24.2. Case Study: Monopoly

Use Cases, etc.

Uses case were skipped, as most know the rules of the game. No update to the SSD is required, and no operations contracts were written.

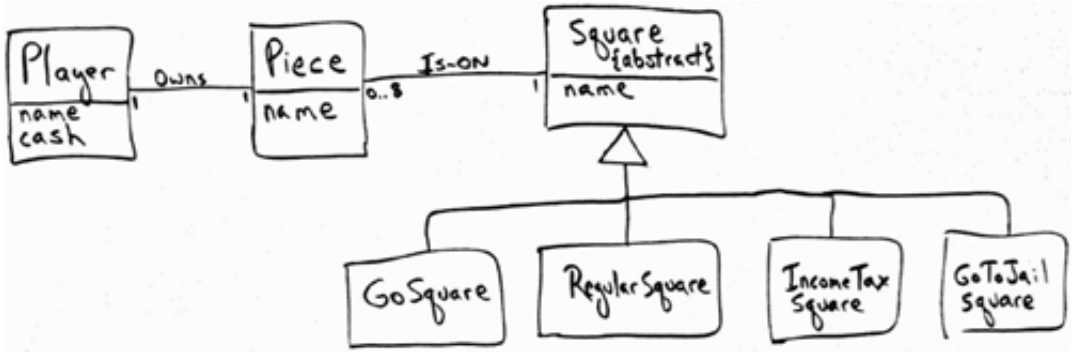
Domain Model

The concepts *Square, GoSquare, IncomeTaxSquare,* and *GoToJailSquare* are all similar—they are variations on a square. In this situation, it is possible (and often useful) to organize them into a **generalization-specialization class hierarchy** (or simply **class hierarchy**) in which the **superclass** *Square* represents a more general concept and the **subclasses** more specialized ones.

In the UML, generalization-specialization relationships are shown with a large triangular arrow pointing from the specialization class to the more general class, as shown in Figure 24.2.

Figure 24.2. Monopoly domain model changes for iteration-2.

[View full size image]



Generalization is the activity of identifying commonality among concepts and defining superclass (general concept) and subclass (specialized concept) relationships. It is a way to construct taxonomic classifications among concepts that are then illustrated in class hierarchies.

The subject of generalization and specialization is covered more thoroughly in a later chapter. See "Generalization" on page 503.

Identifying a superclass and subclasses is of value in a domain model because their presence allows us to understand concepts in more general, refined and abstract terms. It leads to economy of expression, improved comprehension and a reduction in repeated information.

When to show subclasses? The following are common motivations:

Guideline

Create a conceptual subclass of a superclass when:

1. The subclass has additional attributes of interest.
2. The subclass has additional associations of interest.
3. The subclass concept is operated on, handled, reacted to, or manipulated differently than the superclass or other subclasses, in noteworthy ways.

Criteria #3 applies to the case of the different kinds of squares. The *GoSquare* is treated differently than other kinds of squares according to the domain rules. It is a *noteworthy* distinct concept—and the domain model is especially useful as a place to identify noteworthy concepts.

Therefore, an updated domain model is shown in Figure 24.2. Note that each distinct square that is treated differently by the domain rules is shown as a separate class.

Guidelines: A few more domain modeling guidelines and points are illustrated in this model:

- The class *Square* is defined *{abstract}*.
    - ***Guideline***: Declare superclasses abstract. Although this is a conceptual perspective unrelated to software, it is also a common OO guideline that all *software* superclasses be abstract.
- Each subclass name appends the superclass name—*IncomeTaxSquare* rather than *IncomeTax*. That's a good idiom, and also more accurate, as, for example, we really aren't modeling the concept of income tax, but modeling the concept of an income tax square in a monopoly game.
    - ***Guideline***: Append the superclass name to the subclass.
- A *RegularSquare* that does nothing special is also a distinct concept.
- Now that money is involved, the Player has a *cash* attribute.

Introduction

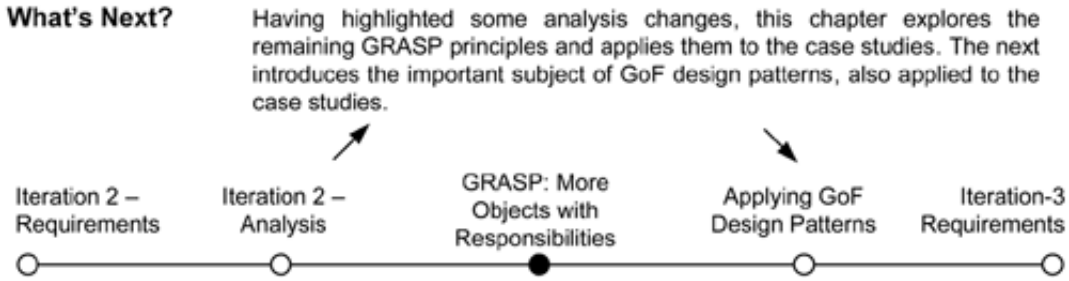
Previously, we applied five GRASP patterns:

