GRASP: Designing Objects with Responsibilities

Applying UML and Patterns

Craig Larman

Chapter 17-18

OOSD

Majority of slides are taken from CSE 432: Object-Oriented Software Engineering class from Lehigh University (http://www.cse.lehigh.edu/~glennb/oose/oose.htm)

Chapter Learning Objectives

- Learn about design patterns
- Learn how to apply five GRASP patterns

- You’ve learned about static class diagrams and dynamic interaction diagrams
- UML is just notation; now you need to learn how to make effective use of the notation
- UML modeling is an art, guided by principles

Design patterns in architecture

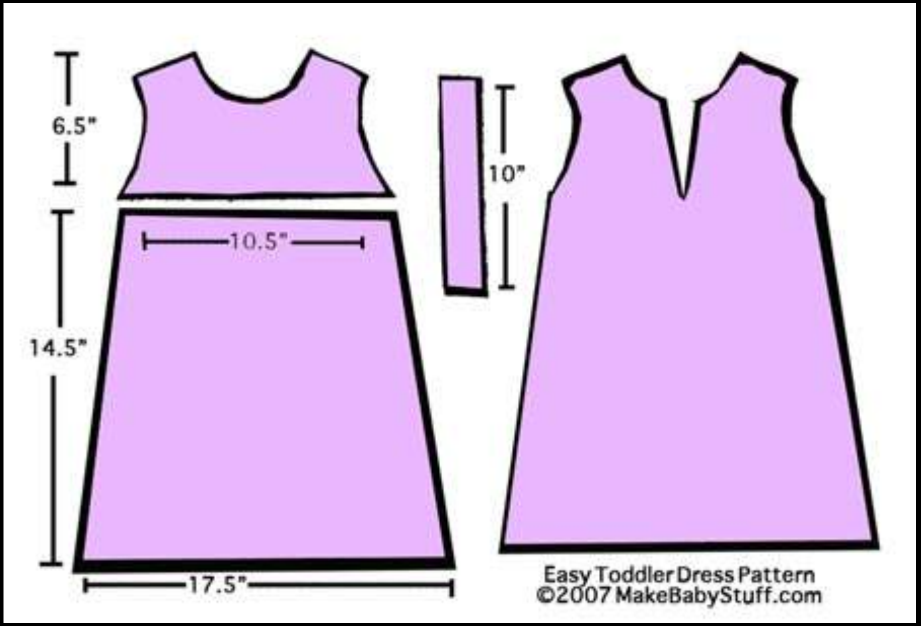
- A *pattern* is a recurring solution to a standard problem, in a context.
- Christopher Alexander, professor of architecture…
  - *Why is what a prof of architecture says relevant to software?*
  - "A pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice."

Design and dress patterns

- Jim Coplein, a software engineer:

“I like to relate this definition to dress patterns …

I could tell you how to make a dress by specifying the route of a scissors through a piece of cloth in terms of angles and lengths of cut. Or, I could give you a pattern. Reading the specification, you would have no idea what was being built or if you had built the right thing when you were finished. The pattern foreshadows the product: it is the rule for making the thing, but it is also, in many respects, the thing itself.”



Patterns in engineering

- *How do other engineers find and use patterns?*
  - Mature engineering disciplines have handbooks describing successful solutions to known problems
  - Automobile designers don't design cars from scratch using the laws of physics
  - Instead, they reuse standard designs with successful track records, learning from experience
  - *Should software engineers make use of patterns? Why?*
- Developing software from scratch is also expensive
  - Patterns support reuse of software architecture design

Definitions and names

- Alexander: “A *pattern* is a recurring solution to a standard problem, in a context.”
- Larman: “In OO design, a *pattern* is a named description of a problem and solution that can be applied in new contexts; ideally, a pattern advises us on how to apply the solution in varying circumstances and considers the forces and trade-offs.”
- How is Larman’s definition similar to Alexander’s?
- How are these definitions significantly different?

Naming Patterns—important!

- Patterns have suggestive names:
  - Arched Columns Pattern, Easy Toddler Dress Pattern, etc.
- Why is naming a pattern or principle helpful?
  - It supports chunking and incorporating that concept into our understanding and memory
  - It facilitates communication

Star and Plume Quilt

GRASP

- Name chosen to suggest the importance of **grasp**ing fundamental principles to successfully design object-oriented software
- Acronym for **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns
- Describe fundamental principles of object design and responsibility
- Expressed as patterns

Doing responsibilities of an object

- 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

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

Responsibilities

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

Responsibility Driven Design

- 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*

Five GRASP patterns:

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

Creator pattern

Name: **Creator**

Problem: Who creates an instance of A?

Solution: 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 aggregates A (in a collection)
- B records A
- B closely uses A
- B has the initializing data for A

Who creates the Squares?

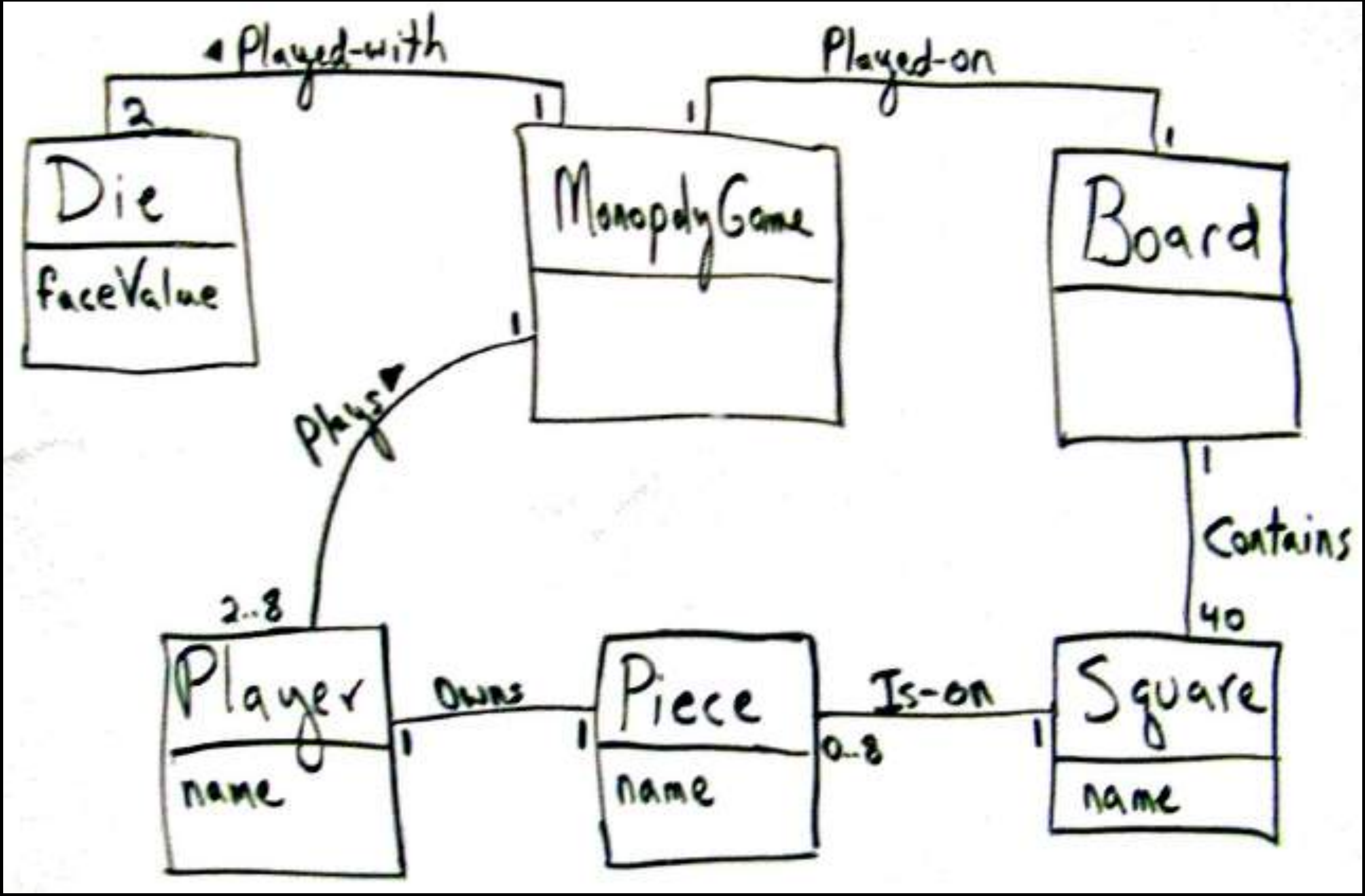


Figure 17.3, page 283

How does Create pattern lead to this partial Sequence diagram?

