Chapter

17

GRASP: DESIGNING OBJECTS WITH RESPONSIBILITIES

Understanding responsibilities is key to good object-oriented design.

—Martin Fowler

Objectives

- Learn to apply five of the GRASP principles or patterns for OOD.

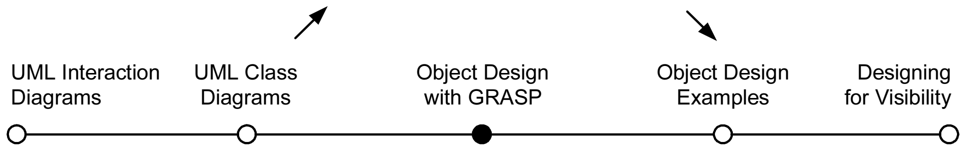
This chapter and the next contribute significantly to an understanding of core OO design (OOD). OOD is sometimes taught as some variation of the following:

> After identifying your requirements and creating a domain model, then add methods to the appropriate classes, and define the messaging between the objects to fulfill the requirements.

Ouch! Such vague advice doesn't help us, because deep principles and issues are involved. Deciding what methods belong where and how objects should interact carries consequences and should be undertaken seriously. Mastering OOD—and this is its intricate charm—involves a large set of soft principles, with many degrees of freedom. It isn't magic—the patterns can be *named* (important!), explained, and applied. Examples help. Practice helps. And this small step helps: After studying these case studies, try recreating (from memory) the Monopoly solution on walls with partners, and apply the principles, such as Information Expert.

What's Next?

Having introduced basic dynamic and static UML notation to support OO design, this chapter introduces design principles. The next provides more detailed case study examples applying these principles and UML modeling.



17.1 UML versus Design Principles

UML and silver bullet thinking p. 12

Since the UML is simply a standard visual modeling language, knowing its details doesn't teach you how to think in objects—that's a theme of this book. The UML is sometimes described as a "design tool" but that's not quite right…

The critical design tool for software development is a mind well educated in design principles. It is not the UML or any other technology.

17.2 Object Design: Example Inputs, Activities, and Outputs

This section summarizes a big-picture example of design in an iterative method:

- What's been done? — Prior activities (e.g., workshop) and artifacts.
- How do things relate? — Influence of prior artifacts (e.g., use cases) on OO design.
- How much design modeling to do, and how?
- What's the output?

Especially, I'd like you to understand how the analysis artifacts relate to object design.

What Are Inputs to Object Design?

Let's start with "process" inputs. Assume we are developers working on the POS NextGen project, and the following scenario is true:

The first ***two-day requirements workshop*** is finished.	The chief architect and business agree to implement and test some ***scenarios of Process Sale in the first three-week*** timeboxed iteration.
Three of the twenty use cases—those that are the most architecturally significant and of high business value—have been analyzed in detail, including, of course, the *Process Sale* use case. (The UP recommends, as typical with iterative methods, analyzing only ***10%–20% of the requirements*** in detail before starting to program.)	***Other artifacts*** have been started: Supplementary Specification, Glossary, and Domain Model.
Programming experiments have resolved the show-stopper technical questions, such as whether a Java Swing UI will work on a touch screen.	The chief architect has drawn some ideas for the ***large-scale logical architecture***, using UML package diagrams. This is part of the UP Design Model.

What are the *artifact* inputs and their relationship to object design?[1] They are summarized in Figure 17.1 and in the following table.

The ***use case text*** defines the visible behavior that the software objects must ultimately support—objects are designed to "realize" (implement) the use cases. In the UP, this OO design is called, not surprisingly, the **use case realization**.	The ***Supplementary Specification*** defines the non-functional goals, such as internalization, our objects must satisfy.
The ***system sequence diagrams*** identify the system operation messages, which are the starting messages on our interaction diagrams of collaborating objects.	The ***Glossary*** clarifies details of parameters or data coming in from the UI layer, data being passed to the database, and detailed item-specific logic or validation requirements, such as the legal formats and validation for product UPCs (universal product codes).
The ***operation contracts*** may complement the use case text to clarify what the software objects must achieve in a system operation. The post-conditions define detailed achievements.	The ***Domain Model*** suggests some names and attributes of software domain objects in the domain layer of the software architecture.

Not all of these artifacts are necessary. Recall that in the UP all elements are optional, possibly created to reduce some risk.

What Are Activities of Object Design?

We're ready to take off our analyst hats and put on our designer-modeler hats.

test first p. 386

Given one or more of these inputs, developers 1) start immediately coding (ideally with **test-first development**), 2) start some UML modeling for the object design, or 3) start with another modeling technique, such as CRC cards.[2]

GRASP p. 277

GoF p. 435

RDD p. 276

In the UML case, the real point is not the UML, but visual modeling—using a language that allows us to explore more visually than we can with just raw text. In this case, for example, we draw both interaction diagrams and complementary class diagrams (dynamic and static modeling) during one **modeling day**. And most importantly, during the drawing (and coding) activity we apply various OO design principles, such as **GRASP** and the **Gang-of-Four (GoF) design patterns**. The overall approach to doing the OO design modeling will be based on the *metaphor* of **responsibility-driven design** (RDD), thinking about how to assign responsibilities to collaborating objects.

This and subsequent chapters explore what it means
to apply RDD, GRASP, and some of the GoF design patterns.

1. Other artifact inputs could include design documents for an existing system being modified. It's also useful to reverse-engineer existing code into UML package diagrams to see the large-scale logical structure and some class and sequence diagrams.
2. All of these approaches are skillful depending on context and person.

On the modeling day, perhaps the team works in small groups for 2–6 hours either at the walls or with software modeling tools, doing different kinds of modeling for the difficult, creative parts of the design. This could include UI, OO, and database modeling with UML drawings, prototyping tools, sketches, and so forth.

During UML drawing, we adopt the realistic attitude (also promoted in agile modeling) that we are drawing the models primarily to *understand and communicate*, not to document. Of course, we expect some of the UML diagrams to be useful input to the definition (or automated code generation with a UML tool) of the code.

On Tuesday—still early in the three-week timeboxed iteration—the team stops modeling and puts on programmer hats to avoid a waterfall mentality of over-modeling before programming.

What Are the Outputs?

Figure 17.1 illustrates some inputs and their relationship to the output of a UML interaction and class diagram. Notice that we may refer to these analysis inputs during design; for example, re-reading the use case text or operation contracts, scanning the domain model, and reviewing the Supplementary Specification.

What's been created during the modeling day (for example)?

- specifically for object design, UML interaction, class, and package diagrams for the difficult parts of the design that we wished to explore before coding
- UI sketches and prototypes
- database models (with UML data modeling profile notation p. 625)
- report sketches and prototypes

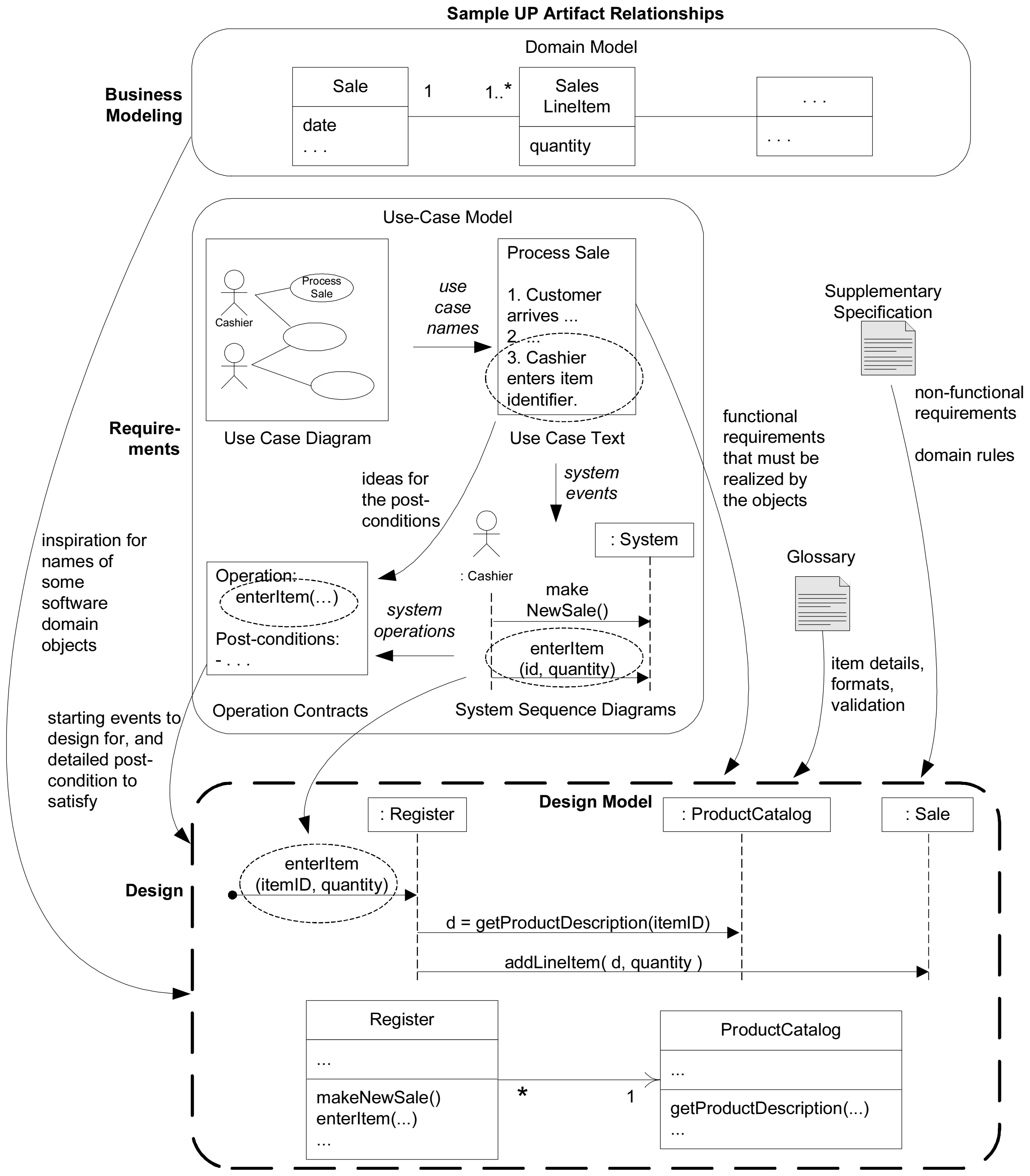


Figure 17.1 Artifact relationships emphasizing influence on OO design.

17.3 Responsibilities and Responsibility-Driven Design

A popular way of thinking about the design of software objects and also larger-scale components[3] is in terms of **responsibilities**, **roles**, and **collaborations**. This is part of a larger approach called **responsibility-driven design** or **RDD** [WM02].

In RDD, we think of software objects as having responsibilities—an abstraction of what they do. The UML defines a **responsibility** as "a contract or obligation of a classifier" [OMG03b]. Responsibilities are related to the obligations or behavior of an object in terms of its role. Basically, these responsibilities are of the following two types: *doing* and *knowing*.

Doing responsibilities of an object include:

- doing something itself, such as creating an object or doing a calculation
- initiating action in other objects
- controlling and coordinating activities in other objects

Knowing responsibilities of an object include:

- knowing about private encapsulated data
- knowing about related objects
- knowing about things it can derive or calculate

Responsibilities are assigned to classes of objects during object design. For example, I may declare that "a *Sale* is responsible for creating *SalesLineItems*" (a doing), or "a *Sale* is responsible for knowing its total" (a knowing).

low representational gap p. 138

Guideline: For software domain objects, the domain model, because of the attributes and associations it illustrates, often inspires the relevant responsibilities related to "knowing." For example, if the domain model *Sale* class has a *time* attribute, it's natural by the goal of **low representational gap** that a software *Sale* class knows its time.

The translation of responsibilities into classes and methods is influenced by the *granularity* of the responsibility. Big responsibilities take hundreds of classes and methods. Little responsibilities might take one method. For example, the responsibility to "provide access to relational databases" may involve two hundred classes and thousands of methods, packaged in a subsystem. By contrast, the responsibility to "create a *Sale*" may involve only one method in one class.

A responsibility is not the same thing as a method—it's an abstraction—but methods fulfill responsibilities.

3. Thinking in terms of responsibilities can apply at any scale of software—from a small object to a system of systems.

RDD also includes the idea of **collaboration**. Responsibilities are implemented by means of methods that either act alone or collaborate with other methods and objects. For example, the *Sale* class might define one or more methods to know its total; say, a method named *getTotal*. To fulfill that responsibility, the *Sale* may collaborate with other objects, such as sending a *getSubtotal* message to each *SalesLineItem* object asking for its subtotal.

RDD is a Metaphor

RDD is a general metaphor for thinking about OO software design. Think of software objects as similar to people with responsibilities who collaborate with other people to get work done. RDD leads to viewing an OO design as a *community of collaborating responsible objects*.

Key point: GRASP names and describes some basic principles to assign responsibilities, so it's useful to know—to support RDD.

17.4 GRASP: A Methodical Approach to Basic OO Design

GRASP: A Learning Aid for OO Design with Responsibilities

It *is* possible to name and explain the detailed principles and reasoning required to grasp basic object design, assigning responsibilities to objects. The **GRASP** principles or patterns are a learning aid to help you understand essential object design and apply design reasoning in a methodical, rational, explainable way. This approach to understanding and using design principles is based on *patterns of assigning responsibilities*.

This chapter—and several others—uses GRASP as a tool to help master the basics of OOD and understanding responsibility assignment in object design.

Understanding how to apply GRASP
for object design is a key goal of the book.

So, GRASP is relevant, but on the other hand, it's just a learning aid to structure and name the principles—once you "grasp" the fundamentals, the specific GRASP terms (Information Expert, Creator, ...) aren't important.

