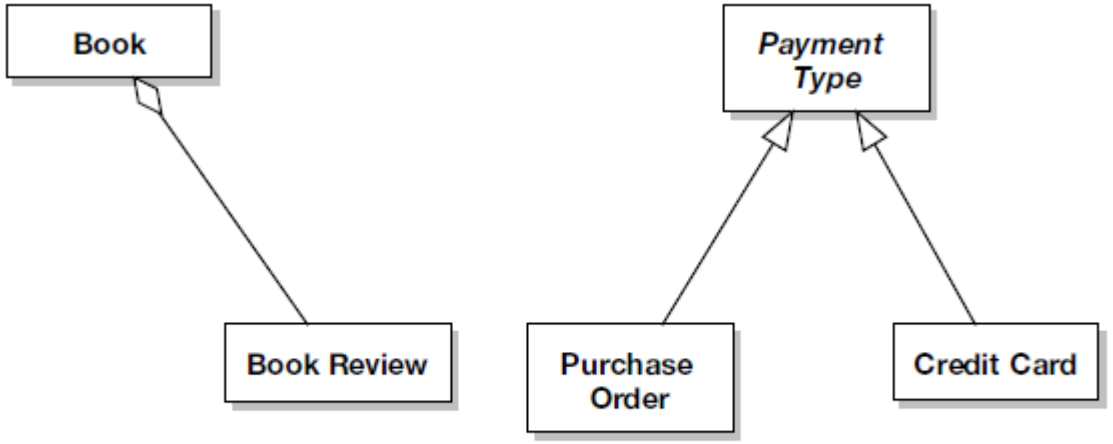


Figure 2-3. *Aggregation and generalization relationships*

The second relationship **(*Purchase Order and Credit Card are both Payment Types)*** ***is called generalization (is-a,*** *because* **a Purchase Order is a Payment Type)**.

8. Limit Your Initial Domain Modeling Efforts to a Couple of Hours

We recommend that **you establish a time budget for building your initial domain model**. A couple of hours is all you should need. You're not going to make it perfect anyway, so do it quickly and expect to fix it as you proceed. You should be vigilant about making necessary adjustments to your analysis-level class model in response to discoveries made during robustness analysis and throughout the project.

You'll discover missing objects as you work through use cases and robustness diagrams. The use case–driven process ***assumes that the domain model is incomplete and provides*** a mechanism for discovering what was missed.

The initial domain modeling session is probably the most important two hours you'll spend on the project! It's likely that you'll discover 80% of your domain classes during that two-hour brainstorming session. If you can get 80% of your domain vocabulary disambiguated, then that's two hours well spent.

7. Organize Your Classes Around Key Abstractions in the Problem Domain

It's generally **good practice to organize your classes around key abstractions in the problem domain**.

Remember that the domain model is a first-cut class diagram that becomes the foundation of your software architecture. This makes the model more resilient in the face of change.

Organizing the architecture around real-world abstractions makes the model more resilient in the face of changing requirements, as the requirements will usually change more frequently than the real world does.

6. Don’t Mistake Your Domain Model for a Data Model

Even though the diagrams might look similar, **remember that what’s good practice on a data model is not likely to be good practice on a class diagram (and vice versa)**.

Classes are small and tables are bigger.

A table in a relational database often relates a number of things.

Conversely, classes are better designed if they’re relatively small packets of data and behavior.

In a class diagram, it’s likely that you’ll have a class that manages a database table, and you might show some sort of TableManager class aggregating a regular domain class. The purpose of these TableManager-type classes is to hide the details of the database management system (DBMS) from the rest of the code base.

5. Don’t Confuse an Object with a Database Table

An object represents a single instance of something.

A database table represents a collection of things.

You don’t have to be as literal-minded as in the Enterprise JavaBeans (EJB) world, where an entity bean generally represents a single row in a table. Domain classes are similar, though.

If you call a domain class Book, then you don’t mean a book table—you mean a single book.

Columns in a table generally map to attributes on a class.

However, database tables typically contain a lot more columns **than** a class contains attributes (tables often have foreign keys, as one example), so there may not be a direct 1:1 mapping between table rows and objects.

4. Use the Domain Model As a Project Glossary

If ambiguous requirements are the enemy, the domain model is the first line of defense.

Ambiguous usage of names by "subject matter experts" is very common and very harmful.

The domain model should serve as a project glossary that helps to ensure consistent usage of terms when describing the problem space.

Using the domain model as a project glossary is the first step toward disambiguating your model.

In every Jumpstart workshop that Doug teaches, he finds at least two or three domain classes where students are using ambiguous names (e.g., "shopping cart," "shopping basket," or "shopping trolley").

3. Do Your Domain Model Before You Write Your Use Cases

Since you're using the domain model to disambiguate your problem domain abstractions, **it would be silly to have your use cases written using ambiguous terms** to describe domain classes.

So spend that two hours working on the domain model before writing your use cases.

Writing the use cases without a domain model to bind everything together stores up lots of problems for later.

2. Don’t Expect Your Final Class Diagrams to Precisely Match Your Domain Model

The class diagrams will become a lot more detailed than the domain model as the design progresses; the domain model is deliberately kept quite simple.

As you’re designing (using sequence diagrams), detailed design constructs such as GUI helpers, factory classes, and infrastructure classes get added to the class diagram, and the domain model diagram will almost certainly be split out into several detailed class diagrams.

However, it should still be possible to trace most classes back to their equivalent domain class.

1. Don’t Put Screens and Other GUI-Specific Classes on Your Domain Model

Doing so opens up Pandora’s box and leads to an overcrowded domain model containing lots of implementation-specific detail.

Performance optimization classes, helper classes, and so on should also be kept out of the domain model.

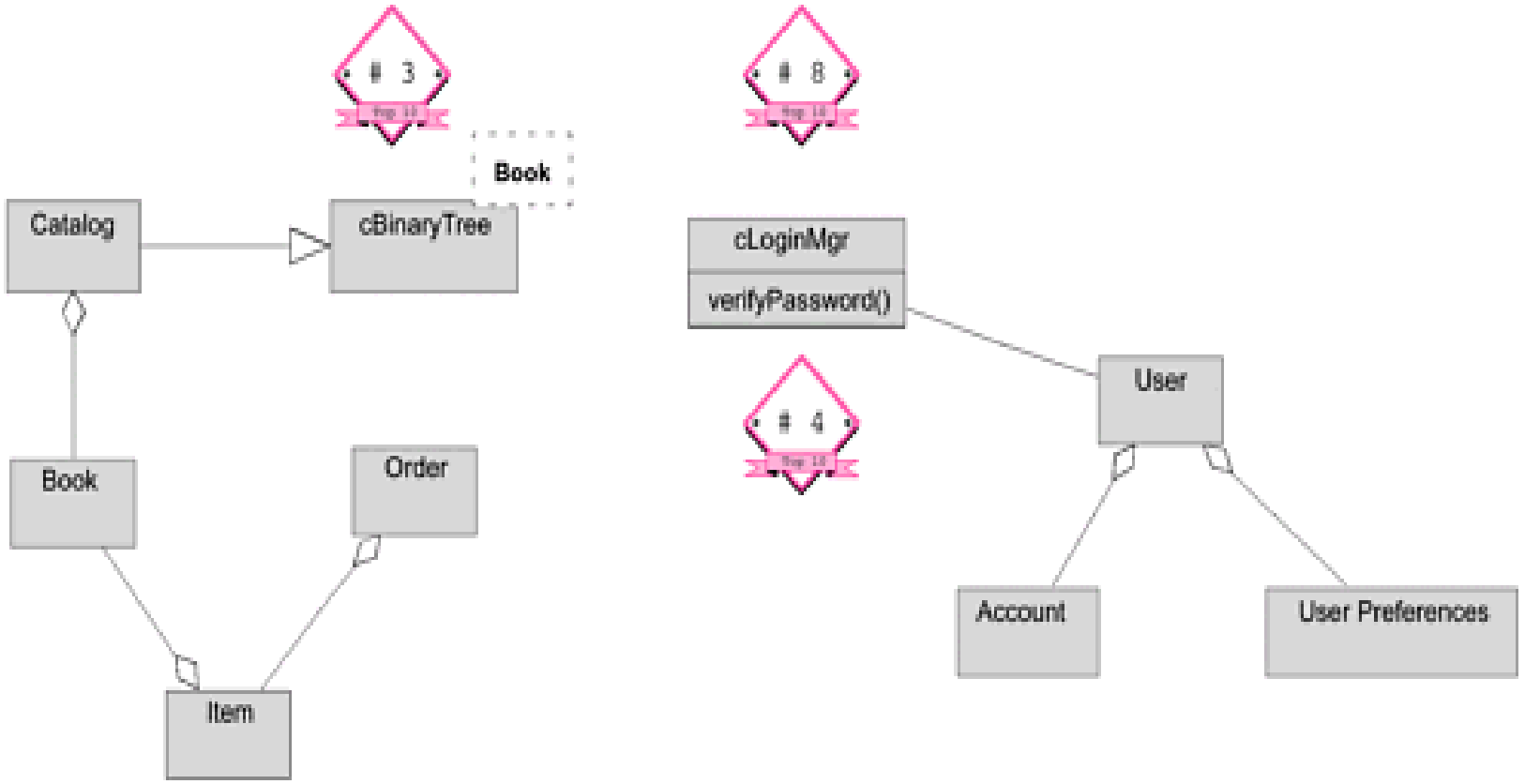
The domain model should focus purely on the problem domain.

The Top 10 Domain Modeling Errors

10. **Start assigning multiplicities to associations right off the bat. Make sure that every association has an explicit multiplicity.**
9. *Do noun and verb analysis so exhaustive that you pass out along the way.*
8. **Assign operations to classes without exploring use cases and sequence diagrams.**
7. *Optimize your code for reusability before making sure you've satisfied the user's requirements.*
6. **Debate whether to use aggregation or composition for each of your "part-of" associations.**
5. *Presume a specific implementation strategy without modeling the problem space.*
4. **Use hard-to-understand names for your classes, like cPortMgrIntf, instead of intuitively obvious ones, like PortfolioManager.**
3. *Jump directly to implementation constructs, such as friend relationships and parameterized classes.*
2. **Create a one-for-one mapping between domain classes and relational database tables.**
1. role="titleicon" *Perform "premature patternization," which involves building cool solutions, from patterns, that have little or no connection to user problems.*

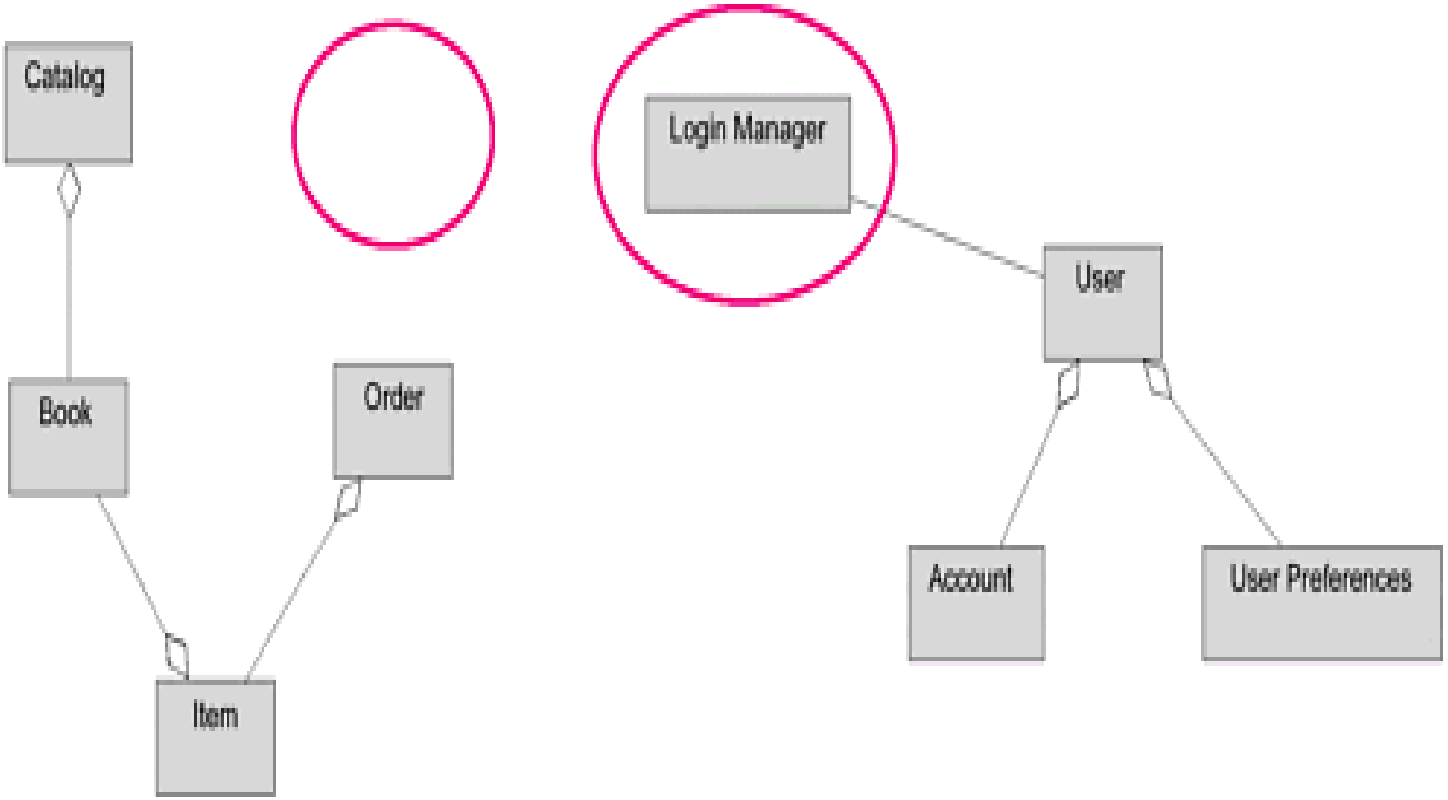
Contoh ke-1

8. Assign operations to classes without exploring use cases and sequence diagrams.
4. Use hard-to-understand names for your classes, like cPortMgrIntf, instead of intuitively obvious ones, like PortfolioManager.
3. Jump directly to implementation constructs, such as friend relationships and parameterized classes.



-The cBinaryTree class is a parameterized class (also known as a template class within the UML). There's no good reason to start defining an implementation construct such as a binary tree at this stage of modeling.
-The cLoginMgr class has an operation named verifyPassword. It's too early to make decisions about which operations go on which classes, and besides, chances are good that the operation belongs on the Account class anyway.
-The name of the class we just discussed was not intuitively obvious.

Ke-1

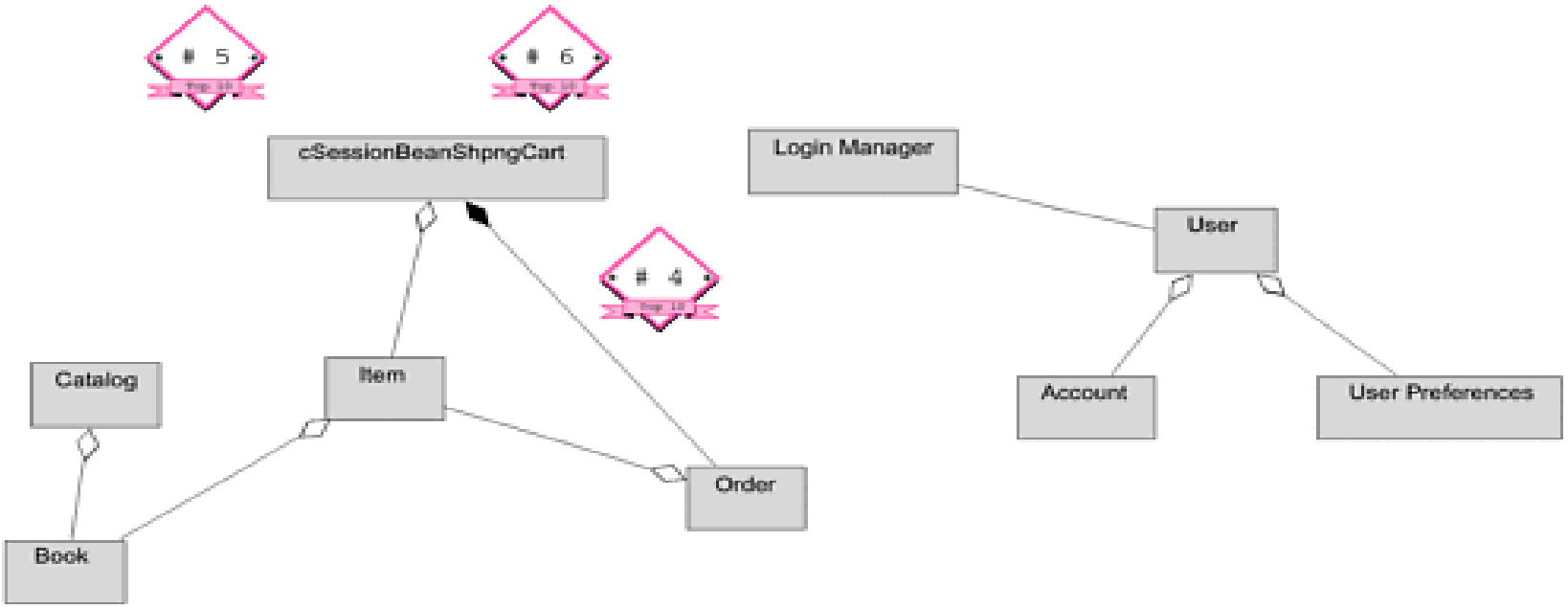


8. Assign operations to classes without exploring use cases and sequence diagrams.
4. Use hard-to-understand names for your classes, like cPortMgrIntf, instead of intuitively obvious ones, like PortfolioManager.
3. Jump directly to implementation constructs, such as friend relationships and parameterized classes.

-The cBinaryTree class is a parameterized class (also known as a template class within the UML). There's no good reason to start defining an implementation construct such as a binary tree at this stage of modeling.
-The cLoginMgr class has an operation named verifyPassword. It's too early to make decisions about which operations go on which classes, and besides, chances are good that the operation belongs on the Account class anyway.
-The name of the class we just discussed was not intuitively obvious.

Contoh ke-2

6. Debate whether to use aggregation or composition for each of your "part-of" associations.
5. Presume a specific implementation strategy without modeling the problem space.
4. Use hard-to-understand names for your classes, like cPortMgrIntf, instead of intuitively obvious ones, like PortfolioManager.

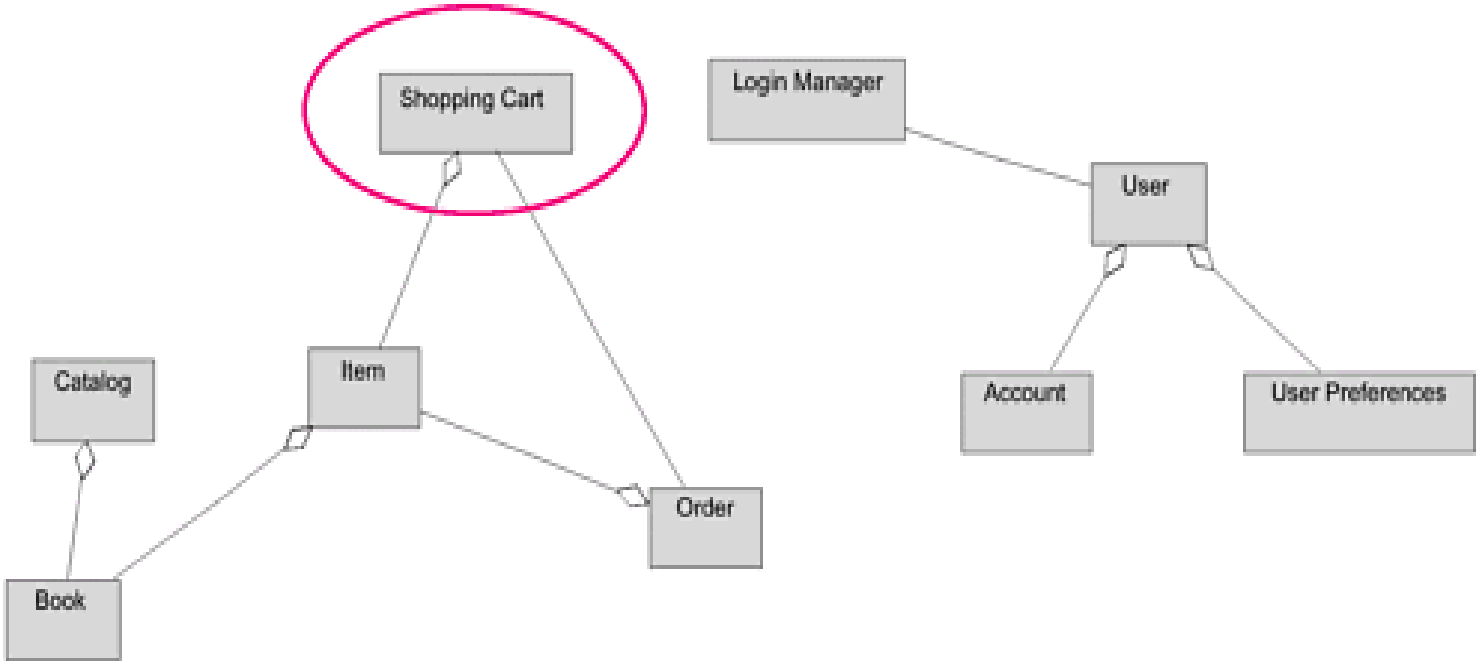


The name of the cSessionBeanShpngCart class indicated that the modeler decided to represent the concept of a shopping cart using a session Enterprise Java Bean (EJB). Robustness analysis, which we'll discuss inChapter 5, is the appropriate stage to start exploring how to map classes to things such as beans.

A class that represents a shopping cart should be called Shopping Cart.

The class we've been discussing had a composition relationship with the Order class. The modeler committed to the idea that an Order disappears when the shopping cart object to which it belongs is destroyed. This may or not make sense in the long run, but it's certainly too soon to be thinking along those lines.

Ke-2



6. Debate whether to use aggregation or composition for each of your "part-of" associations.
5. Presume a specific implementation strategy without modeling the problem space.
4. Use hard-to-understand names for your classes, like cPortMgrIntf, instead of intuitively obvious ones, like PortfolioManager.

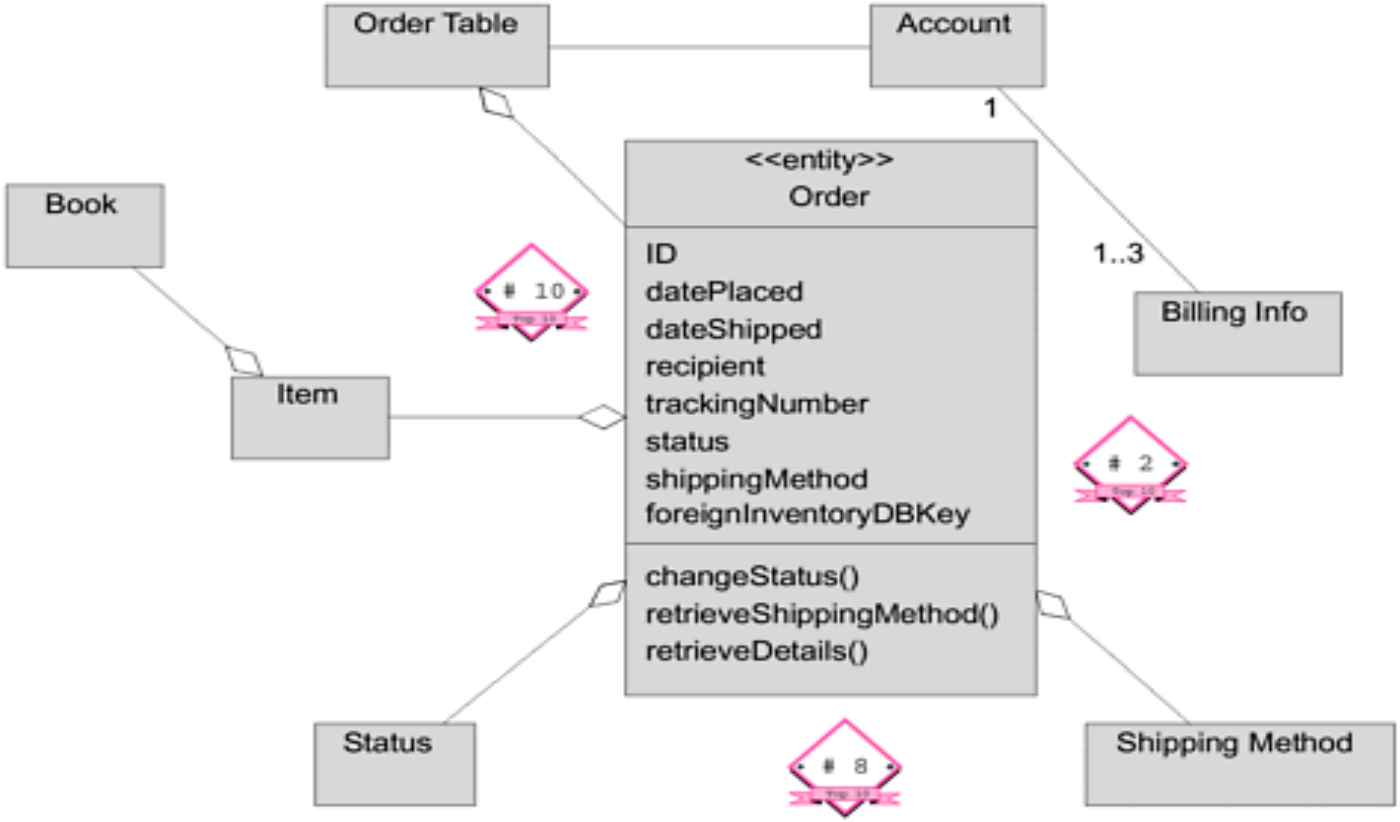
The name of the cSessionBeanShpngCart class indicated that the modeler decided to represent the concept of a shopping cart using a session Enterprise Java Bean (EJB). Robustness analysis, which we'll discuss inChapter 5, is the appropriate stage to start exploring how to map classes to things such as beans.

A class that represents a shopping cart should be called Shopping Cart.

The class we've been discussing had a composition relationship with the Order class. The modeler committed to the idea that an Order disappears when the shopping cart object to which it belongs is destroyed. This may or not make sense in the long run, but it's certainly too soon to be thinking along those lines.

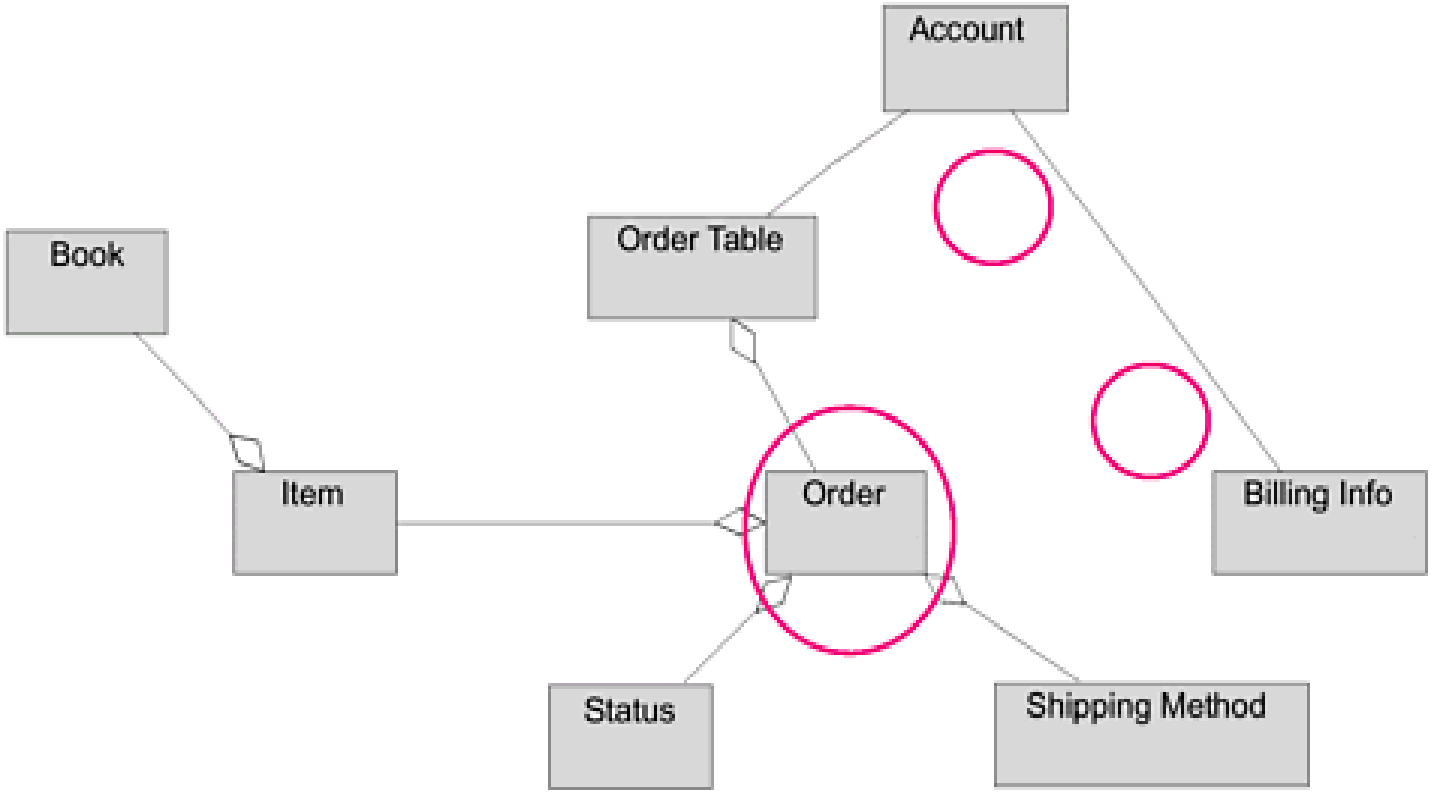
Contoh ke-3

10. Start assigning multiplicities to associations right off the bat. Make sure that every association has an explicit multiplicity.
8. Assign operations to classes without exploring use cases and sequence diagrams.
2. Create a one-for-one mapping between domain classes and relational database tables.