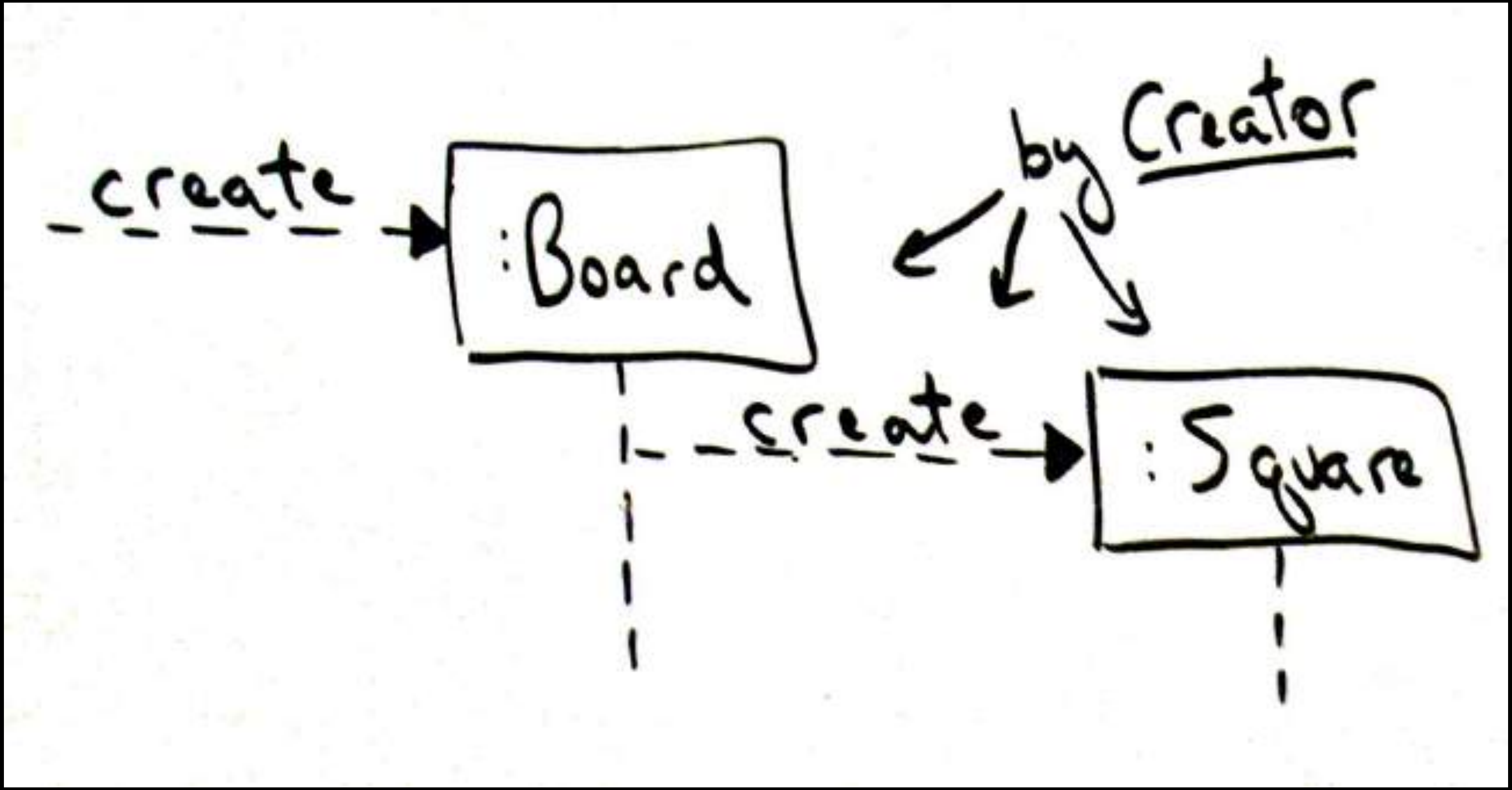


Figure 17.4, page 283

How does Create pattern develop this Design Class Diagram (DCD)?

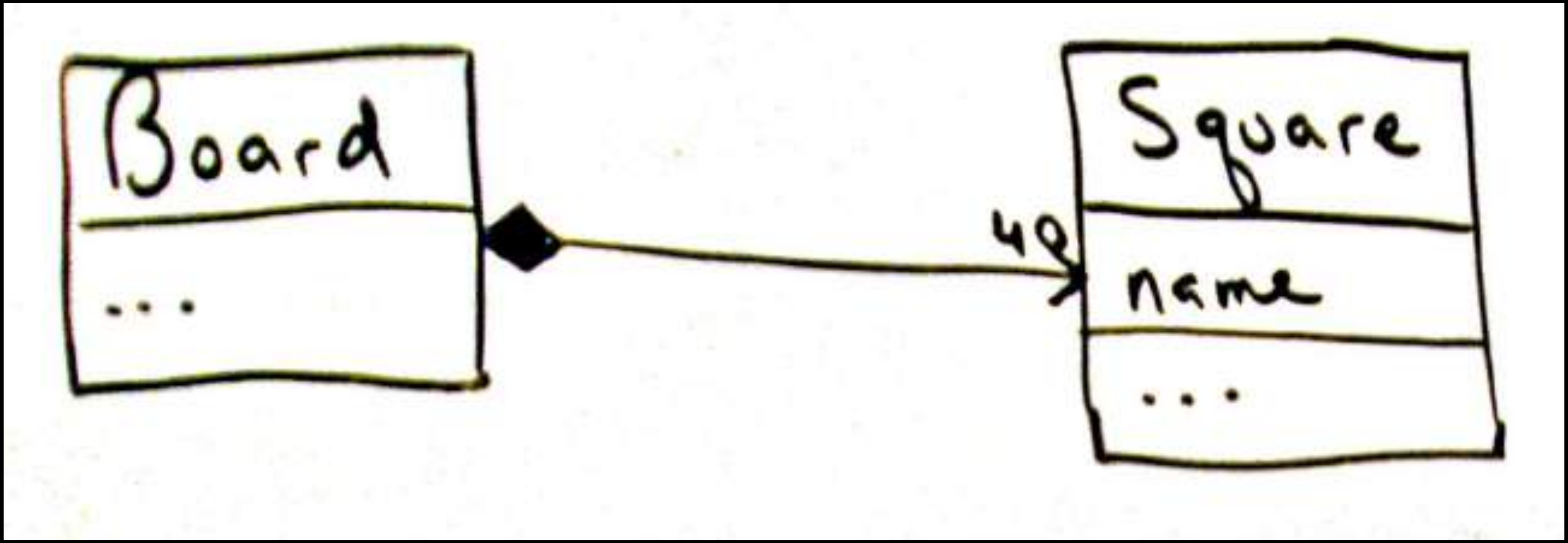
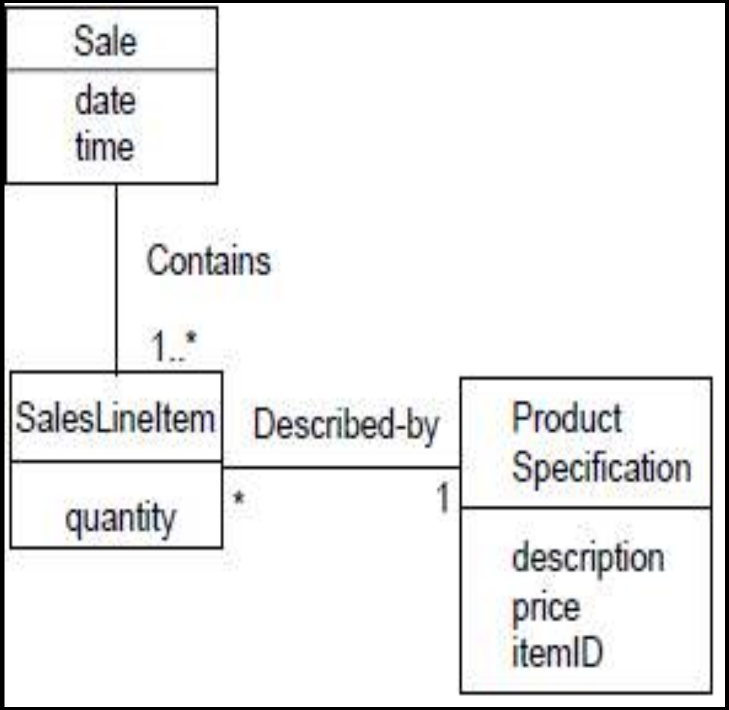


Figure 17.5 , page 283

Board has a composite aggregation relationship with *Square*

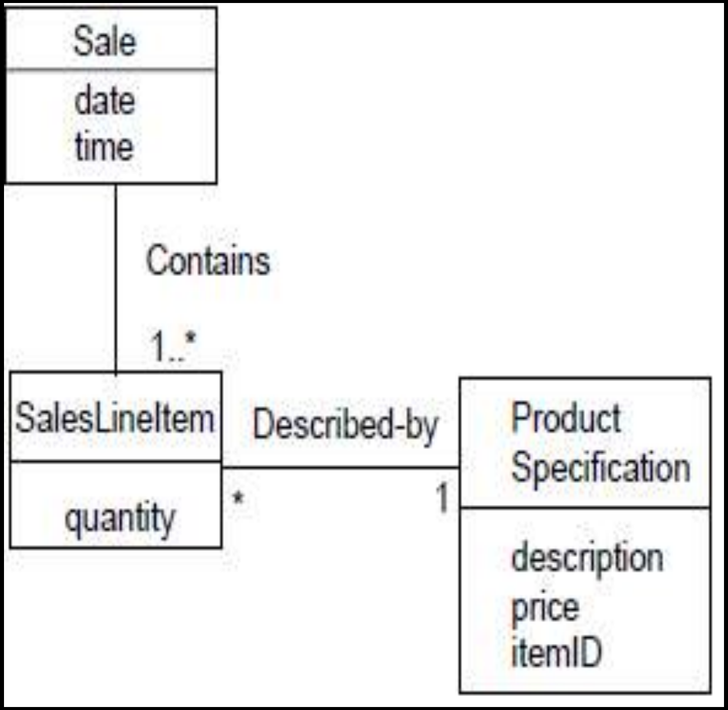
- I.e., Board contains a collection of Squares