17.5 What's the Connection Between Responsibilities, GRASP, and UML Diagrams?

You can think about assigning responsibilities to objects while coding or while modeling. Within the UML, drawing interaction diagrams becomes the occasion for considering these responsibilities (realized as methods).

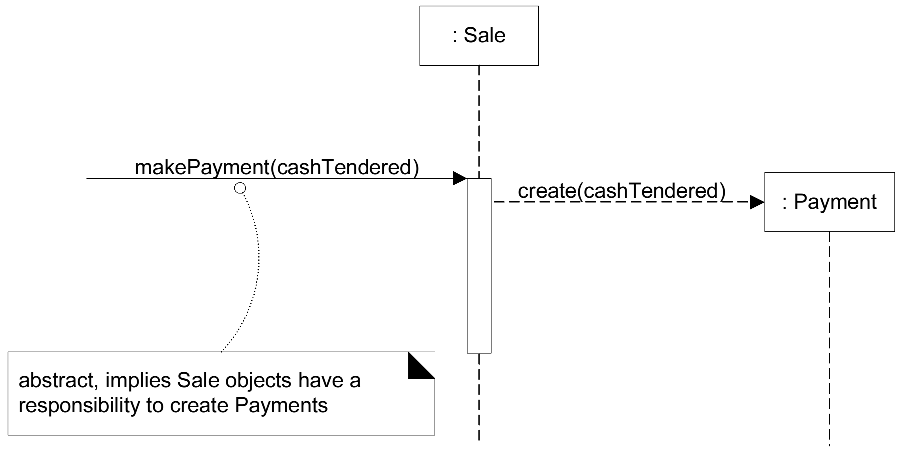

: Sale
makePayment(cashTendered)
create(cashTendered)
: Payment
abstract, implies Sale objects have a responsibility to create Payments

Figure 17.2 Responsibilities and methods are related.

Figure 17.2 indicates that *Sale* objects have been given a responsibility to create *Payments*, which is concretely invoked with a *makePayment* message and handled with a corresponding *makePayment* method. Furthermore, the fulfillment of this responsibility requires collaboration to create the *Payment* object and invoke its constructor.

Therefore, when we draw a UML interaction diagram, we are deciding on responsibility assignments. This chapter emphasizes fundamental principles—expressed in GRASP—to guide choices about assigning responsibilities. Thus, you can apply the GRASP principles while drawing UML interaction diagrams, and also while coding.

17.6 What are Patterns?

Experienced OO developers (and other software developers) build up a repertoire of both general principles and idiomatic solutions that guide them in the creation of software. These principles and idioms, if codified in a structured format describing the problem and solution and named, may be called **patterns**. For example, here is a sample pattern:

Pattern Name:	**Information Expert**
Problem:	What is a basic principle by which to assign responsibilities to objects?
Solution:	Assign a responsibility to the class that has the information needed to fulfill it.

In OO design, a **pattern** is a named description of a problem and solution that can be applied to new contexts; ideally, a pattern advises us on how to apply its solution in varying circumstances and considers the forces and trade-offs. Many patterns, given a specific category of problem, guide the assignment of responsibilities to objects.

Most simply, a good **pattern** is a *named* and *well-known* problem/solution pair that can be applied in new contexts, with advice on how to apply it in novel situations and discussion of its trade-offs, implementations, variations, and so forth.

Patterns Have Names—Important!

Software development is a young field. Young fields lack well-established names for their principles—and that makes communication and education difficult. Patterns have names, such as *Information Expert* and *Abstract Factory.* Naming a pattern, design idea, or principle has the following advantages:

- It supports chunking and incorporating that concept into our understanding and memory.
- It facilitates communication.

When a pattern is named and widely published—and we all agree to use the name—we can discuss a complex design idea in shorter sentences (or shorter diagrams), a virtue of abstraction. Consider the following discussion between two software developers, using a vocabulary of pattern names:

Jill: "Hey Jack, for the persistence subsystem, let's expose the services with a *Facade*. We'll use an *Abstract Factory* for *Mappers*, and *Proxies* for lazy materialization."

Jack: "What the hell did you just say?!?"

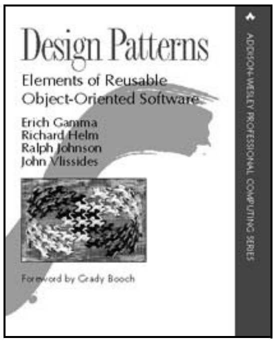
Jill: "Here, read this…"

'New Pattern' is an Oxymoron

New pattern should be considered an oxymoron if it describes a new idea. The very term "pattern" suggests a long-repeating thing. The point of design patterns is *not* to express new design ideas. Quite the opposite—great patterns attempt to codify *existing* tried-and-true knowledge, idioms, and principles; the more honed, old, and widely used, the better.

Consequently, the GRASP patterns don't state new ideas; they name and codify widely used basic principles. To an OO design expert, the GRASP patterns—by idea if not by name—will appear fundamental and familiar. That's the point!

The Gang-of-Four Design Patterns Book


Design Patterns
Elements of Reusable Object-Oriented Software
Erich Gamma
Richard Helm
Ralph Johnson
John Vlissides
Foreword by Grady Booch
ADDISON-WESLEY PROFESSIONAL COMPUTING SERIES

The idea of named patterns in software comes from Kent Beck (also of **Extreme Programming** fame) in the mid 1980s.[4] However, 1994 was a major milestone in the history of patterns, OO design, and software design books: The massive-selling and hugely influential book *Design Patterns* [GHJV95][5] was published, authored by Gamma, Helm, Johnson, and Vlissides. The book, considered the "Bible" of design pattern books, describes 23 patterns for OO design, with names such as *Strategy* and *Adapter.* These 23 patterns, authored by four people, are therefore called the **Gang of Four**[6] (or **GoF**) design patterns.

However, *Design Patterns* isn't an introductory book; it assumes significant prior OO design and programming knowledge, and most code examples are in C++.

Later—intermediate—chapters of *this* book, especially Chapter 26 (p. 435), Chapter 36 (p. 587), and Chapter 37 (p. 621) introduce many of the most frequently used GoF design patterns and apply them to our case studies. Also: See "Contents by Major Topics" on page ix.

It is a key goal of this text to ***learn both GRASP and essential GoF patterns***.

Is GRASP a Set of Patterns or Principles?

GRASP defines nine basic OO design principles or basic building blocks in design. Some have asked, "Doesn't GRASP describe principles rather than patterns?" One answer is in the words of the Gang of Four authors, from the preface of their influential *Design Patterns* book:

One person's pattern is another person's primitive building block.

Rather than focusing on labels, this text focuses on the pragmatic value of using the pattern style as an excellent *learning aid* for naming, presenting, and remembering basic, classic design ideas.

4. The notion of patterns originated with the (building) architectural patterns of Christopher Alexander [AIS77]. Patterns for software originated in the 1980s with Kent Beck, who became aware of Alexander's pattern work in architecture, and then were developed by Beck with Ward Cunningham [BC87, Beck94] at Tektronix.
5. Publishers list the publication date as 1995, but it was released October 1994.
6. Also a subtle joke related to mid-1970s Chinese politics following Mao's death.

17.7 Where are We Now?

So far, this chapter has summarized the background for OO design:

1. The iterative **process background**—Prior artifacts? How do they relate to OO design models? How much time should we spend design modeling?
2. **RDD** as a metaphor for object design—a community of collaborating responsible objects.
3. **Patterns** as a way to name and explain OO design ideas—**GRASP** for basic patterns of assigning responsibilities, and **GoF** for more advanced design ideas. Patterns can be applied during modeling and during coding.
4. **UML** for OO design **visual modeling**, during which time both GRASP and GoF patterns can be applied.

With that understood, it's time to focus on some details of object design.

17.8 A Short Example of Object Design with GRASP

All the GRASP patterns are summarized on the inside front cover of this book.

Following sections explore GRASP in more detail, but let's start with a shorter example to see the big ideas, applied to the Monopoly case study. There are nine GRASP patterns; this example applies the following subset:

- Creator
- Information Expert
- Low Coupling
- Controller
- High Cohesion

Creator

Problem: Who creates the *Square* object?

One of the first problems you have to consider in OO design is: Who creates object X? This is a *doing* responsibility. For example, in the Monopoly case study, who creates a *Square* software object? Now, *any* object can create a *Square*, but what would many OO developers choose? And why?

How about having a *Dog* object (i.e., some arbitrary class) be the creator? No! We can feel it in our bones. Why? Because—and this is the critical point—it doesn't appeal to our mental model of the domain. *Dog* doesn't support **low representational gap** (**LRG**) between how we think of the domain and a straightforward correspondence with software objects. I've done this problem with literally thousands of developers, and virtually every one, from India to the USA, will say, "Make the *Board* object create the *Squares*." Interesting! It reflects an "intuition" that OO software developers often (exceptions are explored later) want "containers" to create the things "contained," such as *Boards* creating *Squares*.

By the way, why we are defining software classes with the names *Square* and *Board,* rather than the names *AB324* and *ZC17*? Answer: By LRG. This connects the UP Domain Model to the UP Design Model, or our mental model of the domain to its realization in the *domain layer* of the software architecture.

With that as background, here's the definition of the **Creator** pattern[7]:

Name:	**Creator**
Problem:	Who creates an A?
Solution: (this can be viewed as advice)	Assign class B the responsibility to create an instance of class A if one of these is true (the more the better): ■ B "contains" or compositely aggregates A. ■ B records A. ■ B closely uses A. ■ B has the initializing data for A.

Notice this has to do with responsibility assignment. Let's see how to apply Creator.

First, a subtle but important point in applying Creator and other GRASP patterns: *B* and *A* refer to *software* objects, not domain model objects. We first try to apply Creator by looking for existing *software* objects that satisfy the role of *B*. But what if we are just starting the OO design, and we have not yet defined any software classes? In this case, by LRG, *look to the domain model* for inspiration.

Thus, for the *Square* creation problem, since no software classes are yet defined, we look at the domain model in Figure 17.3 and see that a *Board contains Squares*. That's a conceptual perspective, not a software one, but of course we can mirror it in the Design Model so that a software *Board* object contains software *Square* objects. And then consistent with LRG and the Creator advice, the *Board* will create *Squares*. Also, *Squares* will always be a part of one *Board*, and *Board* manages their creation and destruction; thus, they are in a *composite aggregation* association with the *Board*.

Recall that an agile modeling practice is to create parallel complementary dynamic and static object models. Therefore, I've drawn *both* a partial sequence diagram and class diagram to reflect this design decision in which I've applied a GRASP pattern while drawing UML diagrams. See Figure 17.4 and Figure 17.5. Notice in Figure 17.4 that when the *Board* is created, it creates a *Square*. For brevity in this example, I'll ignore the side issue of drawing the loop to create all 40 squares.

7. Alternate creation patterns, such as *Concrete Factory* and *Abstract Factory*, are discussed later.

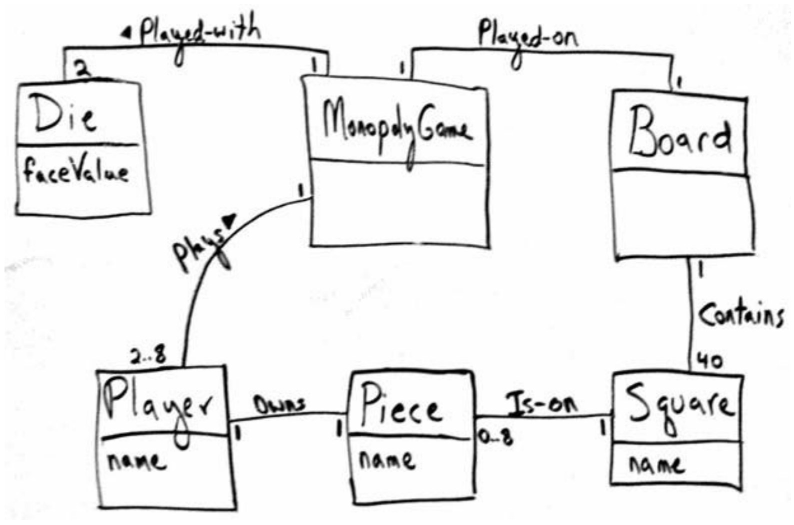


Figure 17.3 Monopoly iteration-1 domain model.

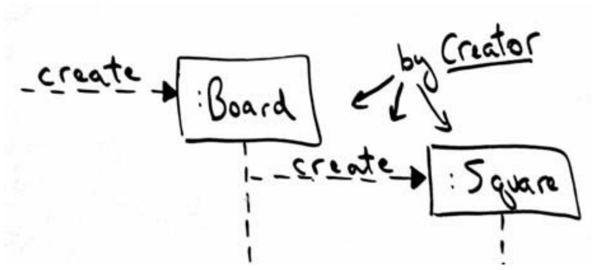


Figure 17.4 Applying the Creator pattern in a dynamic model.

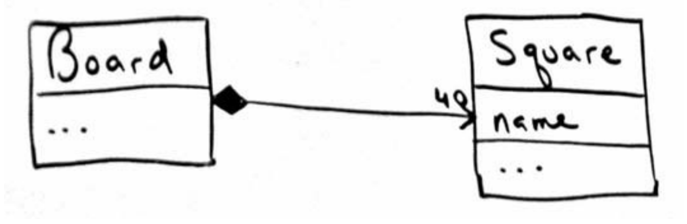


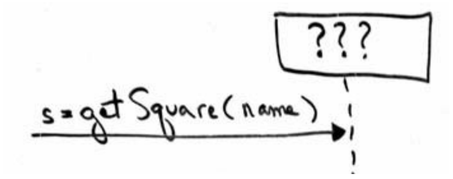
Figure 17.5 In a DCD of the Design Model, *Board* has a composite aggregation association with *Squares*. We are applying Creator in a static model.

Information Expert

Problem: Who knows about a *Square* object, given a key?