The presence of the foreignInventoryDBKey attribute indicates that the modeler is looking ahead toward a relational database. (Note also that classes in your domain model shouldn't have attributes yet, and they certainly shouldn't have operations.)
The Order class has operations assigned to it.
The association between Account and Billing Info has a multiplicity.

ke-3

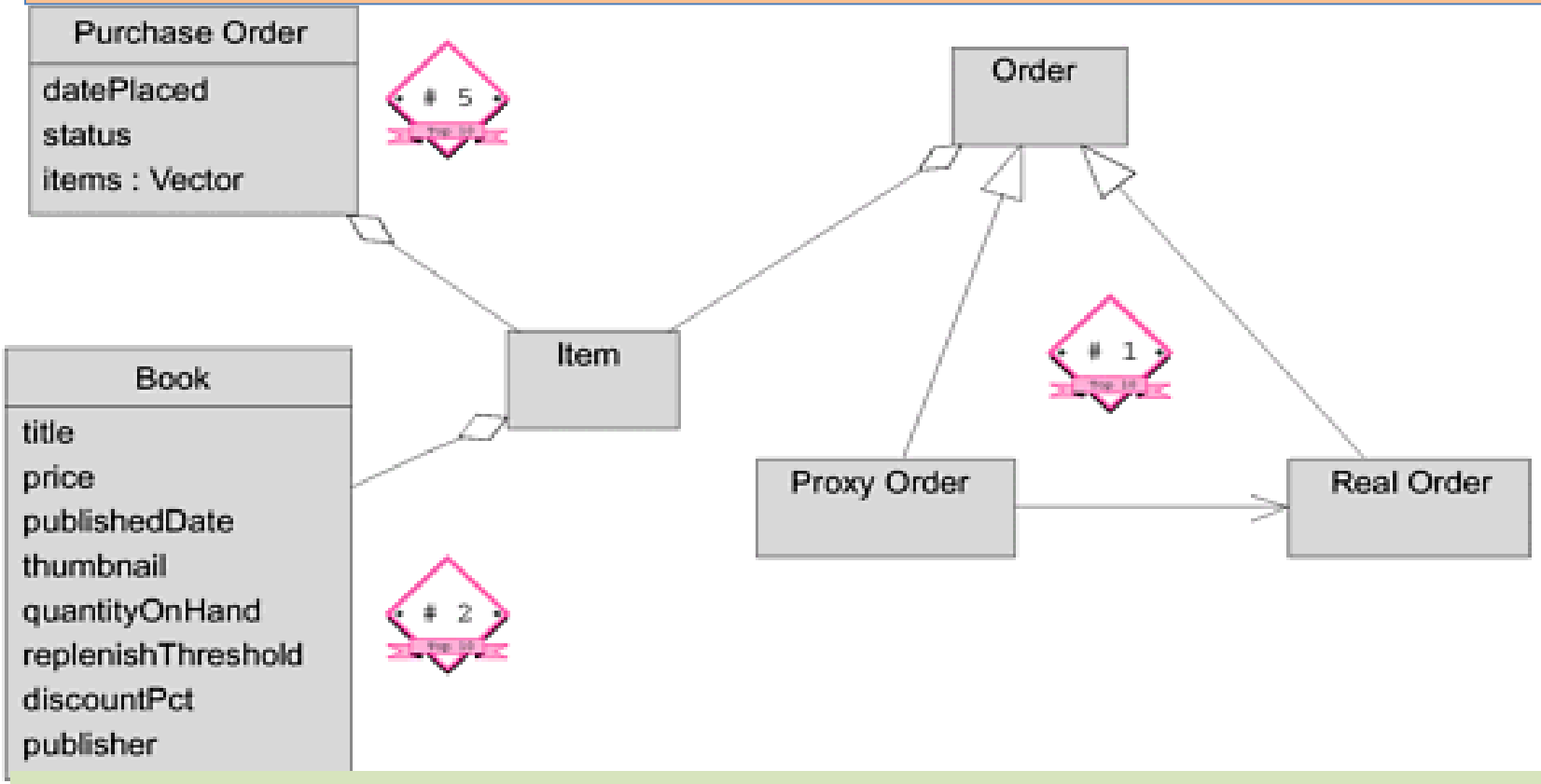


10. Start assigning multiplicities to associations right off the bat. Make sure that every association has an explicit multiplicity.
8. Assign operations to classes without exploring use cases and sequence diagrams.
2. Create a one-for-one mapping between domain classes and relational database tables.

The presence of the foreignInventoryDBKey attribute indicates that the modeler is looking ahead toward a relational database. (Note also that classes in your domain model shouldn't have attributes yet, and they certainly shouldn't have operations.)
The Order class has operations assigned to it.
The association between Account and Billing Info has a multiplicity.

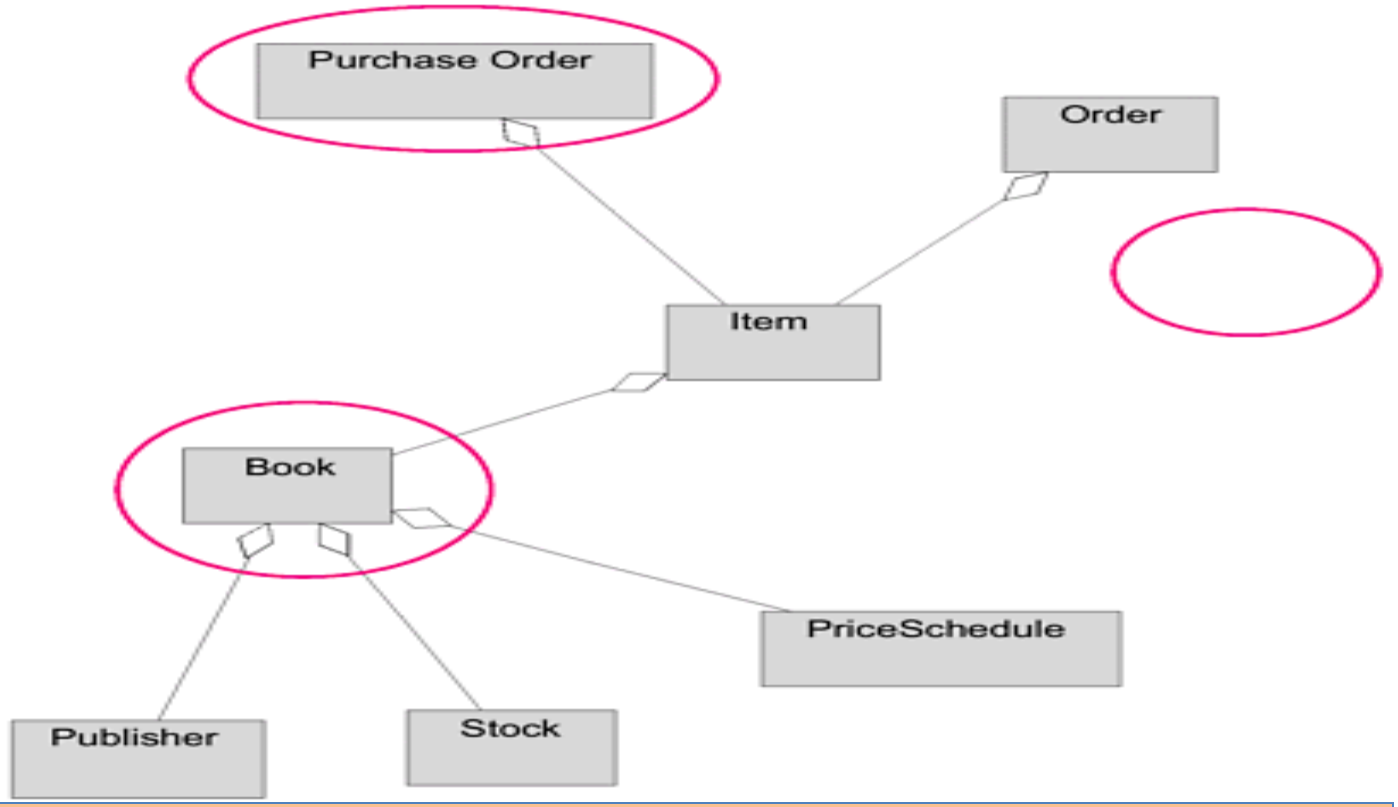
Contoh ke-4

5. Presume a specific implementation strategy without modeling the problem space.
2. Create a one-for-one mapping between domain classes and relational database tables.
1. role="titleicon" ***Perform "premature patternization," which involves building cool solutions, from patterns, that have little or no connection to user problems.***



The presence of attributes named "price," "quantityOnHand," and "publisher," all of which probably belong in associated classes, indicates that the modeler is likely to have mapped an existing Order table directly to the Order class. (Also, as we mentioned for Exercise 3, classes in your domain model shouldn't have attributes yet.)
The Purchase Order class uses the Vector construct from Java.
The modeler has chosen to use the Proxy design pattern; domain modeling is too early to be making this decision.

ke-4

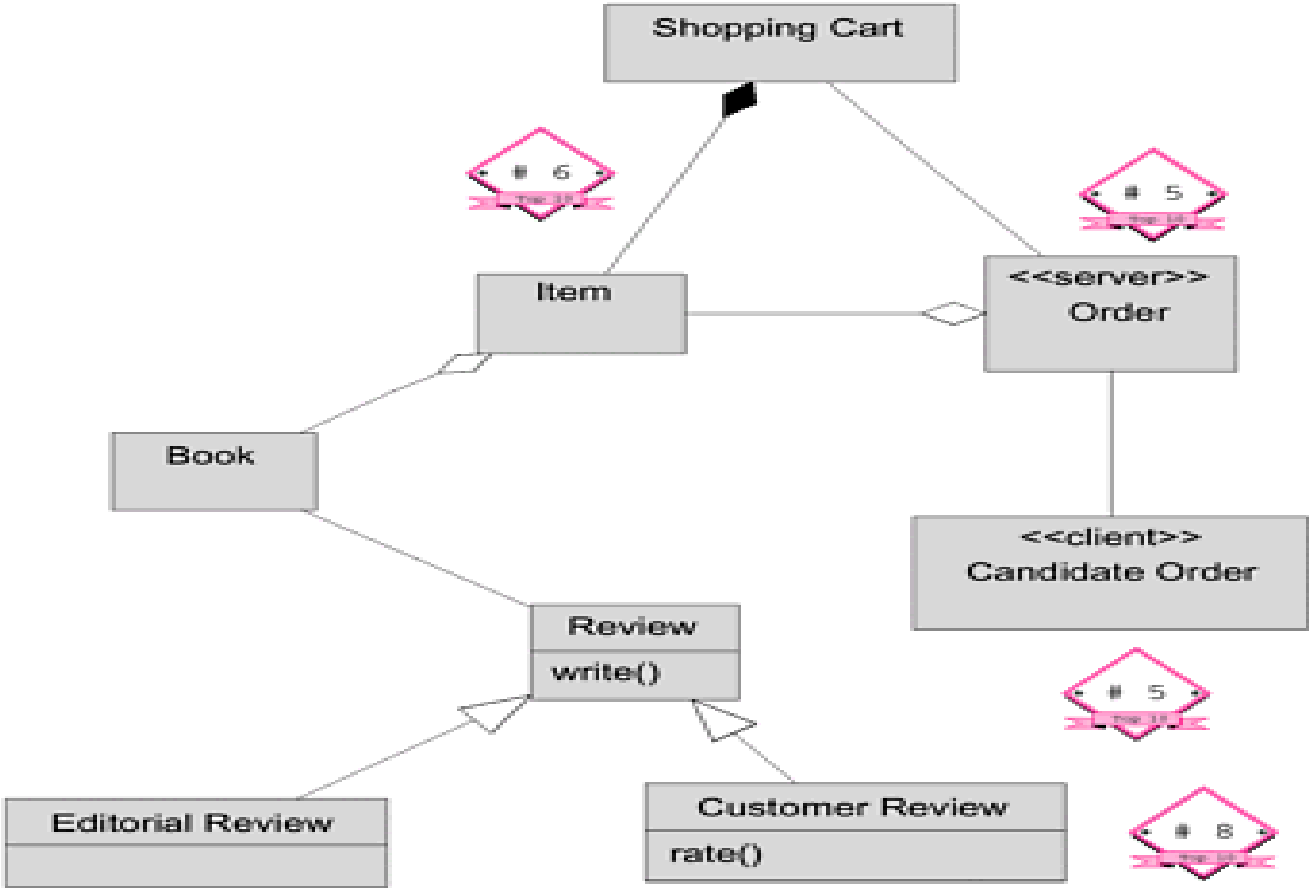


5. Presume a specific implementation strategy without modeling the problem space.
2. Create a one-for-one mapping between domain classes and relational database tables.
1. role="titleicon" *Perform "premature patternization," which involves building cool solutions, from patterns, that have little or no connection to user problems.*

The presence of attributes named "price," "quantityOnHand," and "publisher," all of which probably belong in associated classes, indicates that the modeler is likely to have mapped an existing Order table directly to the Order class. (Also, as we mentioned for Exercise 3, classes in your domain model shouldn't have attributes yet.)
The Purchase Order class uses the Vector construct from Java.
The modeler has chosen to use the Proxy design pattern; domain modeling is too early to be making this decision.

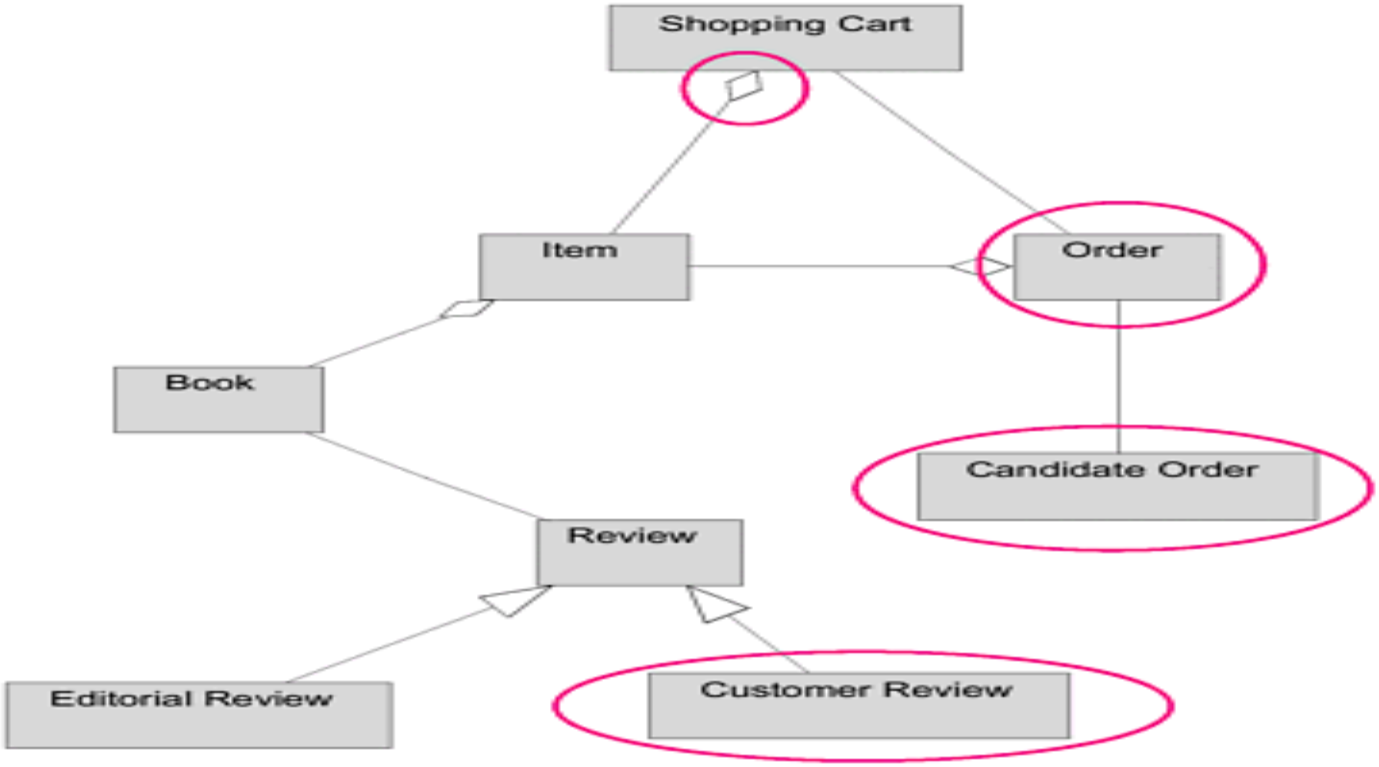
Contoh ke-5

8. Assign operations to classes without exploring use cases and sequence diagrams.
6. Debate whether to use aggregation or composition for each of your "part-of" associations.
5. Presume a specific implementation strategy without modeling the problem space.



.The Customer Review class has an operation.
.The association between Item and Shopping Cart is a composition, but it's too early to know whether this makes sense as opposed to an ordinary aggregation.
.The stereotypes on the Order and Candidate Order classes indicate a premature decision as to the layers to which the classes belong.

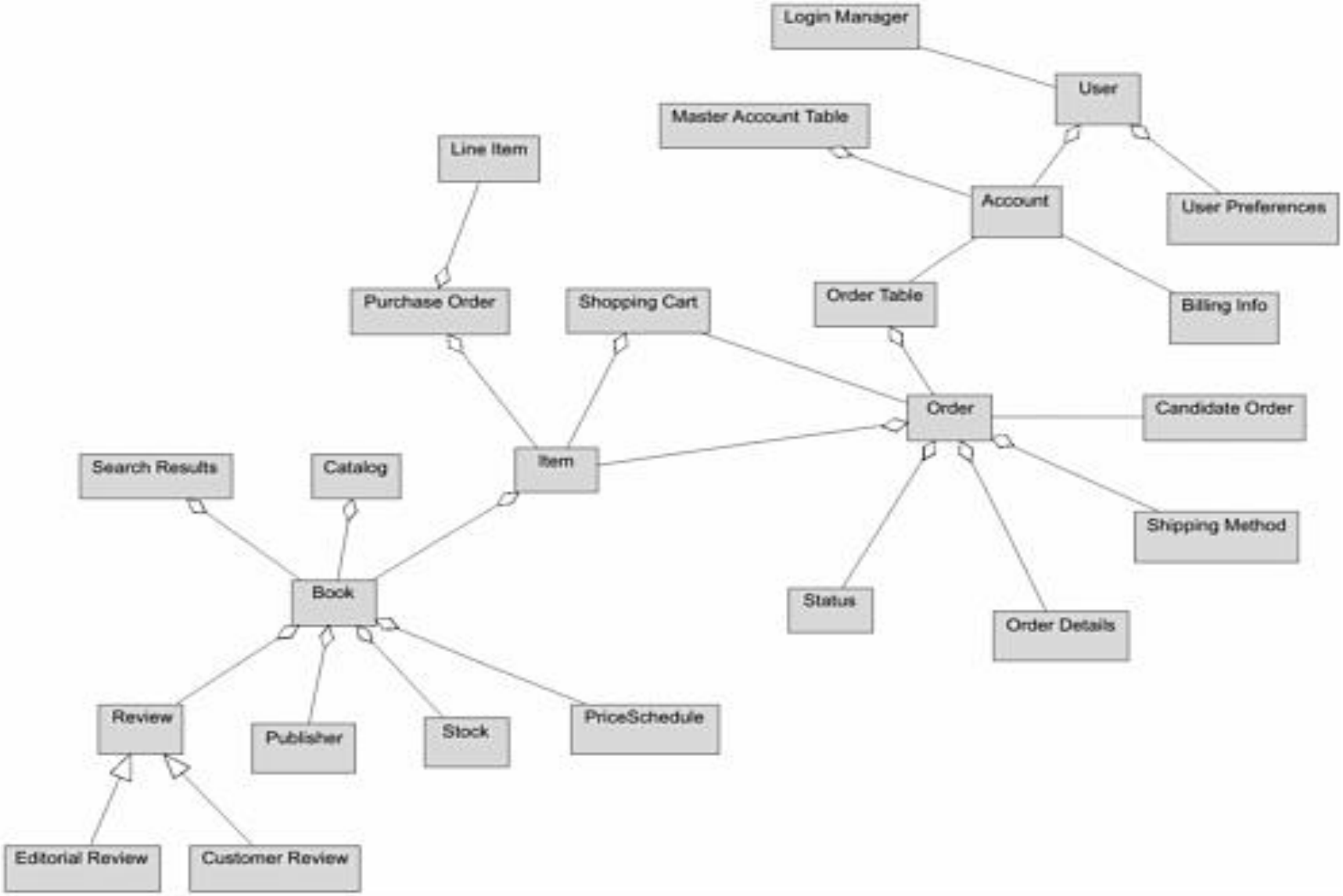
ke-5



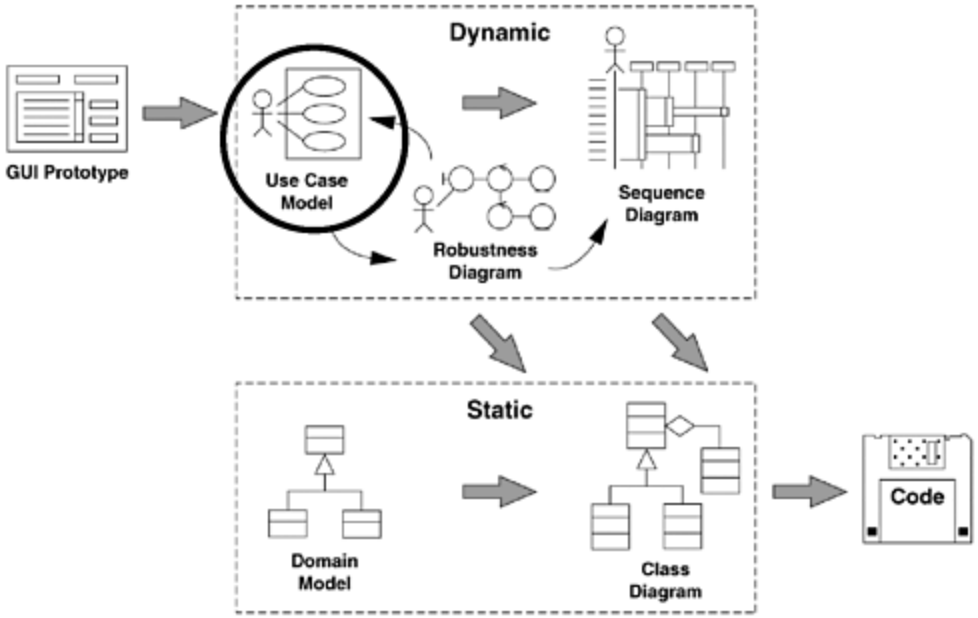
8. Assign operations to classes without exploring use cases and sequence diagrams.
6. Debate whether to use aggregation or composition for each of your "part-of" associations.
5. Presume a specific implementation strategy without modeling the problem space.

.The Customer Review class has an operation.
.The association between Item and Shopping Cart is a composition, but it's too early to know whether this makes sense as opposed to an ordinary aggregation.
.The stereotypes on the Order and Candidate Order classes indicate a premature decision as to the layers to which the classes belong.

Domain Model for The Internet Bookstore



Use Case Modeling



Perilaku Sistem (*System Behavior)*

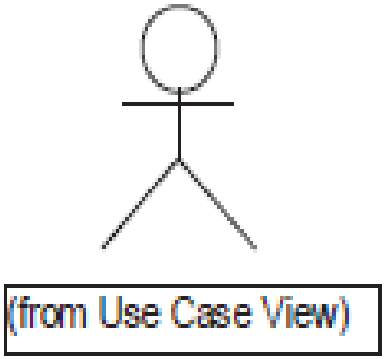
Perilaku dari sistem di bawah pengembangan adalah fungsi yang harus disediakan oleh sistem, didokumentasikan dalam suatu ***Use Case Model yang menjelaskan*** fungsi/operasi yang ada dalam sistem (*Use Case), obyek-obyek yang mempengaruhi* sistem (*Actors) dan hubungan antara Use Case dan Actors (Use Case Diagram).*

Use Case Model berawal di fase permulaan dengan identifikasi dari Actor dan Use Case yang utama dari sistem. Model ini akan menjadi lengkap di fase perluasan, dimana informasi yang lebih detail ditambahkan untuk mengidentifikasikan *Use Case,* dan beberapa *Use Case tambahan ditambahkan sebagai dasar yang dibutuhkan.*

Actors

Actors adalah bagian yang berada di luar system tapi berpengaruh dan juga merupakan pengguna sistem. *Actors hanya menerima informasi dari sistem, memasukkan informasi* ke sistem, dan memasukkan dan menerima dari dan ke sistem. Dengan kata lain *Actors* adalah orang atau obyek yang menggunakan system yang akan dibuat.

Biasanya *Actors ditemukan dalam pernyataan masalah dan berhubungan dengan Customer (pelanggan).*



Pertanyaan berikut mungkin digunakan untuk membantu mengidentifikasi aktor untuk sistem ;

- Siapa yang akan tertarik pada sistem tersebut ?
- Dimana organisasi sistem yang digunakan ?
- Siapa yang akan beruntung dari penggunaan sistem ?
- Siapa yang akan mensuplai sistem dengan informasi ini, menggunakan informasi ini dan memindahkan informasi ini ?
- Siapa yang akan mendukung dan memelihara sistem ?
- Apakah sistem menggunakan sumber eksternal ?
- Apakah satu orang memainkan beberapa peran yang berbeda ?
- Apakah beberapa orang memainkan peran yang sama ?
- Apakah sistem berinteraksi dengan suatu sistem yang lama ?

Bagaimana memilih *Actors yang tepat?*

Apakah seorang mahasiswa baru adalah *Actors yang* berbeda dengan mahasiswa yang sudah lama?

Misalnya anda setuju dengan pernyataan tersebut.

Hal selanjutnya adalah langkah apa yang digunakan agar *Actors baru tersebut* dapat berinteraksi dengan sistem yang sudah ada.

Jika mahasiswa baru menggunakan sistem yang berbeda dengan mahasiswa yang lama, maka mereka dapat dikatakan sebagai aktor yang baru.

Tetapi jika mahasiswa baru tersebut menggunakan sistem yang sama dengan mahasiswa yang lama maka mereka adalah *Actors yang sama.*

Actors yang digunakan dalam Sistem Registrasi Kelas Mata Kuliah di Perguruan Tinggi adalah :

- Mahasiswa ingin mendaftar untuk mengikuti mata kuliah
- Dosen ingin memilih mata kuliah yang akan diajarkan
- Pihak administrasi akan membuat kurikulum dan membuat katalog untuk tiap-tiap semester
- Pihak administrasi harus menjaga semua informasi tentang mata kuliah, Dosen, Mahasiswa
- Billing system harus menerima informasi dari sistem

Berdasarkan hal tersebut diatas, *Actors akan diidentifikasikan sebagai berikut :* **Mahasiswa, Dosen, Pihak Administrasi dan Billing System**.

Dokumentasi *Actors*

Deskripsi untuk setiap *Actor harus ditambahkan pada setiap model. Deskripsi* tersebut harus mengidentifikasi aturan yang dimainkan para *Actor selama berinteraksi* dengan sistem.

Deskripsi *Actors pada Sistem Registrasi Kelas Mata Kuliah pada Perguruan Tinggi* adalah sebagai berikut :

1. Mahasiswa

Mahasiswa adalah obyek diluar system yang menggunakan system dan mahasiswa adalah obyek yang memasukkan ke system dan memperoleh informasi dari system.

2. Dosen

Dosen adalah obyek diluar system yang memasukkan informasi tentang mata kulaih yang diajarkan, dan dosen ini juga memperoleh informasi berapa jumlah mahasiswa yang mengikuti mata kuliahnya.

3. Registrar (BAAK)

Bagian Registrar adalah bagian yang mengolah informasi yang masuk dengan bantuan system yang akan dibuat.

4. Billing (Pembayaran)

Bagian pembayaran (Kasir) menginformasikan biaya yang harus dikeluarkan mahasiswa menurut mata kuliah yang diikuti.

Use Case Model adalah penghubung (dialogue) antara Actors dan sistem, Use Case menerangkan tentang fungsi–fungsi yang di sediakan oleh sistem dan menerangkan pula tentang bagian–bagian dari sistem yang dapat digunakan *Actors.*

Definisi formal dari Use Case adalah sebuah bentuk dari transaksi yang dihasilkan oleh sistem yang akan mengeluarkan hasil untuk aktor.

Notasi Use Case pada UML

Berikut ini pertanyaan yang akan membantu mengidentifikasi *Use Case* dalam sistem :

- Apa tugas masing-masing *Actor?*
- Apakah *Actors akan menghasilkan, mengirimkan, merubah, memindah atau* membaca informasi pada sistem?
- *Use Case apa yang akan membuat, mengirim, merubah, memindah atau* membaca informasi tersebut?
- Apakah tiap *Actor perlu untuk menginformasikan ke sistem tentang* perubahan eksternal?
- Apakah tiap *Actor perlu diberitahu tentang hal-hal umum dari sistem ?*
- *Use Case apa sajakah yang akan mendukung dan menjaga sistem ?*
- Bisakah semua fungsi permintaan dibentuk dari *Use Case?*

Untuk Kebutuhan Sistem berikut:

- *Actor mahasiswa perlu untuk menggunakan sistem untuk mendaftar mata kuliah*
- Setelah proses pemilihan mata kuliah selesai, Billing System harus diisi dengan Billing informasi
- *Actor Dosen perlu untuk menggunakan sistem untuk memilih mata kuliah mana* yang akan diajarkan pada tiap semester. Dan harus bisa menerima perputaran mata kuliah didalam sistem tersebut
- Pihak Administrasi bertanggung jawab untuk membuat katalog mata kuliah pada semester dan juga untuk menjaga semua informasi tentang kurikulum, mahasiswa dan Dosen yang diperlukan oleh sistem.

Dari kebutuhan tersebut, *Use Case yang teridentifikasi adalah sebagai berikut :*

1. Pendaftaran untuk mata kuliah
2. Pilihan untuk mata kuliah yang akan diajarkan
3. Permintaan perputaran mata kuliah
4. Pemeliharaan informasi mata kuliah
5. Pemeliharaan informasi Dosen
6. Pemeliharaan informasi mahasiswa
7. Pembuatan katalog mahasiswa

Use Case Relationships

Sebuah gabungan (*relationships) akan terjadi diantara Actor dan Use Case. Penggunaan Use Case Relationships ini berguna untuk bila terjadi pemakaian Use Case secara* bersama oleh beberapa *Actor. Dalam UML di simbol dengan stereotype (<< >>) yang* dapat diisi dengan *relationships yang terjadi.*

Use Case Diagrams

Use Case Diagrams adalah tampilan grafik dari rancangan relationship yang dibuat untuk dasar sistem yang dibuat.

Pada *Use Case Diagrams ini akan diperlihatkan semua Use Case yang digunakan oleh Actor, semua keterangan yang mengimplementasikan dari* sistem, dan menampilkan semua *relationships dari semua Use Case*

The Key Elements of Use Case Modeling

Use Case Modeling

The Top 10 Use Case Modeling Guidelines

10. Follow the two-paragraph rule.
9. Organize your use cases with actors and use case diagrams.
8. Write your use cases in active voice.
7. Write your use case using an event/response flow, describing both sides of the user/system dialogue.
6. Use GUI storyboards, prototypes, screen mockups, etc.
5. Remember that your use case is really a runtime behavior specification.
4. Write the use case in the context of the object model.
3. Write your use cases using a noun-verb-noun sentence structure.
2. Reference domain classes by name.
1. Reference boundary classes (e.g., screens) by name.