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

The final four GRASP patterns are covered in this chapter. They are:

- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

[View full size image]



Once these have been explained, we will have a rich and shared vocabulary with which to discuss designs. And as some of the "Gang-of-Four" (GoF) design patterns (such as Strategy and Abstract Factory) are also introduced in subsequent chapters, that vocabulary will grow. A short sentence, such as "I suggest a Strategy generated from a Abstract Factory to support Protected Variations and low coupling with respect to <X>" communicates lots of information about the design, since pattern names tersely convey a complex design concept.

Subsequent chapters introduce other useful patterns and apply them to the development of the second iteration of the case studies.

25.1. Polymorphism

Problem

How handle alternatives based on type? How to create pluggable software components?

Alternatives based on type— Conditional variation is a fundamental theme in programs. If a program is designed using if-then-else or case statement conditional logic, then if a new variation arises, it requires modification of the case logic—often in many places. This approach makes it difficult to easily extend a program with new variations because changes tend to be required in several places—wherever the conditional logic exists.

Pluggable software components— Viewing components in client-server relationships, how can you replace one server component with another, without affecting the client?

Solution

When related alternatives or behaviors vary by type (class), assign responsibility for the behavior—using polymorphic operations—to the types for which the behavior varies.[1]

[1] Polymorphism has several related meanings. In this context, it means "giving the same name to services in different objects" [Coad95] when the services are similar or related. The different object types usually implement a common interface or are related in an implementation hierarchy with a common superclass, but this is language-dependent; for example, dynamic binding languages such as Smalltalk do not require this.

Corollary: Do not test for the type of an object and use conditional logic to perform varying alternatives based on type.

Examples

NextGen Problem: How Support Third-Party Tax Calculators?

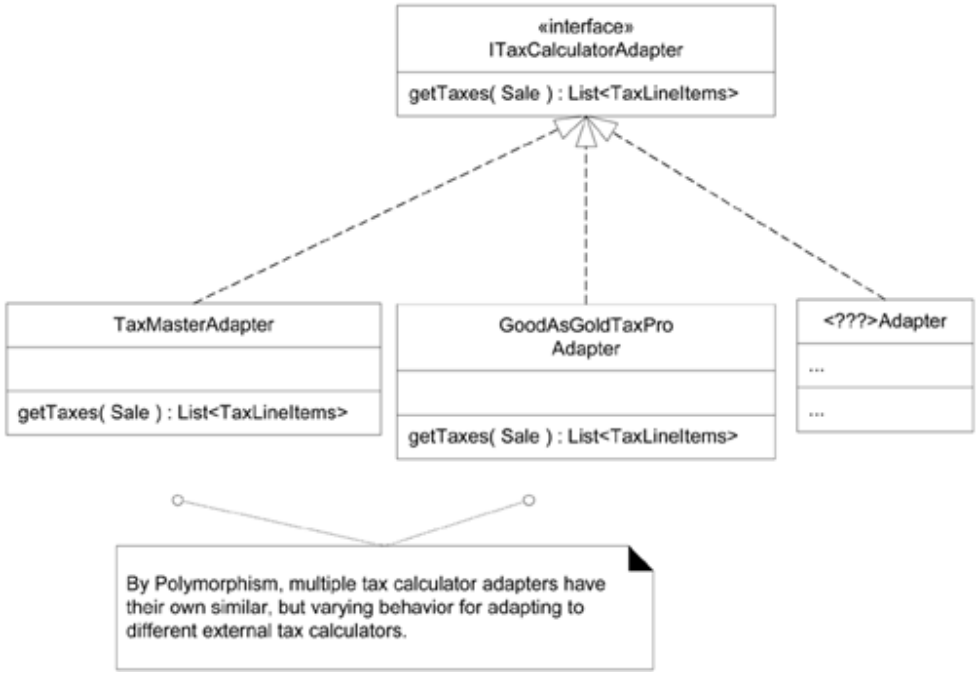
In the NextGen POS application, there are multiple external third-party tax calculators that must be supported (such as Tax-Master and Good-As-Gold TaxPro); the system needs to be able to integrate with different ones. Each tax calculator has a different interface, so there is similar but varying behavior to adapt to each of these external fixed interfaces or APIs. One product may support a raw TCP socket protocol, another may offer a SOAP interface, and a third may offer a Java RMI interface.