The pattern Information Expert (often abbreviated to Expert) is one of the most basic responsibility assignment principles in object design.

Suppose objects need to be able to reference a particular *Square*, given its name. Who should be responsible for knowing a *Square*, given a key? Of course, this is a *knowing* responsibility, but Expert also applies to *doing*.



As with Creator, *any* object can be responsible, but what would many OO developers choose? And why? As with the Creator problem, most OO developers choose the *Board* object. It seems sort of trivially obvious to assign this responsibility to a *Board*, but it is instructive to deconstruct why, and to learn to apply this principle in more subtle cases. Later examples will get more subtle.

Information Expert explains why the *Board* is chosen:

Name:	**Information Expert**
Problem:	What is a basic principle by which to assign responsibilities to objects?
Solution: (advice)	Assign a responsibility to the class that has the information needed to fulfill it.

A responsibility needs information for its fulfillment—information about other objects, an object's own state, the world around an object, information the object can derive, and so forth. In this case, to be able to retrieve and present any one *Square*—given its name—some object must know (have the information) about *all* the *Squares*. We previously decided, as shown in Figure 17.5, that a software *Board* will aggregate all the *Square* objects. Therefore, *Board* has the information necessary to fulfill this responsibility. Figure 17.6 illustrates applying Expert in the context of drawing.

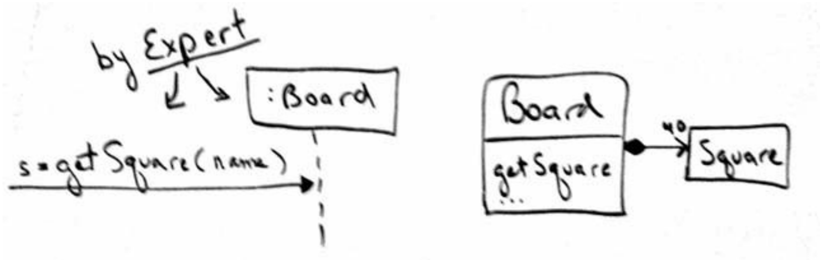


Figure 17.6 Applying Expert.

The next GRASP principle, Low Coupling, explains why Expert is a useful, core principle of OO design.

Low Coupling

Question: Why *Board* over *Dog?*

Expert guides us to assign the responsibility to know a particular *Square*, given a unique name, to the *Board* object because the *Board* knows about all the

Squares (it has the information—it is the Information Expert). But why does Expert give this advice?

The answer is found in the principle of Low Coupling. Briefly and informally, **coupling** is a measure of how strongly one element is connected to, has knowledge of, or depends on other elements. If there is coupling or dependency, then when the depended-upon element changes, the dependant may be affected. For example, a subclass is strongly coupled to a superclass. An object *A* that calls on the operations of object *B* has coupling to *B's* services.

The Low Coupling principle applies to many dimensions of software development; it's really one of the cardinal goals in building software. In terms of object design and responsibilities, we can describe the advice as follows:

Name:	**Low Coupling**
Problem:	How to reduce the impact of change?
Solution: (advice)	Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.

We use Low Coupling to *evaluate* existing designs or to evaluate the choice between new alternatives—all other things being equal, we should prefer a design whose coupling is lower than the alternatives.

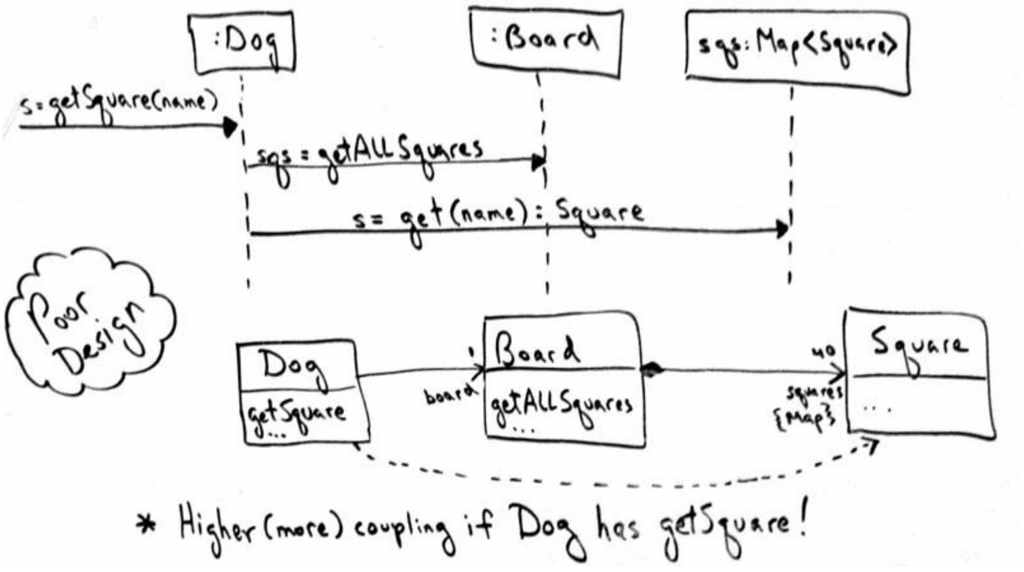


Figure 17.7 Evaluating the effect of coupling on this design.

For example, as we've decided in Figure 17.5, a *Board* object contains many *Squares*. Why not assign *getSquare* to *Dog* (i.e., some arbitrary other class)? Consider the impact in terms of low coupling. If a *Dog* has *getSquare*, as shown in the UML sketch in Figure 17.7, it must collaborate with the *Board* to get the collection of all the *Squares* in the *Board*. They are probably stored in a *Map* collection object, which allows retrieval by a key. Then, the *Dog* can access and return one particular *Square* by the key *name*.

But let's evaluate the total coupling with this poor *Dog* design versus our original design where *Board* does *getSquare*. In the *Dog* case, the *Dog* and the *Board* must both know about *Square* objects (two objects have coupling to *Square*); in the *Board* case, only *Board* must know about *Square* objects (one object has coupling to *Square*). Thus, the overall coupling is lower with the *Board* design, and all other things being equal, it is better than the *Dog* design, in terms of supporting the goal of Low Coupling.

At a higher-goal level, why is Low Coupling desirable? In other words, why would we want to reduce the impact of change? *Because Low Coupling tends to reduce the time, effort, and defects in modifying software.* That's a short answer, but one with big implications in building and maintaining software!

Key Point: Expert Supports Low Coupling

To return to the motivation for Information Expert: it guides us to a choice that supports Low Coupling. Expert asks us to find the object that has most of the information required for the responsibility (e.g., *Board*) and assign responsibility there.

If we put the responsibility anywhere else (e.g., *Dog*), the overall coupling will be higher because more information or objects must be shared away from their original source or home, as the squares in the *Map* collection had to be shared with the *Dog*, away from their home in the *Board*.

Applying UML: Please note a few UML elements in the sequence diagram in Figure 17.7:

- The return value variable *sqs* from the *getAllSquares* message is also used to name the lifeline object in *sqs : Map<Square>* (e.g., a collection of type *Map* that holds *Square* objects). Referencing a return value variable in a lifeline box (to send it messages) is common.
- The variable *s* in the starting *getSquare* message and the variable *s* in the later *get* message refer to the same object.
- The message expression *s = get(name) : Square* indicates that the type of *s* is a reference to a *Square* instance.

Controller

A simple layered architecture has a UI layer and a domain layer, among others. Actors, such as the human observer in the Monopoly game, generate UI events, such as clicking on a button with a mouse to play the game. The UI software objects (in Java for example, a *JFrame* window and a *JButton* button) must then react to the mouse click event and ultimately cause the game to play.

Model-View Separation p. 209

From the Model-View Separation Principle, we know the UI objects should *not* contain application or "business" logic such as calculating a player's move. Therefore, once the UI objects pick up the mouse event, they need to **delegate** (forward the task to another object) the request to *domain objects* in the *domain layer.*

The Controller pattern answers this simple question: What *first* object after or beyond the UI layer should receive the message from the UI layer?

To tie this back to system sequence diagrams, as a review of Figure 17.8 shows, the key system operation is *playGame*. Somehow the human observer generates a *playGame* request (probably by clicking on a GUI button labeled "Play Game") and the system responds.

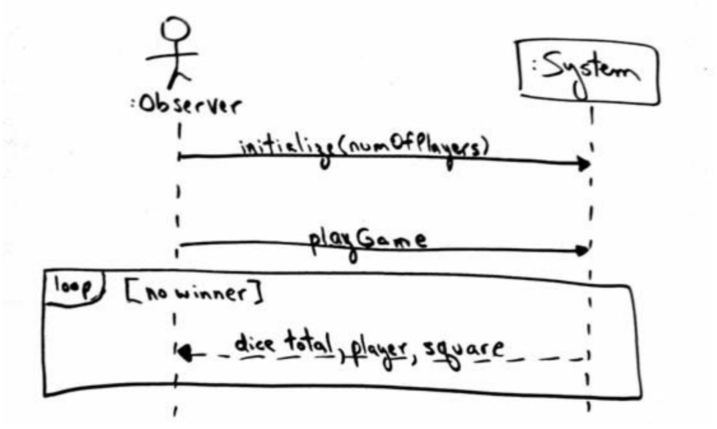


Figure 17.8 SSD for the Monopoly game. Note the *playGame* operation.

Figure 17.9 illustrates a finer-grained look at what's going on, assuming a Java Swing GUI *JFrame* window and *JButton* button.[8] Clicking on a *JButton* sends an *actionPerformed* message to some object, often to the *JFrame* window itself, as we see in Figure 17.9. Then—and this is the ***key point***—the *JFrame* window must adapt that *actionPerformed* message into something more semantically meaningful, such as a *playGame* message (to correspond to the SSD analysis), and delegate the *playGame* message to a domain object in the domain layer.

Do you see the connection between the SSD system operations and the detailed object design from the UI to domain layer? This is important.

Thus, Controller deals with a basic question in OO design: How to connect the UI layer to the application logic layer? Should the *Board* be the first object to receive the *playGame* message from the UI layer? Or something else?

8. Similar objects, messages, and collaboration patterns apply to .NET, Python, etc.

In some OOA/D methods, the name *controller* was given to the application logic object that received and "controlled" (coordinated) handling the request.

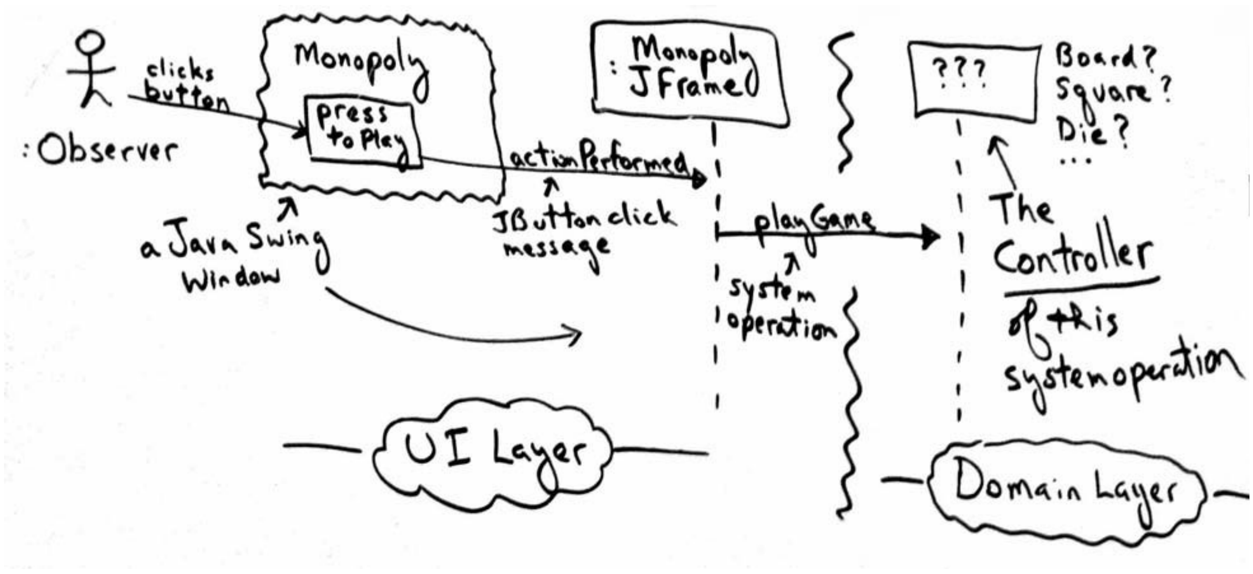


Figure 17.9 Who is the Controller for the *playGame* system operation?

The Controller pattern offers the following advice:

Name:	**Controller**
Problem:	What first object beyond the UI layer receives and coordinates ("controls") a system operation?
Solution: (advice)	Assign the responsibility to an object representing one of these choices:

- Represents the overall "system," a "root object," a device that the software is running within, or a major subsystem (these are all variations of a *facade controller*).
- Represents a use case scenario within which the system operation occurs (a use case or *session controller*)

Let's consider these options:

Option 1: Represents the overall "system," or a "root object"—such as an object called *MonopolyGame*.

Option 1: Represents a device that the software is running within—this option appertains to specialized hardware devices such as a phone or a bank cash machine (e.g., software class *Phone* or *BankCashMachine*); it doesn't apply in this case.

Option 2: Represents the use case or session. The use case that the *playGame* system operation occurs within is called *Play Monopoly Game*. Thus, a software class such as *PlayMonopolyGameHandler* (appending "*…Handler*" or "*…Session*" is an idiom in OO design when this version is used).

Option #1, class *MonopolyGame*, is reasonable if there are only a few system operations (more on the trade-offs when we discuss High Cohesion). Therefore, Figure 17.10 illustrates the design decision based on Controller.