The Top 10 Use Case Modeling Errors

1. Write functional requirements instead of usage scenario text.
2. Describe attributes and methods rather than usage.
3. Write the use cases too tersely.
4. Divorce yourself completely from the user interface.
5. Avoid explicit names for your boundary objects.
6. Write using a perspective other than the user's, in passive voice.
7. Describe only user interactions; ignore system responses.
8. Omit text for alternative courses of action.
9. Focus on something other than what's "inside" a use case, such as how you get there or what happens afterward.
10. Spend a month deciding whether to use includes or extends.

top 10 use case modeling guidelines

10. Follow the two-paragraph rule.
9. Organize your use cases with actors and use case diagrams.
8. Write your use cases in active voice.
7. Write your use case using an event/response flow, describing both sides of the user/system dialogue.
6. Use GUI storyboards, prototypes, screen mockups, etc.
5. Remember that your use case is really a runtime behavior specification.
4. Write the use case in the context of the object model.
3. Write your use cases using a noun-verb-noun sentence structure.
2. Reference domain classes by name.
1. Reference boundary classes (e.g., screens) by name.

The Top 10 Use Case Modeling Errors

- *Write functional requirements instead of usage scenario text.*
- *Describe attributes and methods rather than usage.*
- *Write the use cases too tersely.*
- *Divorce yourself completely from the user interface.*
- *Avoid explicit names for your boundary objects.*
- *Write using a perspective other than the user's, in passive voice.*
- *Describe only user interactions; ignore system responses.*
- *Omit text for alternative courses of action.*
- *Focus on something other than what's "inside" a use case, such as how you get there or what happens afterward.*
- *Spend a month deciding whether to use includes or extends.*

- 10. Follow the two-paragraph rule.

HOW TO WRITE A USE CASE: THE THREE MAGIC QUESTIONS

Well, OK, this whole chapter describes how to write a use case. But when writing use cases, you need to keep asking the following three fundamental questions:[1]

1. What happens?

   (This gets your "sunny-day scenario" started.)

2. And then what happens?

   (Keep asking this question until your "sunny-day scenario" is complete.)

3. What else might happen?

   (Keep asking this one until you've identified all the "rainy-day scenarios" you can think of, and described the related behavior.)

- 9. Organize your use cases with actors and use case diagrams.

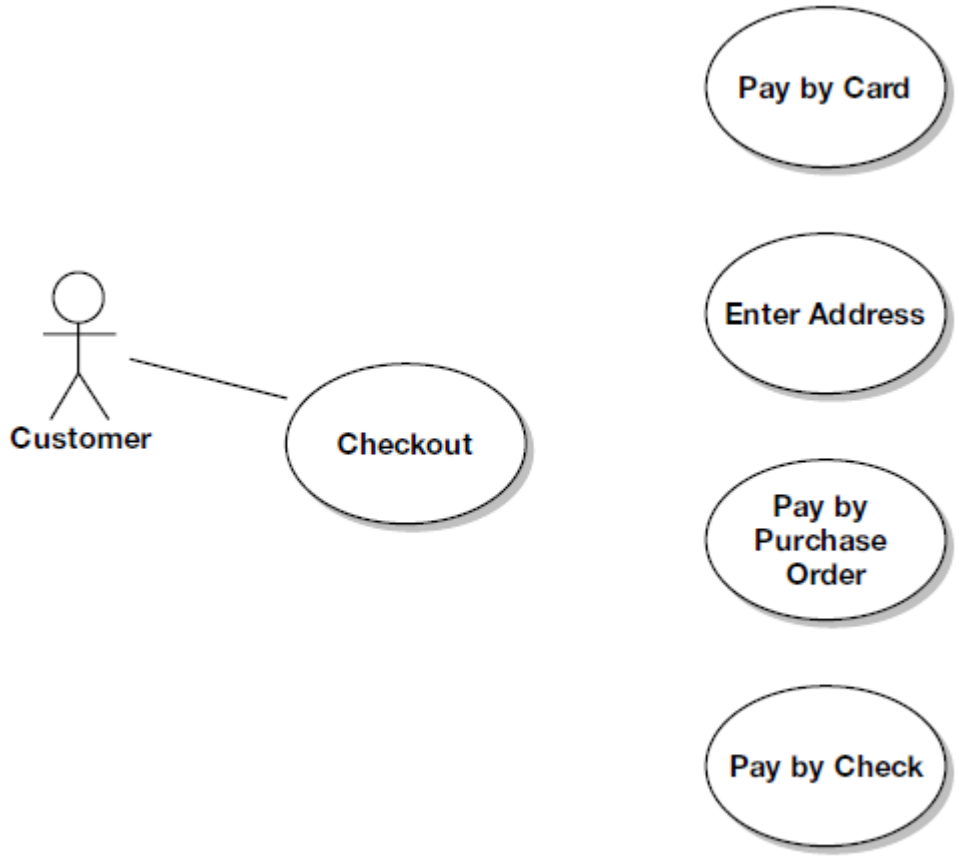


Figure 3-1. *Example use case diagram*

- 8. Write your use cases in active voice.

- 7. Write your use case using an event/response flow, describing both sides of the user/system dialogue.

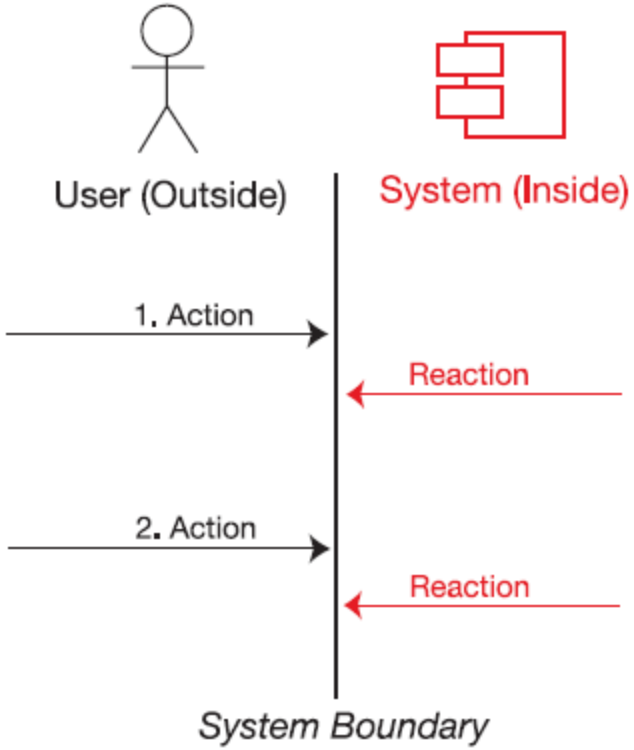


Figure 3-2. *Anatomy of a use case scenario*

- 6. Use GUI storyboards, prototypes, screen mockups, etc.

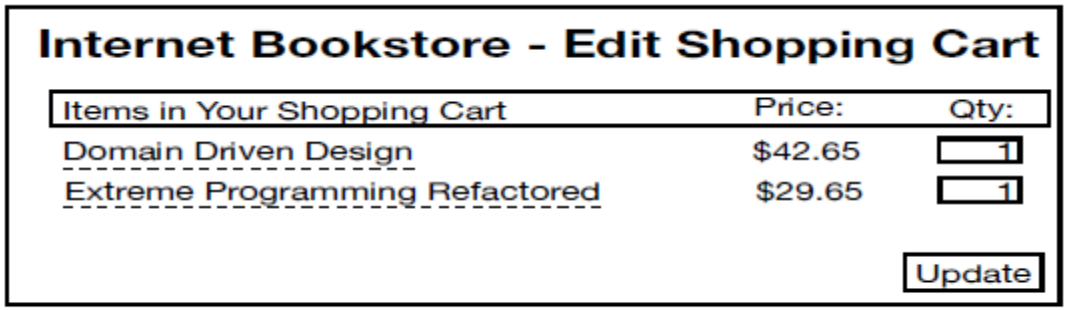


Figure 3-3. *Example UI storyboard*

5. Remember that your use case is really a runtime behavior specification.

Q&A: USE CASE = USER DOCUMENTATION?

You can always think of the use case text as being a narrative describing the user's adventures when interacting with the system. So it goes, "First the user does this; next the user does that."

Q: But don't I also have to describe the system's response?
A: Yes. So the text should really go, "First the user does this; then the system responds with that. Next the user does something; then the system responds with . . ." and so on.

Q: Isn't that similar to writing a user guide?
A: You got it. In fact, being "use case driven" can be summed up like this: *First write the user guide and then write the code.*

Q: Presumably this principle carries all the way through to the design?
A: Yes, the goal is to build something that implements the behavior requirements, so the system you'll be designing will be strongly correlated with the viewpoint of the end users. In other words, first you're describing system usage from the user's perspective, and then you're designing and unit-testing from the user's perspective, too.

Q: What if we're updating a legacy system?
A: The same principle still applies. Simply *work backward from the user guide.* Analyze the existing functionality, and then make changes based on how those functions will be performed in the new system. You should find yourself breaking the legacy user guide down into its fundamental components, from which you can then derive your use case scenarios.

4. Write the use case in the context of the object model.

Repeat this mantra at least a hundred times before breakfast:

> **You can't drive object-oriented designs from use cases unless you tie your use cases to objects.**

3. Write your use cases using a noun-verb-noun sentence structure.

You'll be amazed how much easier it is to create an object-oriented design if your use case text follows the **noun-verb-noun** style. Your use case text will ultimately reside on the margin of your sequence diagram (see Chapter 8). And sequence diagrams are fundamentally geared around nouns and verbs:

- The nouns are the **object instances**. These usually either come from the domain model (entities) or are boundary/GUI objects.
- The verbs are the **messages between objects**. These represent the software functions (controllers) that need to be built.

So, by writing your use case in **noun-verb-noun** format, you're setting yourself up to make the sequence diagramming task considerably easier than it would be otherwise.

2. Reference domain classes by name.

1. Reference boundary classes (e.g., screens) by name.

The Top 10 Use Case Modeling Errors

- *Write functional requirements instead of usage scenario text.*
- *Describe attributes and methods rather than usage.*
- *Write the use cases too tersely.*
- *Divorce yourself completely from the user interface.*
- *Avoid explicit names for your boundary objects.*
- *Write using a perspective other than the user's, in passive voice.*
- *Describe only user interactions; ignore system responses.*
- *Omit text for alternative courses of action.*
- *Focus on something other than what's "inside" a use case, such as how you get there or what happens afterward.*
- *Spend a month deciding whether to use includes or extends.*

Exercise 1

[from Open Account]

Basic Course: The Customer enters the required information. The system validates the information and creates a new Account object.

Alternate Course: If any data is invalid, the system displays an appropriate error message.

[from Search by Author]

The user submits the request. The system displays another page that contains the search results.

[from Log In]

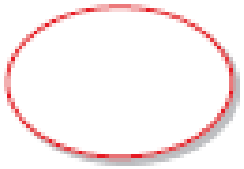
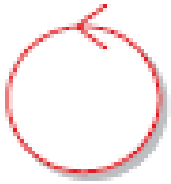
The Customer enters his or her user ID and password, and then clicks the Log In button. The system returns the Customer to the Home Page.

Use Case or Algorithm?

Many people get confused over the difference between use cases and algorithms, as both tend to be described with verb phrases and generally can be thought of as a sequence of steps. So here are some guidelines on how to differentiate between the two.

One of the main differences between a use case and an algorithm is that an algorithm, while it may contain a sequence of steps, **will not represent the dialogue between a user and the system. From the use case perspective, even very complicated algorithms should** just be considered a single step within the user/system dialogue. If you're faced with having to describe a complex algorithm when writing a use case (e.g., generating a list of recommended books, or sorting the list alphabetically), you should specify the algorithm elsewhere, but ***give the algorithm a name (e.g., "Generate Recommendations," "Sort List")*** so that it can be referred to in the use case text. Table 3-2 sums up the differences between a use case and an algorithm.

Table 3-2. *Use Case vs. Algorithm*

 Use Case	 Algorithm
Dialogue between user and system	"Atomic" computation
Event/response sequence	Series of steps
Basic/alternate courses	One step of a use case
Multiple participating objects	Operation on a class
User and System	All System

Exercise 3-1: Search by Author

BASIC COURSE:

The system displays the page with the search form; the user clicks the Author field and types in an author name (e.g., Fred Smith). The user clicks the Search button; the system reads the search form, looks up any books matching that author name, and displays them in a list.

ALTERNATE COURSES:

No matching books found: A page is displayed informing the user that no matching books were found.

Exercise 3-1 Solution: Explicit Boundary Object Names

The same problem can be found several times in this use case: the boundary objects haven't been given explicit names. The fixed version follows.

BASIC COURSE:

The system displays the Search Page; the user clicks the Author field and types in an author name (e.g., Fred Smith). The user clicks the Search button; the system reads the search form, looks up any books matching that author name, and displays the Search Results page showing the resulting Book List.

ALTERNATE COURSES:

No matching books found: The Search Not Found page is displayed.

Exercise 3-2: Edit Shopping Cart

PRECONDITIONS:

The user has logged in.

The user has navigated to the Edit Shopping Cart page.

BASIC COURSE:

The user adds or removes whatever items he wants to change, and then clicks the Update button. The system adds or removes the items, and then displays the page with the updated shopping cart.

ALTERNATE COURSES:

Shopping cart is empty: No items can be removed.

Exercise 3-2 Solution: Vague and Ambiguous

There are at least three problems with this use case.

Problem 1: The use case includes a "Preconditions" clause. Although on very rare occasions, you might find that it's useful to include this clause, most of the time it serves no appreciable purpose. In this example, it actually throws the use case text off course, as the initial "display" action is missed. This would in turn be missed out on the robustness diagram, meaning it would likely be skipped on the design, not estimated for, and not tested.

Problem 2: The basic course text is a bit woolly. It doesn't describe a specific scenario, but instead tries to cover all bases ("The user adds or removes whatever item . . ."). As a result, an important behavioral aspect is missed: the user wouldn't necessarily want to add items from this page, just remove them (or change the quantity).

Problem 3: The alternate course doesn't tie into any particular action in the use case text. There are also several relatively obvious alternate courses that are missing.

The fixed version follows.

BASIC COURSE:

The system displays the Shopping Cart page. The user clicks the Remove button next to a Line Item. The system removes the item from the user's Shopping Cart, and then redisplays the page. The user then clicks the Quantity text field for another Line Item, changes its value from 1 to 2, and clicks the Update button. The system updates the Shopping Cart, recalculates the total amount, and redisplays the page.

ALTERNATE COURSES:

Item not found: The item that the user chose to remove wasn't found in the Shopping Cart (this could happen if the user had two browser tabs open and is viewing an older version of the page). The system refreshes the Shopping Cart page, along with a warning message that the user's action failed because the page was out of date.

Quantity changed to zero: This counts as removing the item, so the item is removed from the Shopping Cart.

Negative value or non-numeric "value" entered: The page is redisplayed with the original Quantity value, and a message next to it informs the user that he entered an invalid value.

Exercise 3-3: Open an Account

BASIC COURSE:

The system displays the Create New Account page and enters the following fields: Username (must be unique), password, confirm password, first name, last name, address (first line), address (second line), city, state, country, zip/postal code, telephone number, and e-mail address. Then the user clicks the Submit button; the system checks that the Username is unique, creates the new user account, and displays the main Hub Page, along with a message indicating that the user account is now created and logged in.

ALTERNATE COURSES:

Password and Confirm Password don't match: The page is redisplayed with a validation message.

Username not unique: The page is redisplayed and the user is asked to choose a different username.

Exercise 3-3 Solution: Too Many Presentation Details

This use case gets bogged down in presentation details; it spends too long listing the fields to be found on the Create New Account page. Instead, these fields should be added as attributes to the appropriate class in your domain model (most likely the Customer class). Then, when you need them later, they'll be right there. The fixed version follows.

BASIC COURSE:

The system displays the Create New Account page and enters the fields to define a new Customer account (username, password, address, etc.). Then the user clicks the Submit button; the system checks that the Username is unique, creates the new user account, and displays the main Hub Page, along with a message indicating that the user account is now created and logged in.

ALTERNATE COURSES:

Password and Confirm Password don't match: The page is redisplayed with a validation message.

Username not unique: The page is redisplayed and the user is asked to choose a different username.

Exercise 1

Basic Course:

The Customer types his or her name, an email address, and a password (twice), and then presses the Create Account button. The system ensures that the Customer has provided valid data, and then creates an Account object using that data. Then the system returns the Customer to the Home Page.

Alternate Courses:

- ***I****f the Customer did not provide a name, the system displays an error message to that effect and prompts the Customer to type a name.*
- *If the Customer provided an email address that's not in the correct form, the system displays an error message to that effect and prompts the Customer to type a different address.*
- *If the Customer provided a password that is too short, the system displays an error message to that effect and prompts the Customer to type a longer password.*
- *If the Customer did not type the same password twice, the system displays an error message to that effect and prompts the Customer to type the password correctly the second time.*

Exercise 2

Basic Course: *The Customer enters his or her user ID and password, and then clicks the Log In button….*

On the Shopping Cart Page, the Customer modifies the quantity of an Item in the Shopping Cart, and then presses the Update button. *The system stores the new quantity, and then computes and displays the new cost for that Item….*

Basic Course: *The system ensures that the Order is cancellable (in other words, that its status isn't "shipping" or "shipped"). Then the system displays the relevant information for the Order on the Cancel Order* Page, including its contents and the shipping address. The Customer presses the Confirm Cancel button. *The system marks the Order status as "deleted," and then invokes the Return Items to Inventory use case.*

Alternate Course: *If the status of the Order is "shipping" or "shipped," the system displays a message indicating that it's too late for the Customer to cancel the order.*

Use Case - Log In

Documentation:

Basic Course

The Customer clicks the Log In button **on the Home Page**. The system **displays the Login Page**. The Customer enters his or her user ID and password and then clicks the Log In button. The system validates the login information against the persistent Account data and then returns the Customer **to the Home Page**.

Alternate Courses

If the Customer clicks the New Account button **on the Login Page**, the system invokes the Open Account use case.

If the Customer clicks the Reminder Word button **on the Login Page**, the system displays the reminder word stored for that Customer, **in a separate dialog box**. When the Customer clicks the OK button, the system returns the Customer **to the Login Page**.

If the Customer enters a user ID that the system does not recognize, the system **displays a message** to that effect and prompts the Customer to either enter a different ID or click the New Account button.

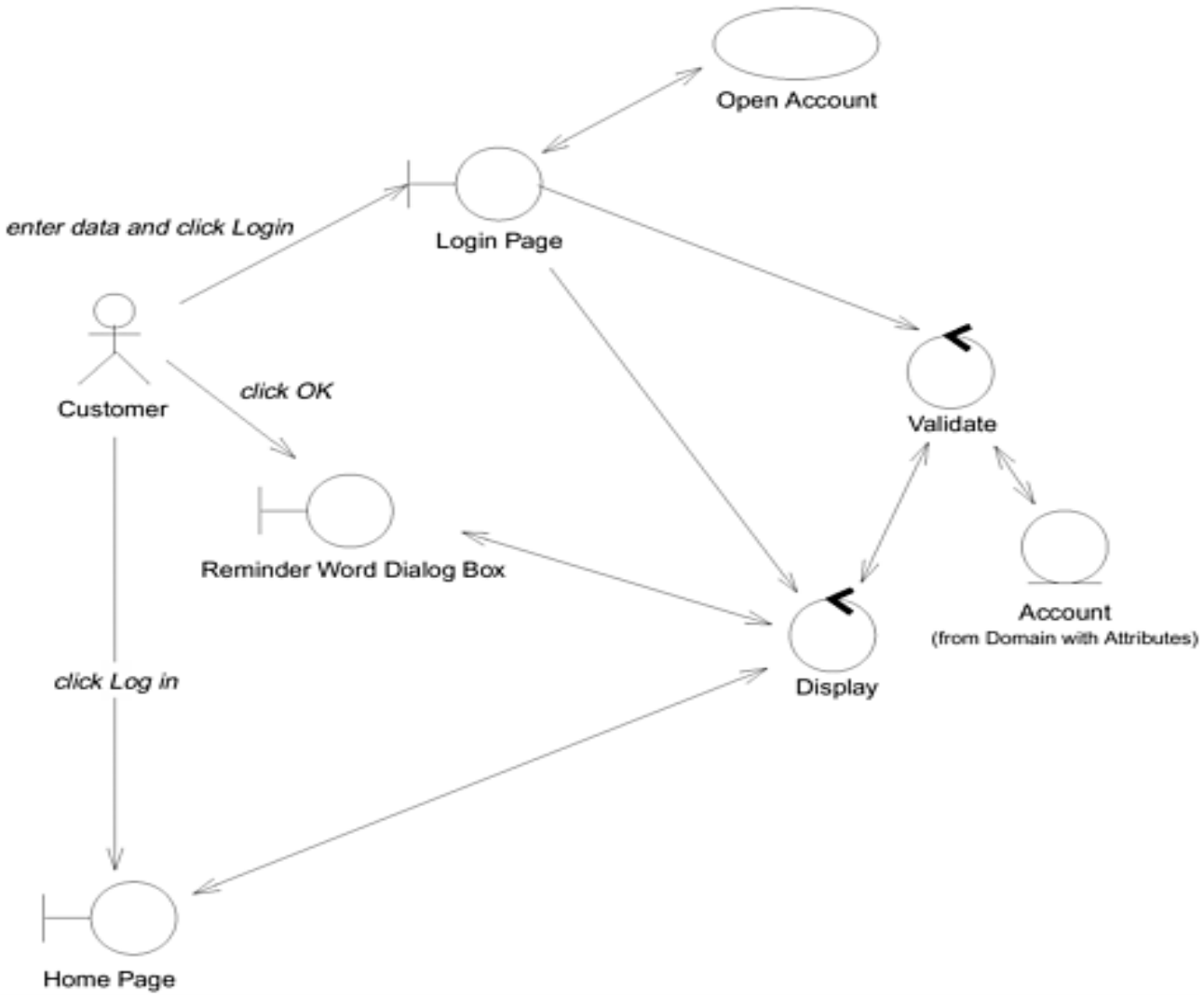
If the Customer enters an incorrect password, the system displays a message to that effect and prompts the Customer to reenter his or her password.

If the Customer enters an incorrect password three times, the system **displays a page** telling the Customer that he or she should contact customer service and also **freezes the Login Page**.

List of Associations

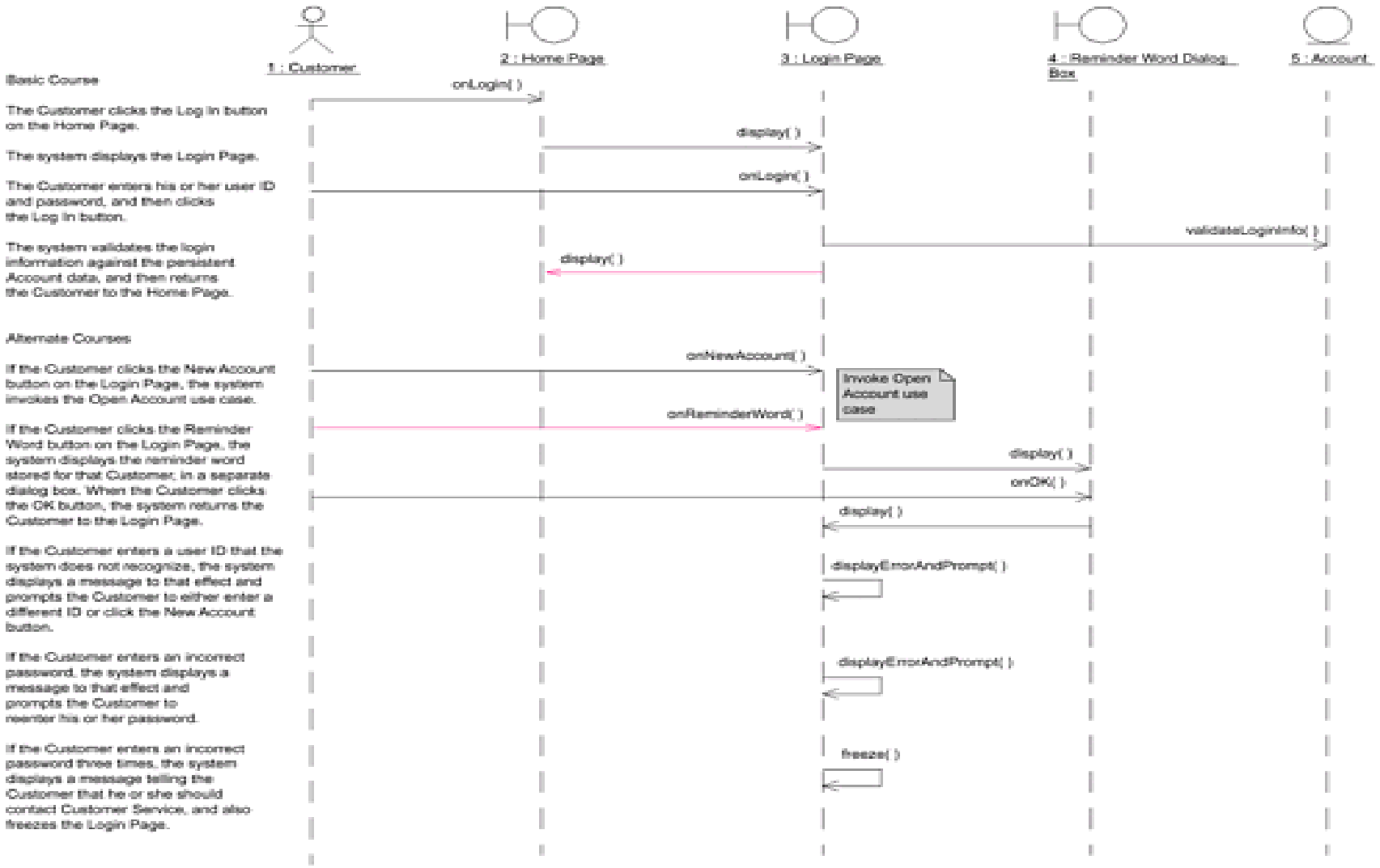
Customer Communicates with Log In

Class Diagram - Log In Robustness



The Customer clicks the Log In button **on the Home Page**. The system **displays the Login Page**. The Customer enters his or her user ID and password and then clicks the Log In button. The system validates the login information against the persistent Account data and then returns the Customer **to the Home Page**.

Interaction Diagram - Log In Sequence



Exercise 3-1: Search by Author

BASIC COURSE:

The system displays the page with the search form; the user clicks the Author field and types in an author name (e.g., Fred Smith). The user clicks the Search button; the system reads the search form, looks up any books matching that author name, and displays them in a list.

ALTERNATE COURSES:

No matching books found: A page is displayed informing the user that no matching books were found.

Exercise 3-1 Solution: Explicit Boundary Object Names

The same problem can be found several times in this use case: the boundary objects haven't been given explicit names. The fixed version follows.

BASIC COURSE:

The system displays the Search Page; the user clicks the Author field and types in an author name (e.g., Fred Smith). The user clicks the Search button; the system reads the search form, looks up any books matching that author name, and displays the Search Results page showing the resulting Book List.

ALTERNATE COURSES:

No matching books found: The Search Not Found page is displayed.

Use Case - Search by Author

Documentation:

Basic Course

The Customer types the name of an Author **on the Search Page** and presses the Search button. The system ensures that the Customer typed a search phrase and then searches the Catalog and retrieves all of the Books with which that Author is associated. The system retrieves the important details about each Book and creates a Search Results object with that information. Then the system **displays the list of Books on the Search Results** Page, with the Books listed in reverse chronological order by publication date. Each entry has a thumbnail of the Book's cover, the Book's title and authors, the average Rating, and an Add to Shopping Cart button. The Customer presses the Add to Shopping Cart button for a particular Book. The system passes control to the Add Item to Shopping Cart use case.

Alternate Courses

If the Customer did not type a search phrase before pressing the Search button, the system displays an error message to that effect and prompts the Customer to type a search phrase.

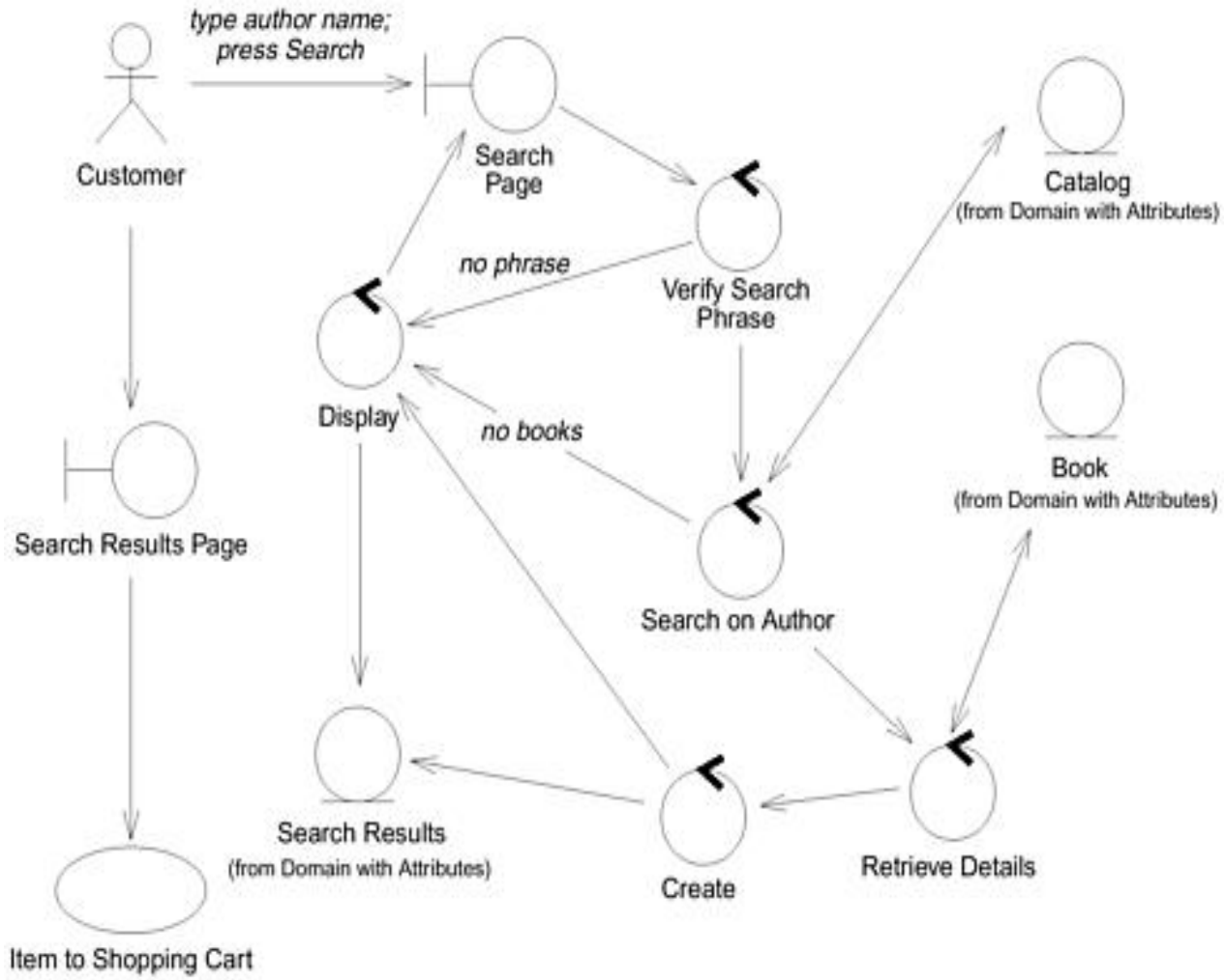
If the system was unable to find any Books associated with the Author that the Customer specified, the system displays a message to that effect and prompts the Customer to perform a different search.