What objects should be responsible for handling these varying external tax calculator interfaces?

Since the behavior of calculator adaptation varies by the type of calculator, by Polymorphism we should assign the responsibility for adaptation to different calculator (or calculator adapter) objects themselves, implemented with a polymorphic *getTaxes* operation (see Figure 25.1).

Figure 25.1. Polymorphism in adapting to different external tax calculators.

[View full size image]



These calculator adapter objects are not the external calculators, but rather, local software objects that represent the external calculators, or the adapter for the calculator. By sending a message to the local object, a call will ultimately be made on the external calculator in its native API.

Each *getTaxes* method takes the *Sale* object as a parameter, so that the calculator can analyze the sale. The implementation of each *getTaxes* method will be different: *TaxMasterAdapter* will adapt the request to the API of Tax-Master, and so on.

UML— Notice the interface and interface realization notation in Figure 25.1.

Monopoly Problem: How to Design for Different Square Actions?

To review, when a player lands on the Go square, they receive $200. There's a different action for landing on the Income Tax square, and so forth. Notice that there is a different rule for different types of squares. Let's review the Polymorphism design principle:

> When related alternatives or behaviors vary by type (class), assign responsibility for the behavior—using polymorphic operations—to the types for which the behavior varies. *Corollary*: Do not test for the type of an object and use conditional logic to perform varying alternatives based on type.

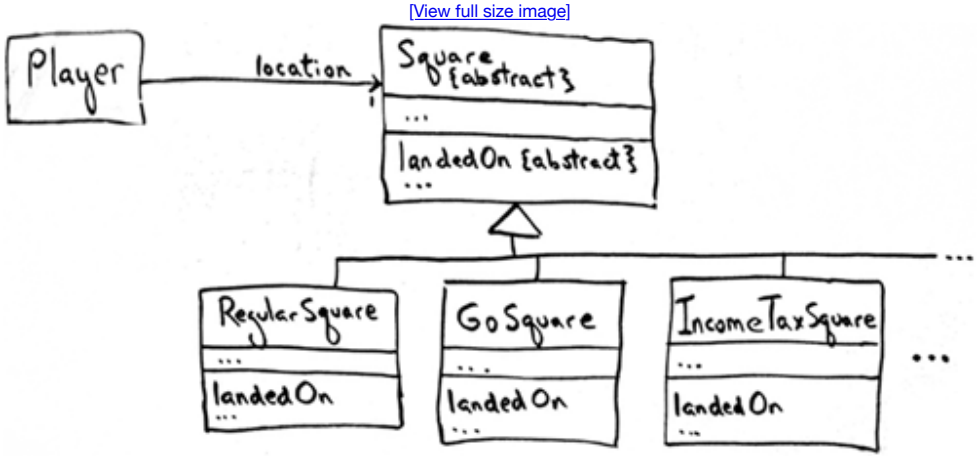
From the corollary, we know we should *not* design with case logic (a *switch* statement in Java or C#) as in the following pseudocode:

```
// bad design
SWITCH ON square.type
CASE GoSquare: player receives $200
CASE IncomeTaxSquare: player pays tax
...
```

Rather, the principle advises us to create a polymorphic operation for each type for which the behavior varies. It varies for the types (classes) *RegularSquare, GoSquare*, and so on. What is the operation that varies? It's what happens when a player lands on a square. Thus, a good name for the polymorphic operation is *landedOn* or some variation. Therefore, by Polymorphism, we'll create a separate class for each kind of square that has a different *landedOn* responsibility, and implement a *landedOn* method in each. Figure 25.2 illustrates the static-view class design.

Figure 25.2. Applying Polymorphism to the Monopoly problem.

[View full size image]



Applying UML: Notice in Figure 25.2 the use of the *{abstract}* keyword for the *landedOn* operation.

Guideline: Unless there is a default behavior in the superclass, declare a polymorphic operation in the superclass to be *{abstract}*.

The remaining interesting problem is the dynamic design: How should the interaction diagrams evolve? What object should send the *landedOn* message to the square that a player lands on? Since a *Player* software object already knows its location square (the one it landed on), then by the principles of Low Coupling and by Expert, class *Player* is a good choice to send the message, as a *Player* already has visibility to the correct square.

Naturally, this message should be sent at the end of the *takeTurn* method. Please review the iteration-1 *takeTurn* design on p. 355 to see our starting point. Figure 25.3 and Figure 25.4 illustrate the evolving dynamic design.

Figure 25.3. Applying Polymorphism.