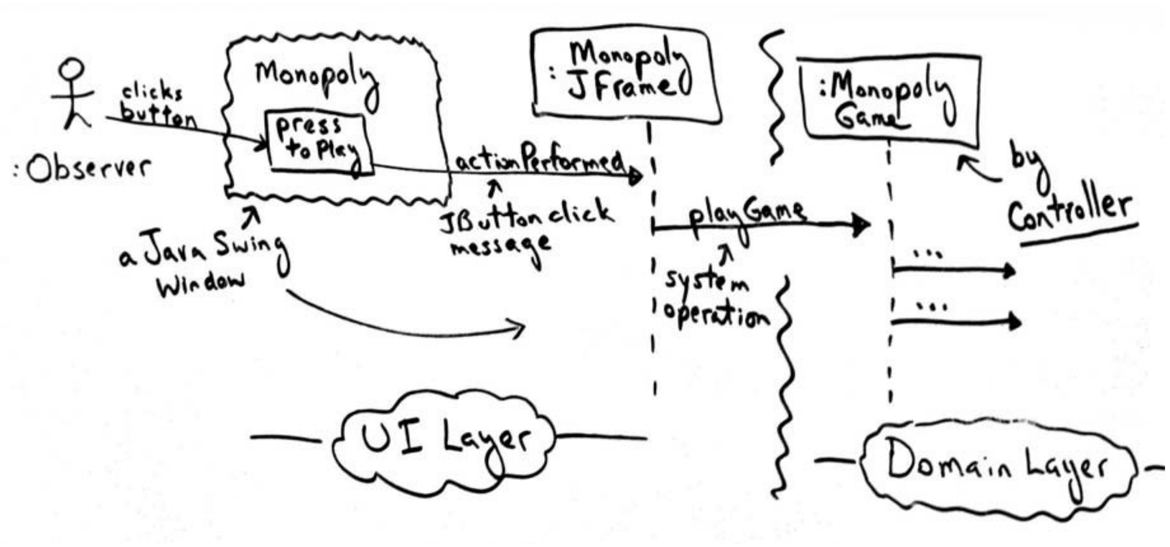
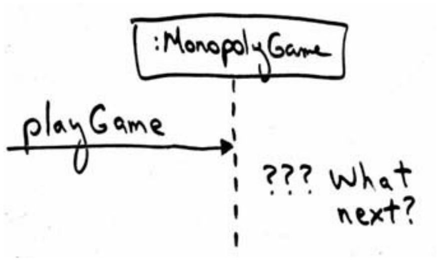


Figure 17.10 Applying the Controller pattern—using *MonopolyGame*. Connecting the UI layer to the domain layer of software objects.

High Cohesion

Based on the Controller decision, we are now at the design point shown in the sequence diagram to the right. The detailed design discussion of what comes next—consistently and methodically applying GRASP—is explored in a following chapter, but right now we have two contrasting design approaches worth considering, illustrated in Figure 17.11.



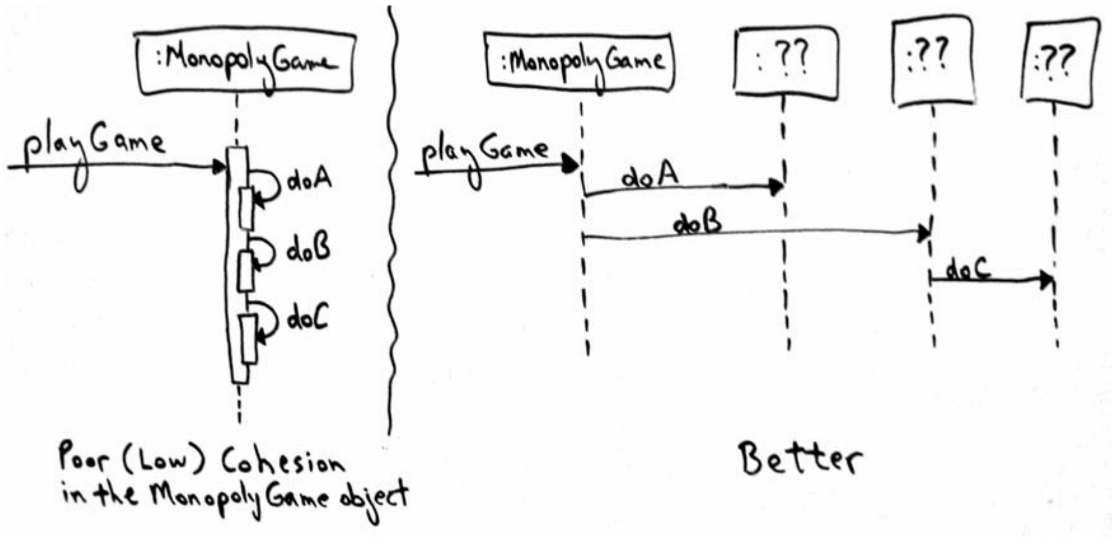


Figure 17.11 Contrasting the level of cohesion in different designs.

Notice in the left-hand version that the *MonopolyGame* object itself does all the work, and in the right-hand version it *delegates* and coordinates the work for the *playGame* request. In software design a basic quality known as **cohesion** informally measures how functionally related the operations of a software element are, and also measures how much work a software element is doing. As a simple contrasting example, an object *Big* with 100 methods and 2,000 source lines of code (SLOC) is doing a lot more than an object *Small* with 10 methods and 200 source lines. And if the 100 methods of *Big* are covering many different areas of responsibility (such as database access *and* random number generation), then *Big* has less focus or functional cohesion than *Small*. In summary, both the amount of code and the relatedness of the code are an indicator of an object's cohesion.

To be clear, bad cohesion (low cohesion) doesn't just imply an object does work only by itself; indeed, a low cohesion object with 2,000 SLOC probably collaborates with many other objects. Now, here's a *key point*: All that interaction tends to *also* create bad (high) coupling. Bad cohesion and bad coupling often go hand-in-hand.

In terms of the contrasting designs in Figure 17.11, the left-hand version of *MonopolyGame* has worse cohesion than the right-hand version, since the left-hand version is making the *MonopolyGame* object itself do all the work, rather than delegating and distributing work among objects. This leads to the principle of High Cohesion, which is used to evaluate different design choices. All other things being equal, prefer a design with higher cohesion.

Name:	**High Cohesion**
Problem:	How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?
Solution: (advice)	Assign responsibilities so that cohesion remains high. Use this to evaluate alternatives.

We can say that the right-hand design better supports High Cohesion than the left-hand version.

17.9 Applying GRASP to Object Design

All nine GRASP patterns are summarized on the inside front cover of this book.

GRASP stands for **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns.[9] The name was chosen to suggest the importance of *grasping* these principles to successfully design object-oriented software.

Understanding and being able to apply the ideas behind GRASP—while coding or while drawing interaction and class diagrams—enables developers new to object technology needs to master these basic principles as quickly as possible; they form a foundation for designing OO systems.

There are nine GRASP patterns:

Creator	Controller	Pure Fabrication
Information Expert	High Cohesion	Indirection
Low Coupling	Polymorphism	Protected Variations

The remainder of this chapter reexamines the first five in more detail; the remaining four are introduced in Chapter 25 starting on p. 413.

17.10 Creator

Problem Who should be responsible for creating a new instance of some class?

The creation of objects is one of the most common activities in an object-oriented system. Consequently, it is useful to have a general principle for the assignment of creation responsibilities. Assigned well, the design can support low coupling, increased clarity, encapsulation, and reusability.

9. Technically, one should write "GRAS Patterns" rather than "GRASP Patterns," but the latter sounds better.

Solution Assign class B the responsibility to create an instance of class A if one of these is true (the more the better):[10]

- B "contains" or compositely aggregates A.
- B records A.
- B closely uses A.
- B has the initializing data for A that will be passed to A when it is created. Thus B is an Expert with respect to creating A.

B is a *creator* of A objects.

If more than one option applies, usually prefer a class B which *aggregates* or *contains* class A.

Example In the NextGen POS application, who should be responsible for creating a *SalesLineItem* instance? By Creator, we should look for a class that aggregates, contains, and so on, *SalesLineItem* instances. Consider the partial domain model in Figure 17.12.

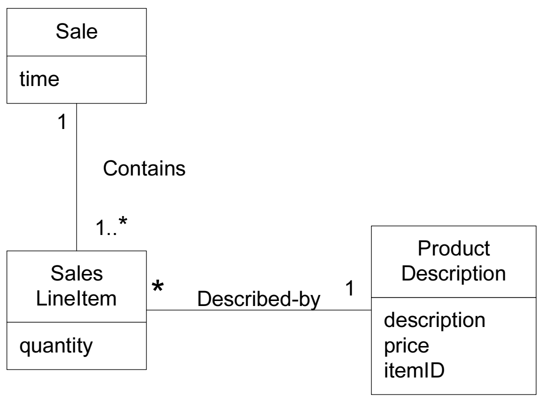


Figure 17.12 Partial domain model.

Since a *Sale* contains (in fact, aggregates) many *SalesLineItem* objects, the Creator pattern suggests that *Sale* is a good candidate to have the responsibility of creating *SalesLineItem* instances. This leads to the design of object interactions shown in Figure 17.13.

This assignment of responsibilities requires that a *makeLineItem* method be defined in *Sale*. Once again, the context in which we considered and decided on these responsibilities was while drawing an interaction diagram. The method section of a class diagram can then summarize the responsibility assignment results, concretely realized as methods.

10. Other creation patterns, such as *Concrete Factory* and *Abstract Factory*, are explored later.

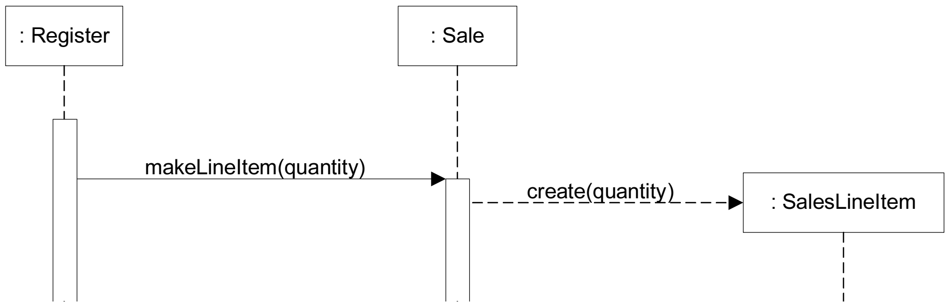


Figure 17.13 Creating a SalesLineItem.

Discussion Creator guides the assigning of responsibilities related to the creation of objects, a very common task. The basic intent of the Creator pattern is to find a creator that needs to be connected to the created object in any event. Choosing it as the creator supports low coupling.

Composite *aggregates* Part, Container *contains* Content, and Recorder *records*. Recorded are all very common relationships between classes in a class diagram. Creator suggests that the enclosing container or recorder class is a good candidate for the responsibility of creating the thing contained or recorded. Of course, this is only a guideline.

composite aggregation p. 264

Note that we turned to the concept of **composition** in considering the Creator pattern. A composite object is an excellent candidate to make its parts.

Sometimes you identify a creator by looking for the class that has the initializing data that will be passed in during creation. This is actually an example of the Expert pattern. Initializing data is passed in during creation via some kind of initialization method, such as a Java constructor that has parameters. For example, assume that a *Payment* instance, when created, needs to be initialized with the *Sale* total. Since *Sale* knows the total, *Sale* is a candidate creator of the *Payment*.

Contraindications Often, creation requires significant complexity, such as using recycled instances for performance, conditionally creating an instance from one of a family of similar classes based upon some external property value, and so forth. In these cases, it is advisable to delegate creation to a helper class called a *Concrete Factory* or an *Abstract Factory* [GHJV95] rather than use the class suggested by *Creator*. Factories are discussed starting on p. 440.

Benefits

- Low coupling is supported, which implies lower maintenance dependencies and higher opportunities for reuse. Coupling is probably not increased because the *created* class is likely already visible to the *creator* class, due to the existing associations that motivated its choice as creator.

Related Patterns or Principles

- Low Coupling
- Concrete Factory and Abstract Factory
- Whole-Part [BMRSS96] describes a pattern to define aggregate objects that support encapsulation of components.

17.11 Information Expert (or Expert)

Problem What is a general principle of assigning responsibilities to objects?

A Design Model may define hundreds or thousands of software classes, and an application may require hundreds or thousands of responsibilities to be fulfilled. During object design, when the interactions between objects are defined, we make choices about the assignment of responsibilities to software classes. If we've chosen well, systems tend to be easier to understand, maintain, and extend, and our choices afford more opportunity to reuse components in future applications.

Solution Assign a responsibility to the information expert—the class that has the *information* necessary to fulfill the responsibility.

Example In the NextGEN POS application, some class needs to know the grand total of a sale.

> Start assigning responsibilities by clearly stating the responsibility.

By this advice, the statement is:

Who should be responsible for knowing the grand total of a sale?

By *Information Expert*, we should look for that class of objects that has the information needed to determine the total.

Now we come to a key question: Do we look in the Domain Model or the Design Model to analyze the classes that have the information needed? The Domain Model illustrates conceptual classes of the real-world domain; the Design Model illustrates software classes.

Answer:

1. If there are relevant classes in the Design Model, look there first.
2. Otherwise, look in the Domain Model, and attempt to use (or expand) its representations to inspire the creation of corresponding design classes.

For example, assume we are just starting design work and there is no, or a minimal, Design Model. Therefore, we look to the Domain Model for information experts; perhaps the real-world *Sale* is one. Then, we add a software class to the Design Model similarly called *Sale*, and give it the responsibility of knowing its total, expressed with the method named *getTotal*. This approach supports *low representational gap* in which the software design of objects appeals to our concepts of how the real domain is organized.

To examine this case in detail, consider the partial Domain Model in Figure 17.14.