If the Customer leaves the page in a way other than by pressing an Add to Shopping Cart button, the system returns control to the use case from which this use case received control.

List of Associations

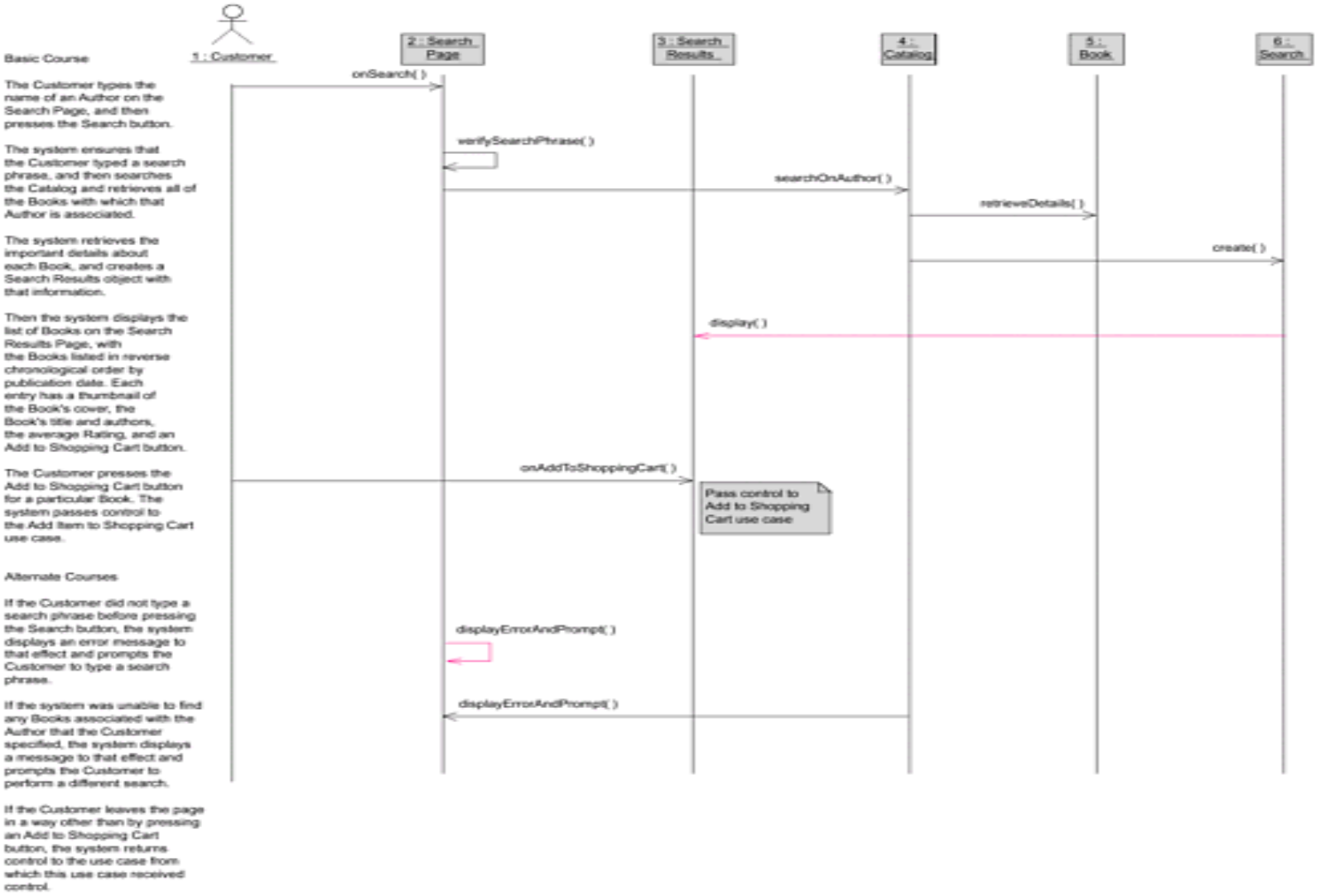
Customer Communicates with Search by Author

Class Diagram - Search by Author Robustness



The Customer types the name of an Author **on the Search Page** and presses the Search button. The system ensures that the Customer typed a search phrase and then searches the Catalog and retrieves all of the Books with which that Author is associated. The system retrieves the important details about each Book and creates a Search Results object with that information. Then the system **displays the list of Books on the Search Results** Page, with the Books listed in reverse chronological order by publication date. Each entry has a thumbnail of the Book's cover, the Book's title and authors, the average Rating, and an Add to Shopping Cart button. The Customer presses the Add to Shopping Cart button for a particular Book. The system passes control to the Add Item to Shopping Cart use case.

Interaction Diagram - Search by Author Sequence



Use Case - Ship Order

Documentation:

Basic Course

The Shipping Clerk ensures that the Items listed on the packing slip for the Order match the physical items. The Clerk waves the bar code on the packing slip under the sensor at the shipping station.

The system changes the status of the Order to “shipping.” Then the system retrieves the Shipping Method that the Customer specified for this Order and displays it on the Shipping Station Console.

The Clerk weighs the set of physical items. The Clerk packages the Items. The Clerk attaches a manifest appropriate for the given shipping method. The Clerk waves the bar code on the manifest under the sensor. The Clerk sends the package out via the associated Shipper.

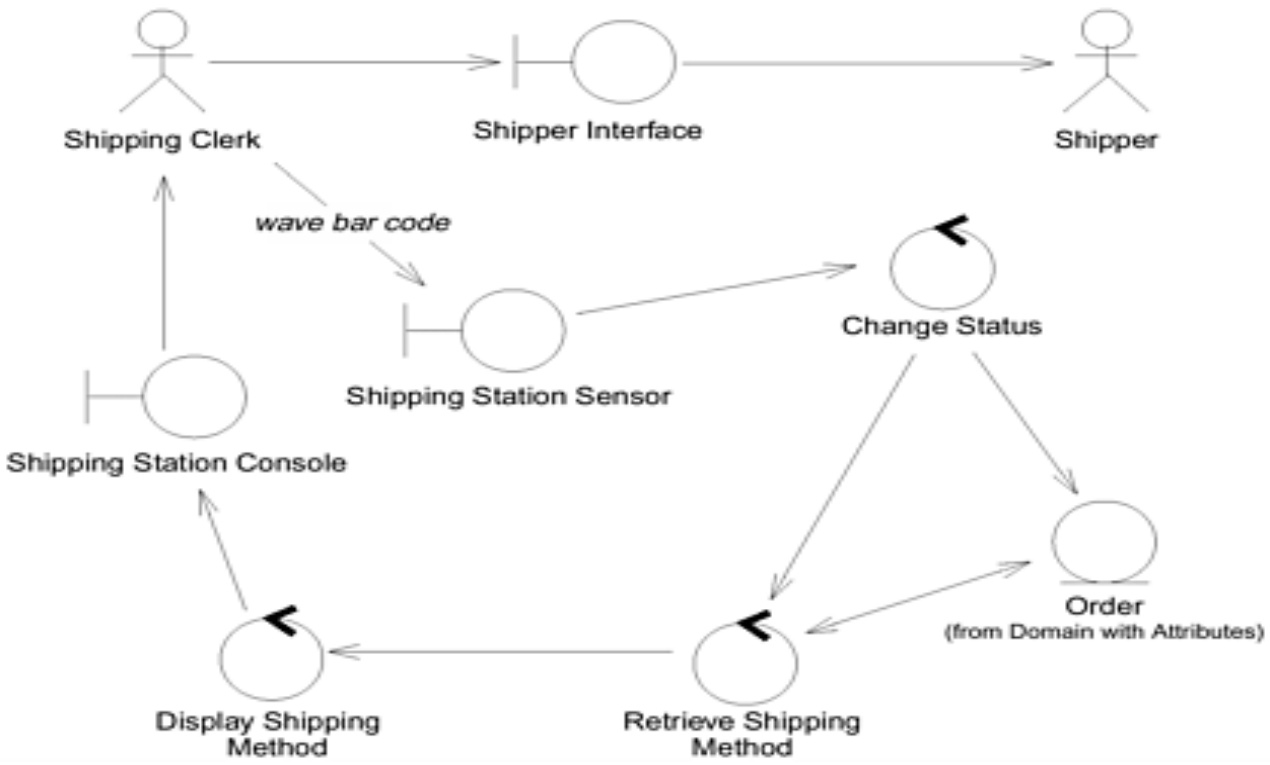
Alternate Course

If the Shipping Clerk finds a mismatch between the Order and the physical items, the Clerk stops processing of the Order until he or she is able to make a match.

List of Associations

- Shipping Clerk Communicates with Ship Order
- Ship Order Communicates with Shipper
- Ship Order Communicates with Shipping Station

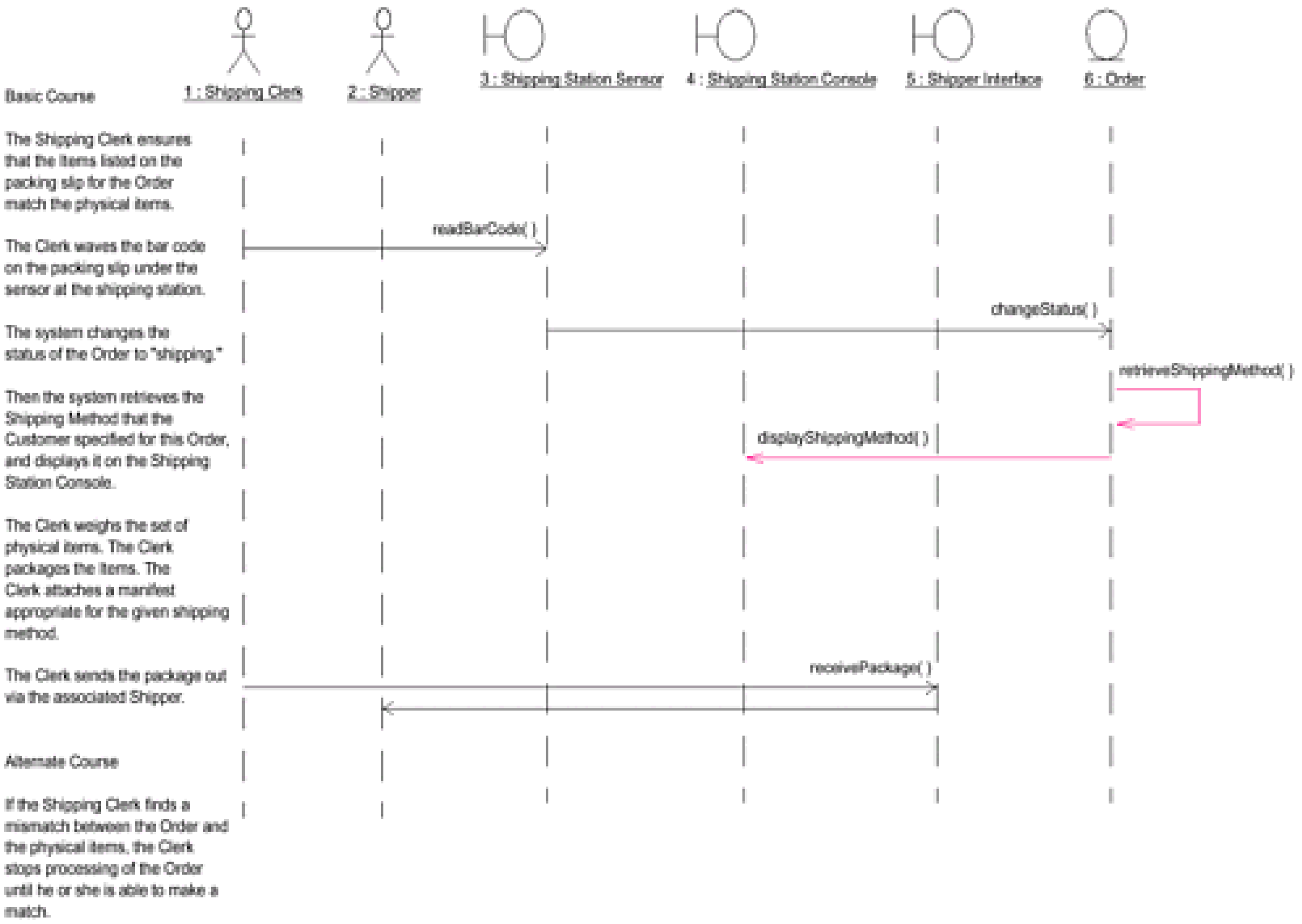
Class Diagram - Ship Order Robustness



The Shipping Clerk ensures that the Items listed on the packing slip for the Order match the physical items. The Clerk waves the bar code on the packing slip **under the sensor at the shipping station.** The system changes the status of the Order to "shipping." Then the system retrieves the Shipping Method that the Customer specified for this Order and displays it **on the Shipping Station Console**.

The Clerk weighs the set of physical items. The Clerk packages the Items. The Clerk attaches a manifest appropriate for the given shipping method. The Clerk waves the bar code on the manifest under the sensor. The Clerk sends the package out via the associated Shipper.

Interaction Diagram - Ship Order Sequence



Use Case - Track Recent Orders

Documentation:

Basic Course

The system retrieves the Orders that the Customer has placed within the last 30 days and displays these Orders on the Order Tracking Page. Each entry has the Order ID (in the form of a link), the Order date, the Order status, the Order recipient, and the Shipping Method by which the Order was shipped. The Customer clicks on a link. The system retrieves the relevant contents of the Order and then displays this information, in view-only mode, on the Order Details Page. The Customer presses OK to return to the Order Tracking Page. Once the Customer has finished viewing Orders, he or she clicks the Account Maintenance link on the Order Tracking Page. The system returns control to the invoking use case.

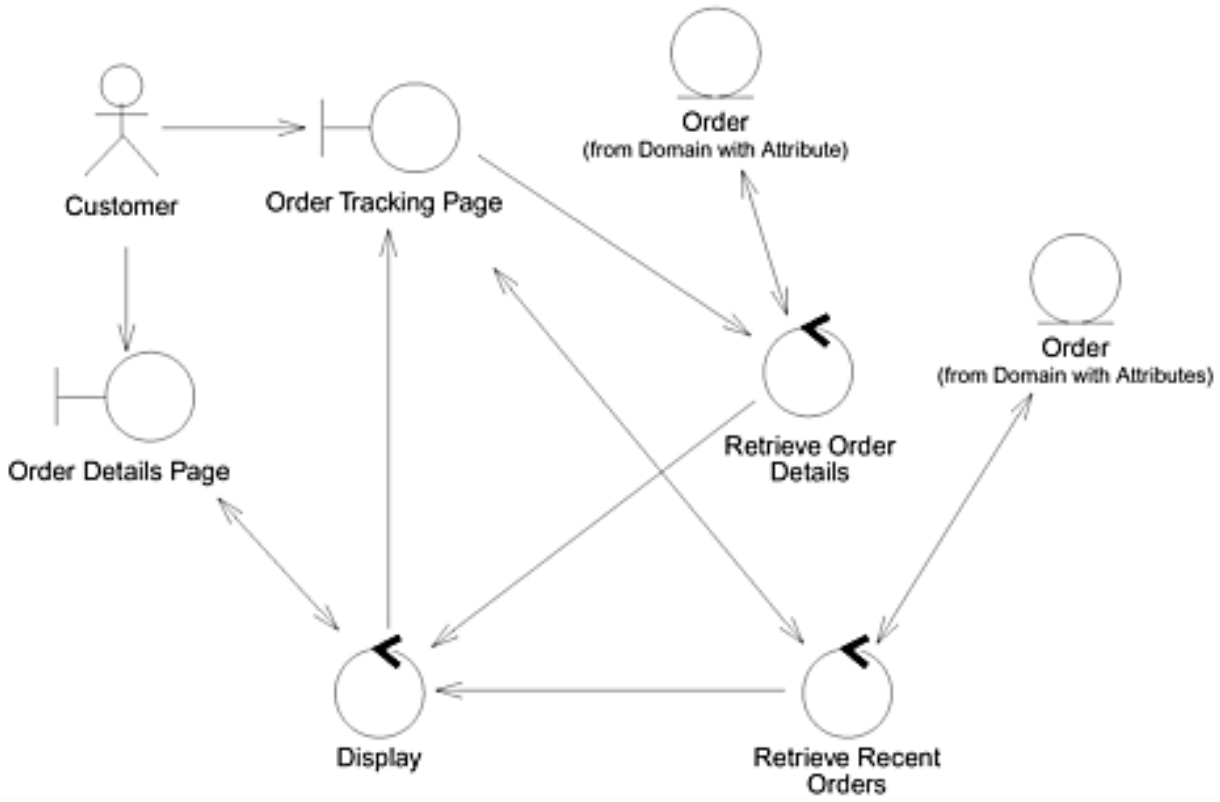
Alternate Course

If the Customer has not placed any Orders within the last 30 days, the system displays a message to that effect on the Order Tracking Page.

List of Associations

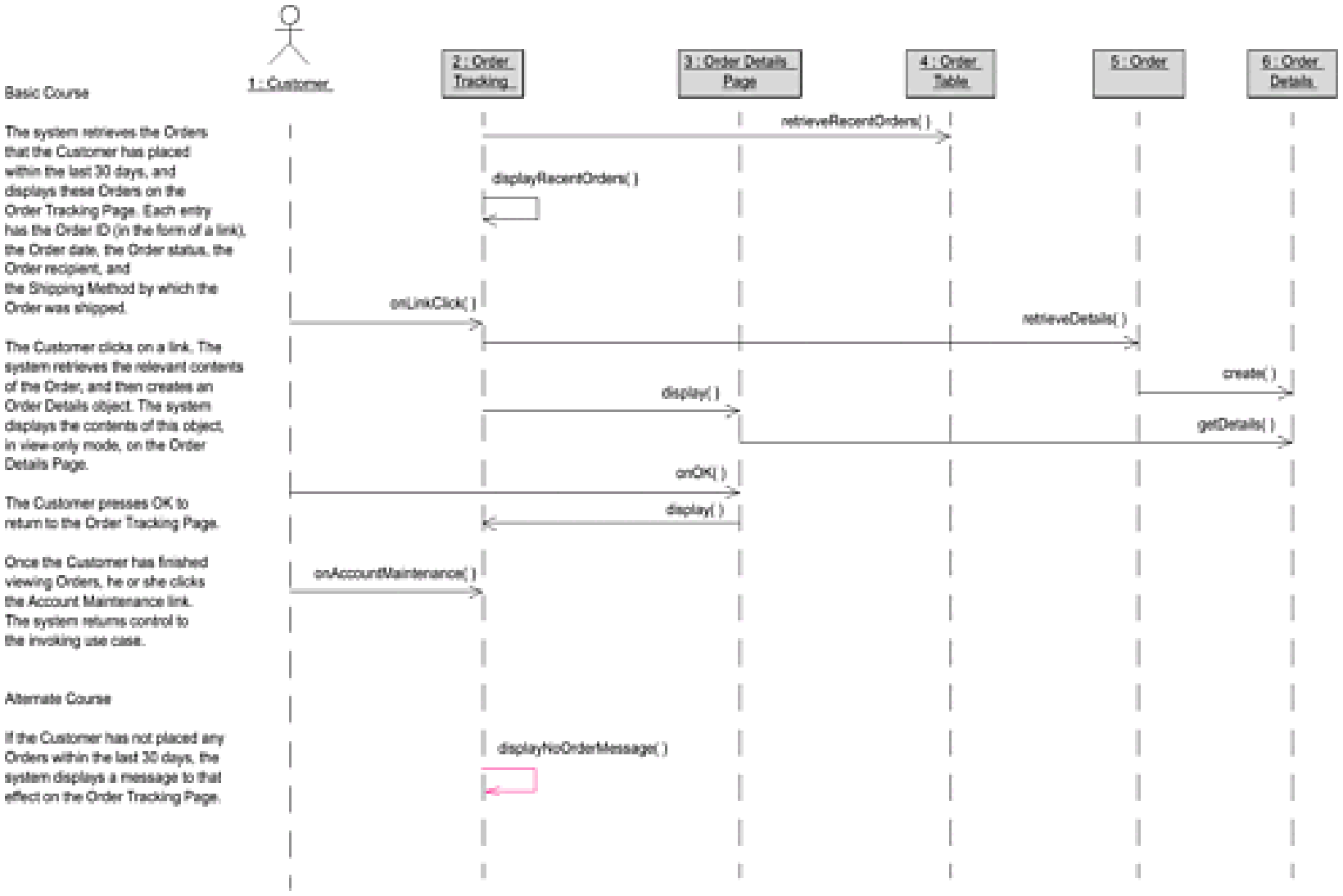
- Customer Communicates with Track Recent Orders

Class Diagram - Track Recent Orders Robustness



The system retrieves the Orders that the Customer has placed within the last 30 days and displays these Orders on the Order Tracking Page. Each entry has the Order ID (in the form of a link), the Order date, the Order status, the Order recipient, and the Shipping Method by which the Order was shipped. The Customer clicks on a link. The system retrieves the relevant contents of the Order and then displays this information, in view-only mode, on the Order Details Page. The Customer presses OK to return to the Order Tracking Page. Once the Customer has finished viewing Orders, he or she clicks the Account Maintenance link on the Order Tracking Page. The system returns control to the invoking use case.

Interaction Diagram - Track Recent Orders Sequence



Use Case - Edit Contents of Shopping Cart

Documentation:

Basic Course

On the Shopping Cart Page, the Customer modifies the quantity of an Item in the Shopping Cart and then presses the Update button. The system stores the new quantity and then computes and displays the new cost for that Item. The Customer presses the Continue Shopping button. The system returns control to the use case from which it received control.

Alternate Courses

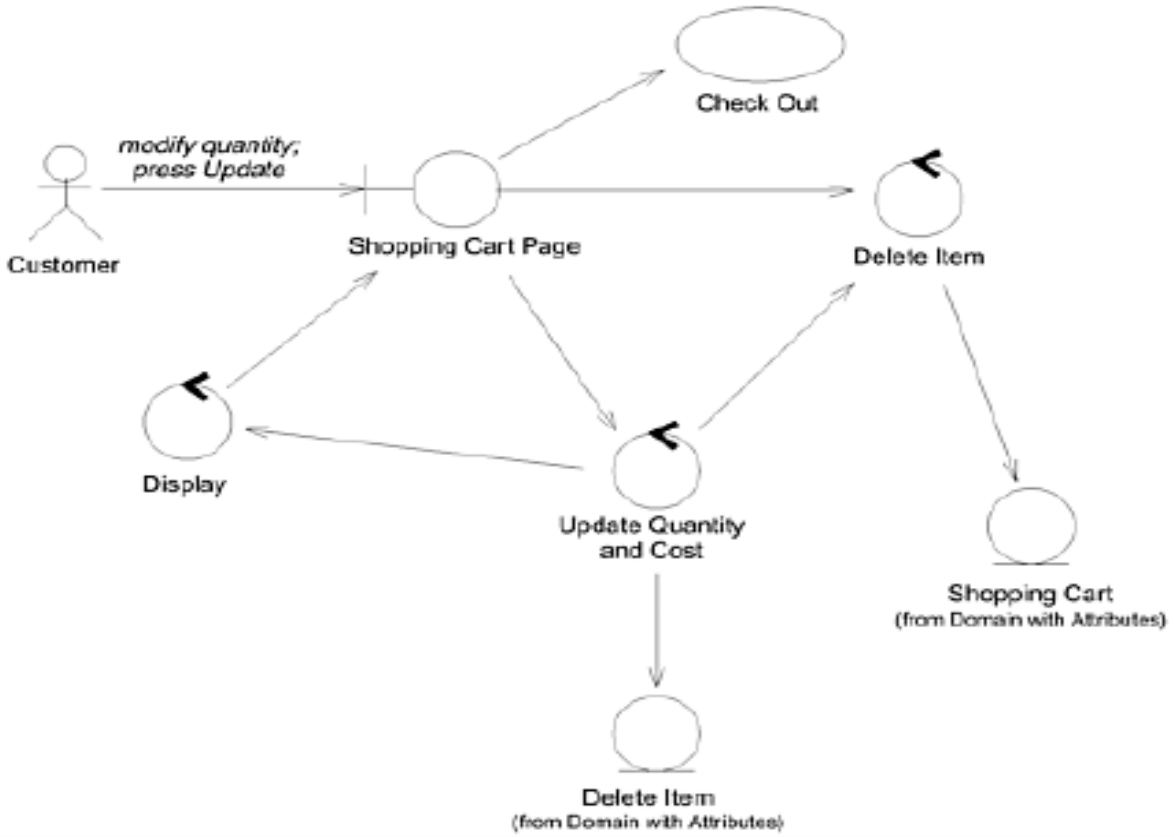
If the Customer changes the quantity of the Item to 0, the system deletes that Item from the Shopping Cart.

If the Customer presses the Delete button instead of the Update button, the system deletes that Item from the Shopping Cart.

If the Customer presses the Check Out button instead of the Continue Shopping button, the system passes control to the Check Out use case.

List of Associations

Customer Communicates with Edit Contents of Shopping Cart



On the Shopping Cart Page, the Customer modifies the quantity of an Item in the Shopping Cart and then presses the Update button. The system stores the new quantity and then computes and displays the new cost for that Item. The Customer presses the Continue Shopping button. The system returns control to the use case from which it received control.

Basic Course

On the Shopping Cart Page, the Customer modifies the quantity of an Item in the Shopping Cart, and then presses the Update button.

The system stores the new quantity, and then computes and displays the new cost for that Item.

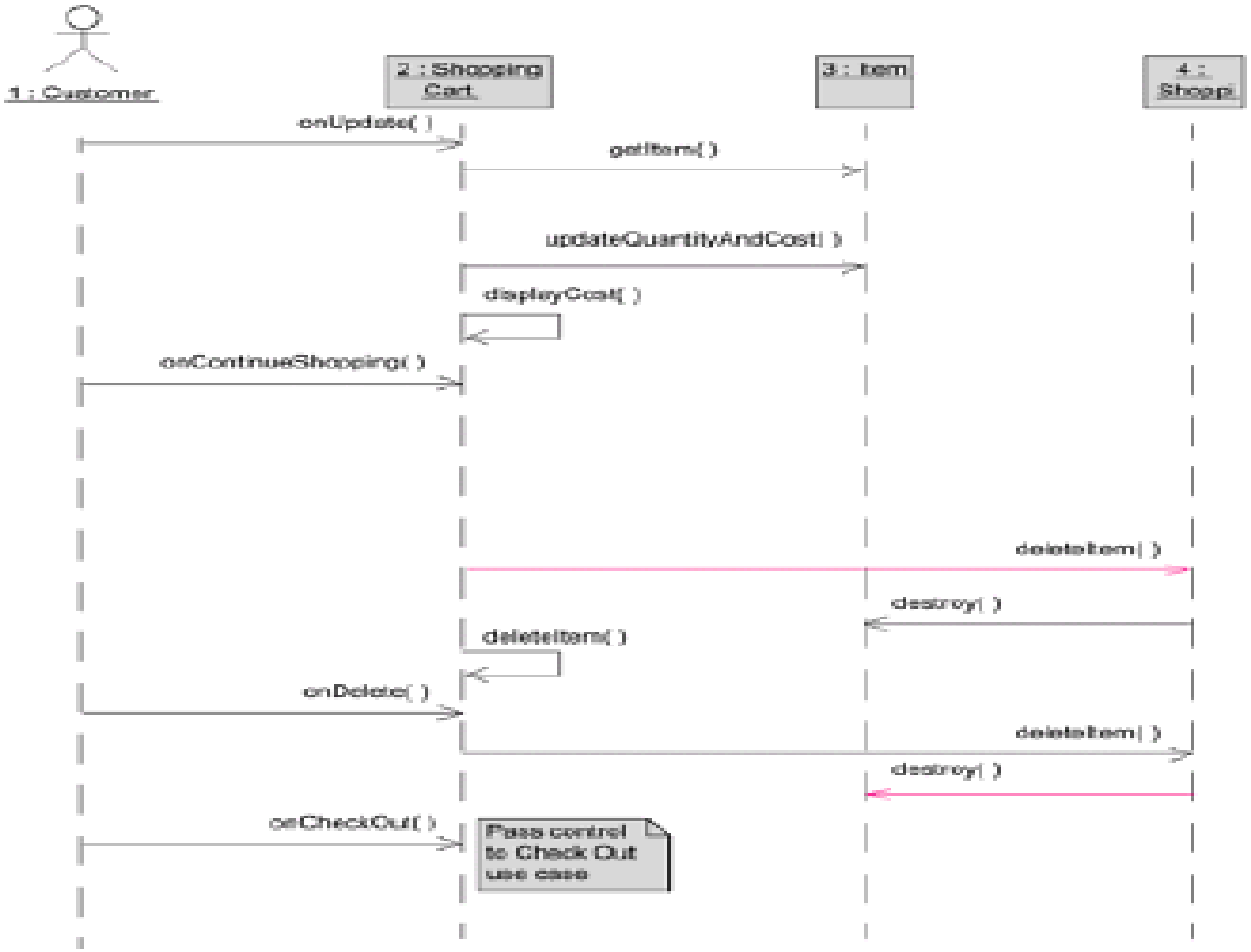
The Customer presses the Continue Shopping button. The system returns control to the use case from which it received control.

Alternate Courses

If the Customer changes the quantity of the Item to 0, the system deletes that Item from the Shopping Cart.

If the Customer presses the Delete button instead of the Update button, the system deletes that Item from the Shopping Cart.

If the Customer presses the Check Out button instead of the Continue Shopping button, the system passes control to the Check Out use case.



Exercise 3-3: Open an Account

BASIC COURSE:

The system displays the Create New Account page and enters the following fields: Username (must be unique), password, confirm password, first name, last name, address (first line), address (second line), city, state, country, zip/postal code, telephone number, and e-mail address. Then the user clicks the Submit button; the system checks that the Username is unique, creates the new user account, and displays the main Hub Page, along with a message indicating that the user account is now created and logged in.

ALTERNATE COURSES:

Password and Confirm Password don't match: The page is redisplayed with a validation message.

Username not unique: The page is redisplayed and the user is asked to choose a different username.

Exercise 3-3 Solution: Too Many Presentation Details

This use case gets bogged down in presentation details; it spends too long listing the fields to be found on the Create New Account page. Instead, these fields should be added as attributes to the appropriate class in your domain model (most likely the Customer class). Then, when you need them later, they'll be right there. The fixed version follows.

BASIC COURSE:

The system displays the Create New Account page and enters the fields to define a new Customer account (username, password, address, etc.). Then the user clicks the Submit button; the system checks that the Username is unique, creates the new user account, and displays the main Hub Page, along with a message indicating that the user account is now created and logged in.

ALTERNATE COURSES:

Password and Confirm Password don't match: The page is redisplayed with a validation message.

Username not unique: The page is redisplayed and the user is asked to choose a different username.