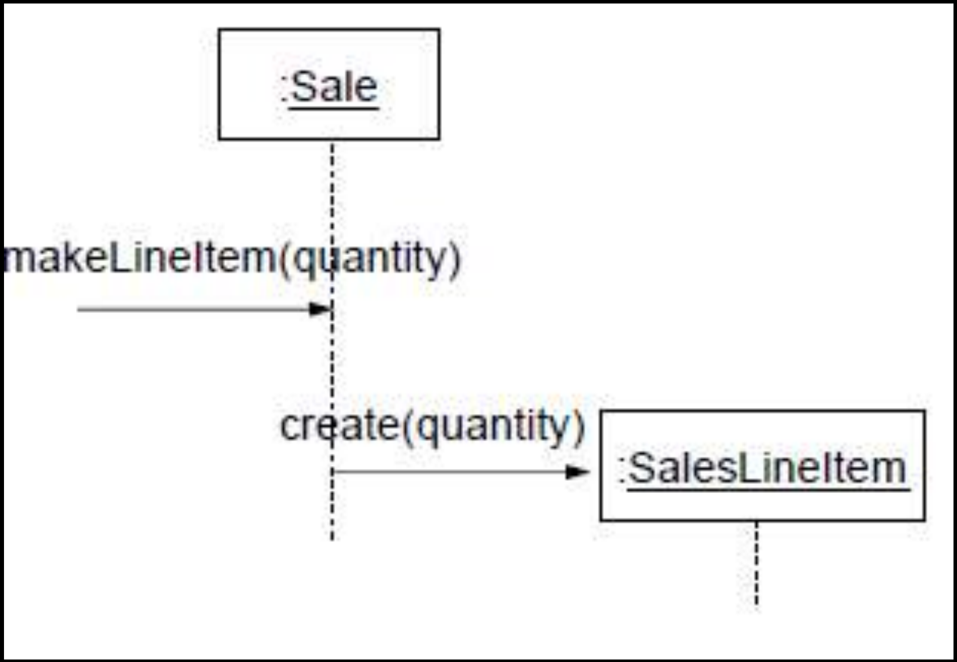
Creator Example - NextGen POS



- who should be responsible for creating a SalesLineItem instance?
- Since a Sale contains many salesLineItem objects, the Creator pattern suggests that Sale is a good Candidate.

Creator Example - NextGen POS





Discussion of Creator pattern

- Responsibilities for object creation are common
- Connect an object to its creator when:
  - Aggregator aggregates Part
  - Container contains Content
  - Recorder records
  - Initializing data passed in during creation

Contraindications or caveats

- Creation may require significant complexity:
  - recycling instances for performance reasons
  - conditionally creating instances from a family of similar classes
- In these instances, other patterns are available…
  - We'll learn about Factory and other patterns later…

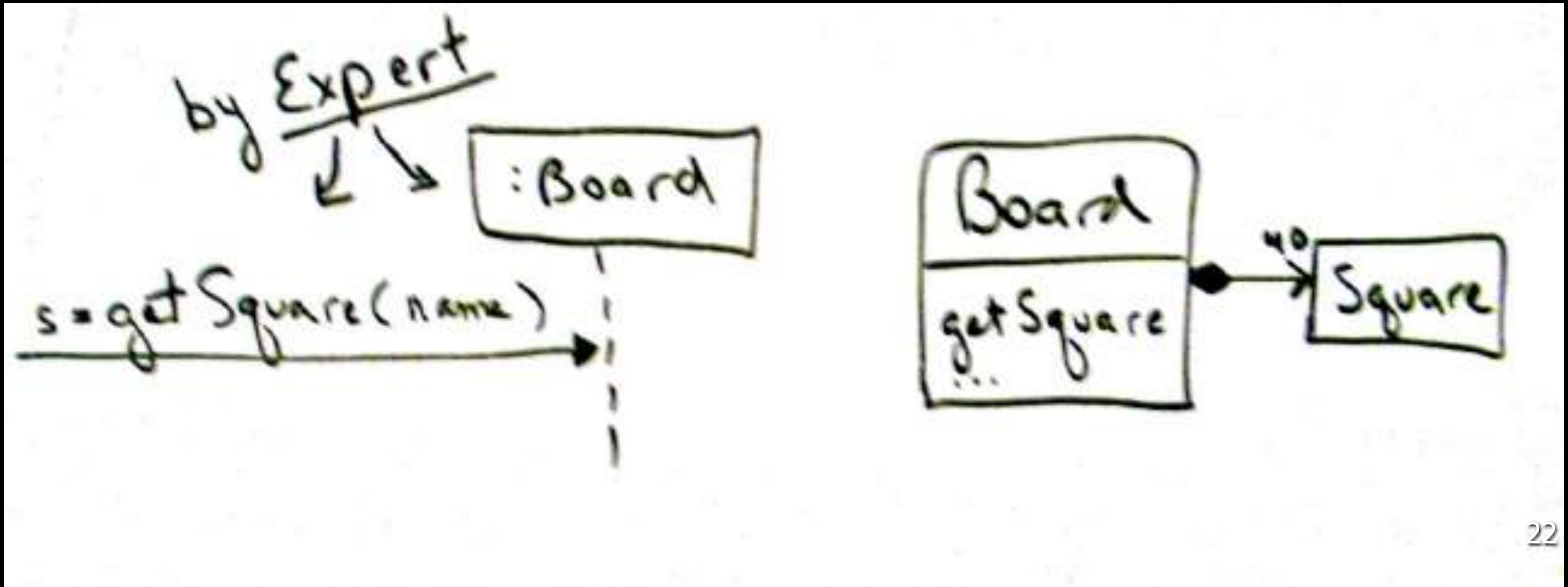
Information Expert pattern or principle

Name: Information Expert

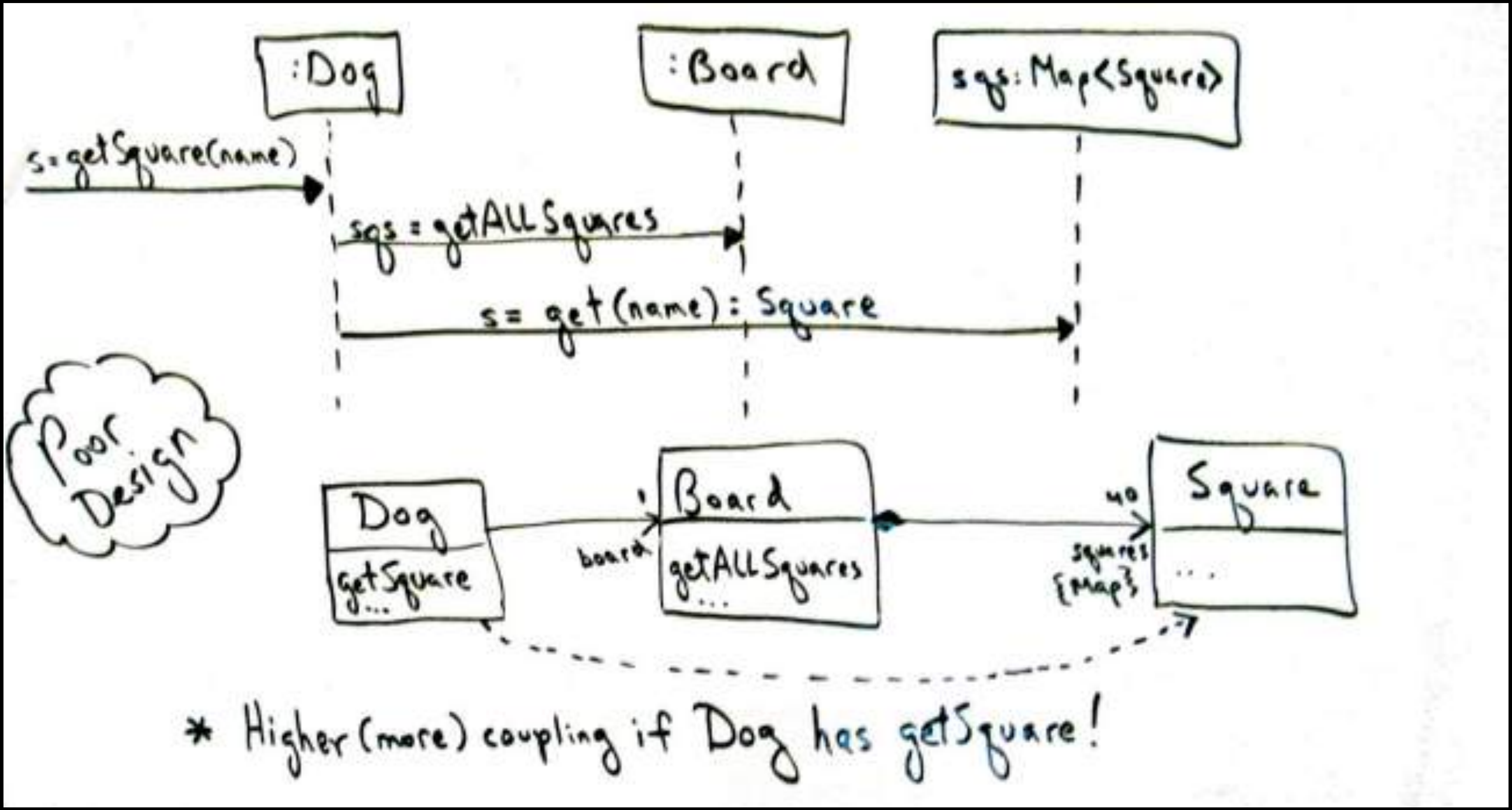
Problem: How to assign responsibilities to objects?