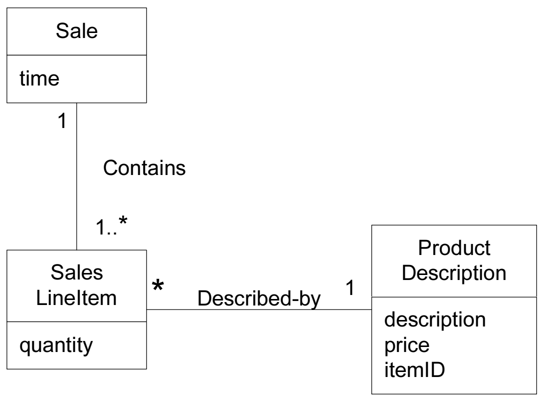


Figure 17.14 Associations of Sale.

What information do we need to determine the grand total? We need to know about all the *SalesLineItem* instances of a sale and the sum of their subtotals. A *Sale* instance contains these; therefore, by the guideline of Information Expert, *Sale* is a suitable class of object for this responsibility; it is an *information expert* for the work.

As mentioned, it is in the context of the creation of interaction diagrams that these questions of responsibility often arise. Imagine we are starting to work through the drawing of diagrams in order to assign responsibilities to objects. A partial interaction diagram and class diagram in Figure 17.15 illustrate some decisions.

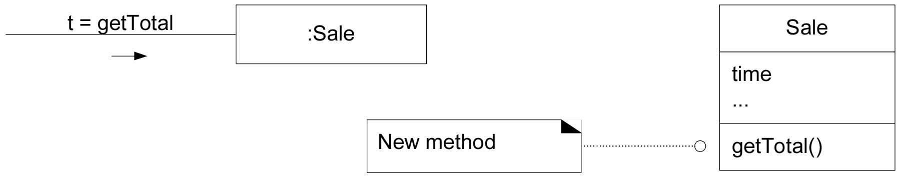


Figure 17.15 Partial interaction and class diagrams.

We are not done yet. What information do we need to determine the line item subtotal? *SalesLineItem.quantity* and *ProductDescription.price*. The *SalesLineItem* knows its quantity and its associated *ProductDescription*; therefore, by Expert, *SalesLineItem* should determine the subtotal; it is the *information expert*.

In terms of an interaction diagram, this means that the *Sale* should send *getSubtotal* messages to each of the *SalesLineItems* and sum the results; this design is shown in Figure 17.16.

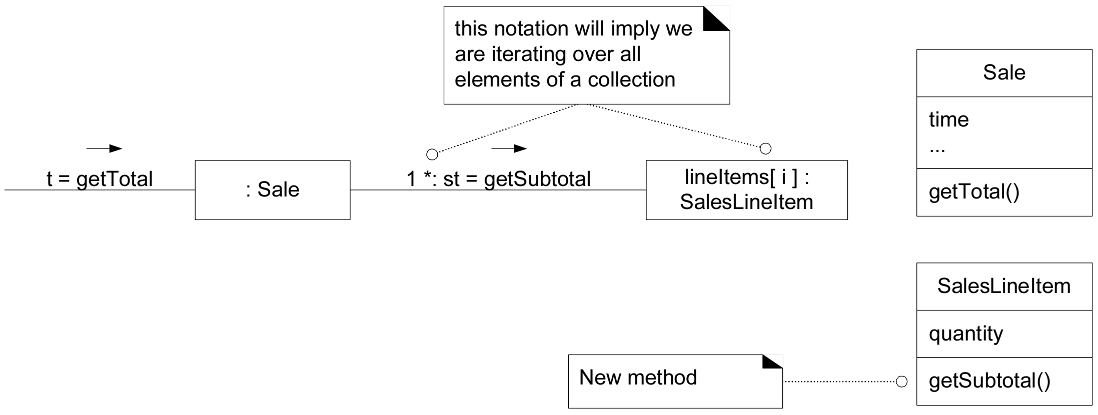

this notation will imply we are iterating over all elements of a collection
t = getTotal
: Sale
1 *: st = getSubtotal
lineItems[i] : SalesLineItem
Sale
time
...
getTotal()
SalesLineItem
quantity
getSubtotal()
New method

Figure 17.16 Calculating the Sale total.

To fulfill the responsibility of knowing and answering its subtotal, a *SalesLineItem* has to know the product price.

The *ProductDescription* is an information expert on answering its price; therefore, *SalesLineItem* sends it a message asking for the product price.

The design is shown in Figure 17.17.

In conclusion, to fulfill the responsibility of knowing and answering the sale's total, we assigned three responsibilities to three design classes of objects as follows.

Design Class	Responsibility
Sale	knows sale total
SalesLineItem	knows line item subtotal
ProductDescription	knows product price

We considered and decided on these responsibilities in the context of drawing an interaction diagram. We could then summarize the methods in the method section of a class diagram.

The principle by which we assigned each responsibility was Information Expert—placing it with the object that had the information needed to fulfill it.

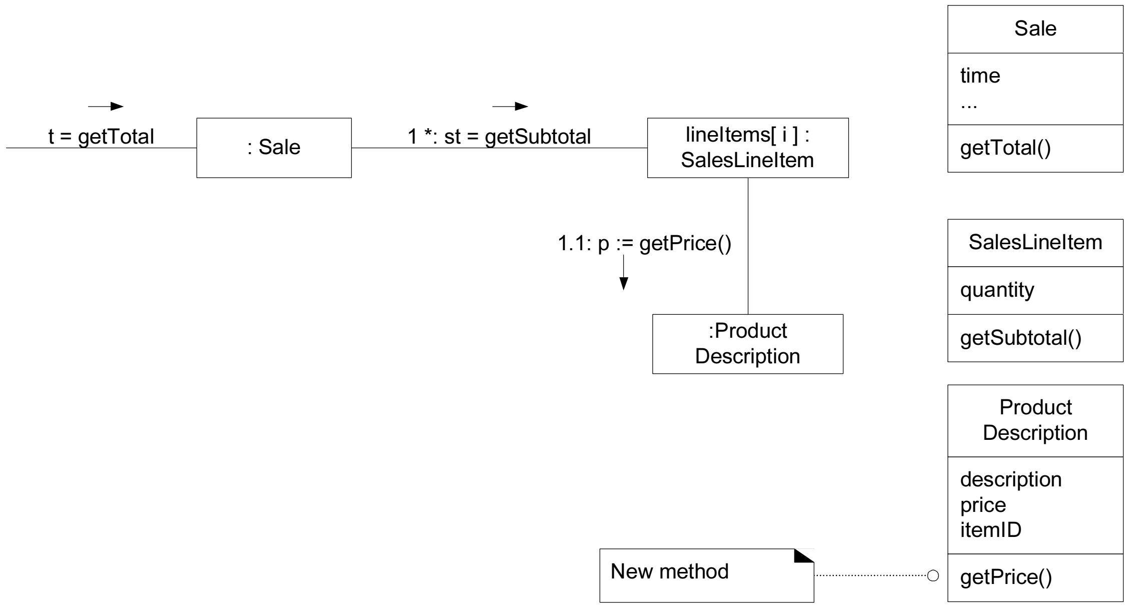


Figure 17.17 Calculating the *Sale* total.

Discussion Information Expert is frequently used in the assignment of responsibilities; it is a basic guiding principle used continuously in object design. Expert is not meant to be an obscure or fancy idea; it expresses the common "intuition" that objects do things related to the information they have.

Notice that the fulfillment of a responsibility often requires information that is spread across different classes of objects. This implies that many "partial" information experts will collaborate in the task. For example, the sales total problem ultimately required the collaboration of three classes of objects. Whenever information is spread across different objects, they will need to interact via messages to share the work.

Expert usually leads to designs where a software object does those operations that are normally done to the inanimate real-world thing it represents; Peter Coad calls this the "Do It Myself" strategy [Coad95]. For example, in the real world, without the use of electro-mechanical aids, a sale does not tell you its total; it is an inanimate thing. Someone calculates the total of the sale. But in object-oriented software land, all software objects are "alive" or "animated," and they can take on responsibilities and do things. Fundamentally, they do things related to the information they know. I call this the "animation" principle in object design; it is like being in a cartoon where everything is alive.

The Information Expert pattern—like many things in object technology—has a real-world analogy. We commonly give responsibility to individuals who have the information necessary to fulfill a task. For example, in a business, who should be responsible for creating a profit-and-loss statement? The person who has access to all the information necessary to create it—perhaps the chief financial officer. And just as software objects collaborate because the information is spread around, so it is with people. The company's chief financial officer may ask accountants to generate reports on credits and debits.

Contraindications In some situations, a solution suggested by Expert is undesirable, usually because of problems in coupling and cohesion (these principles are discussed later in this chapter).

For example, who should be responsible for saving a *Sale* in a database? Certainly, much of the information to be saved is in the *Sale* object, and thus Expert could argue that the responsibility lies in the *Sale* class. And, by logical extension of this decision, each class would have its own services to save itself in a database. But acting on that reasoning leads to problems in cohesion, coupling, and duplication. For example, the *Sale* class must now contain logic related to database handling, such as that related to SQL and JDBC (Java Database Connectivity). The class no longer focuses on just the pure application logic of "being a sale." Now other kinds of responsibilities lower its cohesion. The class must be coupled to the technical database services of another subsystem, such as JDBC services, rather than just being coupled to other objects in the domain layer of software objects, so its coupling increases. And it is likely that similar database logic would be duplicated in many persistent classes.

All these problems indicate violation of a basic architectural principle: design for a separation of major system concerns. Keep application logic in one place (such as the domain software objects), keep database logic in another place (such as a separate persistence services subsystem), and so forth, rather than intermingling different system concerns in the same component.[11]

Supporting a separation of major concerns improves coupling and cohesion in a design. Thus, even though by Expert we could find some justification for putting the responsibility for database services in the *Sale* class, for other reasons (usually cohesion and coupling), we'd end up with a poor design.

Benefits

- Information encapsulation is maintained since objects use their own information to fulfill tasks. This usually supports low coupling, which leads to more robust and maintainable systems. Low Coupling is also a GRASP pattern that is discussed in a following section.
- Behavior is distributed across the classes that have the required information, thus encouraging more cohesive "lightweight" class definitions that are easier to understand and maintain. High cohesion is usually supported (another pattern discussed later).

11. See Chapter 33 for a discussion of separation of concerns.

Related Patterns or Principles

- Low Coupling
- High Cohesion

Also Known As; Similar To "Place responsibilities with data," "That which knows, does," "Do It Myself," "Put Services with the Attributes They Work On."

17.12 Low Coupling

Problem How to support low dependency, low change impact, and increased reuse?

Coupling is a measure of how strongly one element is connected to, has knowledge of, or relies on other elements. An element with low (or weak) coupling is not dependent on too many other elements; "too many" is context dependent, but we examine it anyway. These elements include classes, subsystems, systems, and so on.

A class with high (or strong) coupling relies on many other classes. Such classes may be undesirable; some suffer from the following problems:

- Forced local changes because of changes in related classes.
- Harder to understand in isolation.
- Harder to reuse because its use requires the additional presence of the classes on which it is dependent.

Solution Assign a responsibility so that coupling remains low. Use this principle to evaluate alternatives.

Example Consider the following partial class diagram from a NextGen case study:

Payment
Register
Sale

Assume we need to create a *Payment* instance and associate it with the *Sale*. What class should be responsible for this? Since a *Register* "records" a *Payment* in the real-world domain, the Creator pattern suggests *Register* as a candidate for creating the *Payment*. The *Register* instance could then send an *addPayment* message to the *Sale*, passing along the new *Payment* as a parameter. A possible partial interaction diagram reflecting this is shown in Figure 17.18.

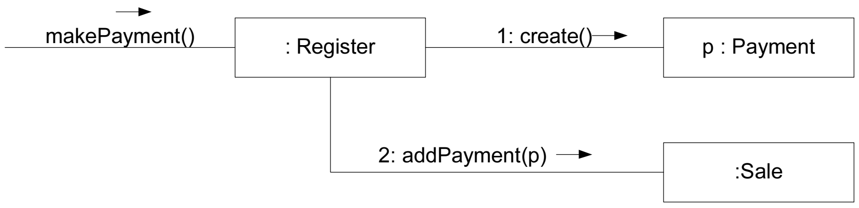


Figure 17.18 Register creates Payment.

This assignment of responsibilities couples the *Register* class to knowledge of the *Payment* class.

Applying UML: Note that the *Payment* instance is explicitly named *p* so that in message 2 it can be referenced as a parameter.

Figure 17.19 shows an alternative solution to creating the *Payment* and associating it with the *Sale*.

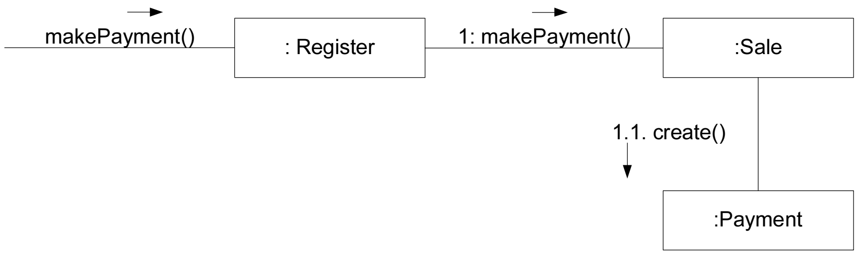


Figure 17.19 Sale creates Payment.

Which design, based on assignment of responsibilities, supports Low Coupling? In both cases we assume the *Sale* must eventually be coupled to knowledge of a *Payment*. Design 1, in which the *Register* creates the *Payment,* adds coupling of *Register* to *Payment*; Design 2, in which the *Sale* does the creation of a *Payment*, does not increase the coupling. Purely from the point of view of coupling, prefer Design 2 because it maintains overall lower coupling. This example illustrates how two patterns—Low Coupling and Creator—may suggest different solutions.

In practice, the level of coupling alone can't be considered in isolation from other principles such as Expert and High Cohesion. Nevertheless, it is one factor to consider in improving a design.