Use Case - Open Account

Documentation:

Basic Course

The Customer types his or her name, an e-mail address, and a password (twice), and then presses the Create Account button. The system ensures that the Customer has provided valid data and then adds an Account to the Master Account Table using that data. Then the system returns the Customer to the Home Page.

Alternate Courses

If the Customer did not provide a name, the system displays an error message to that effect and prompts the Customer to type a name.

If the Customer provided an email address that's not in the correct form, the system displays an error message to that effect and prompts the Customer to type a different address.

If the Customer provided a password that is too short, the system displays an error message to that effect and prompts the Customer to type a longer password.

If the Customer did not type the same password twice, the system displays an error message to that effect and prompts the Customer to type the password correctly the second time.

If the account is already in the Master Account Table, the system tells the Customer.

List of Associations

- Customer Communicates with Open Account
- Login Page Communicates with Open Account

Use Case - Browse List of Books

Documentation:

Basic Course

The Customer clicks on a Category on the Browse Books Page. The system displays the subcategories within that Category. This process continues until there are no more subcategories, at which point the system displays the Books in the lowest subcategory. The Customer clicks on the thumbnail for a Book. The system invokes the Display Book Details use case.

Alternate Course

If the system does not find any Books contained within a given Category, it displays a message to that effect and prompts the Customer to select a different Category.

List of Associations

- Customer Communicates with Browse List of Books

Use Case - Cancel Order

Documentation:

Basic Course

The system ensures that the Order is cancellable (in other words, that its status isn't "shipping" or "shipped"). Then the system displays the relevant information for the Order on the Cancel Order Page, including its contents and the shipping address. The Customer presses the Confirm Cancel button. The system marks the Order status as "deleted" and then invokes the Return Items to Inventory use case.

Alternate Course

If the status of the Order is "shipping" or "shipped," the system displays a message indicating that it's too late for the Customer to cancel the order.

List of Associations

- Search Results Page Communicates with Cancel Order

Use Case - Check Out

Documentation:

Basic Course

The system creates a Candidate Order object that contains the contents of the Customer's Shopping Cart. Then the system retrieves the Shipping Addresses associated with the Customer's Account and displays these addresses on the Shipping Address Page. The Customer selects an address and then presses the Use This Address button. The system associates the given Shipping Address with the Candidate Order. Then the system displays the available Shipping Methods on the Shipping Method Page. The Customer selects a shipping method and then presses the Use This Shipping Method button. The system associates the given Shipping Method with the Candidate Order. Then the system displays the contents of the Billing Info objects associated with the Customer's Account, on the Billing Information Page. The Customer selects a billing method and presses the Use This Billing Information button. The system associates the given Billing Info object with the Candidate Order. Then the system displays the Confirm Order Page. The Customer presses the Confirm Order button. The system converts the Candidate Order to an Order and destroys the Shopping Cart. Then the system returns control to the use case from which this use case received control.

Alternate Courses

If the Customer has not already logged in, the system invokes the Log In use case.

If the system does not find any Shipping Addresses, it invokes the Create Shipping Address use case.

If the system does not find any Billing Info objects, it invokes the Define Billing Information use case.

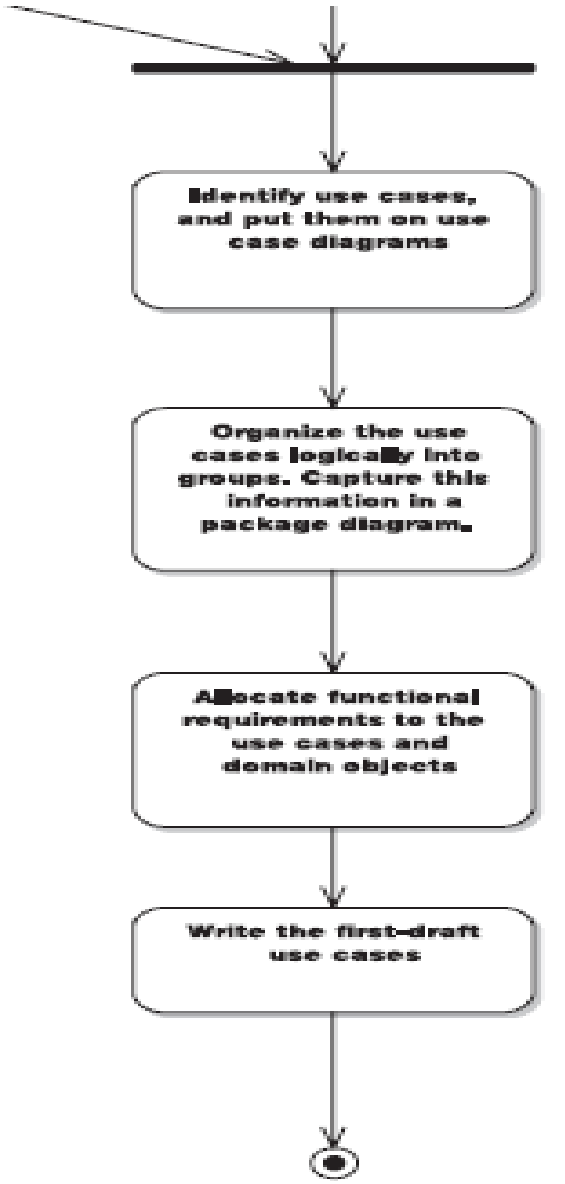
If the Customer presses the Cancel Order button at any time, the system destroys the Candidate Order and returns control to the use case from which this use case received control.

List of Associations

- Customer Communicates with Check Out
- Shopping Cart Page Communicates with Check Out

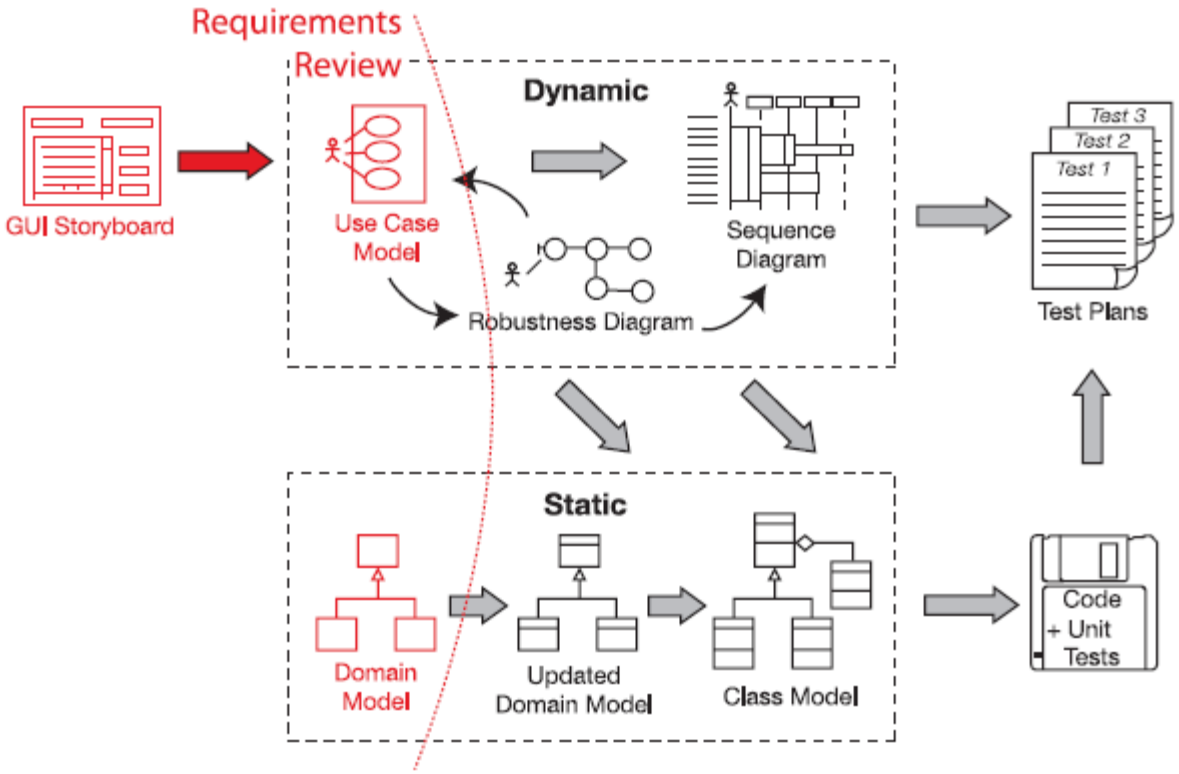

Requirements Analysis
Gather information about the legacy system you're reengineering
Database tables
Identify reusable domain objects
Screens
Put the domain objects on here
Do some rapid prototyping of the proposed new system
Draw the domain model
Identify use cases, and put them on use case diagrams
Organize the use cases logically into groups. Capture this information in a package diagram.
Allocate functional requirements to the use cases and domain objects
Write the first-draft use cases
Milestone 1: Requirements Review


Requirements Analysis
Gather information about the legacy system you're re-engineering
Database tables
Identify real-world domain objects
Screens
Put the domain objects on here
Do some rapid prototyping of the proposed new system
Draw the domain model


Identify use cases, and put them on use case diagrams
Organize the use cases logically into groups. Capture this information in a package diagram.
Allocate functional requirements to the use cases and domain objects
Write the first-draft use cases
Milestone 1: Requirements Review

Milestone 1: Requirements Review

- Chapter 4 (pg.116)



top 10 requirements review guidelines.

10. Make sure your domain model describes at least 80% of the most important abstractions (i.e., real-world objects) from your problem domain, in nontechnical language that your end users can understand.
9. Make sure your domain model shows the is-a (generalization) and has-a (aggregation) relationships between the domain objects.
8. Make sure your use cases describe both basic and alternate courses of action, in active voice.
7. **If you have lists of functional requirements** (i.e., "shall" statements), make sure these are not absorbed into and "intermingled" with the active voice use case text.4
6. Make sure **you've organized your use cases into packages** and that **each package has at least one use case diagram**.
5. Make sure your use cases are written in the context of the object model.
4. Put your use cases in the context of the user interface.
3. Supplement your use case descriptions with some sort of storyboard, line drawing, screen mockup, or GUI prototype.
2. Review the use cases, domain model, and screen mockups/GUI prototypes with end users, stakeholders, and marketing folks, in addition to more technical members of your staff.
1. Structure the review around our "eight easy steps to a better use case" (see Chapter 4).

10. Make Sure Your Domain Model Covers the Problem Domain

Sometimes "the best is the enemy of the good."[3] You don't want to spend endless weeks (and catch a case of analysis paralysis) fiddling around with the domain model, trying to make it perfect. Instead, you should be aiming for the following:

- A domain model diagram that has the most important abstractions from the problem domain
- All the boxes on this diagram have unambiguous names that the users of the system can relate to
- Getting this diagram done quickly!

Further details will be uncovered as you analyze the use cases.

9. Show Generalization and Aggregation Relationships

These generalization (is-a) and aggregation (has-a) relationships go a long way toward making the diagram concrete and specific as opposed to vague and ambiguous.

Occasionally, you might need a "link class" (especially if you have underlying many-to-many relationships in the data model), but it's best not to overuse this construct. You can get most of the way just by using is-a and has-a.

Don't split hairs (yet) between aggregation and composition, either—it's too early to worry about this distinction.

8. Describe Both Basic and Alternate Courses of Action, in Active Voice

You already know this, but the use cases describe how the users are using the system, in present tense, active voice, and they must include both normal (sunny day) usage as well as less typical (rainy day) usage.

We know that you already know this, but we've also seen many of your peers who (in theory) already knew, but didn't do it correctly (in practice). So we're telling you again. We might tell you a couple more times, too.

7. Don't Mix Functional Requirements into Your Use Case Text

Functional requirements are passive voice "shall" requirements (e.g., "The system shall do this"). A common mistake is to include these in the use case descriptions. Instead, the functional requirements should be kept separately and "allocated" to the use cases. That is, the use cases satisfy the functional requirements, but the passive voice "shall" requirements don't compose the use cases. Keep the use cases focused on system usage (active voice).

In Chapter 13, we'll show you how easy it is to allocate requirements to use cases. It's just a single drag and drop if you do it correctly.

6. Organize Your Use Cases into Packages

You can organize your use case model by functionally related areas (subsystems), by release, or both. One useful strategy is to identify all the use cases within a functional area on the use case diagram, but write the narrative descriptions only for the use cases that will participate in the current release.

Tip It's useful to use color-coding on the use case diagram to show which use cases will be implemented in the current release.

By all means don't get hung up on trying to perfect your use case diagram with a flawless use of stereotypes—here again, "the best is the enemy of the good" (i.e., you don't have time on your schedule for endless fiddling around). **The use case diagram really serves as a visual table of contents** for the package, and so it should be clear and easy to understand, but the real "meat" of the use case model is the text of the use cases and the corresponding robustness and sequence diagrams for each use case.

5. Write Your Use Cases in the Context of the Object Model

By far the most effective way to base your use cases on the domain model is to **do the domain model first**, and **write the use cases to explicitly reference the domain objects by name**.

Much of the ambiguity in use cases arises from the fact that they are often written entirely in "user terms" without explicitly and precisely referring to the particular "problem domain elements" that are affected by the scenario in question. So the first step in disambiguating your use case text is to make that text explicitly refer to the appropriate domain objects.

4. Put Your Use Cases in the Context of the User Interface

To base your use cases on the user interface, name the screens that will participate in the use case, and use those names in the use case text.

The use cases really need to link not only the object model but also the GUI model to the narrative behavior descriptions. In practical terms, this usually means that you should **name your screens explicitly and use those names in the use case text**.

While there is a theory that preaches keeping your use cases completely divorced from the user interface, our experience is that in practice, this inevitably leads to vagueness and ambiguity in the use cases. So name your screens. Trust us, you'll need to give them names anyway. You'll be happy that you've named them. Really.

Tip Name your screens, and use those names in your use case descriptions.

---

3. Use Storyboards, Line Drawings, Screen Mock-ups, or GUI Prototypes

Make sure all of the user's behavior (e.g., buttons they can click, menus they can pick from) is accounted for in the use cases.

Once you've given your screen a name, you have no excuse for not creating some sort of storyboard or mock-up that helps you to walk through the actions that users can take when they interact with the system. It's amazing how much system behavior might be tied to, let's say, the Cancel button in, for example, a transaction-oriented system. You might have to roll back to the previously completed transaction. But if you don't draw some sort of visual aid that shows what's on the screen, you might forget to write the "On Click Cancel" behavior in your use case text. And you wouldn't want to forget that, would you?

Remember that these mock-ups don't need to be high-fidelity renderings with animated buttons ("the best is the enemy of the good," again). They need to be simple, clear illustrations that focus on what system behavior can be triggered by the user, and they need to be quick to create (otherwise you'll ignore our advice and skip drawing them—admit it!).

Tip Don't be afraid to build some exploratory prototypes to help validate your requirements.

If you do have any exploratory prototypes, walk through them in conjunction with reviewing the use cases to gain insight into the required behavior. Sometimes a storyboard might not be enough, and you want to string a few screens together before a presentation to your users. If you need to do this, go right ahead.

2. Review the Behavioral Requirements with the Right People

Review the use cases, domain model, and screen mock-ups/GUI prototypes with end users, stakeholders, and marketing folks, in addition to more technical members of your staff. Make sure the requirements are well understood and agreed upon by all parties before proceeding toward design.

Your requirements review is a collaborative workshop that should be attended by the customer (or representatives of the customer), the business analyst (i.e., the senior analyst responsible for the use cases), and a senior member of your development team, plus any project managers who will be closely involved in the project. Ideally, an actual end user of the proposed (or current) system should also be there, to provide real-world feedback on whether the new system matches up with what is really needed.

The goal of your requirements review session is to achieve basic agreement among all parties that the use cases capture the behavioral requirements of the system.[4] And in order to achieve that goal, you must ensure that the required system behavior is unambiguously understood by all.[5]

In fact, you're not just reviewing the use cases; you also need to review the domain model and whatever prototype elements are in place. This works best when everyone is in a room together, with a facilitator/moderator who keeps the conversations on track and a scribe who records the results and the action items.

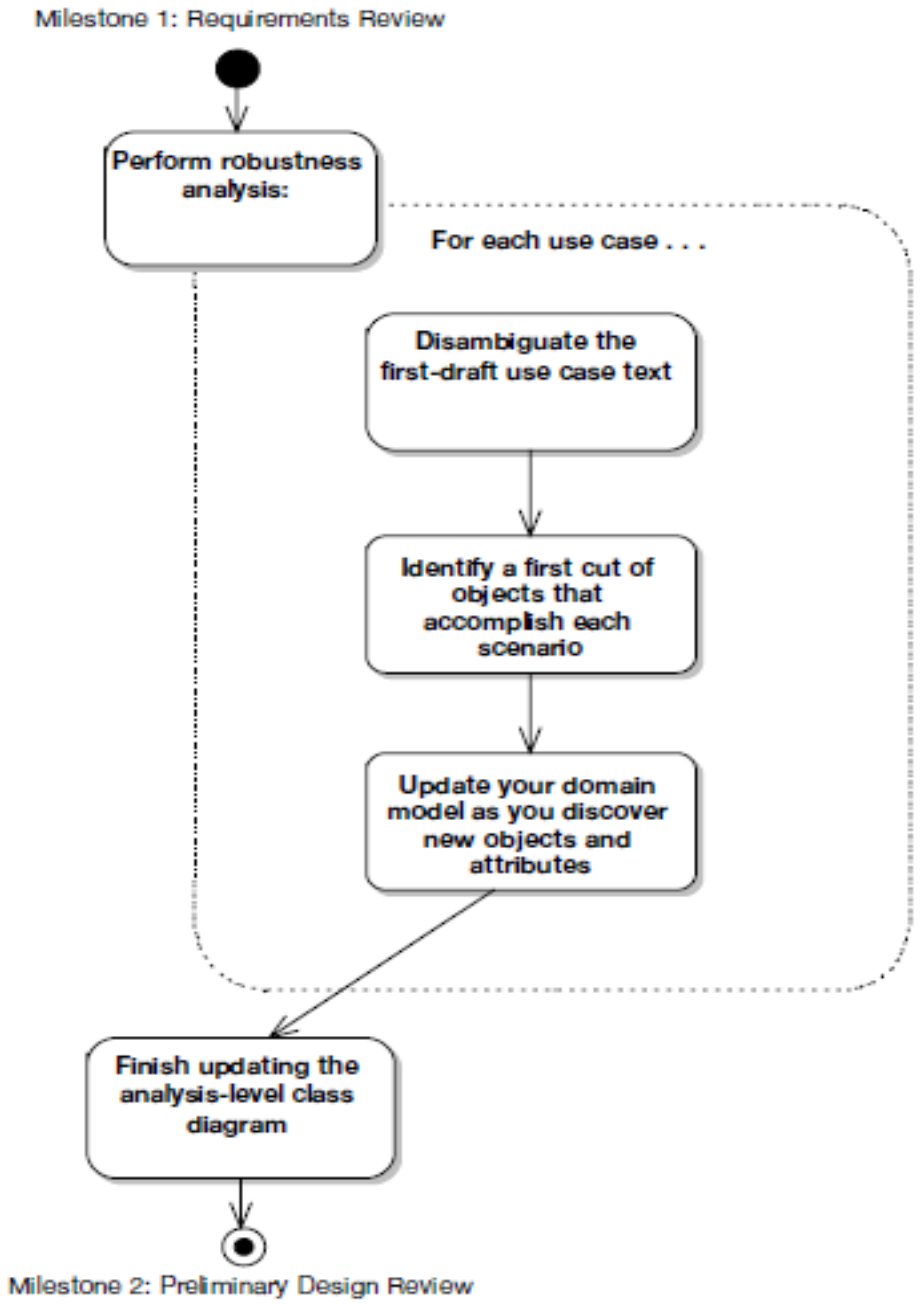
1. Structure the Review Around Our "Eight Easy Steps to a Better Use Case"

Once you have everyone in the room, and they all understand why they're there and what they must achieve during the review session, then the following steps can be performed. As you'll see shortly, these steps form the bulk of the review session:

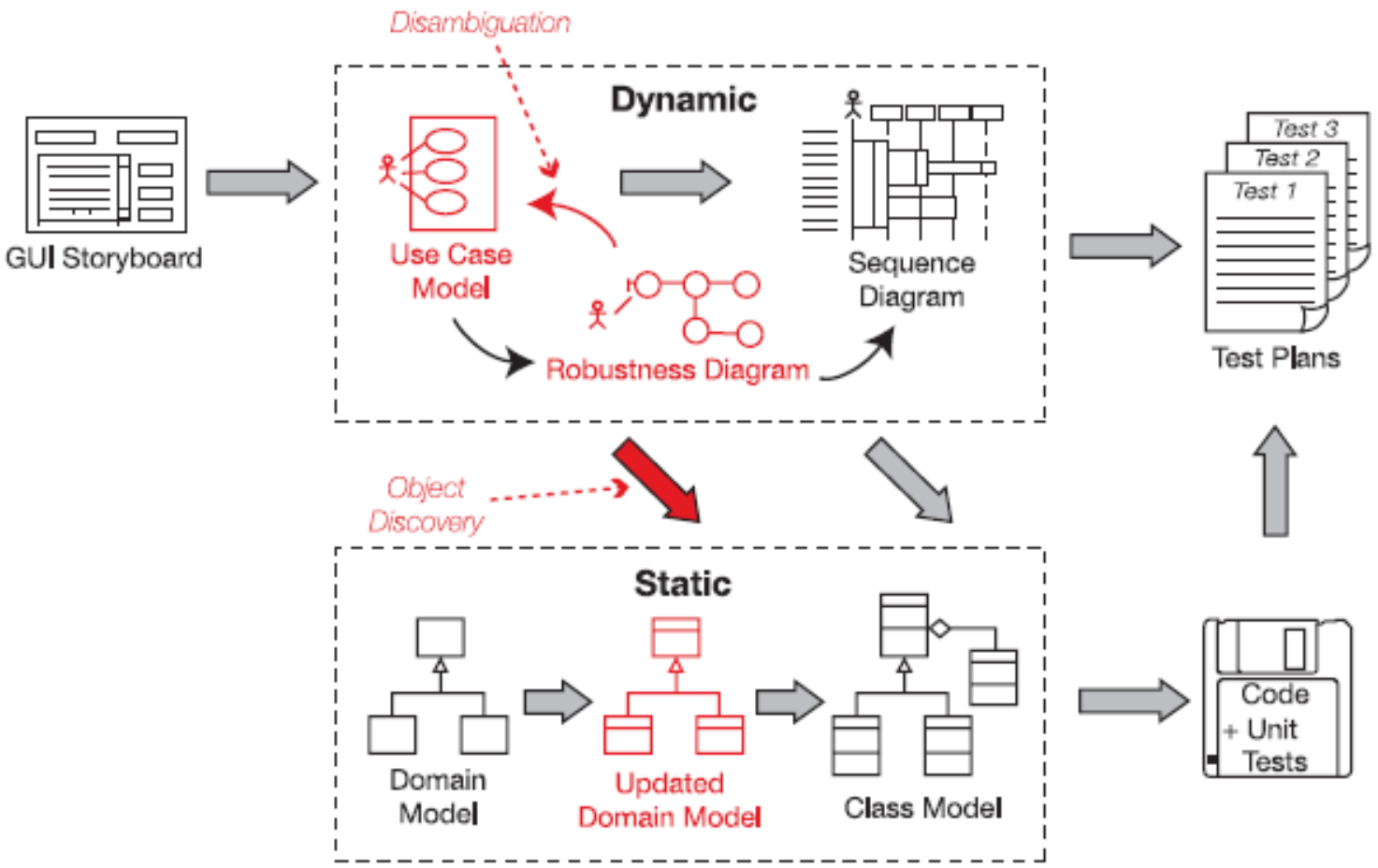
1. Remove everything that's out of scope.
2. Change passive voice to active voice.
3. Check that your use case text isn't too abstract.
4. Accurately reflect the GUI.
5. Name participating domain objects.
6. Make sure you have all the alternate courses.
7. Trace each requirement to its use cases.
8. Make each use case describe what the users are trying to do.

These eight steps will help you transmogrify your use case from a vague and ambiguous piece of mush to a razor-sharp, concise, and precise behavior specification. In the "Requirements Review in Practice" section, we walk through an example requirements review that illustrates most of these checks.

Analysis/preliminary design



Robustness Analysis



Where Does Robustness Analysis Fit into the Process?

Looking at Figure 5-1, robustness analysis sort of takes place in the murky middle ground between analysis and design. If you think of analysis (i.e., the use cases) as the "what" and design as the "how," then robustness analysis is really preliminary design. During this phase, you start making some preliminary assumptions about your design, and you start to think about the technical architecture and to think through the various possible design strategies. So it's part analysis and part design. It's also an important technique to remove ambiguity from (disambiguate) your use case text.

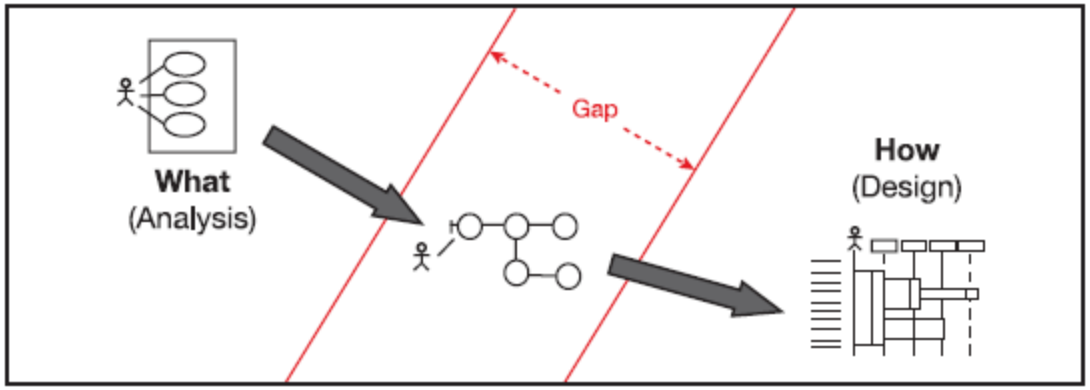


Figure 5-1. *Bridging the gap between "what" and "how"*

Like Learning to Ride a Bicycle

Learning this technique has a bit in common with learning to ride a bicycle. Until you "get it," robustness analysis can seem devilishly difficult, but once you do get it, it's really very simple. To jump-start your understanding, we'll walk through plenty of examples in this chapter. Experience has shown us that you usually need to draw six or so robustness diagrams before the penny drops and you suddenly get it. Just remember, **a robustness diagram is an object picture of a use case**.

Once you get the hang of it, you should be able to rattle off a robustness diagram in about ten minutes (or less) for each use case. Actually, as you'll see, **the trick is in writing your use case correctly**. If a robustness diagram takes more than ten minutes to draw, you can bet you're spending most of that time rewriting your use case text.

Anatomy of a Robustness Diagram

A robustness diagram is somewhat of a hybrid between a class diagram and an activity diagram. It's a pictorial representation of the behavior described by a use case, showing both participating classes and software behavior, although it intentionally avoids showing which class is responsible for which bits of behavior. Each class is represented by a graphical stereotype icon (see Figure 5-2). However, a robustness diagram reads more like an **activity diagram** (or a flowchart), in the sense that one object "talks to" the next object. This flow of action is represented by a line between the two objects that are talking to each other.

There's a direct 1:1 correlation between the flow of action in the robustness diagram and the steps described in the use case text.

The three class stereotypes shown in Figure 5-2 are as follows:

- **Boundary objects:** The "interface" between the system and the outside world (think back to Figure 3-2). Boundary objects are typically screens or web pages (i.e., the presentation layer that the actor interacts with).
- **Entity objects:** Classes from the domain model.
- **Controllers:** The "glue" between the boundary and entity objects.

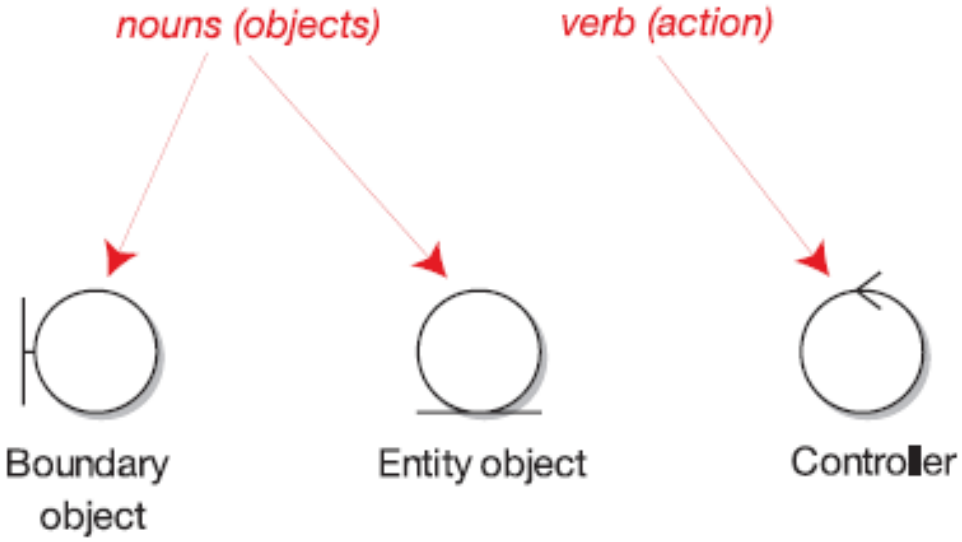


Figure 5-2. *Robustness diagram symbols*

1. **Boundary objects**, which actors use in communicating with the system
2. **Entity objects**, which are usually objects from the domain model (the subject of Chapter 2)
3. **Control objects** (which we usually call controllers because they often aren't real objects), which serve as the "glue" between boundary objects and entity objects

Figure 5-1. Robustness Diagram Symbols

It’s useful to think of boundary objects and entity objects as being nouns, and controllers as being verbs. Keep the following rules in mind when drawing your robustness diagrams:

- Nouns can talk to verbs (and vice versa).
- Nouns can’t talk to other nouns.
- Verbs can talk to other verbs.

We’ll revisit these rules later in this chapter (see Figures 5-8 and 5-9).

Four basic rules apply:

1. Actors can only talk to boundary objects.
2. Boundary objects can only talk to controllers and actors.
3. Entity objects can only talk to controllers.
4. Controllers can talk to boundary objects, entity objects, and other controllers, but not to actors.