Solution: Assign responsibility to the class that has the information needed to fulfill it?

- E.g., Board information needed to get a Square

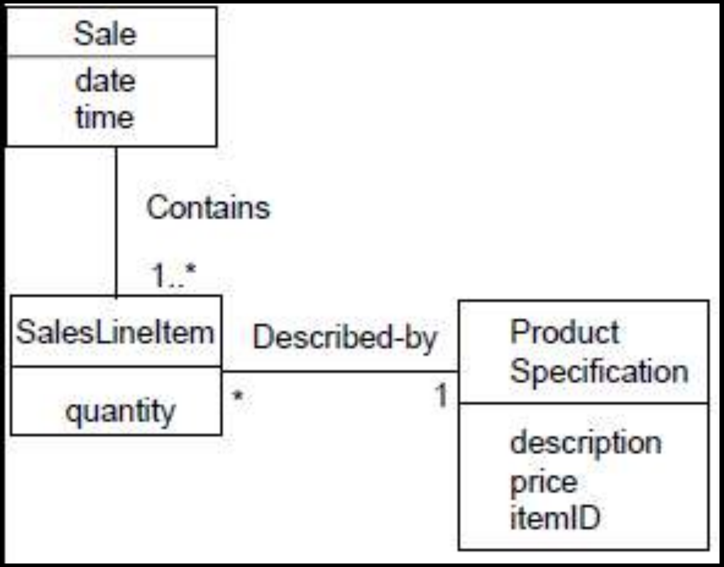


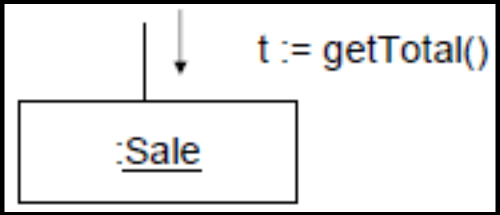
Why does the following design violate Low Coupling?



Why is a better idea to leave getSquare responsibility in Board?

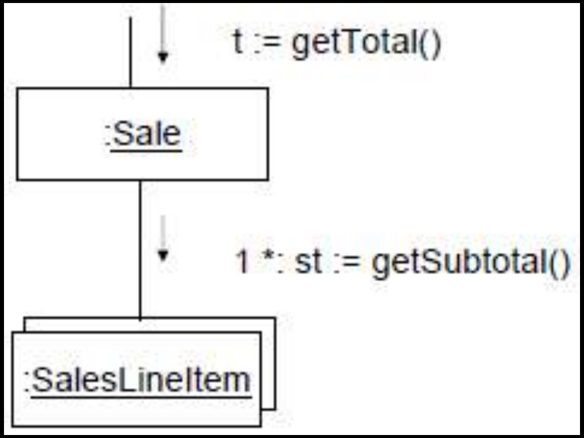
Information Expert (or Expert)





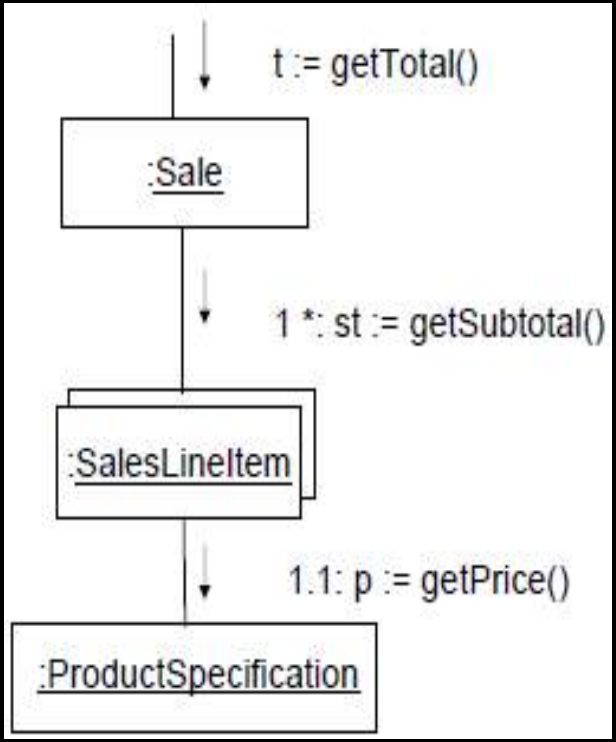
- It is necessary to know about all the SalesLineItem instances of a sale and the sum of the subtotals.
- A Sale instance contains these, i.e. it is an information expert for this responsibility.

Information Expert (or Expert)



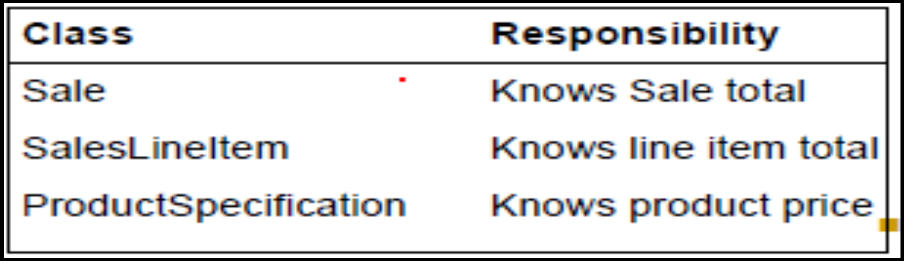
- What information is needed to determine the line item subtotal?
  – quantity and price.
- SalesLineItem should determine the subtotal.
- This means that Sale needs to send getSubtotal() messages to each of the SalesLineItems and sum the results.

Information Expert (or Expert)



- To fulfil the responsibility of knowing and answering its subtotal, a SalesLineItem needs to know the product price.
- The ProductSpecification is the information expert on answering its price.

Information Expert (or Expert)



Class	Responsibility
Sale	Knows Sale total
SalesLineItem	Knows line item total
ProductSpecification	Knows product price

- To fulfil the responsibility of knowing and answering the sale’s total, three responsibilities were assigned to three design classes
- The fulfillment of a responsibility often requires information that is spread across different classes of objects. This implies that there are many “partial experts” who will collaborate in the task

Benefits and Contraindications

- Facilitates information encapsulation: *why?*
  - Classes use their own info to fulfill tasks
- Encourages cohesive, lightweight class definitions

But:

- Information expert may contradict patterns of Low Coupling and High Cohesion
- Remember separation of concerns principle for large sub-systems
- I.e., keep “business” or application logic in one place, user interface in other place, database access in another place, etc.

Low Coupling Pattern

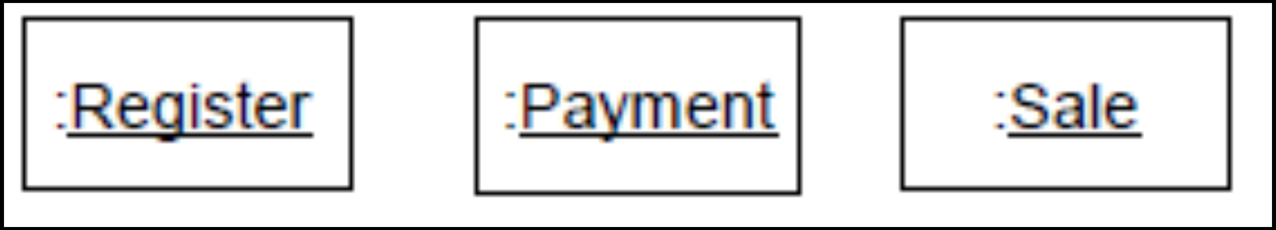
- Name: **Low Coupling**
- Problem: How to reduce the impact of change and encourage reuse?
- Solution: Assign a responsibility so that coupling (linking classes) remains low.