Discussion Low Coupling is a principle to keep in mind during all design decisions; it is an underlying goal to continually consider. It is an **evaluative principle** that you apply while evaluating all design decisions.

In object-oriented languages such as C++, Java, and C#, common forms of coupling from *TypeX* to *TypeY* include the following:

- *TypeX* has an attribute (data member or instance variable) that refers to a *TypeY* instance, or *TypeY* itself.
- A *TypeX* object calls on services of a *TypeY* object.
- *TypeX* has a method that references an instance of *TypeY*, or *TypeY* itself, by any means. These typically include a parameter or local variable of type *TypeY*, or the object returned from a message being an instance of *TypeY*.
- *TypeX* is a direct or indirect subclass of *TypeY*.
- *TypeY* is an interface, and *TypeX* implements that interface.

Low Coupling encourages you to assign a responsibility so that its placement does not increase the coupling to a level that leads to the negative results that high coupling can produce.

Low Coupling supports the design of classes that are more independent, which reduces the impact of change. It can't be considered in isolation from other patterns such as Expert and High Cohesion, but rather needs to be included as one of several design principles that influence a choice in assigning a responsibility.

A subclass is strongly coupled to its superclass. Consider carefully any decision to derive from a superclass since it is such a strong form of coupling. For example, suppose that objects must be stored persistently in a relational or object database. In this case, you could follow the relatively common design practice of creating an abstract superclass called *PersistentObject* from which other classes derive. The disadvantage of this subclassing is that it highly couples domain objects to a particular technical service and mixes different architectural concerns, whereas the advantage is automatic inheritance of persistence behavior.

You cannot obtain an absolute measure of when coupling is too high. What is important is that you can gauge the current degree of coupling and assess whether increasing it will lead to problems. In general, classes that are inherently generic in nature and with a high probability for reuse should have especially low coupling.

The extreme case of Low Coupling is no coupling between classes. This case offends against a central metaphor of object technology: a system of connected objects that communicate via messages. Low Coupling taken to excess yields a poor design—one with a few incohesive, bloated, and complex active objects that do all the work, and with many passive zero-coupled objects that act as simple data repositories. Some moderate degree of coupling between classes is normal and necessary for creating an object-oriented system in which tasks are fulfilled by a collaboration between connected objects.

Contraindications High coupling to stable elements and to pervasive elements is seldom a problem. For example, a J2EE application can safely couple itself to the Java libraries (*java.util,* and so on), because they are stable and widespread.

Pick Your Battles

It is not high coupling per se that is the problem; it is high coupling to elements that are *unstable* in some dimension, such as their interface, implementation, or mere presence.

This is an important point: As designers, we can add flexibility, encapsulate details and implementations, and in general design for lower coupling in many areas of the system. But, if we put effort into "future proofing" or lowering the coupling when we have no realistic motivation, this is not time well spent.

You must pick your battles in lowering coupling and encapsulating things. Focus on the points of realistic high instability or evolution. For example, in the NextGen project, we know that different third-party tax calculators (with unique interfaces) need to be connected to the system. Therefore, designing for low coupling at this variation point is practical.

Benefits

- not affected by changes in other components
- simple to understand in isolation
- convenient to reuse

Background Coupling and cohesion are truly fundamental principles in design, and should be appreciated and applied as such by all software developers. Larry Constantine, also a founder of structured design in the 1970s and a current advocate of more attention to usability engineering [CL99], was primarily responsible in the 1960s for identifying and communicating coupling and cohesion as critical principles [Constantine68, CMS74].

Related Patterns

- Protected Variation

17.13 Controller

Problem What first object beyond the UI layer receives and coordinates ("controls") a system operation?

System operations were first explored during the analysis of SSD. These are the major input events upon our system. For example, when a cashier using a POS terminal presses the "End Sale" button, he is generating a system event indicating "the sale has ended." Similarly, when a writer using a word processor presses the "spell check" button, he is generating a system event indicating "perform a spell check."

A **controller** is the first object beyond the UI layer that is responsible for receiving or handling a system operation message.

Solution Assign the responsibility to a class representing one of the following choices:

- Represents the overall "system," a "root object," a device that the software is running within, or a major subsystem—these are all variations of a *facade controller.*
- Represents a use case scenario within which the system event occurs, often named <UseCaseName>Handler, <UseCaseName>Coordinator, or <UseCaseName>Session (*use case or session controller*).
    - Use the same controller class for all system events in the same use case scenario.
    - Informally, a session is an instance of a conversation with an actor. Sessions can be of any length but are often organized in terms of use cases (use case sessions).

Corollary: Note that "window," "view," and "document" classes are not on this list. Such classes should *not* fulfill the tasks associated with system events; they typically receive these events and delegate them to a controller.

Example Some get a better sense of applying this pattern with code examples. Look ahead in the Implementation section on p. 309 for Java examples of both rich client and Web UIs.

The NextGen application contains several system operations, as illustrated in Figure 17.20. This model shows the system itself as a class (which is legal and sometimes useful when modeling).

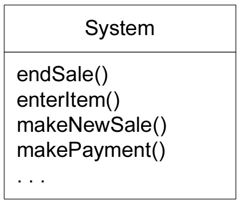


Figure 17.20 Some system operations of the NextGen POS application.

During analysis, system operations may be assigned to the class *System* in some analysis model, to indicate they are system operations. However, this does *not* mean that a software class named *System* fulfills them during design. Rather, during design, a controller class is assigned the responsibility for system operations (see Figure 17.21).

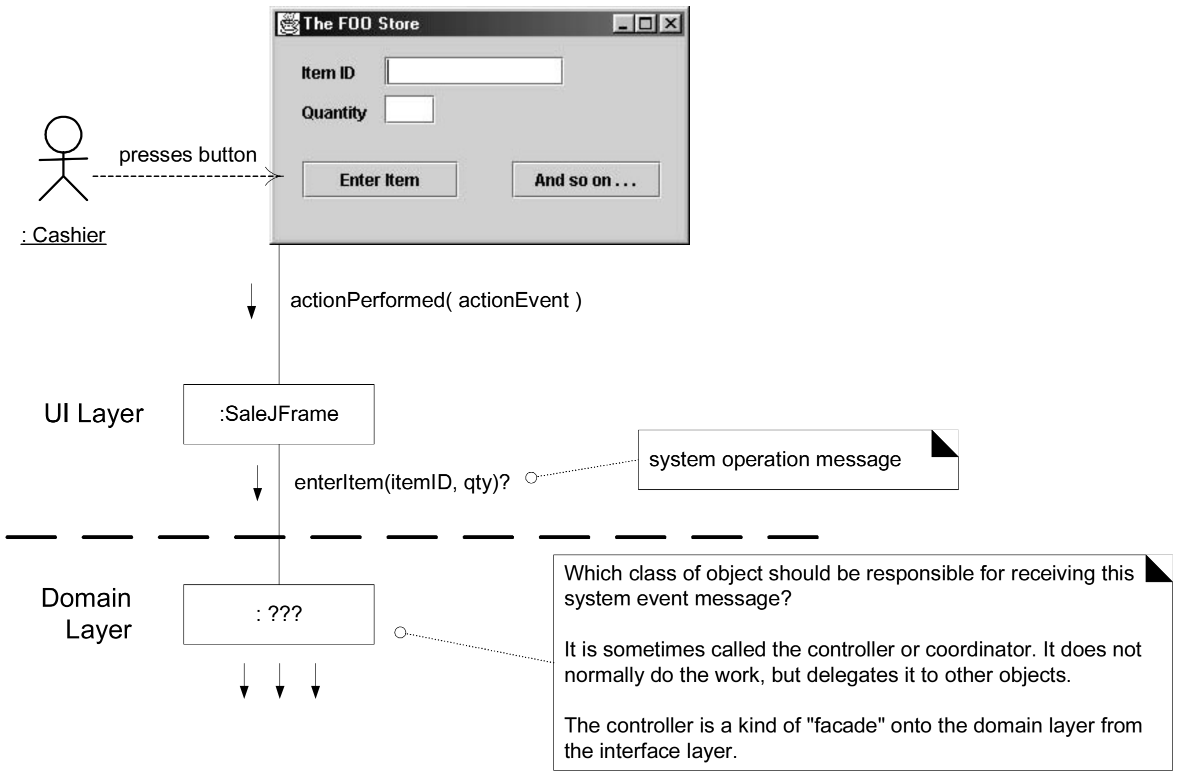

The FOO Store
Item ID
Quantity
Enter Item
And so on . . .
presses button
: Cashier
actionPerformed(actionEvent)
UI Layer
:SaleJFrame
enterItem(itemID, qty)?
system operation message
Domain Layer
: ???
Which class of object should be responsible for receiving this system event message?
It is sometimes called the controller or coordinator. It does not normally do the work, but delegates it to other objects.
The controller is a kind of "facade" onto the domain layer from the interface layer.

Figure 17.21 What object should be the Controller for enterItem?

Who should be the controller for system events such as *enterItem* and *endSale*?

By the Controller pattern, here are some choices:

Represents the overall “system,” “root object,” device, or subsystem.	*Register, POSSystem*
Represents a receiver or handler of all system events of a use case scenario.	*ProcessSaleHandler, ProcessSaleSession*

Note that in the domain of POS, a *Register* (called a *POS Terminal*) is a specialized device with software running in it.

In terms of interaction diagrams, one of the examples in Figure 17.22 could be useful.

The choice of which of these classes is the most appropriate controller is influenced by other factors, which the following section explores.

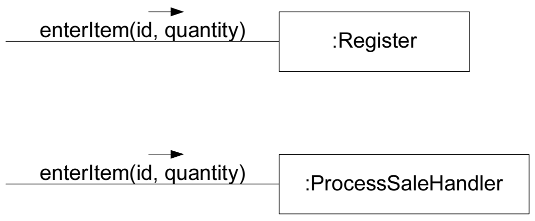


Figure 17.22 Controller choices.

During design, the system operations identified during system behavior analysis are assigned to one or more controller classes, such as *Register*, as shown in Figure 17.23.

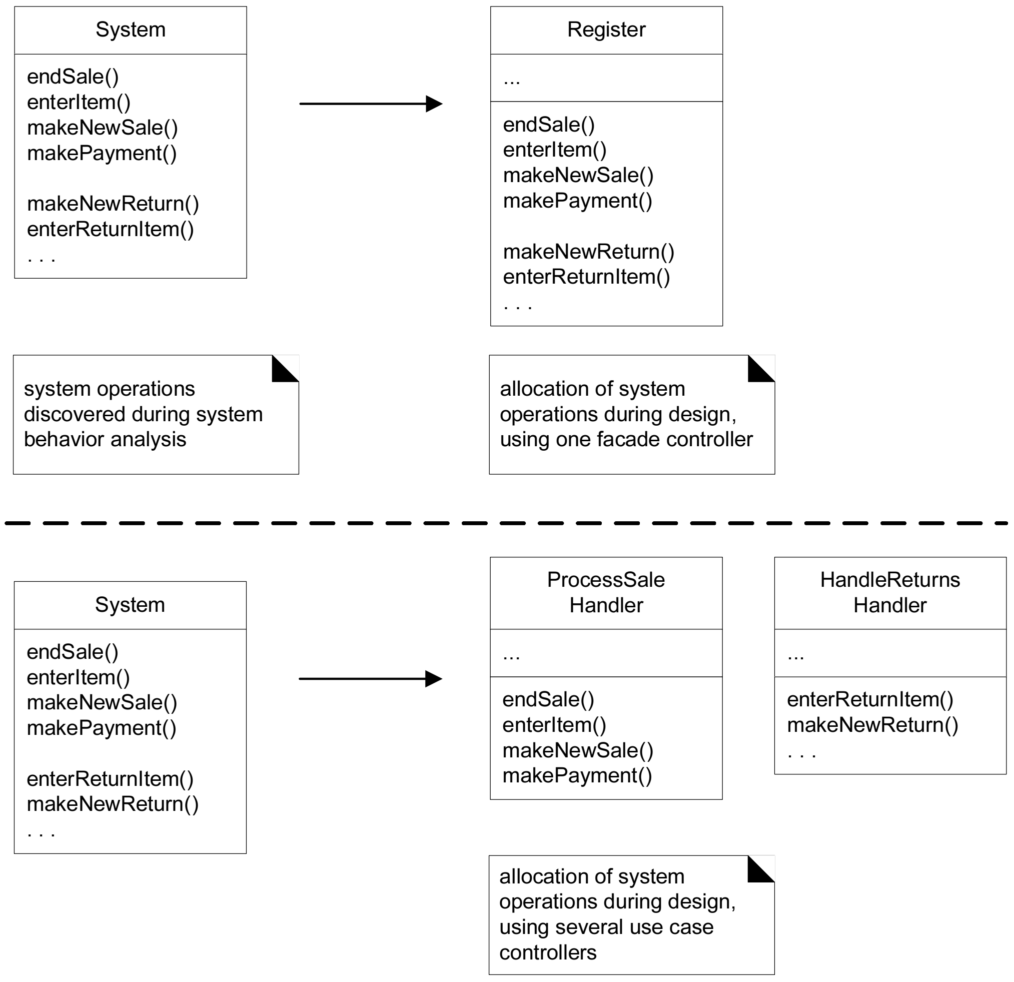


Figure 17.23 Allocation of system operations.

Discussion Some get a better sense of applying this pattern with code examples. Look ahead in the Implementation section on p. 309 for examples in Java for both rich client and Web UIs.

Simply, this is a delegation pattern. In accordance with the understanding that the UI layer shouldn't contain application logic, UI layer objects must delegate work requests to another layer. When the "other layer" is the domain layer, the Controller pattern summarizes common choices that you, as an OO developer, make for the domain object delegate that receives the work requests.

Systems receive external input events, typically involving a GUI operated by a person. Other mediums of input include external messages, such as in a call-processing telecommunications switch or signals from sensors such as in process control systems.