[View full size image]

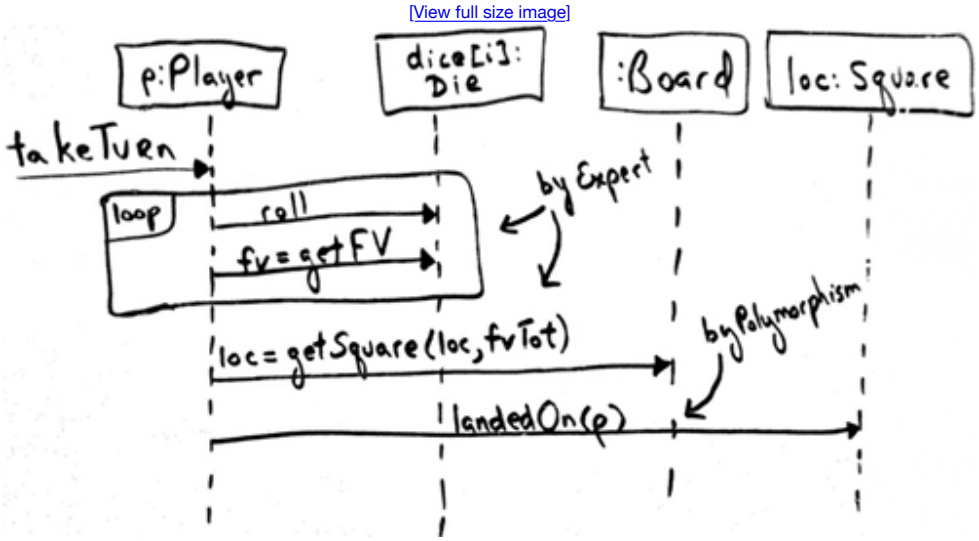
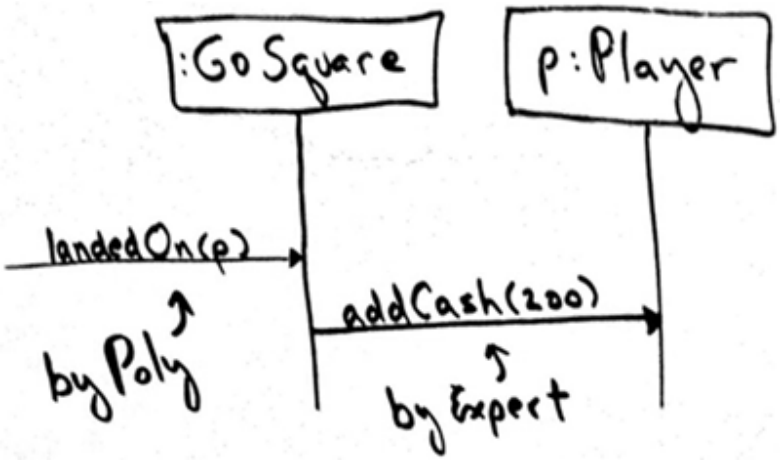


Figure 25.4. The *GoSquare* case.



Applying UML:

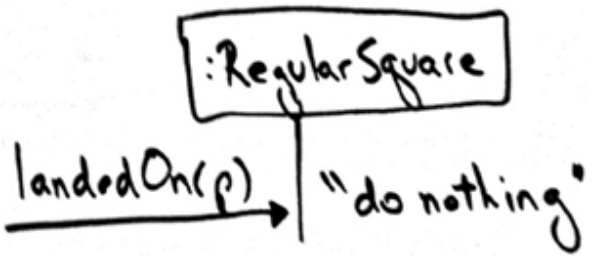
UML frames p. 235

- Notice in Figure 25.3 and Figure 25.4 the informal approach to showing the polymorphic cases in separate diagrams when sketching UML. An alternative—especially when using a UML tool—is to use *sd* and *ref* frames.
- Notice in Figure 25.3 that the *Player* object is labeled 'p' so that in the *landedOn* message we can refer to that object in the parameter list. (You will see in Figure 25.4 that it is useful for the *Square* to have parameter visibility to the *Player*.)
- Notice in Figure 25.3 that the *Square* object is labeled *loc* (short for 'location') and this is the same label as the return value variable in the *getSquare* message. This implies they are the same object.