Figure 5-5. Robustness Diagram Rules

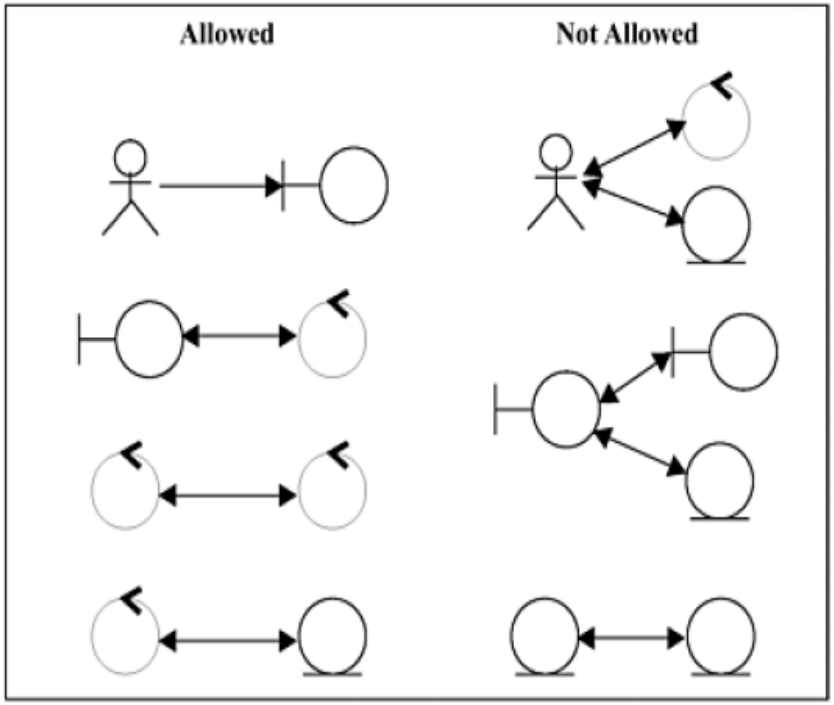
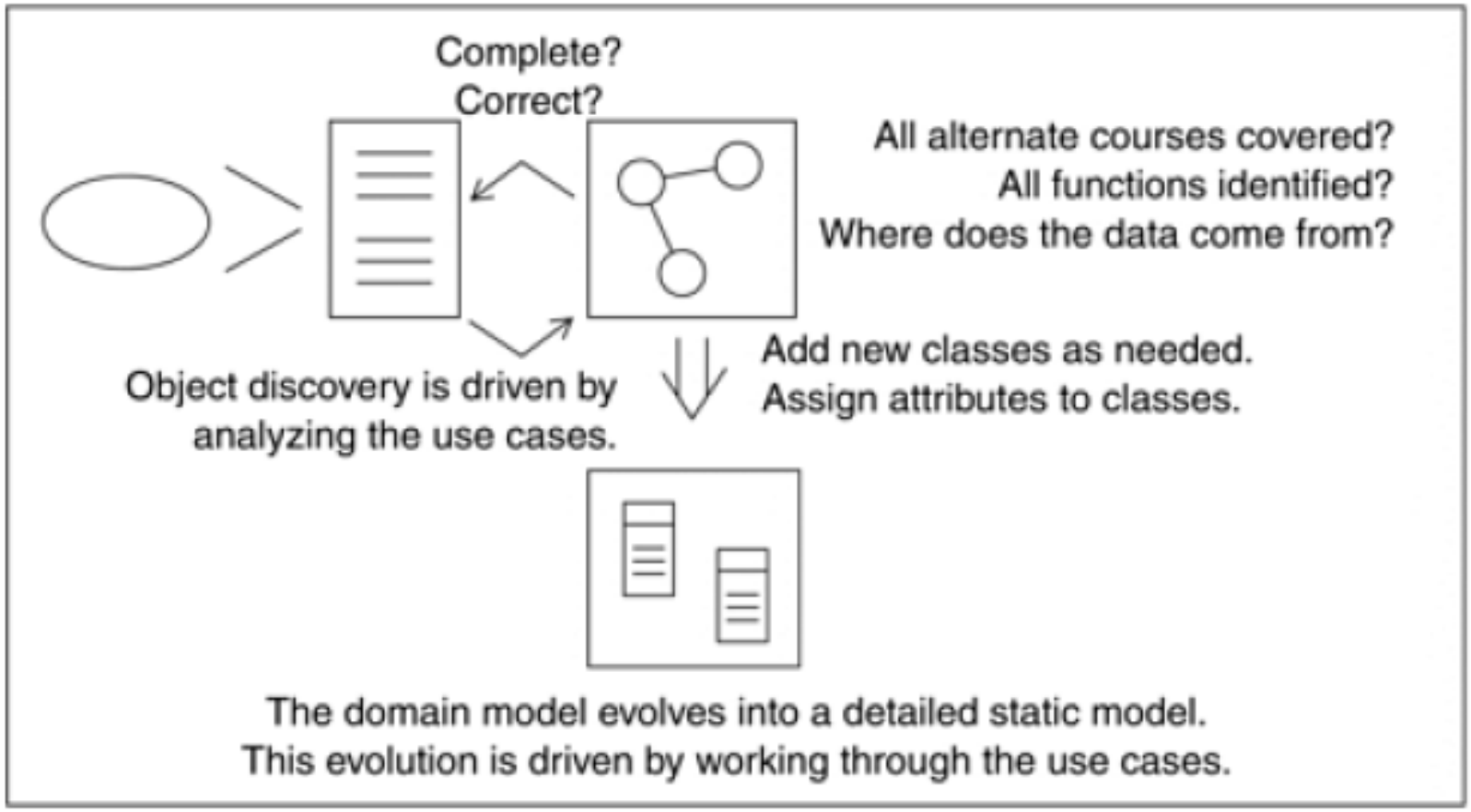


Figure 5-4. Robustness Model–Static Model Feedback Loop



top 10 robustness analysis guidelines

10. Paste the use case text directly onto your robustness diagram.
9. Take your entity classes from the domain model, and add any that are missing.
8. Expect to rewrite (disambiguate) your use case while drawing the robustness diagram.
7. Make a boundary object for each screen, and name your screens unambiguously.
6. Remember that controllers are only occasionally real control objects; they are typically logical software functions.
5. Don't worry about the direction of the arrows on a robustness diagram.
4. It's OK to drag a use case onto a robustness diagram if it's invoked from the parent use case.
3. The robustness diagram represents a preliminary conceptual design of a use case, not a literal detailed design.
2. Boundary and entity classes on a robustness diagram will generally become object instances on a sequence diagram, while controllers will become messages.
1. Remember that a robustness diagram is an "object picture" of a use case, whose purpose is to force refinement of both use case text and the object model.

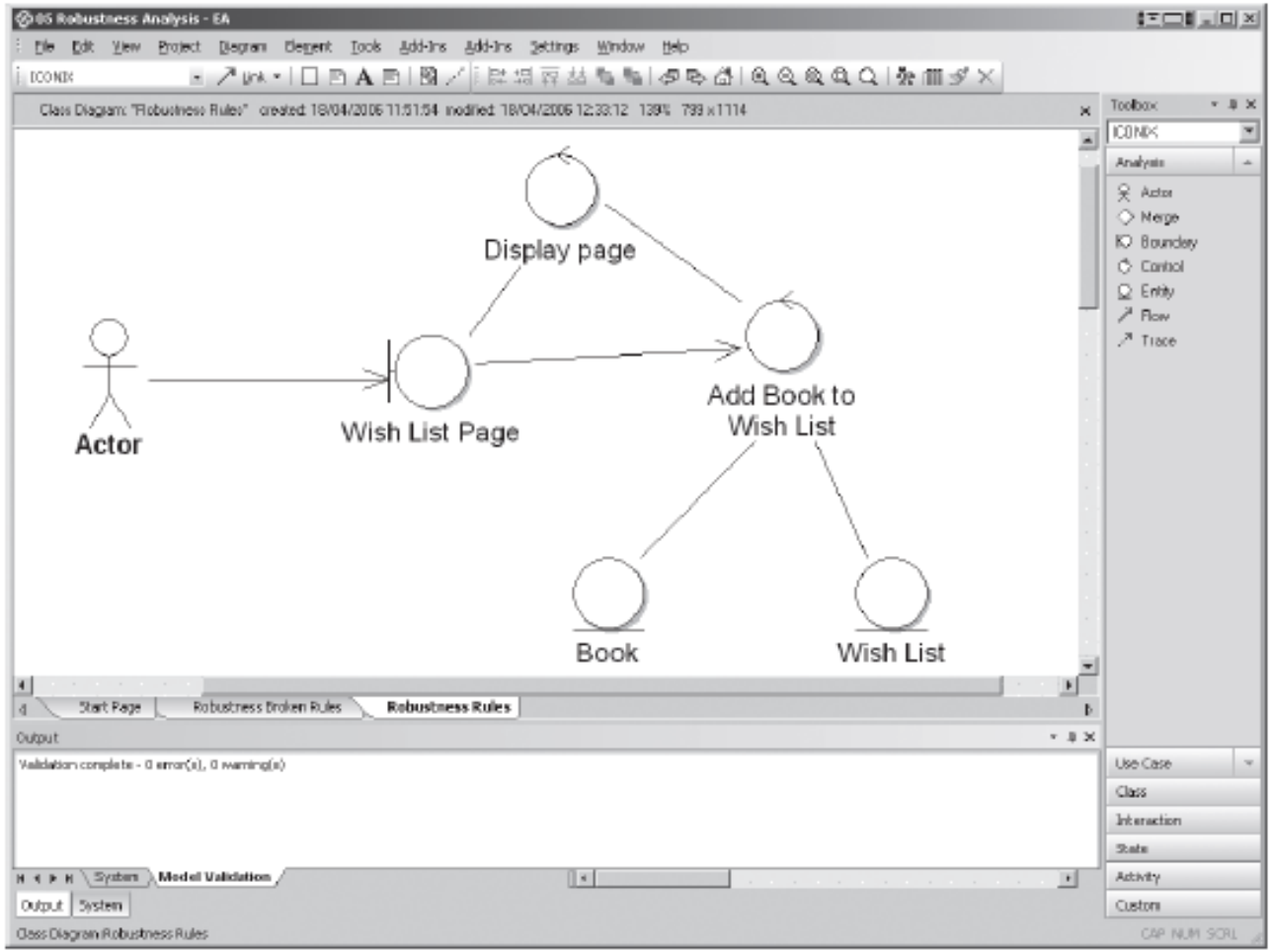


Figure 5-8. *All possible* ***valid*** *robustness diagram relationships*

The relationships shown in Figure 5-8 are allowed because

- An Actor can talk to a Boundary Object.
- Boundary Objects and Controllers can talk to each other (Noun <-> Verb).
- A Controller can talk to another Controller (Verb <-> Verb).
- Controllers and Entity Objects can talk to each other (Verb <-> Noun).

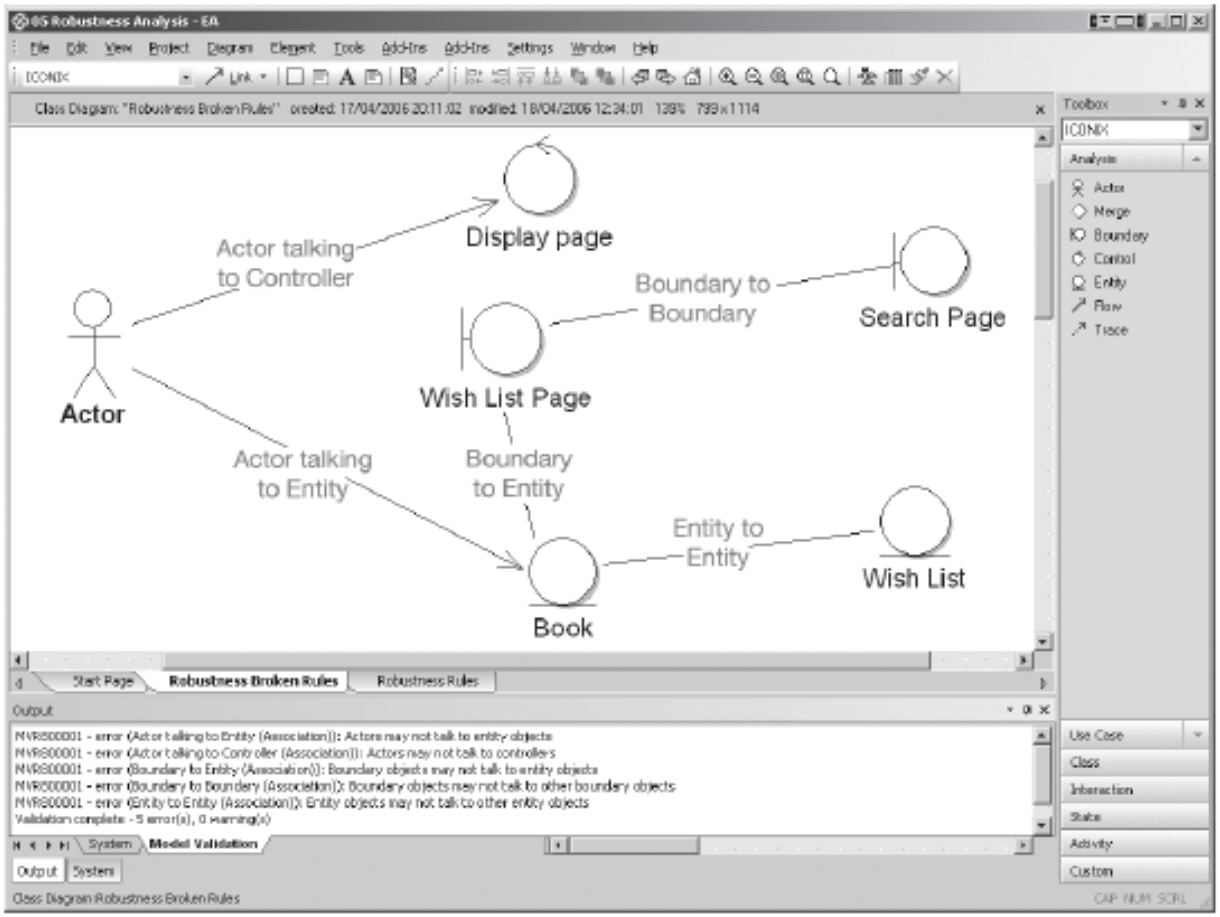


Figure 5-9. *Robustness diagram rule checker in EA showing all possible* ***invalid*** *relationships*

The relationships shown in Figure 5-9 are **not allowed** because

- An Actor can't talk directly to a Controller or an Entity (must talk to a Boundary Object).
- Boundary Objects and Entity Objects can't talk directly to each other (must go via a Controller).
- Entities can't talk directly to other Entities (must go via a Controller).
- Boundary Objects can't talk directly to other Boundary Objects (must go via a Controller).

The Top 10 Robustness Analysis Errors

10. Violate one or more of the noun/verb robustness diagram rules.

9. Don't use robustness analysis to help you use a consistent format for your use case text.

8. Don't include alternate courses on robustness diagrams.

7. Don't use robustness analysis to ensure consistency between class names on class diagrams and in use case text.

6. Allocate behavior to classes on your robustness diagrams.

5. Include too few or two many controllers.

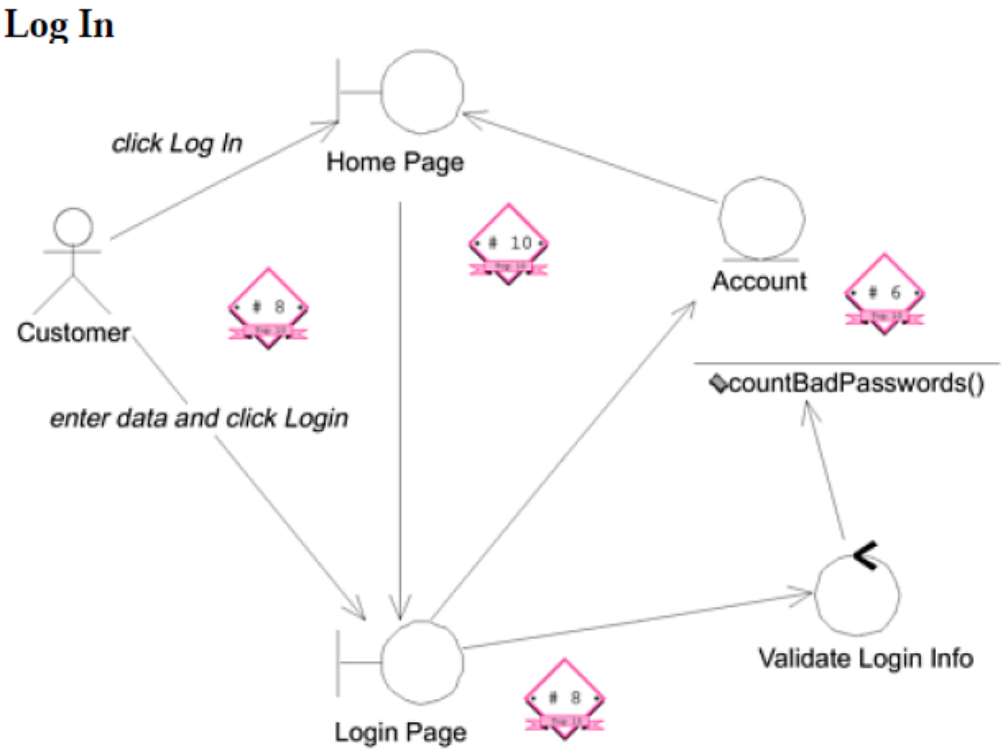
4. Take too much time trying to perfect robustness diagrams.

3. Try to do detailed design on robustness diagrams.

2. Don't perform a visual trace between the use case text and the robustness diagram.

1. Don't update your static model.

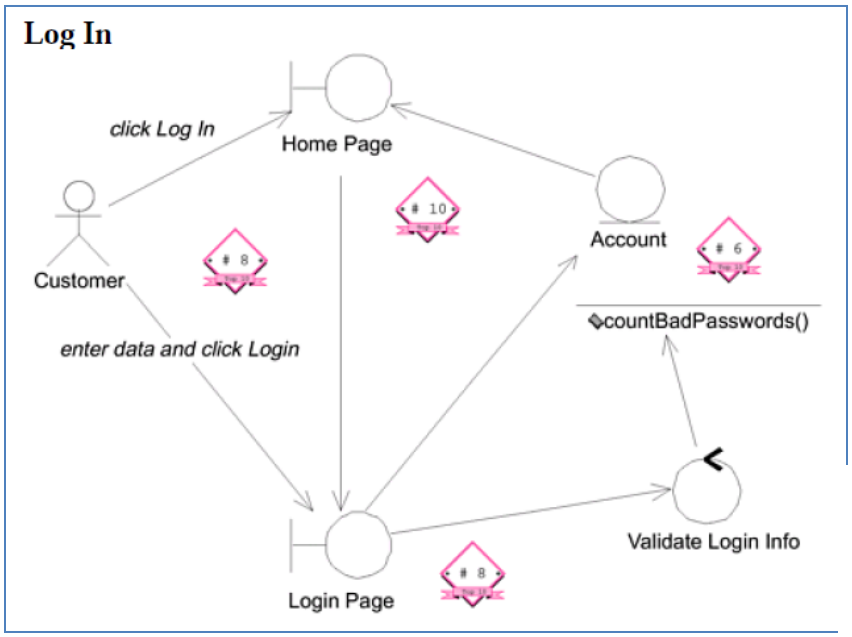
- The HomePage boundary object talked to the Login Page boundary object and the Account entity object.
- The Account object had a method assigned to it.
- No alternate courses were represented.

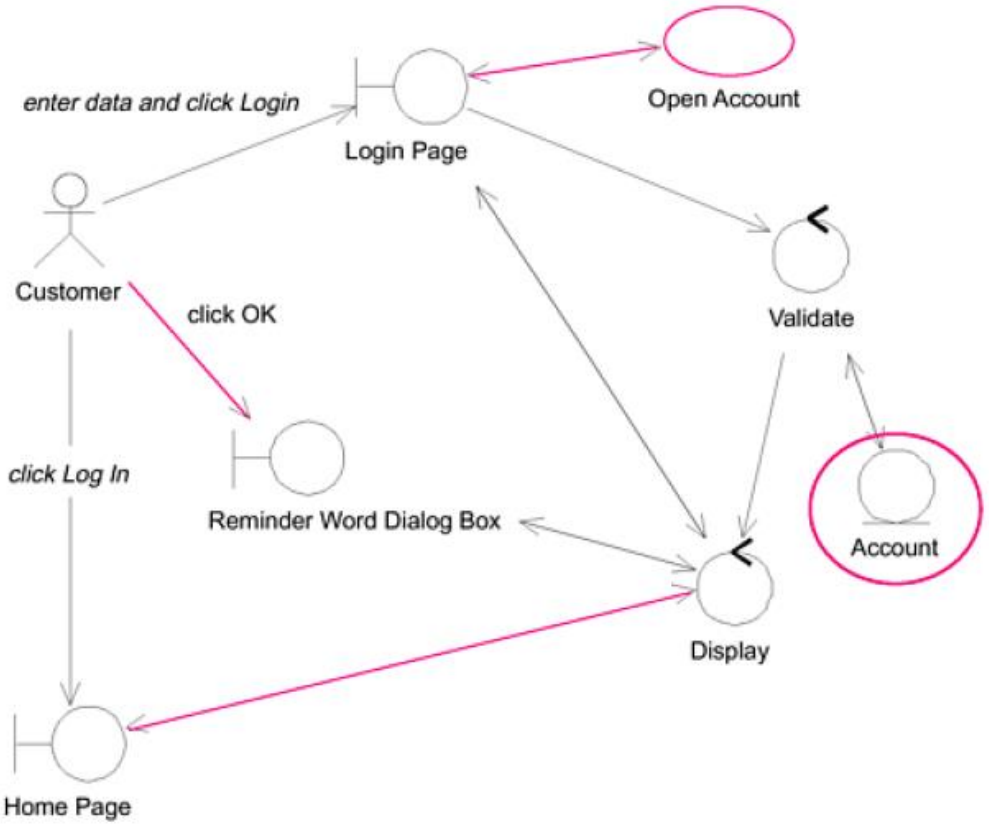


Basic Course: The Customer clicks the Log In button on the Home Page. The system displays the Login Page. The Customer enters his or her user ID and password and then clicks the Log In button. The system validates the login information against the persistent Account data and then returns the Customer to the Home Page.

Alternate Courses:

If the Customer clicks the New Account button on the Login Page, the system invokes the Open Account use case. If the Customer clicks the Reminder Word button on the Login Page, the system displays the reminder word stored for that Customer, in a separate dialog box. When the Customer clicks the OK button, the system returns the Customer to the Login Page.


Log In
click Log In
Home Page
Customer
8
10
Account
# 6
countBadPasswords()
enter data and click Login
Validate Login Info
Login Page
8


enter data and click Login
Login Page
Open Account
Customer
click OK
Validate
click Log In
Reminder Word Dialog Box
Account
Display
Home Page

BASIC COURSE:
The user clicks the login link from any of a number of pages; the system displays the login page. The user enters their username and password and clicks Submit. The system checks the master account list to see if the user account exists. If it exists, the system then checks the password. The system retrieves the account information, starts an authenticated session, and redisplays the previous page with a welcome message.

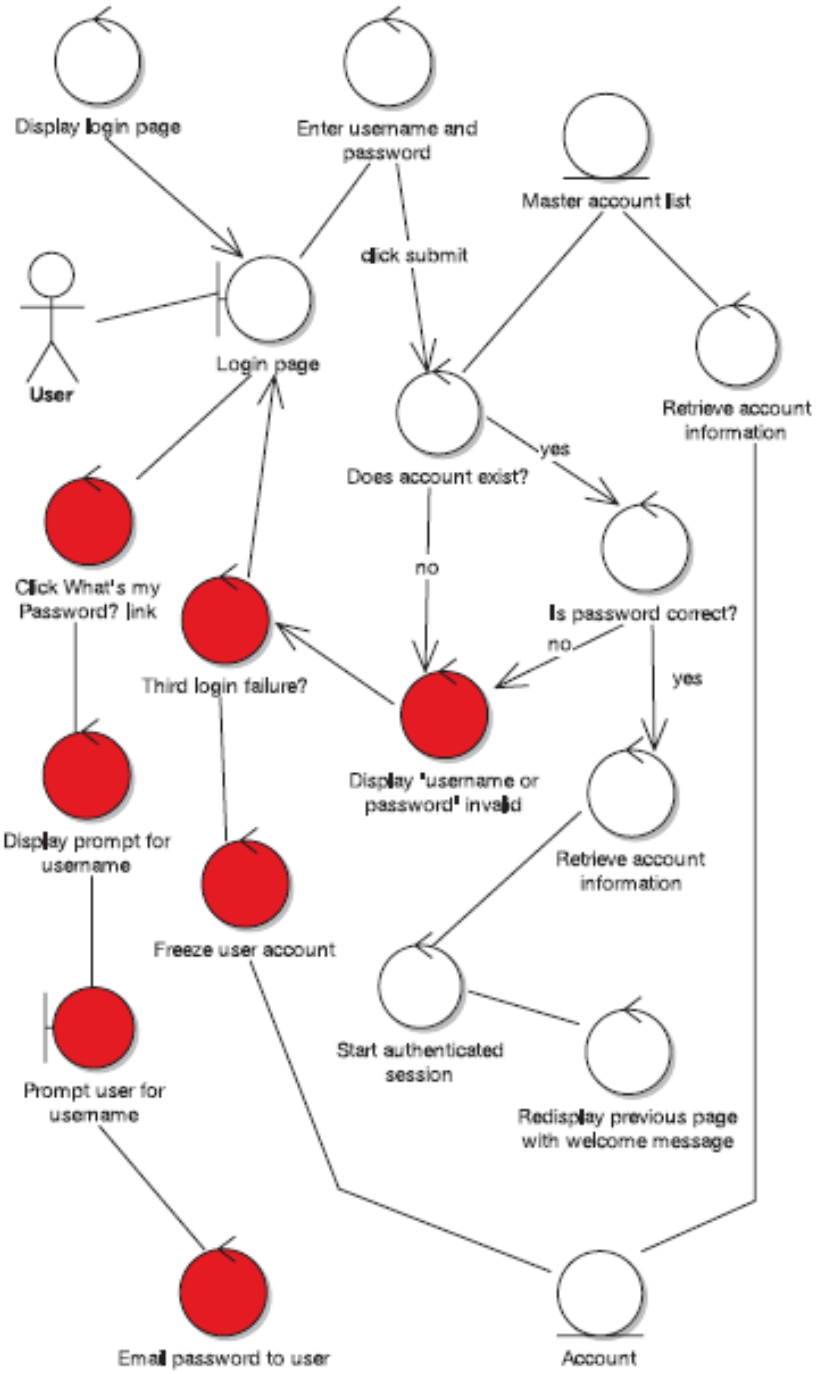
ALTERNATE COURSES:
User forgot the password: The user clicks the What's my Password? link. The system prompts the user for their username if not already entered, retrieves the account info, and emails the user their password.

Invalid account: The system displays a message saying that the "username or password" was invalid, and prompts them to reenter it.

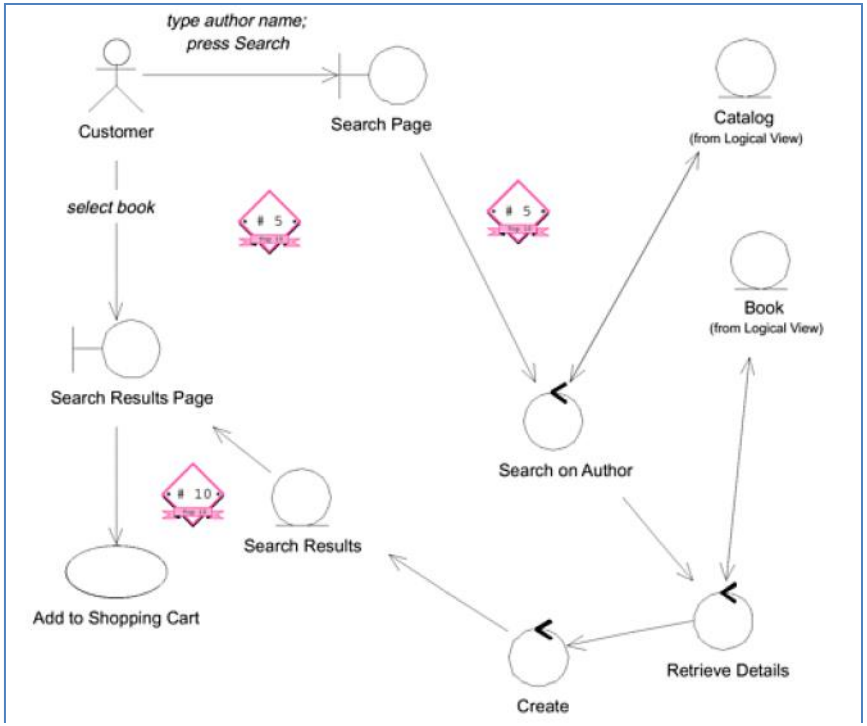
Invalid password: The system displays a message that the "username or password" was invalid, and prompts them to reenter it.

User cancels login: The system redisplays the previous page.

Third login failure: The system locks the user's account, so the user must contact Customer Support to reactivate it.