Low Coupling Pattern

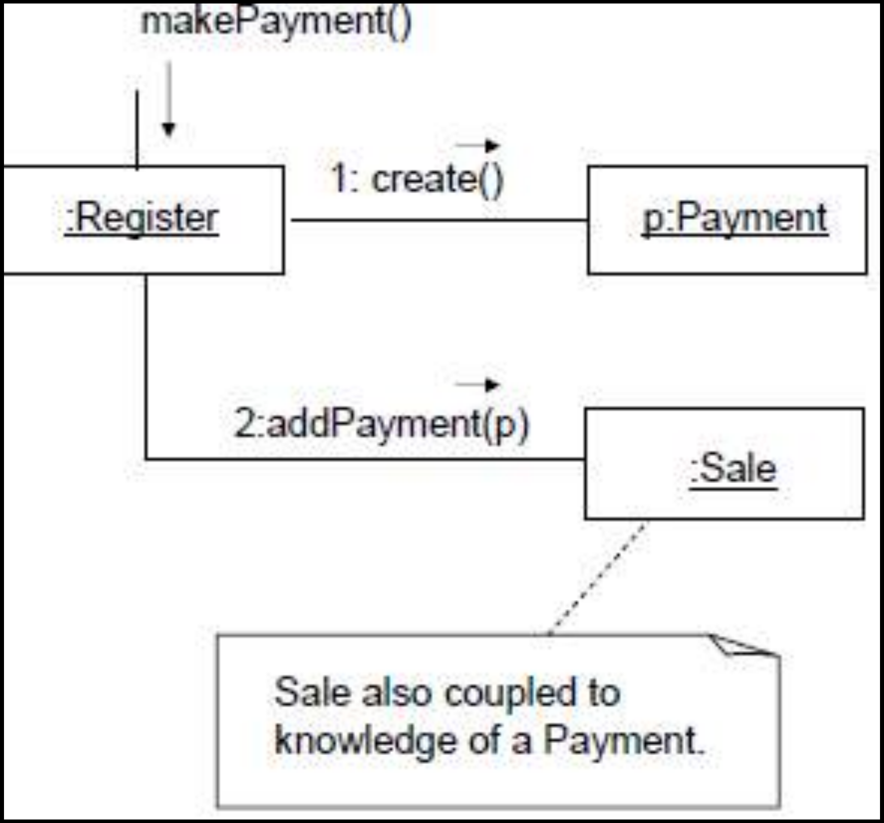
- Coupling: it is a measure of how strongly one element is connected to, has knowledge of, or relies upon other elements.
- A class with high coupling depends on many other classes (libraries, tools).
- Problems because of a design with high coupling:
  - Changes in related classes force local changes.
  - Harder to understand in isolation; need to understand other classes.
  - Harder to reuse because it requires additional presence of other classes.
- Problem: How to support low dependency, low change impact and increased reuse?
- Solution: Assign a responsibility so that coupling remains low.

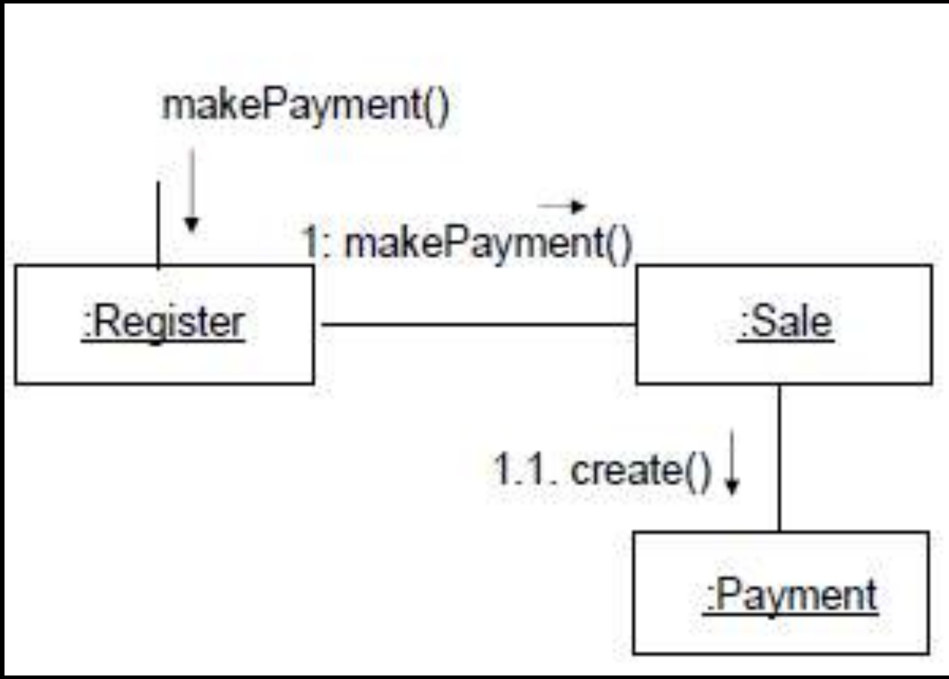
Low Coupling

- Assume we need to create a Payment instance and associate it with the Sale.
- What class should be responsible for this?
- By Creator, Register is a candidate.



Low Coupling





Low Coupling

- Some of the places where coupling occurs:
  - Attributes: X has an attribute that refers to a Y instance.
  - Methods: e.g. a parameter or a local variable of type Y is found in a method of X.
  - Subclasses: X is a subclass of Y.
  - Types: X implements interface Y.
- There is no specific measurement for coupling, but in general,
  - classes that are generic and simple to reuse have low coupling.
- There will always be some coupling among objects, otherwise, there would be no collaboration.

Benefits & Contraindications

- Understandability: Classes are easier to understand in isolation
- Maintainability: Classes aren’t affected by changes in other components
- Reusability: easier to grab hold of classes

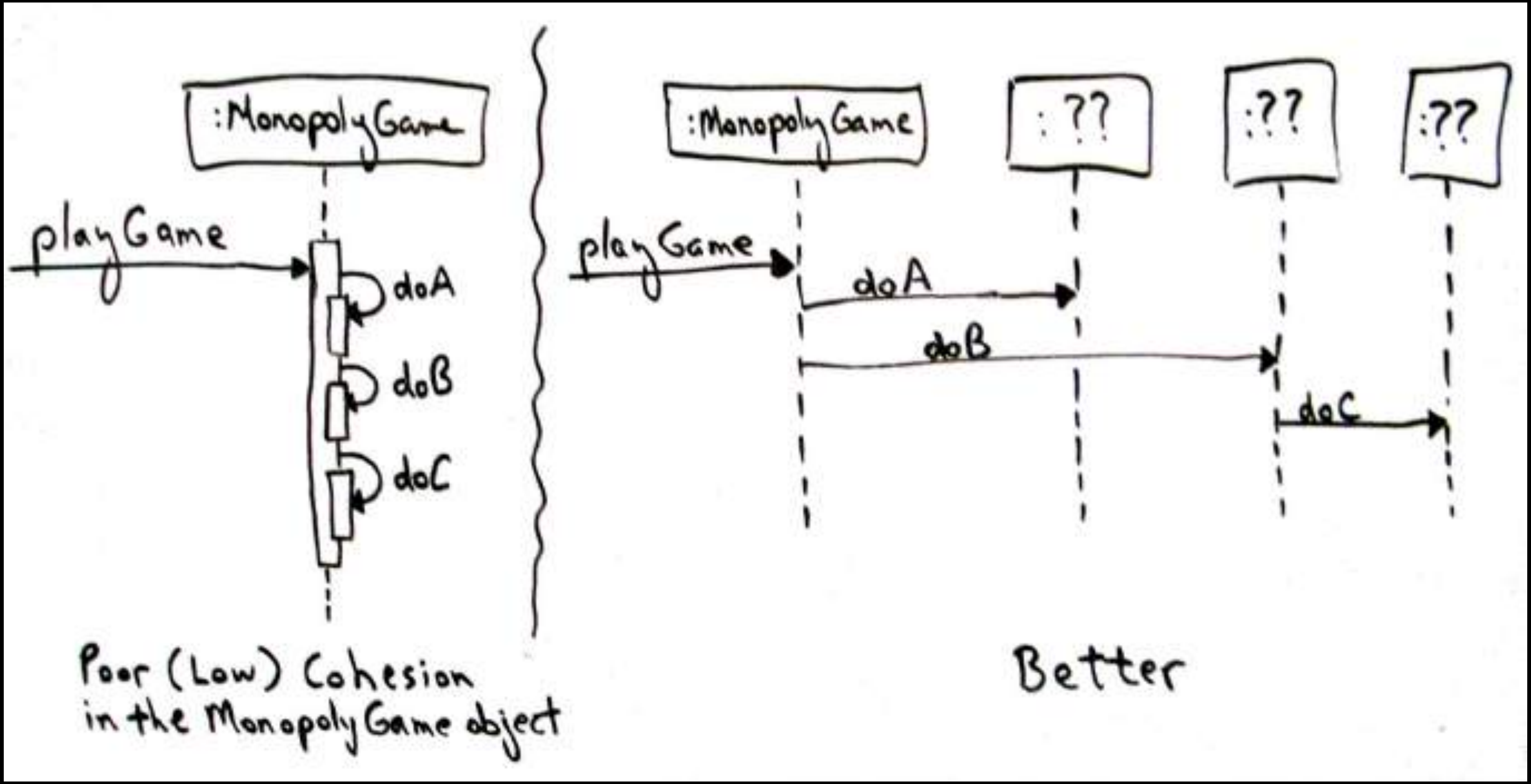
But:

- Don’t sweat coupling to stable classes (in libraries or pervasive, well-tested classes)