In all cases, you must choose a handler for these events. Turn to the Controller pattern for guidance toward generally accepted, suitable choices. As illustrated in Figure 17.21, the controller is a kind of facade into the domain layer from the UI layer.

You will often want to use the same controller class for all the system events of one use case so that the controller can maintain information about the state of the use case. Such information is useful, for example, to identify out-of-sequence system events (for example, a *makePayment* operation before an *endSale* operation). Different controllers may be used for different use cases.

A common defect in the design of controllers results from over-assignment of responsibility. A controller then suffers from bad (low) cohesion, violating the principle of High Cohesion.

Guideline

Normally, a controller should *delegate* to other objects the work that needs to be done; it coordinates or controls the activity. It does not do much work itself.

Please see the "Issues and Solutions" section for elaboration.

The first category of controller is a facade controller representing the overall system, device, or a subsystem. The idea is to choose some class name that suggests a cover, or facade, over the other layers of the application and that provides the main point of service calls from the UI layer down to other layers. The facade could be an abstraction of the overall physical unit, such as a *Register*[12], *TelecommSwitch, Phone*, or *Robot*; a class representing the entire software sys-

12. Various terms are used for a physical POS unit, including register, point-of-sale terminal (POST), and so forth. Over time, "register" has come to embody the notion of both a physical unit and the logical abstraction of the thing that registers sales and payments.

tem, such as *POSSystem*; or any other concept which the designer chooses to represent the overall system or a subsystem, even, for example, *ChessGame* if it was game software.

Facade controllers are suitable when there are not "too many" system events, or when the user interface (UI) cannot redirect system event messages to alternating controllers, such as in a message-processing system.

If you choose a use case controller, then you will have a different controller for each use case. Note that this kind of controller is not a domain object; it is an artificial construct to support the system (a *Pure Fabrication* in terms of the GRASP patterns). For example, if the NextGen application contains use cases such as *Process Sale* and *Handle Returns*, then there may be a *ProcessSaleHandler* class and so forth.

When should you choose a use case controller? Consider it an alternative when placing the responsibilities in a facade controller leads to designs with low cohesion or high coupling, typically when the facade controller is becoming "bloated" with excessive responsibilities. A use case controller is a good choice when there are many system events across different processes; it factors their handling into manageable separate classes and also provides a basis for knowing and reasoning about the state of the current scenario in progress.

In the UP and Jacobson's older Objectory method [Jacobson92], there are the (optional) concepts of boundary, control, and entity classes. **Boundary objects** are abstractions of the interfaces, **entity objects** are the application-independent (and typically persistent) domain software objects, and **control objects** are use case handlers as described in this Controller pattern.

A important corollary of the Controller pattern is that UI objects (for example, window or button objects) and the UI layer should not have responsibility for fulfilling system events. In other words, system operations should be handled in the application logic or domain layers of objects rather than in the UI layer of a system. See the "Issues and Solutions" section for an example.

Web UIs and Server-Side Application of Controller

Please see p. 310 for a server-side example using Java Struts—a popular framework.

A similar delegation approach can be used in ASP.NET and WebForms: The "code behind" file that contains event handlers for Web browser button clicks will obtain a reference to a domain controller object (e.g., a *Register* object in the POS case study), and then delegate the request for work. This is in contrast to the common, fragile style of ASP.NET programming in which developers insert application logic handling in the "code behind" file, thus mixing application logic into the UI layer.

Server-side Web UI frameworks (such as Struts) embody the concept of the Web-MVC (Model-View-Controller) pattern. The "controller" in Web-MVC differs from this GRASP controller. The former is part of the UI layer and controls the

UI interaction and page flow. The GRASP controller is part of the domain layer and controls or coordinates the handling of the work request, essentially unaware of what UI technology is being used (e.g., a Web UI, a Swing UI, ...).

Also common with server-side designs when Java technologies are used is delegation from the Web UI layer (e.g., from a Struts *Action* class) to an Enterprise JavaBeans (**EJB**) *Session* object. Variant #2 of the Controller pattern—an object representing a user session or use case scenario—covers this case. In this case, the EJB *Session* object may itself delegate farther on to the domain layer of objects, and again, you can apply the Controller pattern to choose a suitable receiver in the pure domain layer.

All that said, the appropriate handling of server-side systems operations is strongly influenced by the chosen server technical frameworks and continues to be a moving target. But the underlying principle of Model-View Separation can and does still apply.

Even with a rich-client UI (e.g., a Swing UI) that interacts with a server, the Controller pattern still applies. The client-side UI forwards the request to the local client-side controller, and the controller forwards all or part of the request handling to remote services. This design lowers the coupling of the UI to remote services and makes it easier, for example, to provide the services either locally or remotely, through the *indirection* of the client-side controller.

Benefits

- *Increased potential for reuse* and *pluggable interfaces*—These benefits ensure that application logic is *not* handled in the interface layer. The responsibilities of a controller could technically be handled in an interface object, but such a design implies that program code and the fulfillment of application logic would be embedded in interface or window objects. An interface-as-controller design reduces the opportunity to reuse logic in future applications, since logic that is bound to a particular interface (for example, window-like objects) is seldom applicable in other applications. By contrast, delegating a system operation responsibility to a controller supports the reuse of the logic in future applications. And since the application logic is not bound to the interface layer, it can be replaced with a different interface.
- *Opportunity to reason about the state of the use case*—Sometimes we must ensure that system operations occur in a legal sequence, or we want to be able to reason about the current state of activity and operations within the use case that is underway. For example, we may have to guarantee that the *makePayment* operation cannot occur until the *endSale* operation has occurred. If so, we need to capture this state information somewhere; the controller is one reasonable choice, especially if we use the same controller throughout the use case (as recommended).

Implementation The following examples use Java technologies for two common cases, a rich client in Java Swing and a Web UI with Struts on the server (a Servlet engine).

Please note that you should apply a similar approach in .NET **WinForms** and ASP.NET **WebForms**. A good practice in well-designed .NET (often ignored by MS programmers who violate the Model-View Separation Principle) is to *not* insert application logic code in the event handlers or in the "code behind" files (those are both part of the UI layer). Rather, in the .NET event handlers or "code behind" files, simply obtain a reference to a domain object (e.g., a *Register* object), and delegate to it.

Implementation with Java Swing: Rich Client UI

This section assumes you are familiar with basic Swing. The code contains comments to explain the key points. A few comments: Notice at ❶ that the *ProcessSaleJFrame* window has a reference to the domain controller object, the *Register*. At ❷ I define the handler for the button click. At ❸ I show the key message—sending the *enterItem* message to the controller in the domain layer.

```
package com.craiglarman.nextgen.ui.swing;

   // imports…


   // in Java, a JFrame is a typical window
public class ProcessSaleJFrame extends JFrame
{

   // the window has a reference to the 'controller' domain object

❶ private Register register;


   // the window is passed the register, on creation
public ProcessSaleJFrame(Register _register)
{
   register = _register;
}

   // this button is clicked to perform the
   // system operation "enterItem"
private JButton BTN_ENTER_ITEM;


   // this is the important method!
   // here i show the message from the UI layer to domain layer
private JButton getBTN_ENTER_ITEM()
{
      // does the button exist?
   if (BTN_ENTER_ITEM != null)
      return BTN_ENTER_ITEM;


      // ELSE button needs to be initialized...
   BTN_ENTER_ITEM = new JButton();
   BTN_ENTER_ITEM.setText("Enter Item");


      // THIS IS THE KEY SECTION!
      // in Java, this is how you define
      // a click handler for a button
❷     BTN_ENTER_ITEM.addActionListener(new ActionListener()
         {
         public void actionPerformed(ActionEvent e)
         {
                 // Transformer is a utility class to
                 // transform Strings to other data types
                 // because the JTextField GUI widgets have Strings
              ItemID id = Transformer.toItemID(getTXT_ID().getText());
              int qty = Transformer.toInt(getTXT_QTY().getText());


                 // here we cross the boundary from the
                 // UI layer to the domain layer
                 // delegate to the 'controller'
                 // > > > THIS IS THE KEY STATEMENT < < <

❸             register.enterItem(id, qty);
         }
         } ); // end of the addActionListener call

      return BTN_ENTER_ITEM;
      } // end of method

   // …
   } // end of class
```

Implementation with Java Struts: Client Browser and WebUI

This section assumes you are familiar with basic Struts. Notice at ❶ that to obtain a reference to the *Register* domain object on the server side, the *Action* object must dig into the Servlet context. At ❷ I show the key message—sending the *enterItem* message to the domain controller object in the domain layer.

```
package com.craiglarman.nextgen.ui.web;

// … imports


   // in Struts, an Action object is associated with a
   // web browser button click, and invoked (on the server)
   // when the button is clicked.
public class EnterItemAction extends Action {


   // this is the method invoked on the server
   // when the button is clicked on the client browser
public ActionForward execute( ActionMapping mapping,
                                ActionForm form,
                                HttpServletRequest request,
                                HttpServletResponse response )
                              throws Exception
{

      // the server has a Repository object that
      // holds references to several things, including
      // the POS "register" object
   Repository repository = (Repository)getServlet().
      getServletContext().getAttribute(Constants.REPOSITORY_KEY);
❶    Register register = repository.getRegister();


         // extract the itemID and qty from the web form
     String txtId = ((SaleForm)form).getItemID();
     String txtQty = ((SaleForm)form).getQuantity();

         // Transformer is a utility class to
         // transform Strings to other data types
     ItemID id = Transformer.toItemID(txtId);
     int qty = Transformer.toInt(txtQty);


         // here we cross the boundary from the
         // UI layer to the domain layer
         // delegate to the 'domain controller'
         // > > > THIS IS THE KEY STATEMENT < < <

❷    register.enterItem(id, qty);

     // …
  } // end of method
  } // end of class
```

Issues and Solutions

Bloated Controllers

Poorly designed, a controller class will have low cohesion—unfocused and handling too many areas of responsibility; this is called a **bloated controller**. Signs of bloating are:

- There is only a *single* controller class receiving *all* system events in the system, and there are many of them. This sometimes happens if a facade controller is chosen.
- The controller itself performs many of the tasks necessary to fulfill the system event, without delegating the work. This usually involves a violation of Information Expert and High Cohesion.
- A controller has many attributes, and it maintains significant information about the system or domain, which should have been distributed to other objects, or it duplicates information found elsewhere.

Among the cures for a bloated controller are these two:

1. Add more controllers—a system does not have to need only one. Instead of facade controllers, employ use case controllers. For example, consider an application with many system events, such as an airline reservation system.

It may contain the following controllers:

Use case controllers
MakeReservationHandler
ManageSchedulesHandler
ManageFaresHandler

2. Design the controller so that it primarily delegates the fulfillment of each system operation responsibility to other objects.

UI Layer Does Not Handle System Events

To reiterate: An important corollary of the Controller pattern is that UI objects (for example, window objects) and the UI layer should not have responsibility for handling system events. As an example, consider a design in Java that uses a *JFrame* to display the information.

Assume the NextGen application has a window that displays sale information and captures cashier operations. Using the Controller pattern, Figure 17.24 illustrates an acceptable relationship between the *JFrame* and the controller and other objects in a portion of the POS system (with simplifications).

Notice that the *SaleJFrame* class—part of the UI layer—delegates the *enterItem* request to the *Register* object. It did not get involved in processing the operation or deciding how to handle it; the window only delegated it to another layer.

Assigning the responsibility for system operations to objects in the application or domain layer by using the Controller pattern rather than the UI layer can increase reuse potential. If a UI layer object (like the *SaleJFrame*) handles a system operation that represents part of a business process, then business process logic would be contained in an interface (for example, window-like) object; the opportunity for reuse of the business logic then diminishes because of its coupling to a particular interface and application. Consequently, the design in Figure 17.25 is undesirable.

Placing system operation responsibility in a domain object controller makes it easier to reuse the program logic supporting the associated business process in future applications. It also makes it easier to unplug the UI layer and use a different UI framework or technology, or to run the system in an offline "batch" mode.

Message Handling Systems and the Command Pattern

Some applications are message-handling systems or servers that receive requests from other processes. A telecommunications switch is a common example. In such systems, the design of the interface and controller is somewhat different. The details are explored in a later chapter, but in essence, a common solution is to use the Command pattern [GHJV95] and Command Processor pattern [BMRSS96], introduced in Chapter 37.

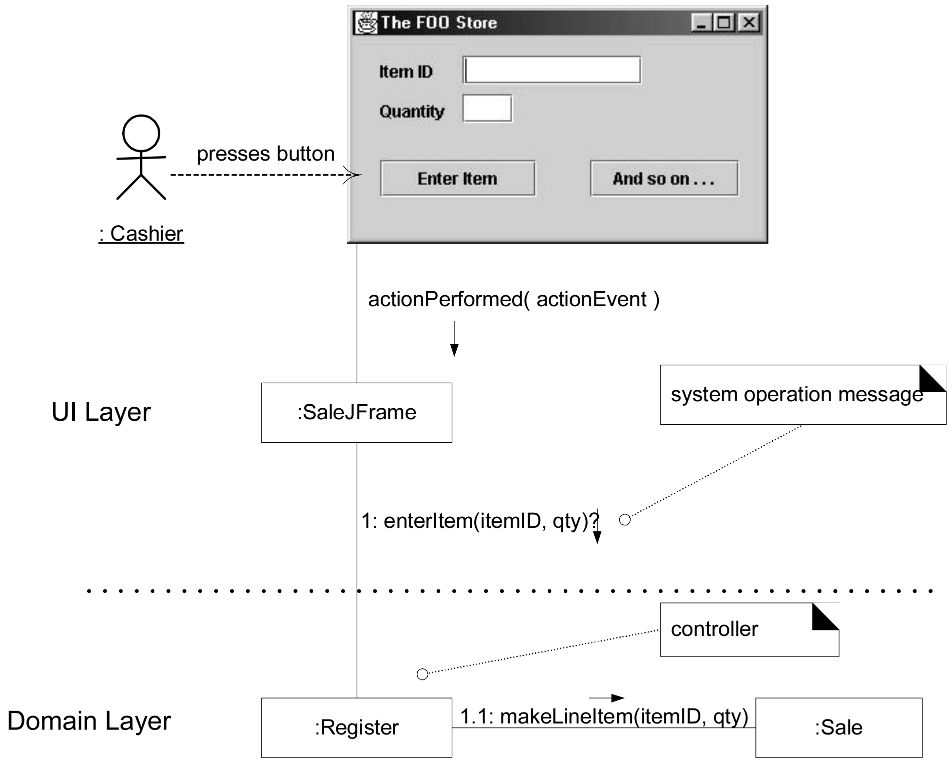


Figure 17.24 Desirable coupling of UI layer to domain layer.