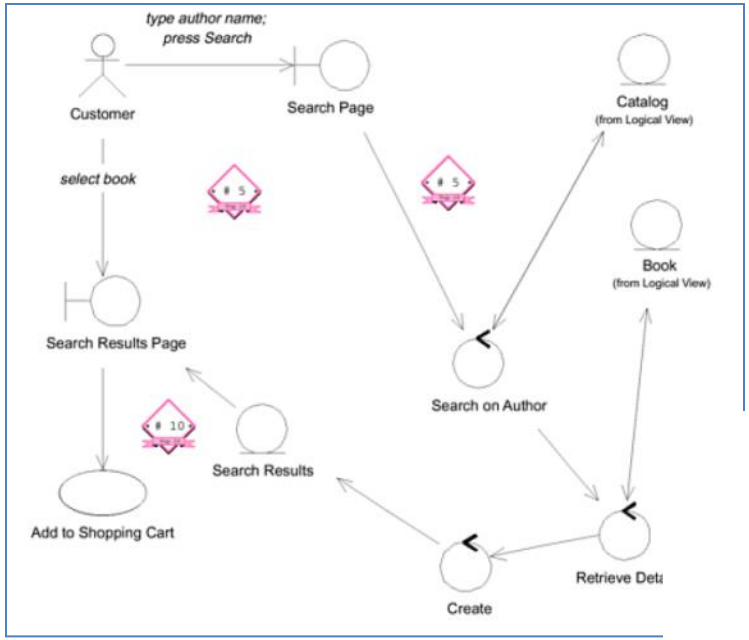


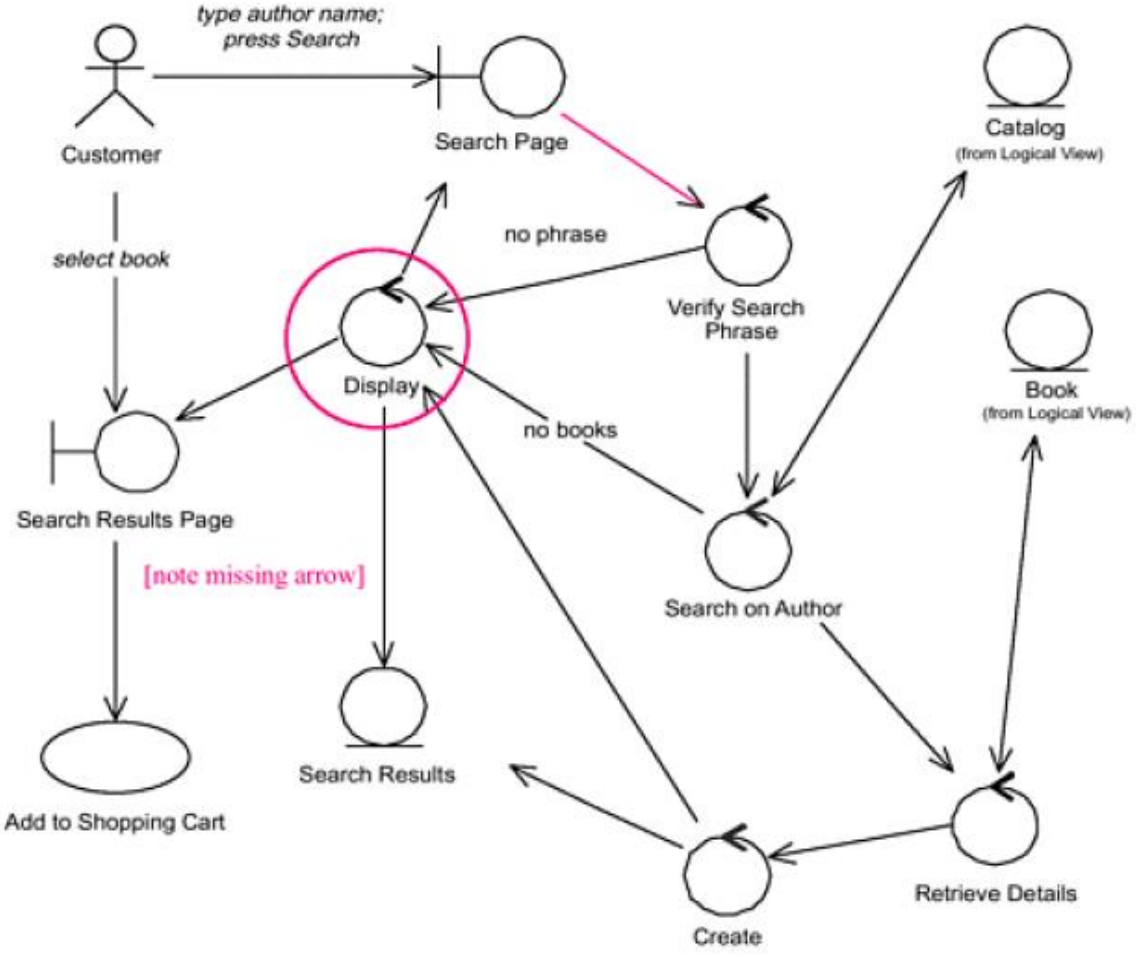
- There are too few controllers. Verify Search Phrase enables the system to avoid performing a search with no search phrase, while Display is a standard controller associated with Web pages. (These controllers also reflect alternate courses that the previous diagram didn't.)
- The Search Results entity object is talking to the Search Results Page boundary object.
- The use case text doesn't reflect the creation of the Search Results object.



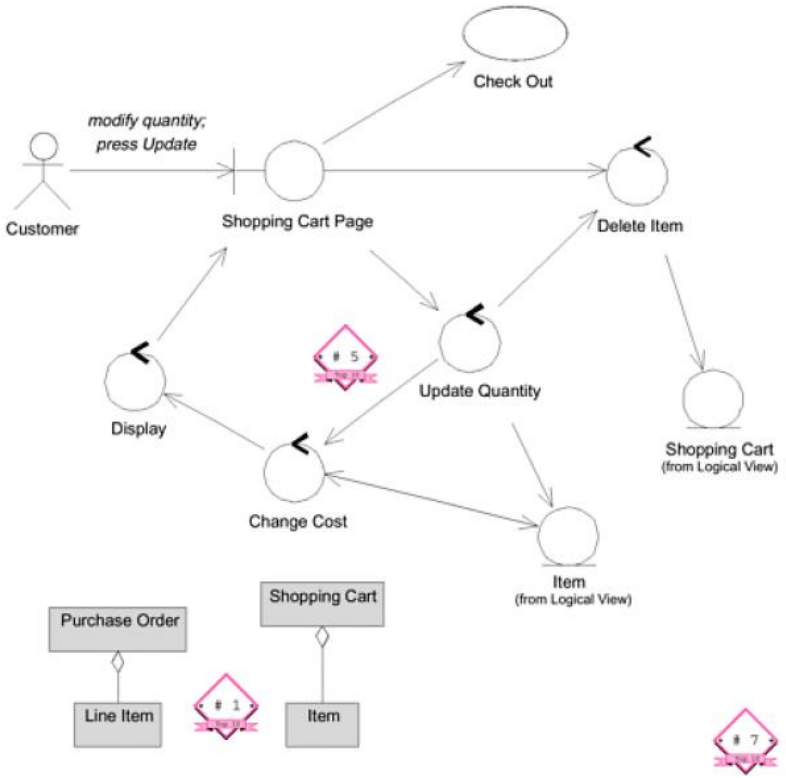
Basic Course: The Customer types the name of an Author on the Search Page and then presses the Search button. The system ensures that the Customer typed a valid search phrase, Author and then searches the Catalog and retrieves all of the Books with which that is associated. The the system retrieves the important details about each Book.

Then the system displays the list of Books on the Search Results Page, with the Books listed in reverse chronological order by publication date. Each entry has a thumbnail of the Book's cover, the Book's title and authors, the average Rating, and an Add to Shopping Cart button. The Customer presses the Add to Shopping Cart button for a particular Book. The system passes control to the Add Item to Shopping Cart use case.


type author name;
press Search
Customer
Search Page
Catalog
(from Logical View)
select book
Book
(from Logical View)
Search Results Page
Search on Author
Search Results
Add to Shopping Cart
Create

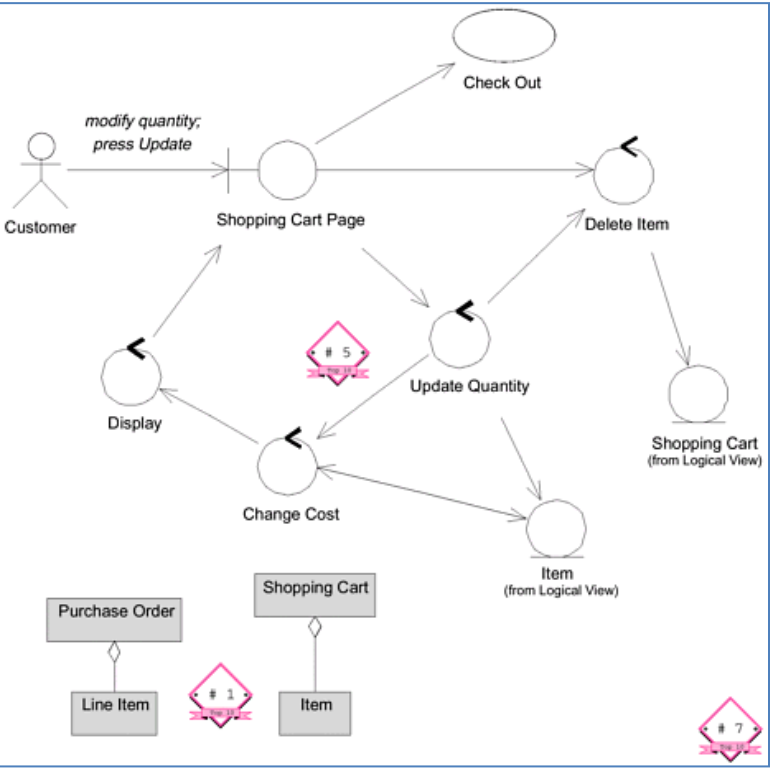

type author name;
press Search
Customer
Search Page
Catalog
(from Logical View)
select book
no phrase
Verify Search
Phrase
Display
Book
(from Logical View)
no books
Search Results Page
[note missing arrow]
Search on Author
Search Results
Add to Shopping Cart
Retrieve Details
Create

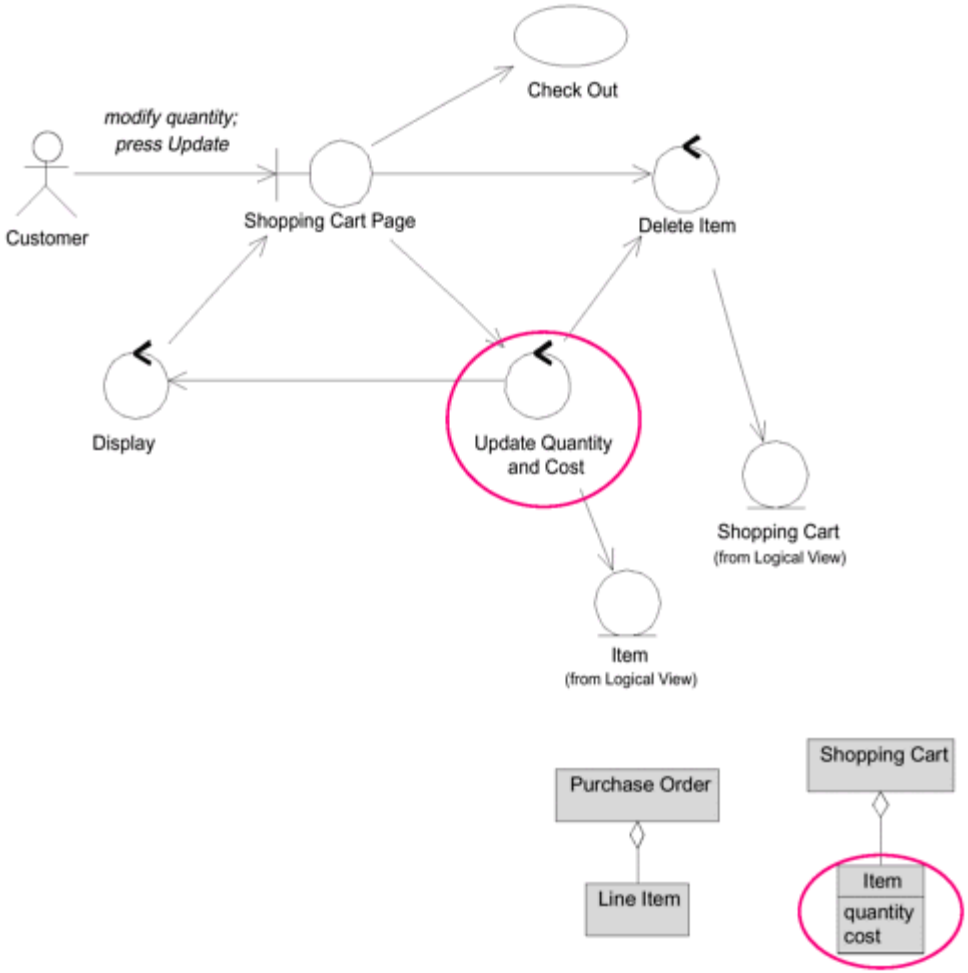
•· The Change Cost controller is unnecessary, since both it and the Update Quantity controller operate on the Item object.
•· The use case text refers to Line Item, but it's clear from the class diagram excerpt and the robustness diagram that the text should refer to Item instead. (This kind of name usage inconsistency can be deadly.)
•· The class diagram excerpt doesn't reflect the attributes that are mentioned in the use case text.



Basic Course: On the Shopping Cart Page, the Customer modifies the quantity of a Line Item in the Shopping Cart and then presses the Update button. The system stores the new quantity and then computes and displays the new cost for that Line Item. The Customer Presses the Continue Shopping button. The system returns control to the use case from which it received control.

Alternate Courses: (1) If the Customer changes the quantity of the Item to 0, the system deletes that Item from the Shopping Cart. (2) If the Customer presses the Delete button instead of the Update button, the system deletes that Item from the Shopping Cart. (3) If the Customer presses the Check Out button instead of the Continue Shopping button, the system passes control to the Check Out use case.


Check Out
modify quantity;
press Update
Customer
Shopping Cart Page
Delete Item
Display
5
Update Quantity
Shopping Cart
(from Logical View)
Change Cost
Item
(from Logical View)
Purchase Order
Shopping Cart
Line Item
1
Item
7


Check Out
modify quantity;
press Update
Customer
Shopping Cart Page
Delete Item
Display
Update Quantity
and Cost
Shopping Cart
(from Logical View)
Item
(from Logical View)
Purchase Order
Shopping Cart
Line Item
Item
quantity
cost

Robustness Diagram for the “Show Book Details” Use Case (hal. 115)

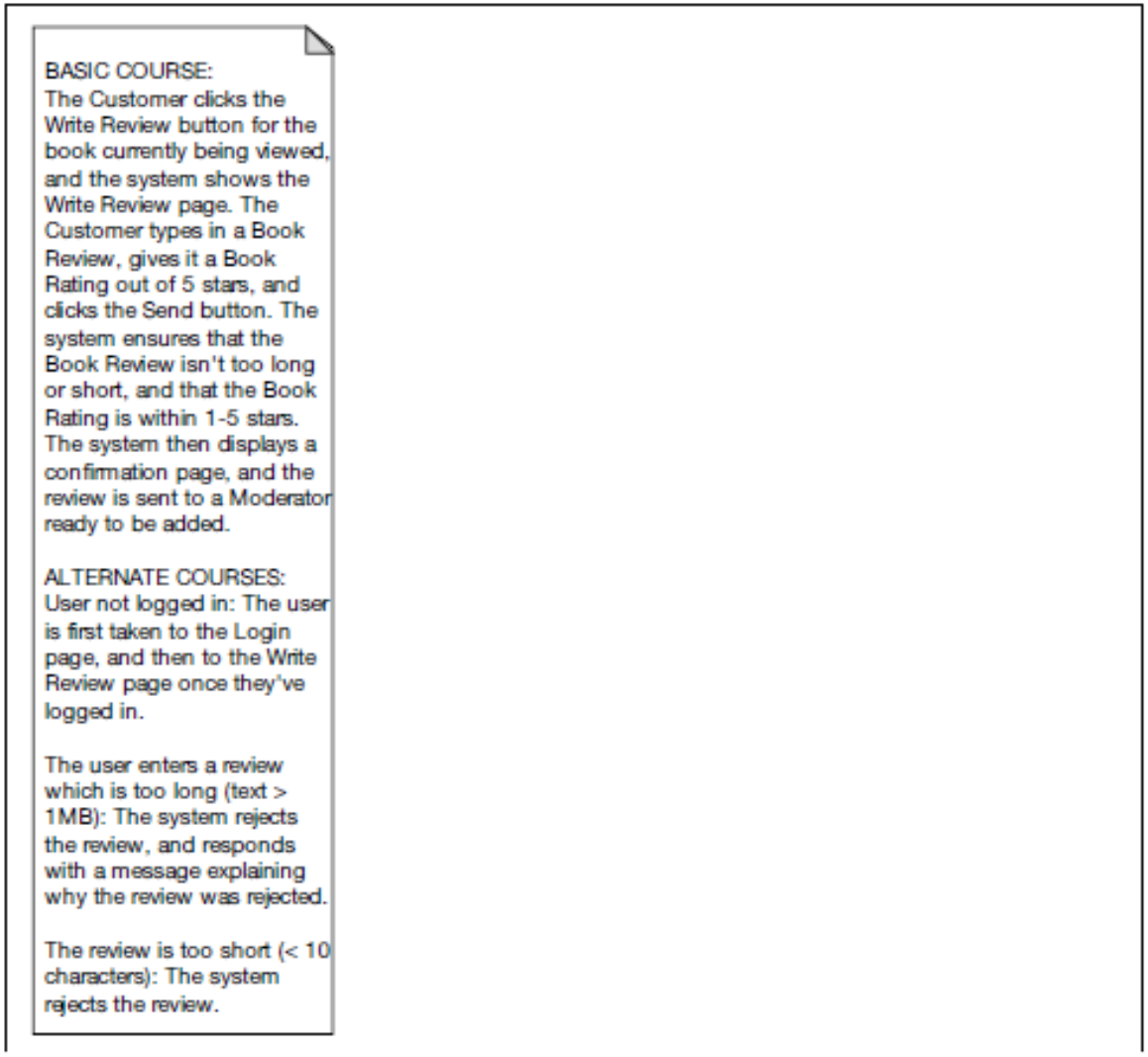


Figure 5-12. *Step 1: Create a new, blank* Write Customer Review *robustness diagram*

BASIC COURSE:
The Customer clicks the Write Review button for the book currently being viewed, and the system shows the Write Review page. The Customer types in a Book Review, gives it a Book Rating out of 5 stars, and clicks the Send button. The system ensures that the Book Review isn't too long or short, and that the Book Rating is within 1-5 stars. The system then displays a confirmation page, and the review is sent to a Moderator ready to be added.

ALTERNATE COURSES:
User not logged in: The user is first taken to the Login page, and then to the Write Review page once they've logged in.

The user enters a review which is too long (text > 1MB): The system rejects the review, and responds with a message explaining why the review was rejected.

The review is too short (< 10 characters): The system rejects the review.

Figure 5-13. *Step 2: Spot the deliberate mistake.*

BASIC COURSE:
The Customer clicks the Write Review button for the book currently being viewed, and the system shows the Write Review page. The Customer types in a Book Review, gives it a Book Rating out of 5 stars, and clicks the Send button. The system ensures that the Book Review isn't too long or short, and that the Book Rating is within 1-5 stars. The system then displays a confirmation page, and the review is sent to a Moderator ready to be added.

ALTERNATE COURSES:
User not logged in: The user is first taken to the Login page, and then to the Write Review page once they've logged in.

The user enters a review which is too long (text > 1MB): The system rejects the review, and responds with a message explaining why the review was rejected.

The review is too short (< 10 characters): The system rejects the review.

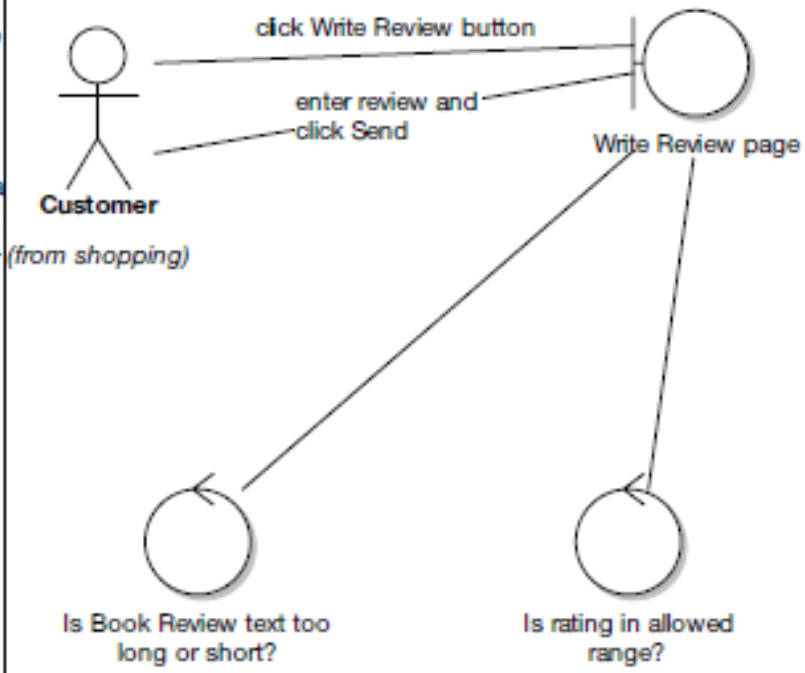


Figure 5-14. *Step 3: We've now corrected the mistake and added some validation controllers.*

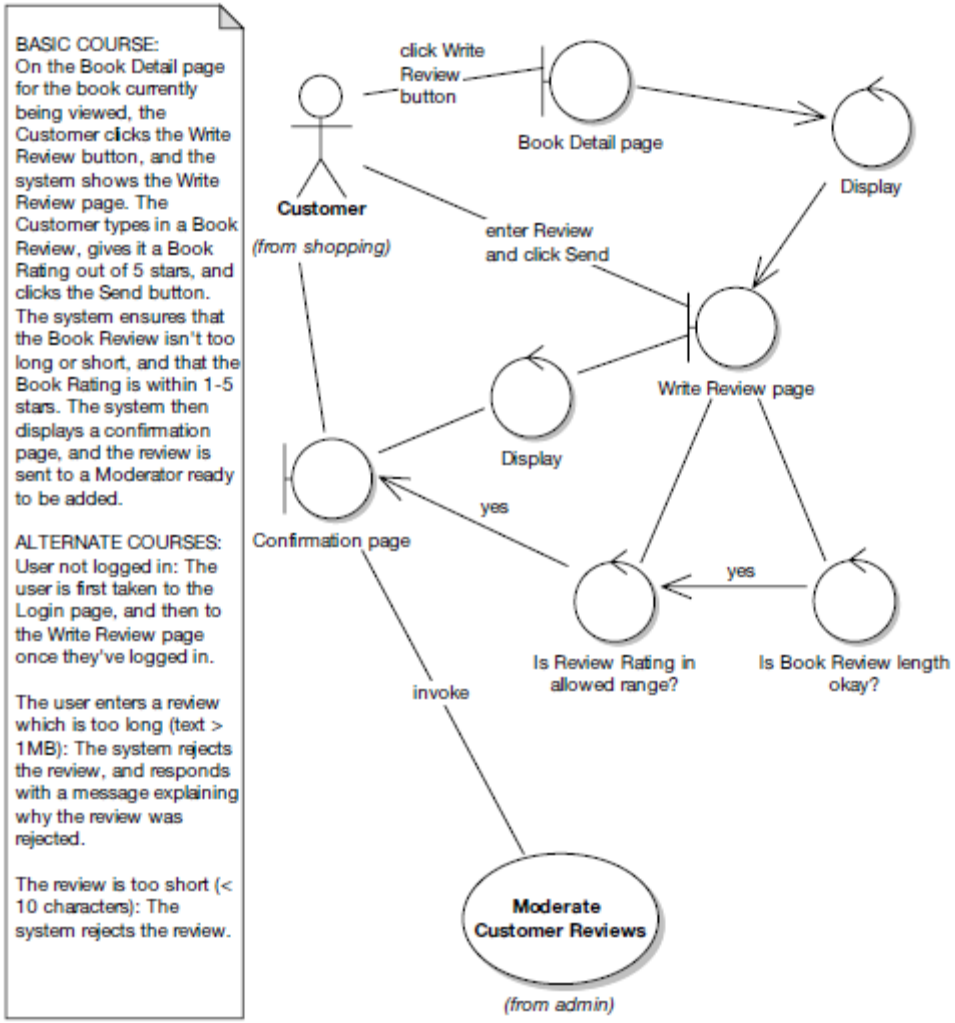


Figure 5-15. *Step 4: The remainder of the basic course text in graphic form*

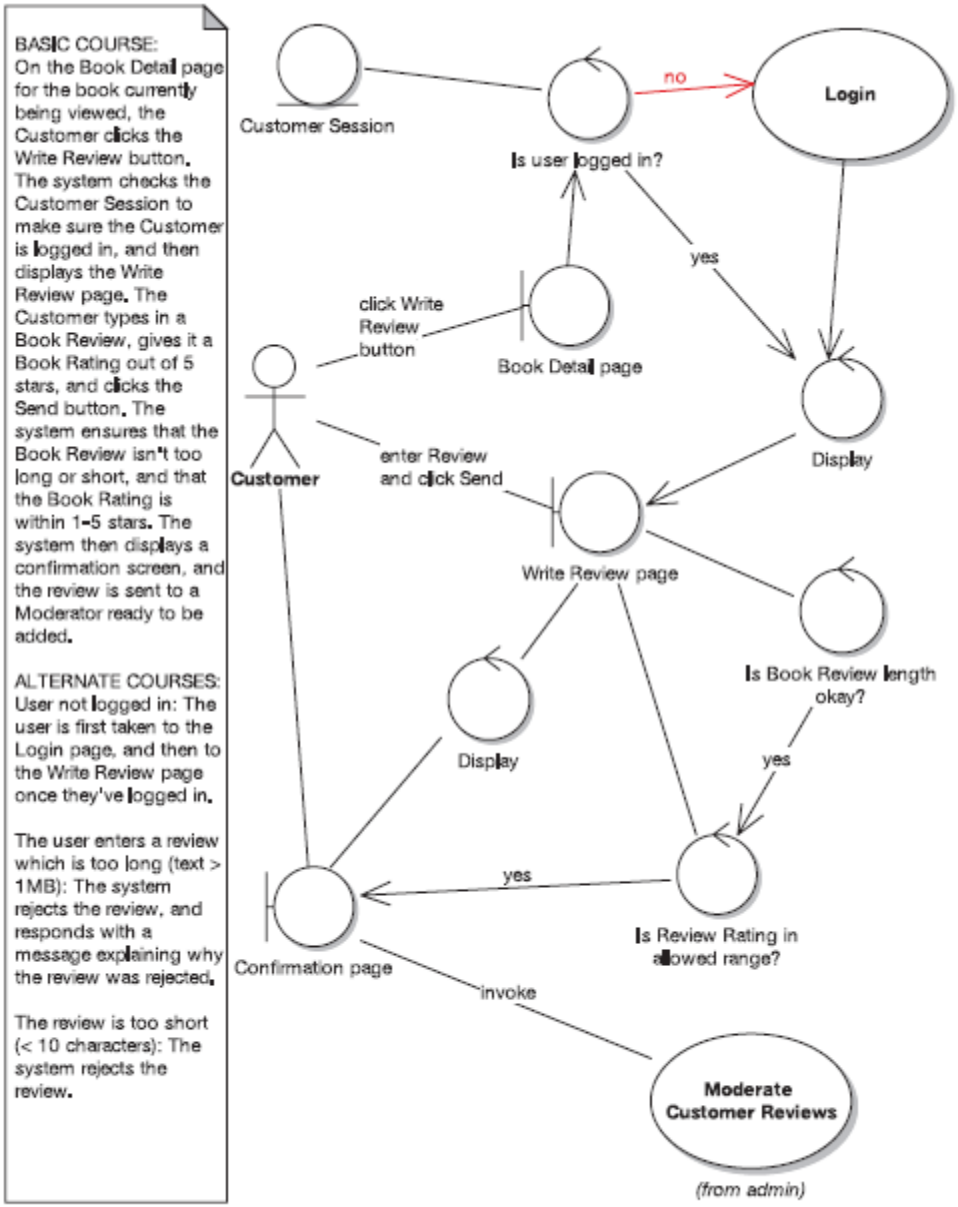


Figure 5-16. *Step 5: The first alternate course has been added.*

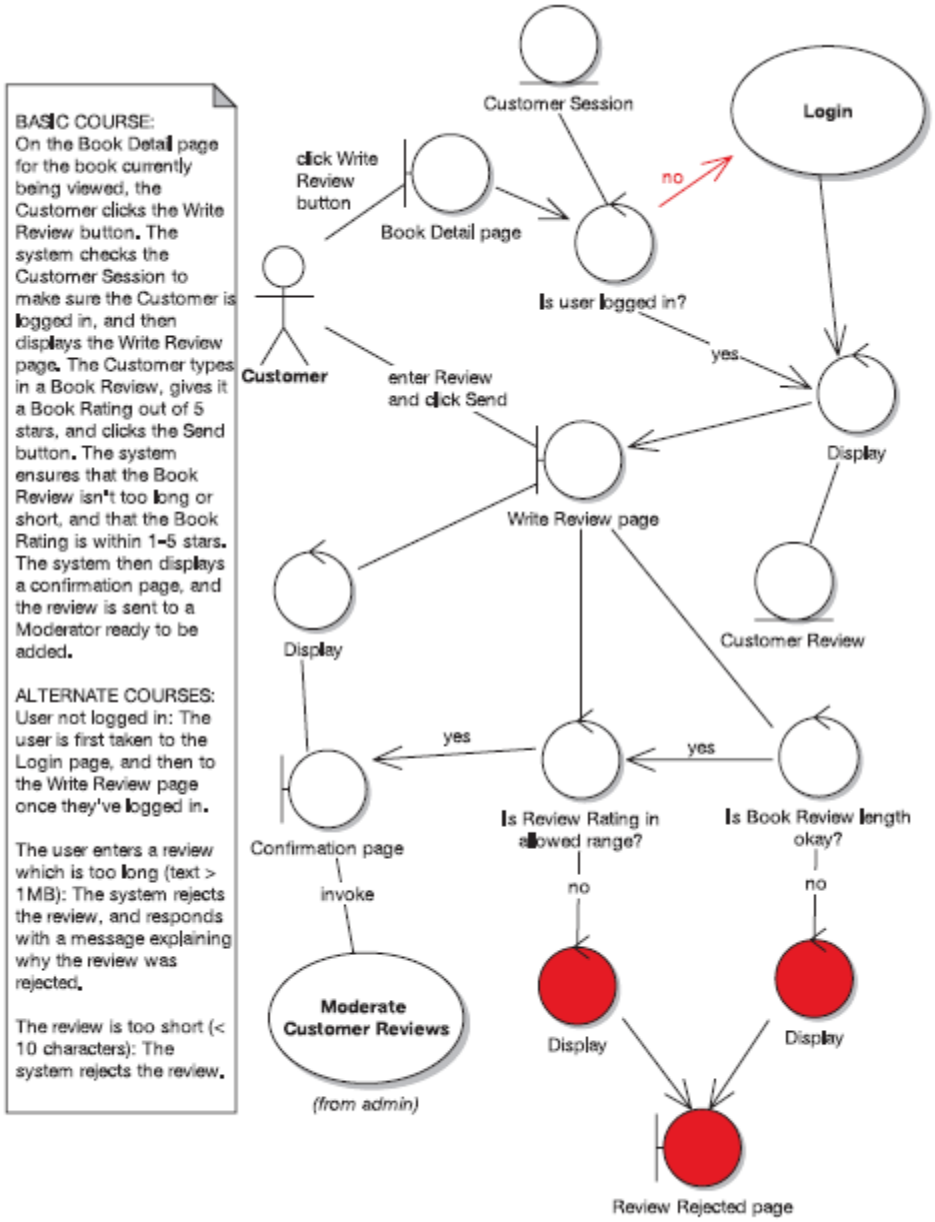


Figure 5-17. *Step 6: Robustness diagram with the last two alternate courses added*