High Cohesion pattern

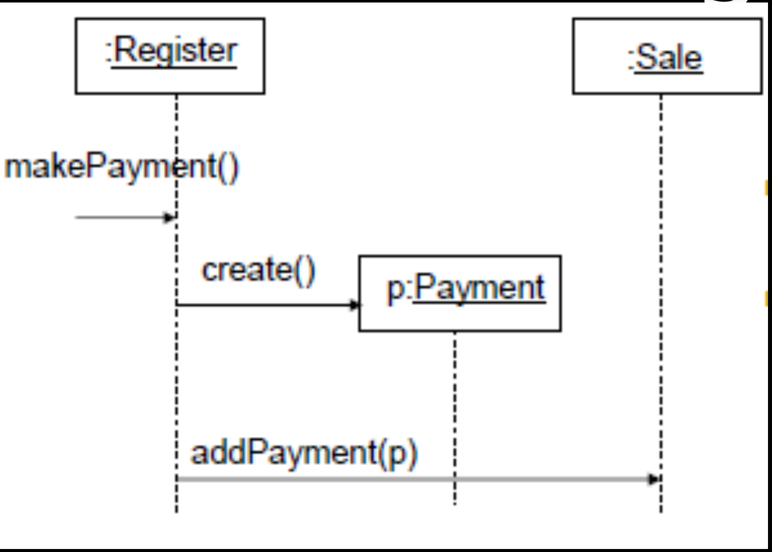
- Cohesion measures how strongly related and focused are the responsibilities of an element
- Name: **High Cohesion**
- Problem: How to keep classes focused and manageable?
- Solution: Assign responsibility so that cohesion remains high.
- Rule of thumb: a class with high cohesion has a relative low number of methods, with highly related functionality, and doesn’t do much work. It collaborates and delegates.

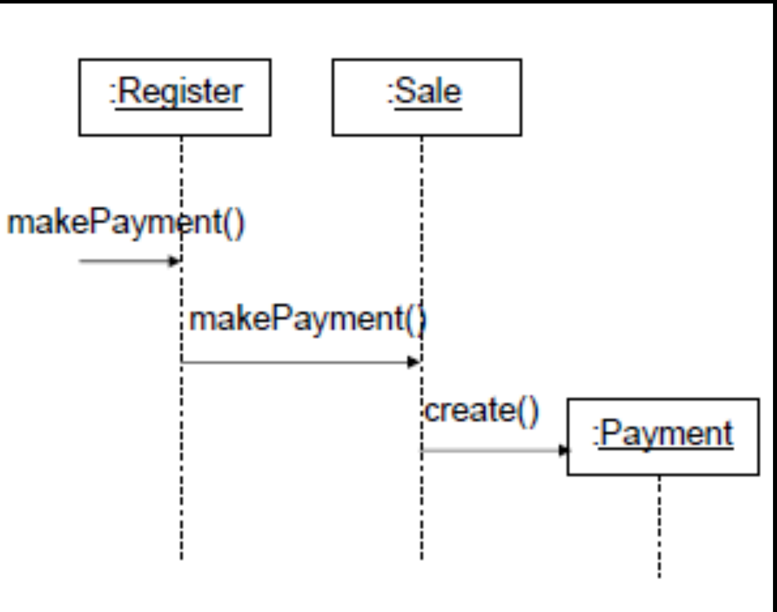
How does the design on right promote high cohesion?



Delegate responsibility & coordinate work

High Cohesion





- Assume we need to create a Payment instance and associate it with Sale. What class should be responsible for this?
- By Creator, Register is a candidate.
- Register may become bloated if it is assigned more and more system operations.
- An alternative design delegates the Payment creation responsibility to the Sale, which supports higher cohesion in the Register.
- This design supports high cohesion and low coupling.

High Cohesion

- Scenarios that illustrate varying degrees of functional cohesion

1. Very low cohesion: class responsible for many things in many different areas.

   e.g.: a class responsible for interfacing with a data base and remote-procedure-calls.

2. Low cohesion: class responsible for complex task in a functional area.

   – e.g.: a class responsible for interacting with a relational database.

Benefits & Contraindications

- Understandability, maintainability
- Complements Low Coupling

But:

- Avoid grouping of responsibilities or code into one class or component to simplify maintenance by one person. *Why?*
- Sometimes desirable to create less cohesive server objects that provide an interface for many operations, due to performance needs associated with remote objects and remote communication

Controller pattern

- Name: **Controller**
  (see Model-View-Controller architecture)
- Problem: Who should be responsible for UI events?
- Solution: Assign responsibility for receiving or handling a system event in one of two ways:
  - Represent the overall system (*façade* pattern)
  - Represent a use case scenario within which the system event occurs (a *session* controller)

Who is the controller of playGame operation?

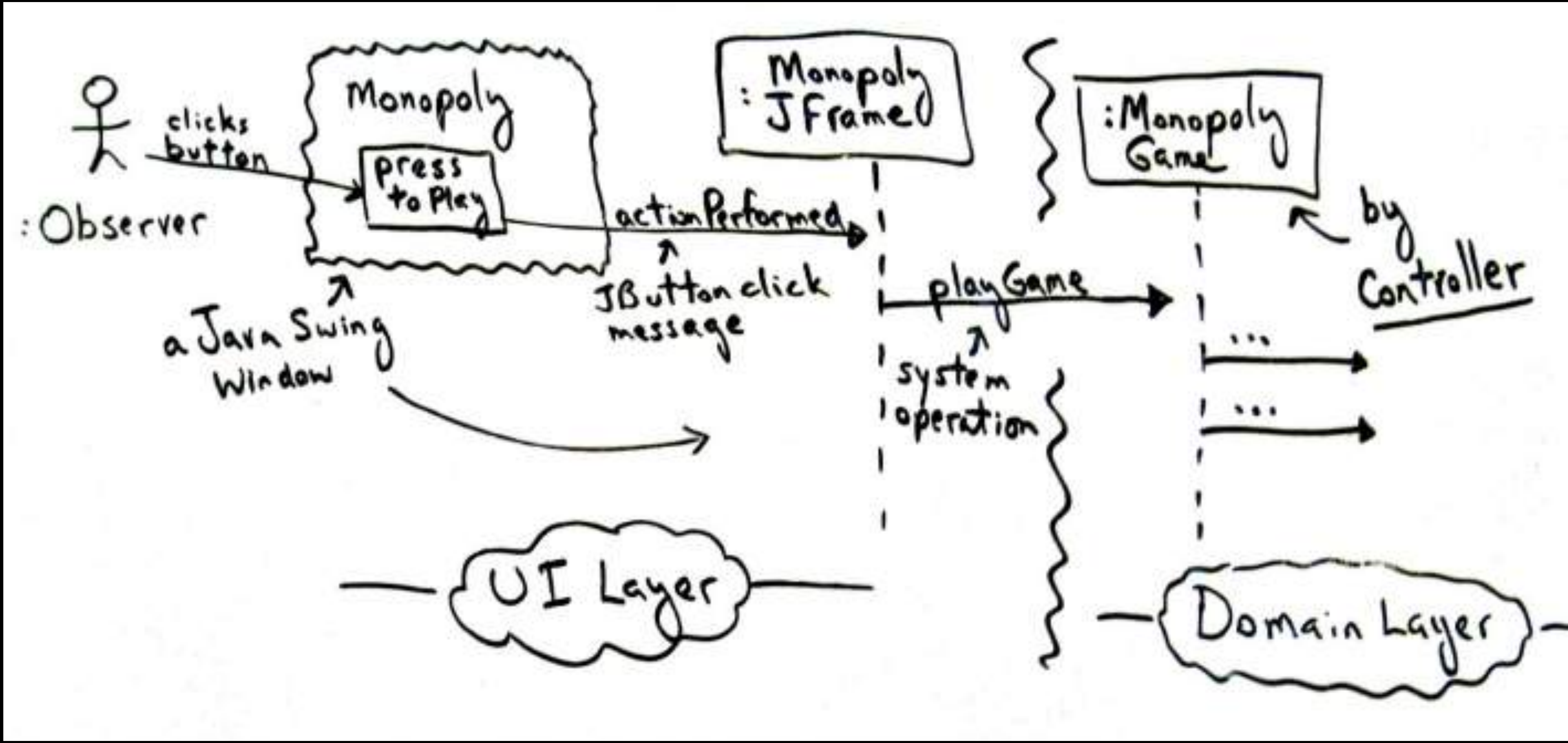
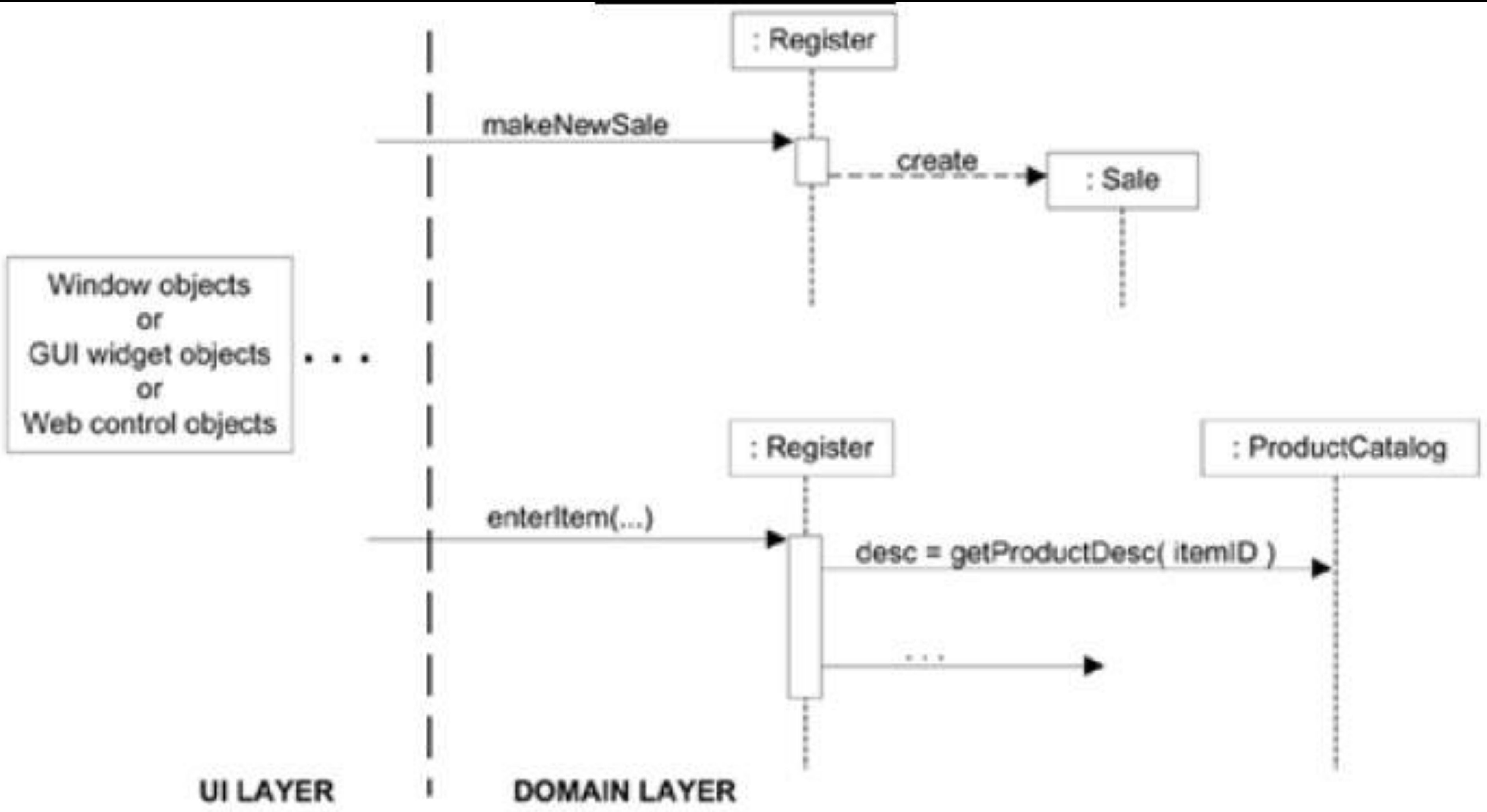