Let's consider each of the polymorphic cases in terms of GRASP and the design issues:

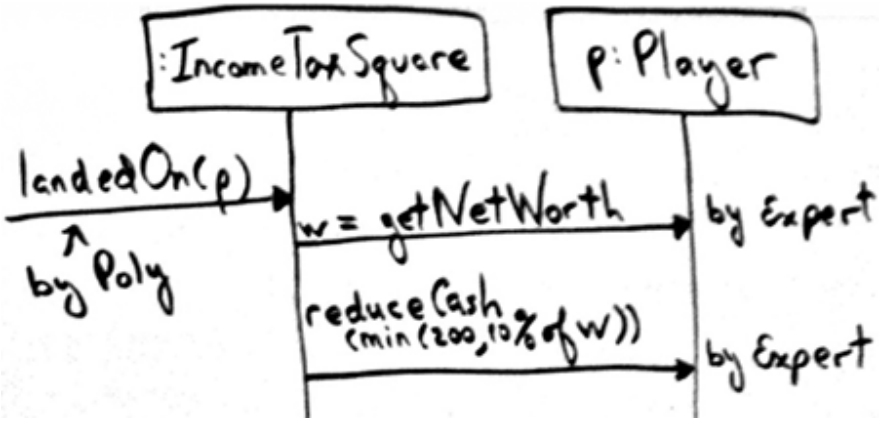
- *GoSquare*— See Figure 25.4. By low representational gap, the *Player* should know its cash. Therefore, by Expert, it should be sent an *addCash* message. Thus the square needs visibility to the *Player* so it can send the message; consequently, the *Player* is passed as a parameter 'p' in the *landedOn* message to achieve parameter visibility.
- *RegularSquare*— See Figure 25.5. In this case, *nothing* happens. I've informally labeled the diagram to indicate this, though a UML note box could be used as well. In code, the body of this method will be empty—sometimes called a NO-OP (no operation) method. Note that to make the magic of polymorphism work, we need to use this approach to avoid special case logic.

Figure 25.5. The *RegularSquare* case.



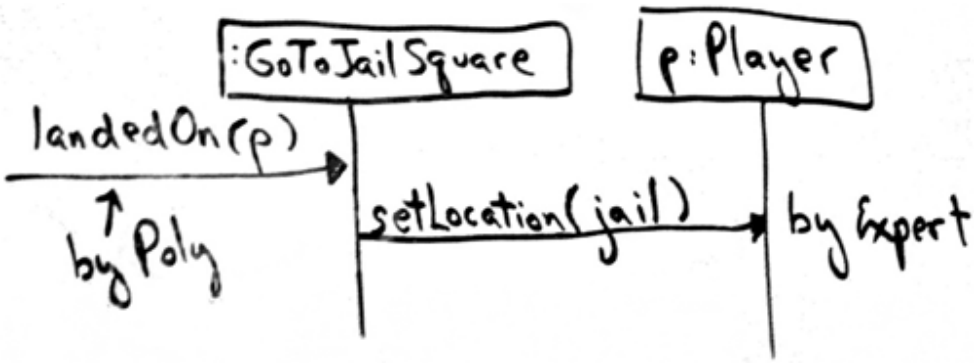
- *IncomeTaxSquare*— See Figure 25.6. We need to calculate 10% of the player's net worth. By low representational gap and by Expert, who should know this? The *Player*. Thus the square asks for the player's worth, and then deducts the appropriate amount.

Figure 25.6. The *IncomeTaxSquare* case.



- *GoToJailSquare*— See Figure 25.7. Simply, the Player's location must be changed. By Expert, it should receive a *setLocation* message. Probably, the *GoToJailSquare* will be initialized with an attribute referencing the *JailSquare*, so that it can pass this square as a parameter to the *Player*.

Figure 25.7. The *GoToJailSquare* case.



UML as Sketch: Notice in Figure 25.4 that the vertical lifeline is drawn as a solid line, rather than the traditional dashed line. This is more convenient when hand sketching. Furthermore, UML 2 allows either format—although in any event conformance to correct UML is not so important when sketching, only that the participants understand each other.

Improving the Coupling

As a small OO design refinement, notice in Figure 18.25 on p. 357 for iteration-1 that the *Piece* remembers the square location but the *Player* does not, and thus the *Player* must extract the location from the *Piece* (to send the *getSquare* message to the *Board*), and then re-assign the new location to the *Piece*. That's a weak design point, and in this iteration, when the *Player* must also send the *landedOn* message to its *Square*, it becomes even weaker. Why? What's wrong with it? Answer: Problems in *coupling*.

Clearly the *Player* needs to permanently know its own *Square* location object rather than the *Piece*, since the *Player* keeps collaborating with its *Square*. You should see this as a refactoring opportunity to improve coupling—when object A keeps needing the data in object B it implies either 1) object A should hold that data, or 2) object B should have the responsibility (by Expert) rather than object A.

Therefore, in iteration-2 I've refined the design so that the *Player* rather than the *Piece* knows its square; this is reflected in both the DCD of Figure 25.2 and the interaction diagram of Figure 25.3.