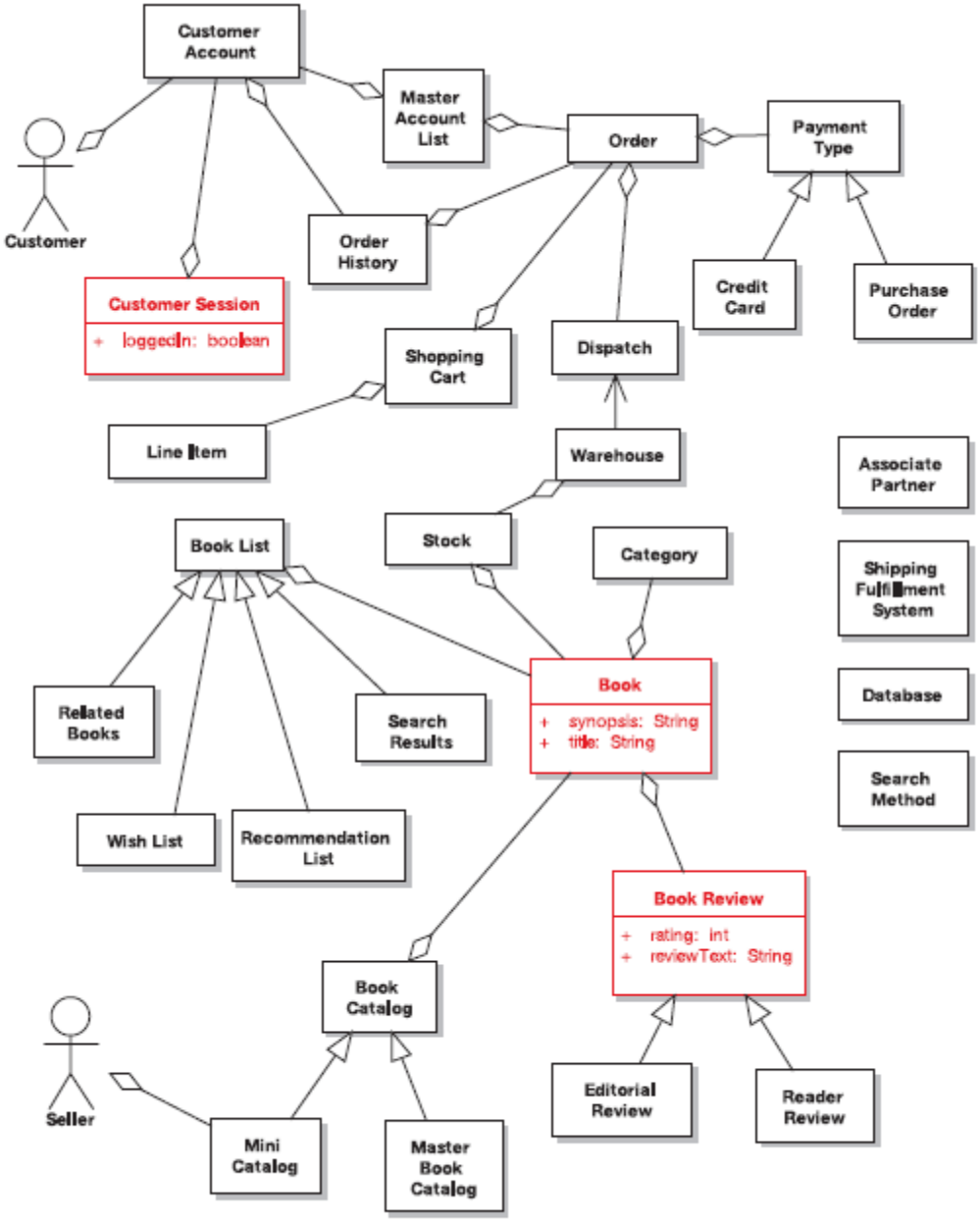


Figure 5-19. *Static model for the Internet Bookstore, after robustness analysis for two use cases*

---Batas---

Milestone 2: Preliminary Design Review

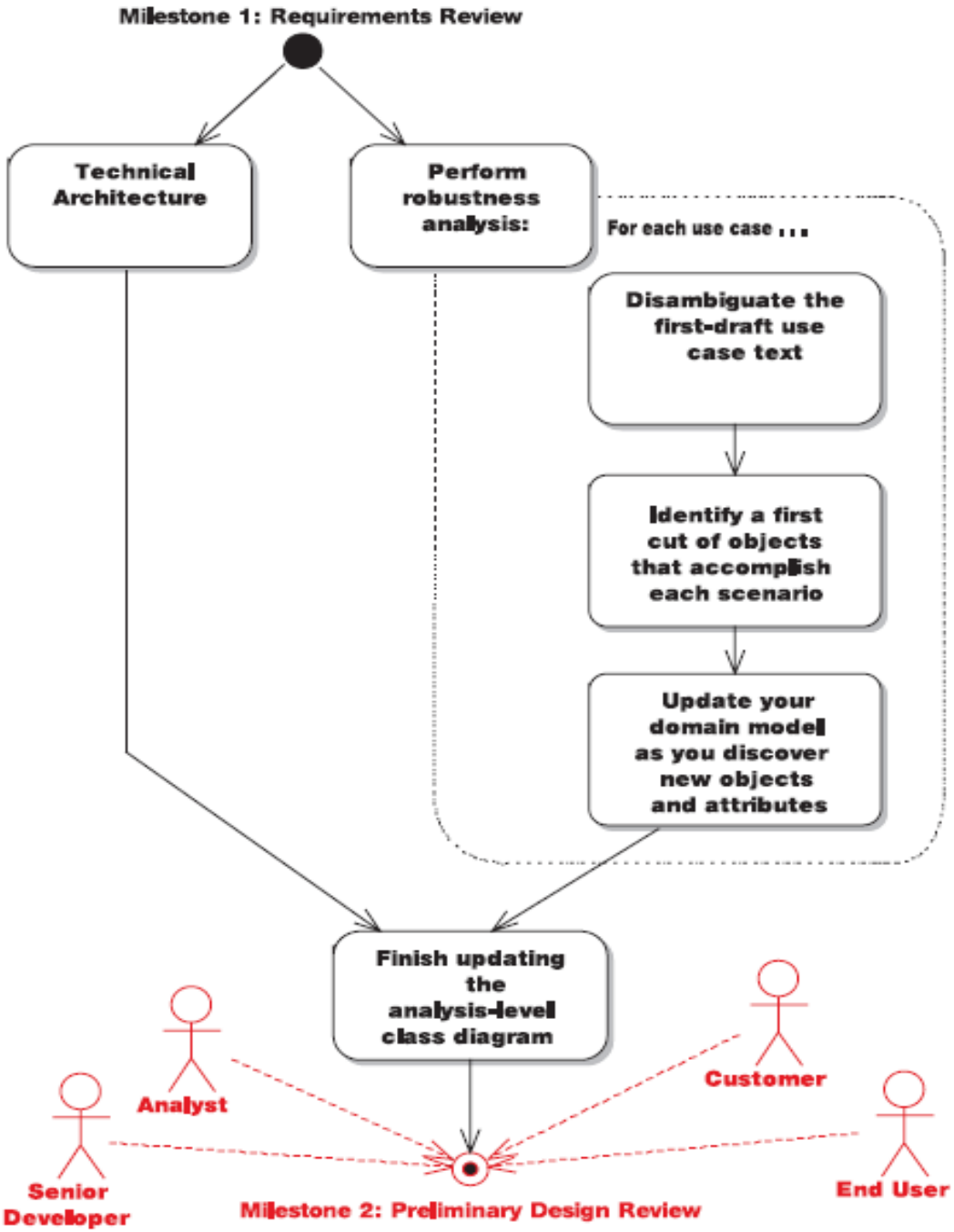
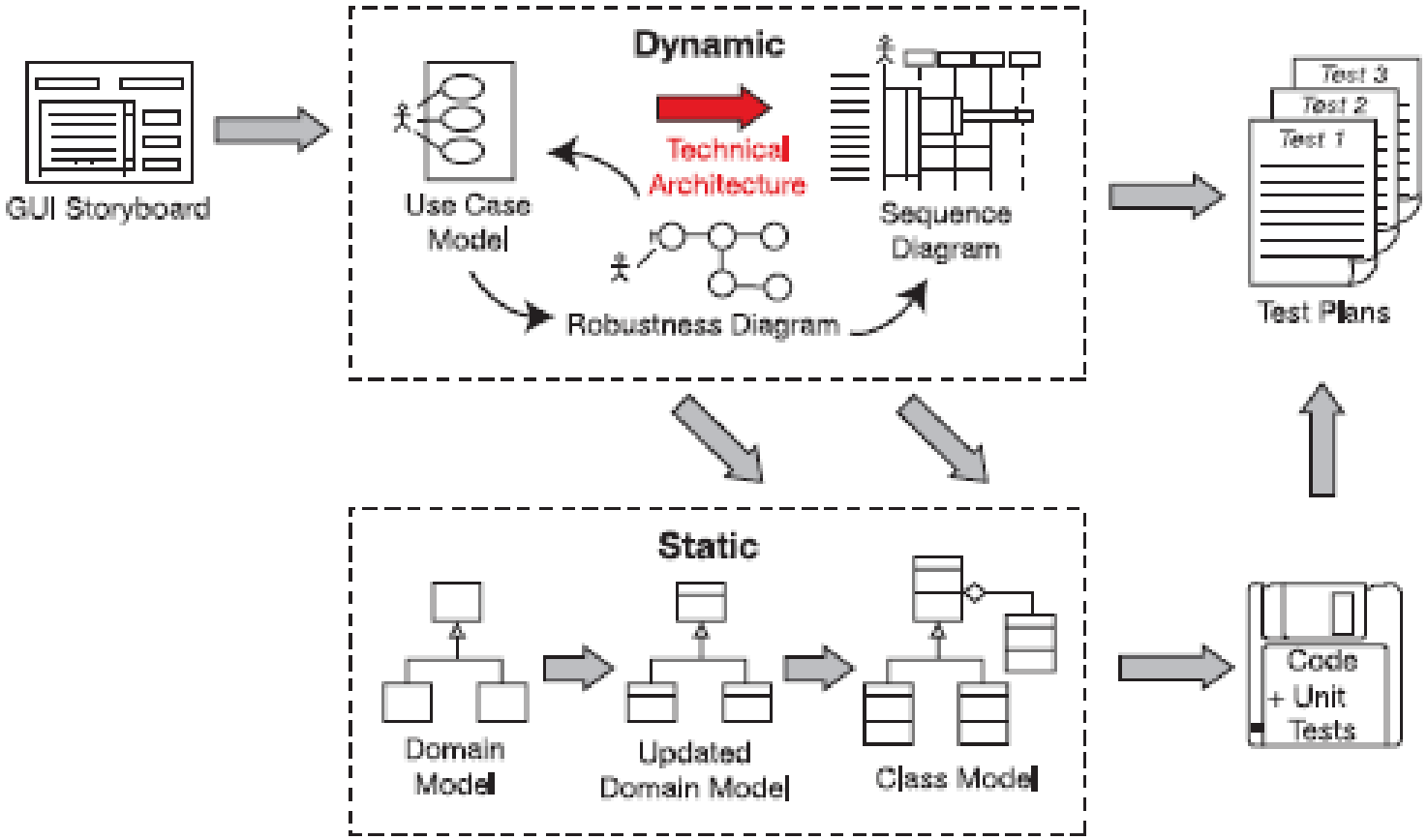


Figure 6-8. *Analysis and Preliminary Design*

Technical Architecture



What Is Technical Architecture?

Technical architecture (also referred to as *system architecture* and *software architecture*) generally describes the system you're intending to build in terms of *structure*. The architecture is built to satisfy the *business* and *service-level* requirements of the system you're going to build. The architecture includes (but isn't limited to) the system topology (the server nodes, physical location on the network, choice of application server[s], etc.).

A good TA will be based on some thorough analysis of the "numbers" involved—that is, the number of people who will be using the system at any time, whether there are peak usage hours (and what those peaks are likely to be), the number of transactions per minute, failover criteria, and so on. These numbers will play a huge role in deciding such factors as what sort of application server (or web server) should be used, how many licenses to buy, and which server- and client-side technologies the project should use. These are not decisions for the fainthearted!

Documented architectures range in depth and formality from several volumes of detailed specs (with every "i" dotted and "t" crossed) to a bunch of e-mails and Visio diagrams. The ideal level lies somewhere between the two extremes, though of course the needs will vary depending on the nature and size of the project.

What Are the Duties of a Technical Architect?

In addition to simply creating the right architecture to solve the problem posed by the requirements, the architect must also document the architecture in an expressive and unambiguous written format, and ***make sure the latest version is available to everyone on the project***. The technical architect must also truly believe in the TA he or she has created, and be prepared to evangelize it and communicate its intent to all the project stakeholders. This is an important point, because an architect who doesn't follow the courage of his or her convictions will end up with a disjointed system, where individual teams or team members head off in different directions and do slightly different things. For example, one team will use WebWork for their presentation tier, another team will use Velocity, another JSP, and so on. Nothing will quite fit together properly, and it will never quite be clear whether the requirements have been fully met. If one part of the system meets the failover requirements, but another part has been designed slightly differently, has the requirement been met?

Top 10 Technical Architecture Guidelines

10. Separate functional, data, and system architecture.

Architectures generally cover three broad areas:

- The deployment model (network and application servers, and how they fit together; system topology; web browsers supported; etc.)
- The package/component model (separation of concerns to different strata/components)
- The data model

9. Understand why you're creating an architecture.

   Before you even think about the system's architecture, it's important to understand precisely why an architecture is even needed.

8. Base the architecture objectively on the requirements.

   It's tempting to base the architecture on the latest technology or whatever happens to be the "flavor of the month," rather than listening to what the requirements are trying to tell you and making an objective decision based on what's needed. Budget considerations are also important. If you decide that, technically speaking, the best application server for the project is "BankBreaker 8.0 Service Pack 12," is the budget available to handle this? Are there cheaper (and more robust) alternatives that match the requirements just as well?

7. Consider factors such as scalability, security, and availability.

6. Consider internationalization and localization.

5. Pose hard questions to all the people involved.

   Questions regarding such issues as security, auditing, portability, and scalability need to be answered now, not six months into the project.

4. If you don't get the answers you need, ask again—and keep on asking!

3. Consider testability.

   Our co-author on *Agile Development with ICONIX Process* (Apress, 2005), Mark Collins-Cope, described a trading system he'd been working on. According to the spec, orders should time out after between 1 and 30 days. To test this, the team couldn't really wait around (a tad boring), so they built functionality to enable the UI to set the date and time, specifically for testing. This is just one example of how testing must be considered even in the early stages when you're thinking about the architecture.

2. Explore which external systems you'll need to interface with.

   Scour the requirements for anything relating to synchronous or asynchronous external system interaction. For synchronous systems, think about external system availability. Is it a requirement that your system be able to operate without the other system?

1. Have the courage to believe that your architecture is right and the strength to push its adoption throughout the project.

   Building software is a complex process. It's all too easy to end up with a big plate of spaghetti, tangled and amorphous, rather than an elegant haute cuisine plate of pristine perfection (see the next section on layering for the "lasagna model"). You would think that those involved would pull together to achieve the latter instead of the former. But people are people, and if they're simply left to it, each individual or group will form its own "micro-project" with its own direction, set of goals, and set of standards—even its own "mini-architecture." The technical architect (actually, the chief architect) must communicate his or her documented architecture and make sure everyone understands it, not just at the beginning of the project but throughout.

Milestone 2: Preliminary Design Review

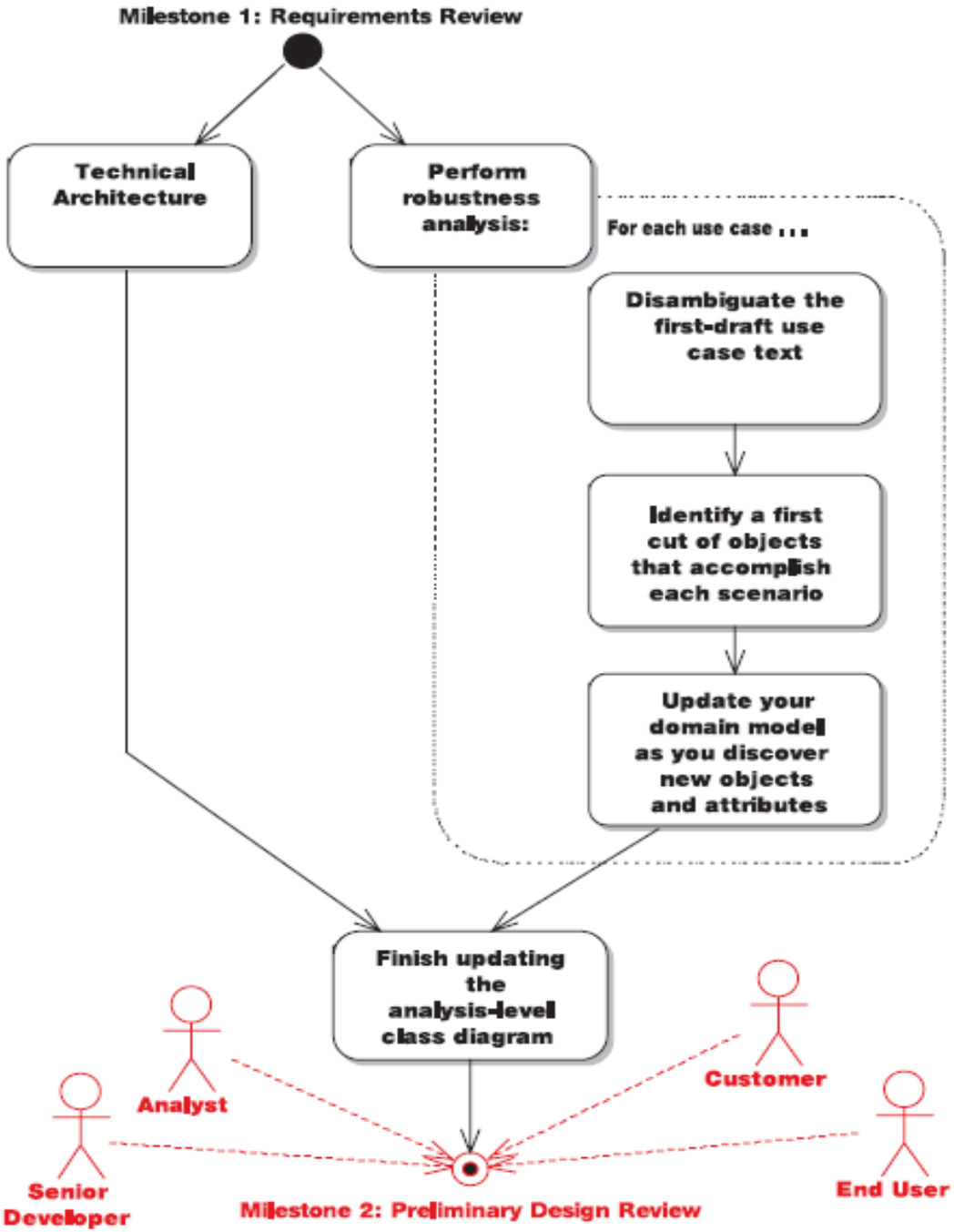
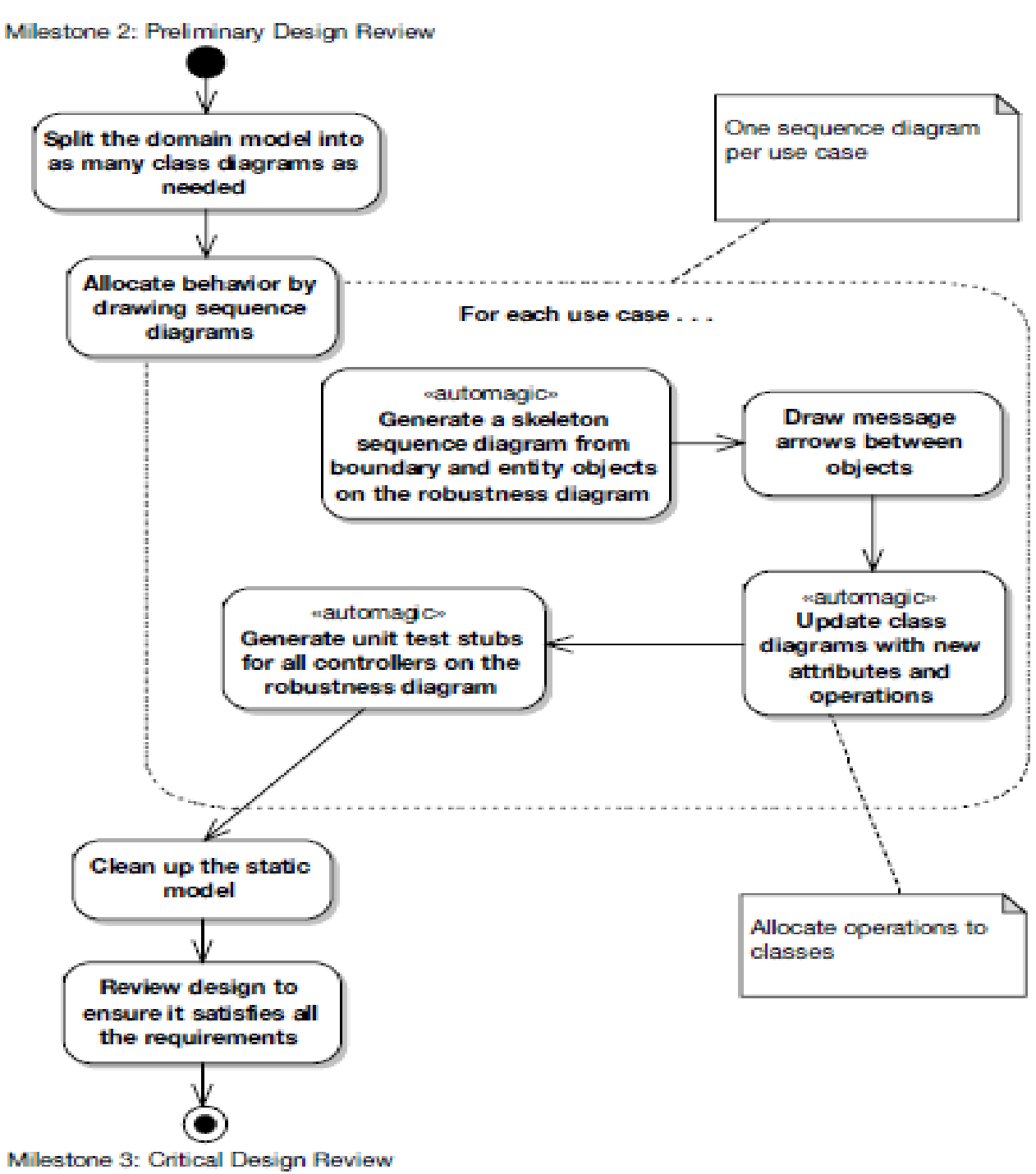


Figure 6-8. *Analysis and Preliminary Design*

top 10 PDR guidelines

10. For each use case, make sure the use case text matches the robustness diagram, using the highlighter test.
9. Make sure that all the entities on all robustness diagrams appear within the updated domain model.
8. Make sure that you can trace data flow between entity classes and screens.
7. Don't forget the alternate courses, and don't forget to write behavior for each of them when you find them.
6. Make sure each use case covers both sides of the dialogue between user and system.
5. Make sure you haven't violated the syntax rules for robustness analysis,
4. Make sure that this review includes both nontechnical (customer, marketing team, etc.) and technical folks (programmers).
3. Make sure your use cases are in the context of the object model and in the context of the GUI.
2. Make sure your robustness diagrams (and the corresponding use case text) don't attempt to show the same level of detail that will be shown on the sequence diagrams (i.e., don't try to do detailed design yet).
1. Follow our "six easy steps" to a better preliminary design (see Chapter 6).

Detailed Design



Sequence Diagramming (Allocate Behavior to Classes)

top 10 sequence diagramming guidelines

10. Understand why you're drawing a sequence diagram, to get the most out of it.
9. Do a sequence diagram for every use case, with both basic and alternate courses on the same diagram.
8. Start your sequence diagram from the boundary classes, entity classes, actors, and use case text that result from robustness analysis.
7. Use the sequence diagram to show how the behavior of the use case (i.e., all the controllers from the robustness diagram) is accomplished by the objects.
6. Make sure your use case text maps to the messages being passed on the sequence diagram. Try to line up the text and message arrows.
5. Don't spend too much time worrying about focus of control.
4. Assign operations to classes while drawing messages. Most visual modeling tools support this capability.
3. Review your class diagrams frequently while you're assigning operations to classes, to make sure all the operations are on the appropriate classes.
2. Prefactor your design on sequence diagrams before coding.
1. Clean up the static model before proceeding to the CDR.

Cleaning Up the Static Model

Milestone 3: Critical Design Review

The CDR helps you to achieve three important goals, before you begin coding for the current batch of use cases:

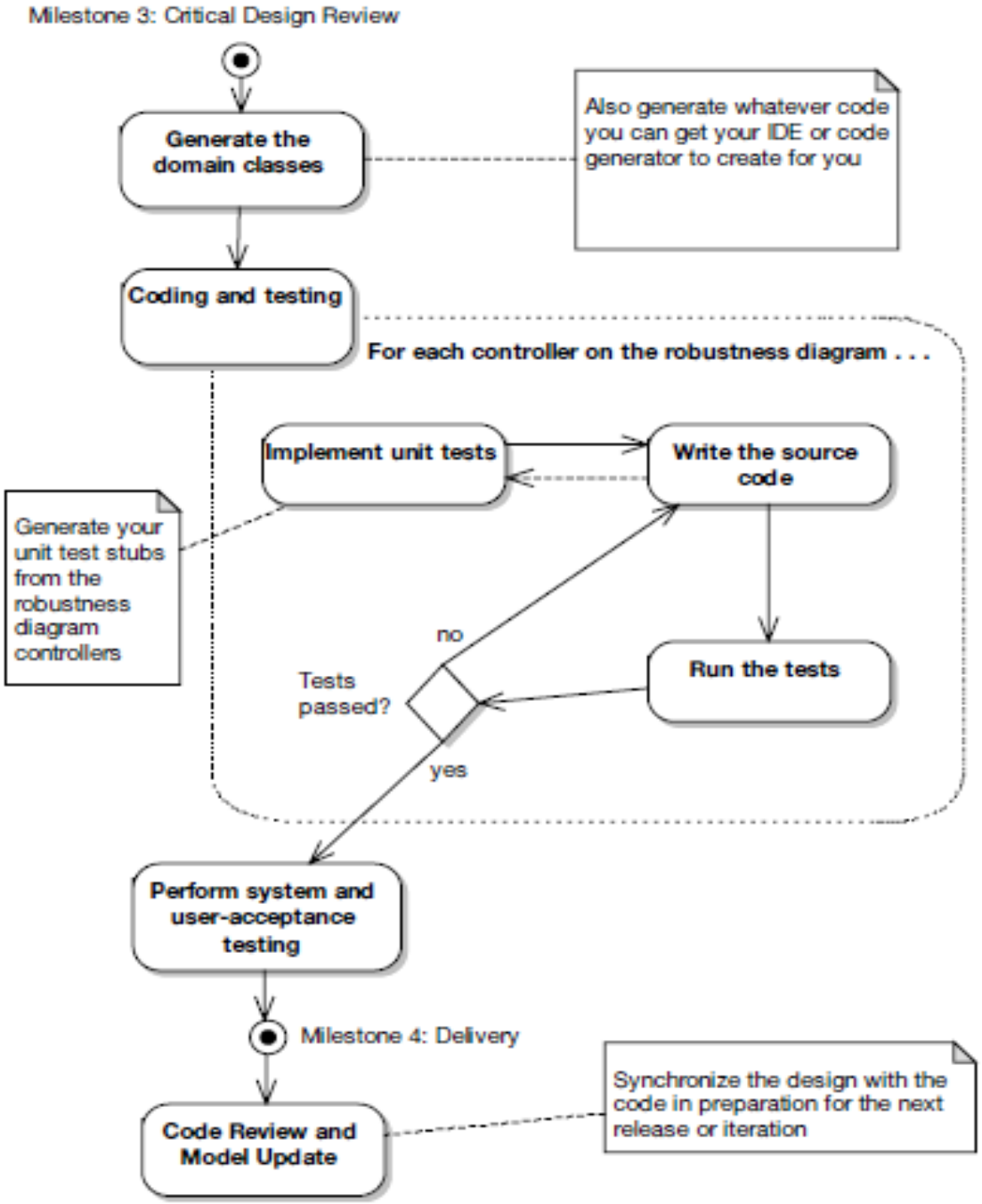
- Ensure that the “how” of detailed design matches up with the “what” specified in your requirements.
- Review the quality of your design.
- Check for continuity of messages on your sequence diagrams (iron out “leaps of logic” in the design).

top 10 CDR guidelines

10. Make sure the sequence diagram matches the use case text.
9. Make sure (yes, again) that each sequence diagram accounts for both basic and alternate courses of action.
8. Make sure that operations have been allocated to classes appropriately.
7. Review the classes on your class diagrams to ensure they all have an appropriate set of attributes and operations.
6. If your design reflects the use of patterns or other detailed implementation constructs, check that these details are reflected on the sequence diagram.
5. Trace your functional (and nonfunctional) requirements to your use cases and classes to ensure you have covered them all.
4. Make sure your programmers “sanity check” the design and are confident that they can build it and that it will work as intended.
3. Make sure all your attributes are typed correctly, and that return values and parameter lists on your operations are complete and correct.
2. Generate the code headers for your classes, and inspect them closely.
1. Review the test plan for your release.

Implementation (Coding)

Implementation



top 10 implementation guidelines

10. Be sure to drive the code directly from the design.
9. If coding reveals the design to be wrong in some way, change it. But also review the process.
8. Hold regular code inspections.
7. Always question the framework's design choices.
6. Don't let framework issues take over from business issues.
5. If the code starts to get out of control, hit the brakes and revisit the design.
4. Keep the design and the code in sync.
3. Focus on unit testing while implementing the code.
2. Don't overcomment your code (it makes your code less maintainable and more difficult to read).
1. Remember to implement the alternate courses as well as the basic courses.

Unit Testing

top 10 unit testing guidelines

10. Adopt a “testing mind-set” wherein every bug found is a victory and not a defeat. If you find (and fix) the bug in testing, the users won’t find it in the released product.
9. Understand the different kinds of testing, and when and why you’d use each one.
8. When unit testing, create one or more unit tests for each controller on each robustness diagram.
7. For real-time systems, use the elements on state diagrams as the basis for test cases.
6. Do requirement-level verification (i.e., check that each requirement you have identified is accounted for).
5. Use a traceability matrix to assist in requirement verification.
4. Do scenario-level acceptance testing for each use case.
3. Expand threads in your test scenarios to cover a complete path through the appropriate part of the basic course plus each alternate course in your scenario testing.
2. Use a testing framework like JUnit to store and organize your unit tests.
1. Keep your unit tests fine-grained.

Expand Threads for Integration and Scenario Testing

This activity involves expanding the sunny day/rainy day threads of the use cases. The integration tests come from the use cases, in the form of testing the following:

1. The entire sunny-day scenario (the basic course)
2. Part of the sunny-day scenario plus each individual rainy day scenario (the alternate courses)

For example, a use case with three alternate courses would need (at minimum) four integration test scenarios: one for the basic course and one for each alternate course (including whichever part of the basic course goes along with it).

Code Review and Model Update

top 10 Code Review and Model Update guidelines

10. Prepare for the review, and make sure all participants have read the relevant review material prior to the meeting.
9. Create a high-level list of items to review, based on the use cases.
8. If necessary, break down each item in the list into a smaller checklist.
7. Review code at several different levels.
6. Gather data during the review, and use it to accumulate boilerplate checklists for future reviews.
5. Follow up the review with a list of action points e-mailed to all people involved.
4. Try to focus on error detection during the review, not error correction.
3. Use an integrated code/model browser that hot-links your modeling tool to your code editor.
2. Keep it “just formal enough” with checklists and follow-up action lists, but don’t overdo the bureaucracy.
1. Remember that it’s also a Model Update session, not just a Code Review.