Figure 17.9, p. 288

What class represents the overall system or relevant use case scenario?


: Register
makeNewSale
create
: Sale
Window objects or GUI widget objects or Web control objects
. . .
: Register
: ProductCatalog
enterItem(...)
desc = getProductDesc(itemID)
. . .
UI LAYER
DOMAIN LAYER

Model-View Seperation Principle

- UI Objects should **not** contain application or “business” logic
  - such as calculating a player’s move
  - Once the UI object pick up the mouse event, they need to delegate (forward the task another object) the request to *domain objects* in *domain model*

Summary

- Skillful assignment of responsibilities is extremely important in object-oriented design (CRC cards are one technique)
- Patterns are named problem/solution pairs that codify good advice and principles related to assignment of responsibilities
- GRASP identifies five patterns or principles:
  - Creator, Information Expert, Controller, Low Coupling and High Cohesion

More GRASP Patterns

- Polymorphism
- Indirection
- Pure Fabrication
- Protected Variations

Polymorphism

- Definition:
  - 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

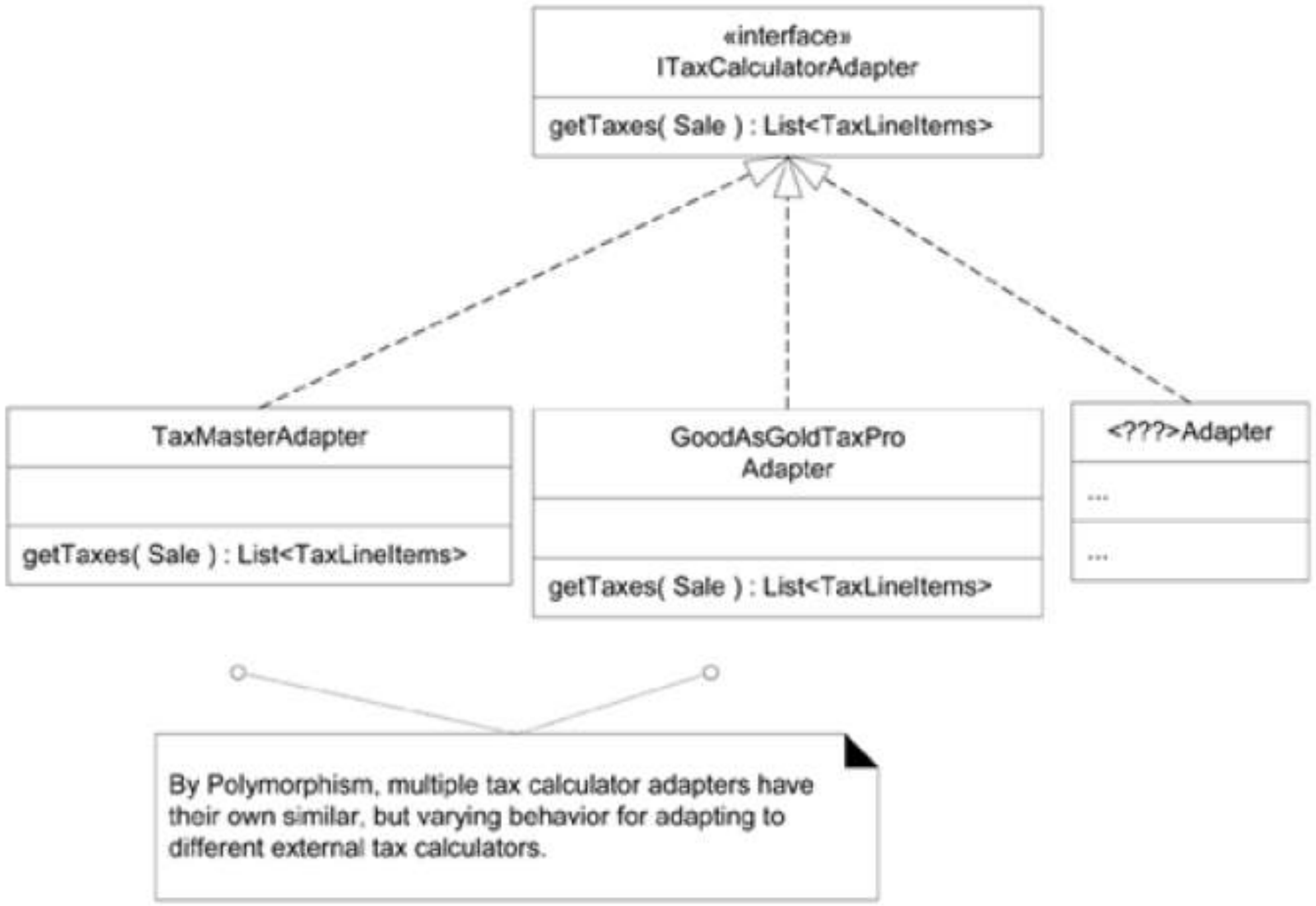
Polymorphism

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

Polymorphism

- *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 <u>easily extend a program with new variations</u> 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?

NextGen Problem: How Support Third-Party Tax Calculators


«interface»
ITaxCalculatorAdapter
getTaxes(Sale) : List<TaxLineItems>
TaxMasterAdapter
getTaxes(Sale) : List<TaxLineItems>
GoodAsGoldTaxPro Adapter
getTaxes(Sale) : List<TaxLineItems>
<???>Adapter
...
...
By Polymorphism, multiple tax calculator adapters have their own similar, but varying behavior for adapting to different external tax calculators.

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

Monopoly Problem: How to Design for Different Square Actions?

- 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.