In fact, one can even question if the *Piece* is a useful object in the Design Model. In the real world, a little plastic piece sitting on the board is a useful proxy for a human, because we're big and go to the kitchen for cold beer! But in software, the *Player* object (being a tiny software blob) can fulfill the role of the *Piece*.

Discussion

Polymorphism is a fundamental principle in designing how a system is organized to handle similar variations. A design based on assigning responsibilities by Polymorphism can be easily extended to handle new variations. For example, adding a new calculator adapter class with its own polymorphic *getTaxes* method will have minor impact on the existing design.

Guideline: When to Design with Interfaces?

Polymorphism implies the presence of abstract superclasses or interfaces in most OO languages. When should you consider using an interface? The general answer is to introduce one when you want to support polymorphism without being committed to a particular class hierarchy. If an abstract superclass AC is used without an interface, any new polymorphic solution must be a subclass of AC, which is very limiting in single-inheritance languages such as Java and C#. As a rule-of-thumb, if there is a class hierarchy with an abstract superclass C1, consider making an interface I1 that corresponds to the public method signatures of C1, and then declare C1 to implement the I1 interface. Then, even if there is no immediate motivation to *avoid* subclassing under C1 for a new polymorphic solution, there is a flexible evolution point for unknown future cases.

Contraindications

Sometimes, developers design systems with interfaces and polymorphism for speculative “future-proofing” against an unknown possible variation. If the variation point is definitely motivated by an immediate or very probable variability, then the effort of adding the flexibility through polymorphism is of course rational. But critical evaluation is required, because it is not uncommon to see unnecessary effort being applied to future-proofing a design with polymorphism at variation points that in fact are improbable and will never actually arise. Be realistic about the true likelihood of variability before investing in increased flexibility.

Benefits

- Extensions required for new variations are easy to add.
- New implementations can be introduced without affecting clients.

Related Patterns

- Protected Variations
- A number of popular GoF design patterns [GHJV95], which will be discussed in this book, rely on polymorphism, including Adapter, Command, Composite, Proxy, State, and Strategy.

Also Known As; Similar To

Choosing Message, Don't Ask “What Kind?”

25.2. Pure Fabrication

Problem

What object should have the responsibility, when you do not want to violate High Cohesion and Low Coupling, or other goals, but solutions offered by Expert (for example) are not appropriate?

Object-oriented designs are sometimes characterized by implementing as software classes representations of concepts in the real-world problem domain to lower the representational gap; for example a *Sale* and *Customer* class. However, there are many situations in which assigning responsibilities only to domain layer software classes leads to problems in terms of poor cohesion or coupling, or low reuse potential.

Solution

Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concept—something made up, to support high cohesion, low coupling, and reuse.

Such a class is a *fabrication* of the imagination. Ideally, the responsibilities assigned to this fabrication support high cohesion and low coupling, so that the design of the fabrication is very clean, or *pure*—hence a pure fabrication.

Finally, in English *pure fabrication* is an idiom that implies making something up, which we do when we're desperate!

Examples

NextGen Problem: Saving a Sale Object in a Database

For example, suppose that support is needed to save *Sale* instances in a relational database. By Information Expert, there is some justification to assign this responsibility to the *Sale* class itself, because the sale has the data that needs to be saved. But consider the following implications:

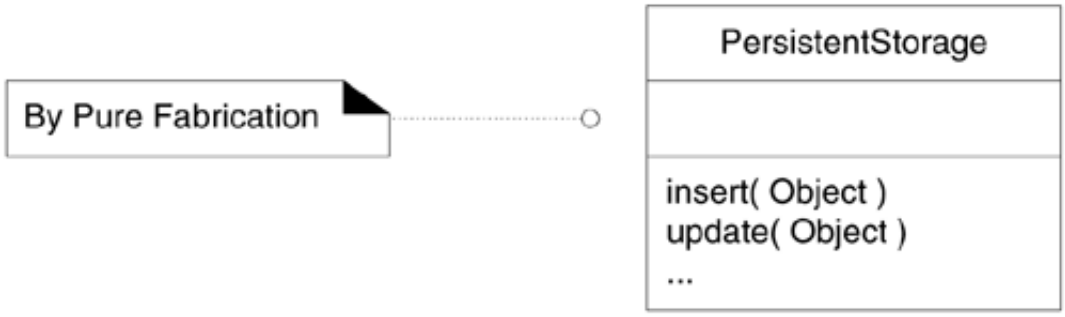
- The task requires a relatively large number of supporting database-oriented operations, none related to the concept of sale-ness, so the *Sale* class becomes incohesive.
- The *Sale* class has to be coupled to the relational database interface (such as JDBC in Java technologies), so its coupling goes up. And the coupling is not even to another domain object, but to a particular kind of database interface.
- Saving objects in a relational database is a very general task for which many classes need support. Placing these responsibilities in the *Sale* class suggests there is going to be poor reuse or lots of duplication in other classes that do the same thing.