Related Patterns

- **Command**—In a message-handling system, each message may be represented and handled by a separate Command object [GHJV95].
- **Facade**—A facade controller is a kind of Facade [GHJV95].
- **Layers**—This is a POSA pattern [BMRSS96]. Placing domain logic in the domain layer rather than the presentation layer is part of the Layers pattern.
- **Pure Fabrication**—This GRASP pattern is an arbitrary creation of the designer, not a software class whose name is inspired by the Domain Model. A use case controller is a kind of Pure Fabrication.

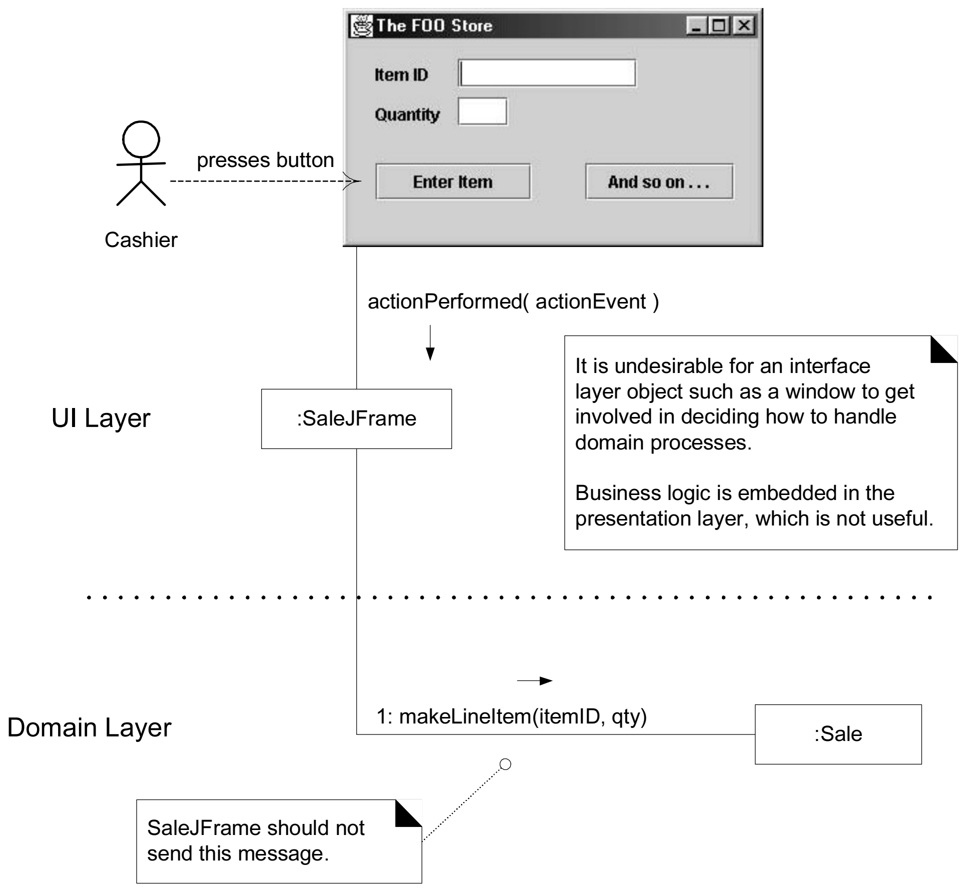

The FOO Store
Item ID
Quantity
Enter Item
And so on . . .
presses button
Cashier
actionPerformed(actionEvent)
UI Layer
:SaleJFrame
It is undesirable for an interface layer object such as a window to get involved in deciding how to handle domain processes.
Business logic is embedded in the presentation layer, which is not useful.
Domain Layer
1: makeLineItem(itemID, qty)
:Sale
SaleJFrame should not send this message.

Figure 17.25 Less desirable coupling of interface layer to domain layer.

17.14 High Cohesion

Problem How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?

In terms of object design, **cohesion** (or more specifically, functional cohesion) is a measure of how strongly related and focused the responsibilities of an element are. An element with highly related responsibilities that does not do a tremendous amount of work has high cohesion. These elements include classes, subsystems, and so on.

Solution Assign a responsibility so that cohesion remains high. Use this to evaluate alternatives.

A class with low cohesion does many unrelated things or does too much work. Such classes are undesirable; they suffer from the following problems:

- hard to comprehend
- hard to reuse
- hard to maintain
- delicate; constantly affected by change

Low cohesion classes often represent a very "large grain" of abstraction or have taken on responsibilities that should have been delegated to other objects.

Example Let's take another look at the example problem used in the Low Coupling pattern and analyze it for High Cohesion.

Assume we have a need to create a (cash) *Payment* instance and associate it with the *Sale*. What class should be responsible for this? Since *Register* records a *Payment* in the real-world domain, the Creator pattern suggests *Register* as a candidate for creating the *Payment*. The *Register* instance could then send an *addPayment* message to the *Sale*, passing along the new *Payment* as a parameter, as shown in Figure 17.26.

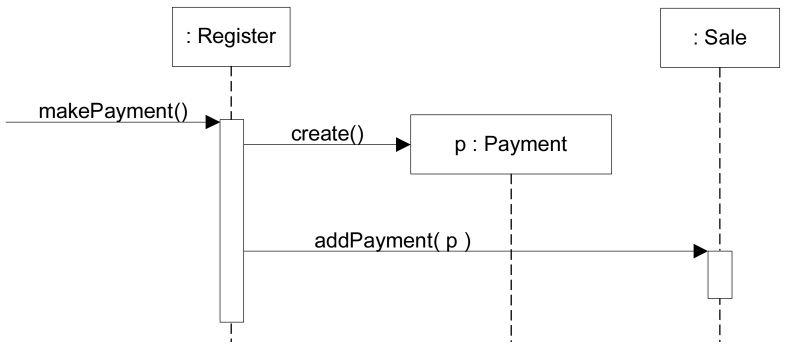


Figure 17.26 Register creates Payment.

This assignment of responsibilities places the responsibility for making a payment in the *Register*. The *Register* is taking on part of the responsibility for fulfilling the *makePayment* system operation.

In this isolated example, this is acceptable; but if we continue to make the *Register* class responsible for doing some or most of the work related to more and more system operations, it will become increasingly burdened with tasks and become incohesive.

Imagine fifty system operations, all received by *Register*. If *Register* did the work related to each, it would become a "bloated" incohesive object. The point is not that this single *Payment* creation task in itself makes the *Register* incohesive, but as part of a larger picture of overall responsibility assignment, it may suggest a trend toward low cohesion.

And most important in terms of developing skills as object designers, regardless of the final design choice, is the valuable achievement that at least we know to consider the impact on cohesion.

By contrast, as shown in Figure 17.27, the second design delegates the payment creation responsibility to the *Sale* supports higher cohesion in the *Register.*

Since the second design supports both high cohesion and low coupling, it is desirable.

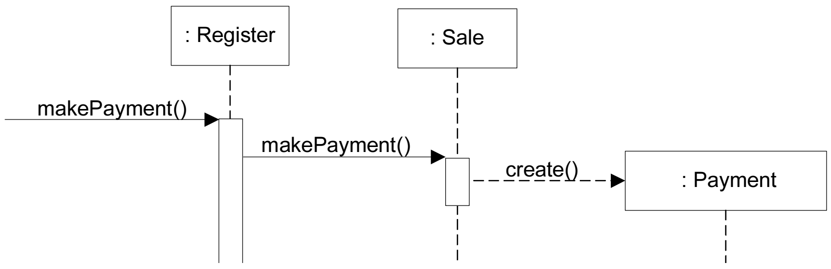


Figure 17.27 Sale creates Payment.

In practice, the level of cohesion alone can't be considered in isolation from other responsibilities and other principles such as Expert and Low Coupling.

Discussion Like Low Coupling, High Cohesion is a principle to keep in mind during all design decisions; it is an underlying goal to continually consider. It is an evaluative principle that a designer applies while evaluating all design decisions.

Grady Booch describes high functional cohesion as existing when the elements of a component (such as a class) "all work together to provide some well-bounded behavior" [Booch94].

Here are some scenarios that illustrate varying degrees of functional cohesion:

1. *Very low cohesion*—A class is solely responsible for many things in very different functional areas.
    - ❍ Assume the existence of a class called *RDB-RPC-Interface* which is completely responsible for interacting with relational databases and for handling remote procedure calls. These are two vastly different functional areas, and each requires lots of supporting code. The responsibilities should be split into a family of classes related to RDB access and a family related to RPC support.
2. *Low cohesion*—A class has sole responsibility for a complex task in one functional area.
    - ❍ Assume the existence of a class called *RDBInterface* which is completely responsible for interacting with relational databases. The

methods of the class are all related, but there are lots of them, and a tremendous amount of supporting code; there may be hundreds or thousands of methods. The class should split into a family of lightweight classes sharing the work to provide RDB access.

3. *High cohesion*—A class has moderate responsibilities in one functional area and collaborates with other classes to fulfill tasks.
    - Assume the existence of a class called *RDBInterface* that is only partially responsible for interacting with relational databases. It interacts with a dozen other classes related to RDB access in order to retrieve and save objects.
4. *Moderate cohesion*—A class has lightweight and sole responsibilities in a few different areas that are logically related to the class concept but not to each other.
    - Assume the existence of a class called *Company* that is completely responsible for (a) knowing its employees and (b) knowing its financial information. These two areas are not strongly related to each other, although both are logically related to the concept of a company. In addition, the total number of public methods is small, as is the amount of supporting code.

As a rule of thumb, a class with high cohesion has a relatively small number of methods, with highly related functionality, and does not do too much work. It collaborates with other objects to share the effort if the task is large.

A class with high cohesion is advantageous because it is relatively easy to maintain, understand, and reuse. The high degree of related functionality, combined with a small number of operations, also simplifies maintenance and enhancements. The fine grain of highly related functionality also supports increased reuse potential.

The High Cohesion pattern—like many things in object technology—has a real-world analogy. It is a common observation that if a person takes on too many unrelated responsibilities—especially ones that should properly be delegated to others—then the person is not effective. This is observed in some managers who have not learned how to delegate. These people suffer from low cohesion; they are ready to become "unglued."

Another Classic Principle: Modular Design

Coupling and cohesion are old principles in software design; designing with objects does not imply ignoring well-established fundamentals. Another of these—which is strongly related to coupling and cohesion—is to promote **modular design**. To quote:

> Modularity is the property of a system that has been decomposed into a set of cohesive and loosely coupled modules [Booch94].

We promote a modular design by creating methods and classes with high cohesion. At the basic object level, we achieve modularity by designing each method with a clear, single purpose and by grouping a related set of concerns into a class.

Cohesion and Coupling; Yin and Yang

Bad cohesion usually begets bad coupling, and vice versa. I call cohesion and coupling the *yin and yang of software engineering* because of their interdependent influence. For example, consider a GUI widget class that represents and paints a widget, saves data to a database, and invokes remote object services. Not only is it profoundly incohesive, but it is coupled to many (and disparate) elements.

Contraindications In a few cases, accepting lower cohesion is justified.

One case is the grouping of responsibilities or code into one class or component to simplify maintenance by one person—although be warned that such grouping may also worsen maintenance. But suppose an application contains embedded SQL statements that by other good design principles should be distributed across ten classes, such as ten "database mapper" classes. Now, commonly only one or two SQL experts know how to best define and maintain this SQL. Even if dozens of object-oriented (OO) programmers work on the project, few OO programmers may have strong SQL skills. Suppose the SQL expert is not even a comfortable OO programmer. The software architect may decide to group all the SQL statements into one class, *RDBOperations*, so that it is easy for the SQL expert to work on the SQL in one location.

Another case for components with lower cohesion is with distributed server objects. Because of overhead and performance implications associated with remote objects and remote communication, it is sometimes desirable to create fewer and larger, less cohesive server objects that provide an interface for many operations. This approach is also related to the pattern called **Coarse-Grained Remote Interface**. In that pattern the remote operations are made more coarse-grained so that they can to do or request more work in remote operation calls to alleviate the performance penalty of remote calls over a network. As a simple example, instead of a remote object with three fine-grained operations *setName*, *setSalary*, and *setHireDate*, there is one remote operation, *setData*, which receives a set of data. This results in fewer remote calls and better performance.

Benefits

- Clarity and ease of comprehension of the design is increased.
- Maintenance and enhancements are simplified.
- Low coupling is often supported.
- Reuse of fine-grained, highly related functionality is increased because a cohesive class can be used for a very specific purpose.

17.15 Recommended Resources

The metaphor of RDD especially emerged from the influential object work in Smalltalk at Tektronix in Portland, from Kent Beck, Ward Cunningham, Rebecca Wirfs-Brock, and others. *Designing Object-Oriented Software* [WWW90] is the landmark text, and is as relevant today as when it was written. Wirfs-Brock has more recently released another RDD text, *Object Design: Roles, Responsibilities, and Collaborations* [WM02].

Two other recommended texts emphasizing fundamental object design principles are *Object-Oriented Design Heuristics* by Riel and *Object Models* by Coad.