Thus, even though *Sale* is a logical candidate by virtue of Information Expert to save itself in a database, it leads to a design with low cohesion, high coupling, and low reuse potential—exactly the kind of desperate situation that calls for making something up.

A reasonable solution is to create a new class that is solely responsible for saving objects in some kind of persistent storage medium, such as a relational database; call it the *PersistentStorage*.[2] This class is a Pure

Fabrication—a figment of the imagination.

[2] In a real persistence framework, more than a single pure fabrication class is ultimately necessary to create a reasonable design. This object will be a front-end facade on to a large number of back-end helper objects.



Notice the name: *PersistentStorage*. This is an understandable concept, yet the name or concept "persistent storage" is not something one would find in the Domain Model. And if a designer asked a business-person in a store, "Do you work with persistent storage objects?" they would not understand. They understand concepts such as "sale" and "payment." *PersistentStorage* is not a domain concept, but something made up or fabricated for the convenience of the software developer.

This Pure Fabrication solves the following design problems:

- The *Sale* remains well-designed, with high cohesion and low coupling.
- The *PersistentStorage* class is itself relatively cohesive, having the sole purpose of storing or inserting objects in a persistent storage medium.
- The *PersistentStorage* class is a very generic and reusable object.

Creating a pure fabrication in this example is exactly the situation in which their use is called for—eliminating a bad design based on Expert, with poor cohesion and coupling, with a good design in which there is greater potential for reuse.

Note that, as with all the GRASP patterns, the emphasis is on where responsibilities should be placed. In this example the responsibilities are shifted from the *Sale* class (motivated by Expert) to a Pure Fabrication.

Monopoly Problem: Handling the Dice

In the refactoring chapter, I used the example of dice rolling behavior (rolling and summing the dice totals) to apply Extract Method (p. 391) in the *Player.takeTurn* method. At the end of the example I also mentioned that the refactored solution itself was not ideal, and a better solution would be presented later.

In the current design, the *Player* rolls all the dice and sums the total. Dice are very general objects, usable in many games. By putting this rolling and summing responsibility in a Monopoly game *Player*, the summing service is not generalized for use in other games. Another weakness: It is not possible to simply ask for the current dice total without rolling the dice again.

But, choosing any other object inspired from the Monopoly game domain model leads to the same problems. And that leads us to Pure Fabrication—make something up to conveniently provide related services.

Although there is no cup for the dice in Monopoly, many games do use a dice cup in which one shakes all the dice and rolls them onto a table. Therefore, I propose a Pure Fabrication called *Cup* (notice that I'm still trying to use similar domain-relevant vocabulary) to hold all the dice, roll them, and know their total. The new design is shown in Figure 25.8 and Figure 25.9. The *Cup* holds a collection of many *Die* objects. When one sends a *roll*

message to a *Cup*, it sends a *roll* message to all its dice.

Figure 25.8. DCD for a *Cup*.

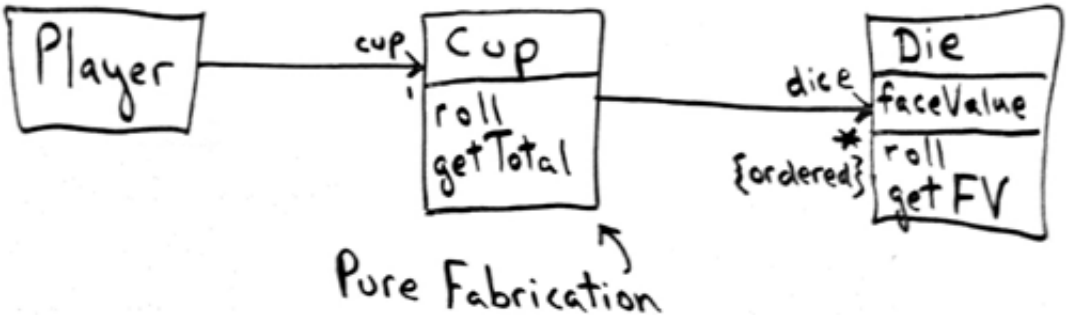
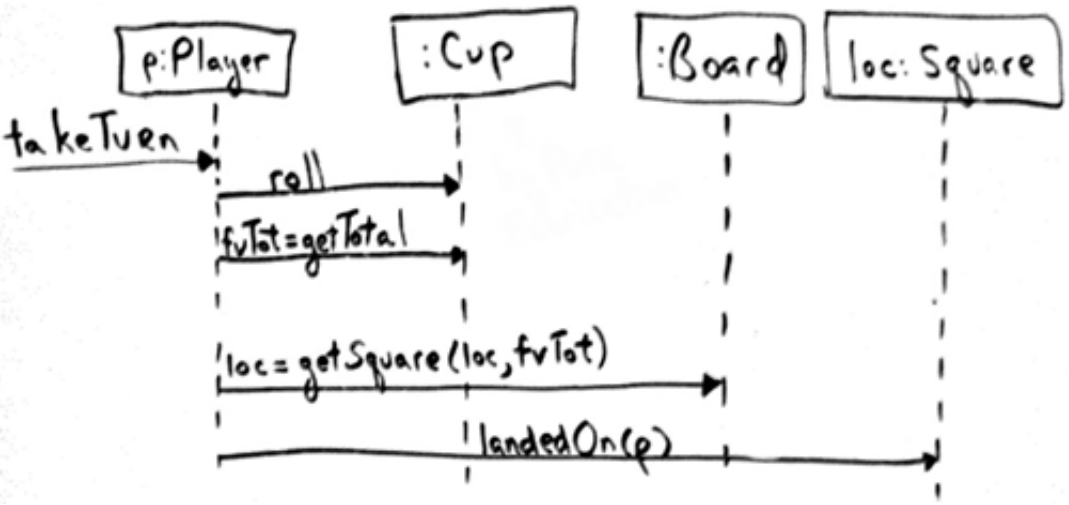


Figure 25.9. Using the *Cup* in the Monopoly game.



Discussion

The design of objects can be broadly divided into two groups:

1. Those chosen by representational decomposition.
2. Those chosen by behavioral decomposition.

For example, the creation of a software class such as *Sale* is by representational decomposition; the software class is related to or represents a thing in a domain. Representational decomposition is a common strategy in object design and supports the goal of low representational gap. But sometimes, we desire to assign responsibilities by grouping behaviors or by algorithm, without any concern for creating a class with a name or purpose that is related to a real-world domain concept.

A good example is an “algorithm” object such as a *TableOfContentsGenerator,* whose purpose is (surprise!) to generate a table of contents and was created as a helper or convenience class by a developer, without any concern for choosing a name from the domain vocabulary of books and documents. It exists as a convenience class conceived by the developer to group together some related behavior or methods, and is thus motivated by

behavioral decomposition.

To contrast, a software class named *TableOfContents* is inspired by *representational decomposition*, and should contain information consistent with our concept of the real domain (such as chapter names).

Identifying a class as a Pure Fabrication is not critical. It's an educational concept to communicate the general idea that some software classes are inspired by representations of the domain, and some are simply "made up" as a convenience for the object designer. These convenience classes are usually designed to group together some common behavior, and are thus inspired by behavioral rather than representational decomposition.

Said another way, a Pure Fabrication is usually partitioned based on related functionality, so it is a kind of function-centric or behavioral object.

Many existing object-oriented design patterns are examples of Pure Fabrications: Adapter, Strategy, Command, and so on [GHJV95].

As a final comment worth reiterating: Sometimes a solution offered by Information Expert is not desirable. Even though the object is a candidate for the responsibility by virtue of having much of the information related to the responsibility, in other ways, its choice leads to a poor design, usually due to problems in cohesion or coupling.

Benefits

- High Cohesion is supported because responsibilities are factored into a fine-grained class that only focuses on a very specific set of related tasks.
- Reuse potential may increase because of the presence of fine-grained Pure Fabrication classes whose responsibilities have applicability in other applications.

Contraindications

Behavioral decomposition into Pure Fabrication objects is sometimes overused by those new to object design and more familiar with decomposing or organizing software in terms of functions. To exaggerate, functions just become objects. There is nothing inherently wrong with creating "function" or "algorithm" objects, but it needs to be balanced with the ability to design with representational decomposition, such as the ability to apply Information Expert so that a representational class such as *Sale* also has responsibilities. Information Expert supports the goal of co-locating responsibilities with the objects that know the information needed for those responsibilities, which tends to support lower coupling. If overused, Pure Fabrication could lead to too many behavior objects that have responsibilities *not* co-located with the information required for their fulfillment, which can adversely affect coupling. The usual symptom is that most of the data inside the objects is being passed to other objects to reason with it.

Related Patterns and Principles

- Low Coupling.
- High Cohesion.
- A Pure Fabrication usually takes on responsibilities from the domain class that would be assigned those responsibilities based on the Expert pattern.
- All GoF design patterns [GHJV95], such as Adapter, Command, Strategy, and so on, are Pure Fabrications.
- Virtually all other design patterns are Pure Fabrications.