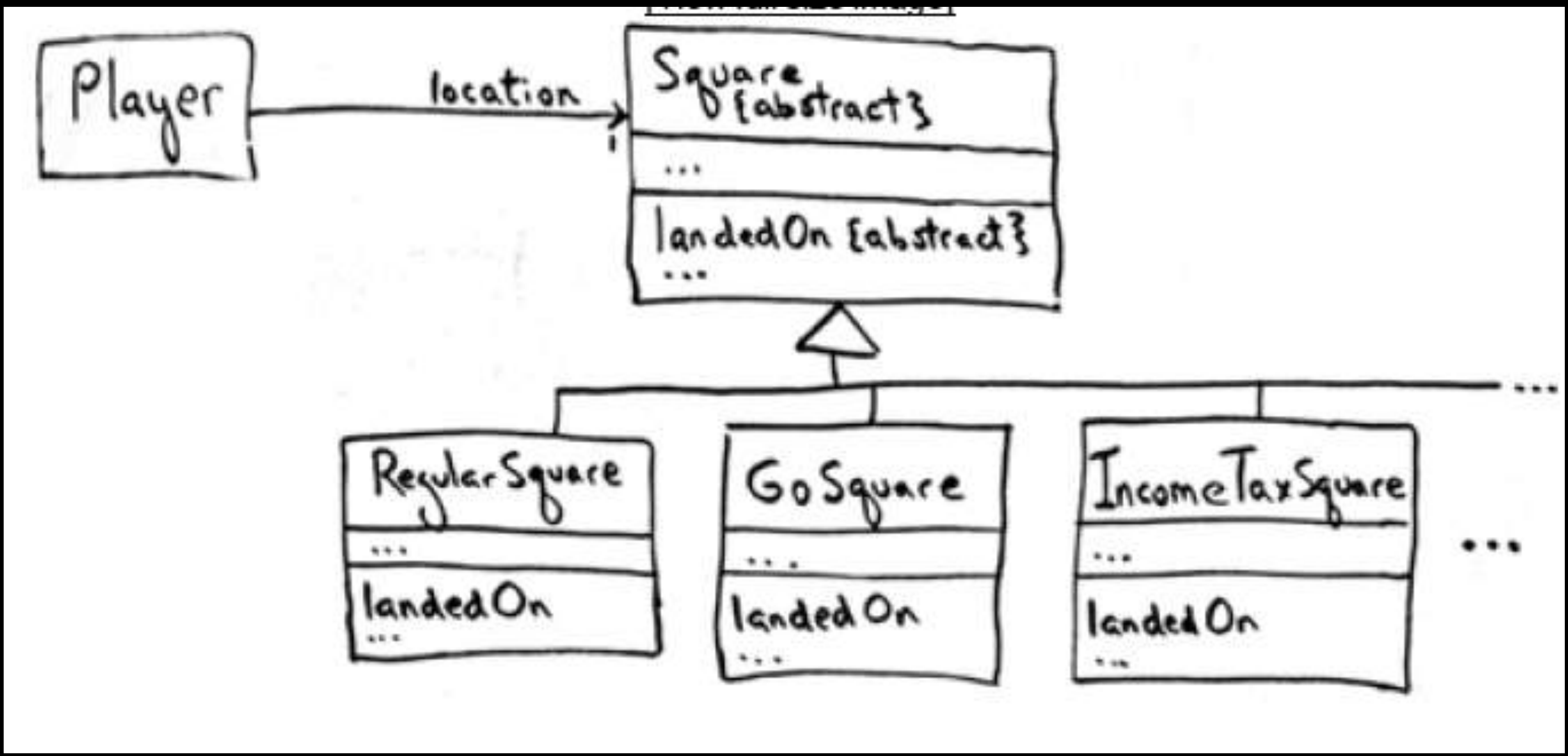
Monopoly Problem: How to Design for Different Square Actions?

- Bad design!

```
   // bad design
SWITCH ON square.type

CASE GoSquare: player receives $200
CASE IncomeTaxSquare: player pays tax
…
```

- Good design!



Player
location
1
Square {abstract}
...
landedOn {abstract}
...
RegularSquare
...
landedOn
...
GoSquare
...
landedOn
...
IncomeTaxSquare
...
landedOn
...
...
...

Figure 25.3. Applying Polymorphism.

[View full size image]

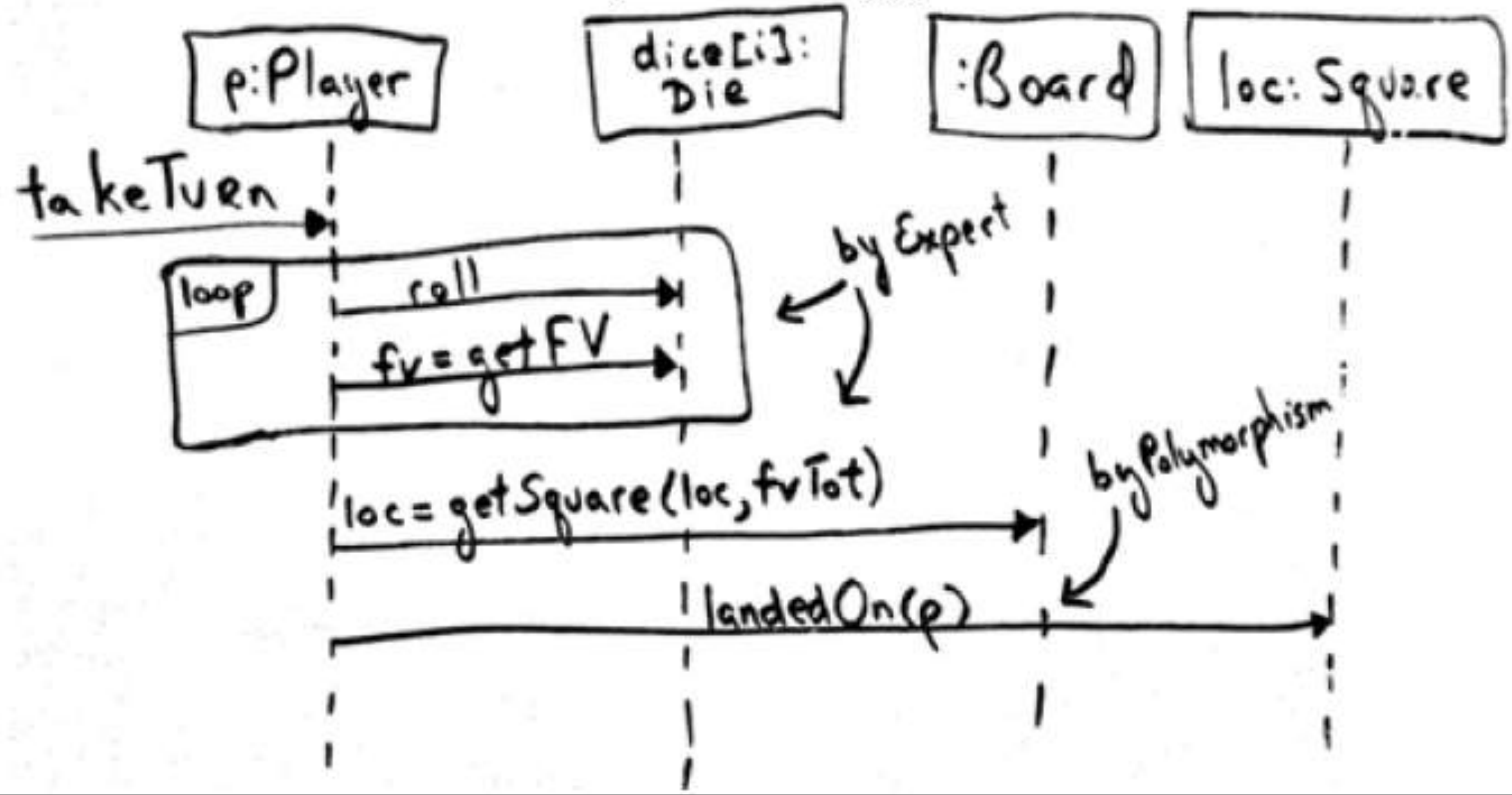
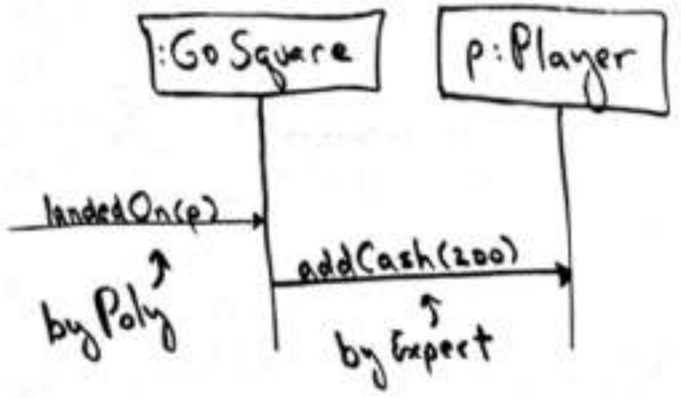


Figure 25.4. The *GoSquare* case.



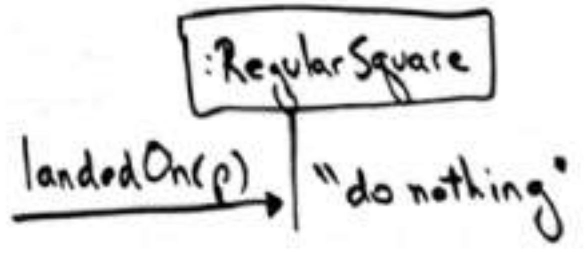


Figure 25.6. The *IncomeTaxSquare* case.

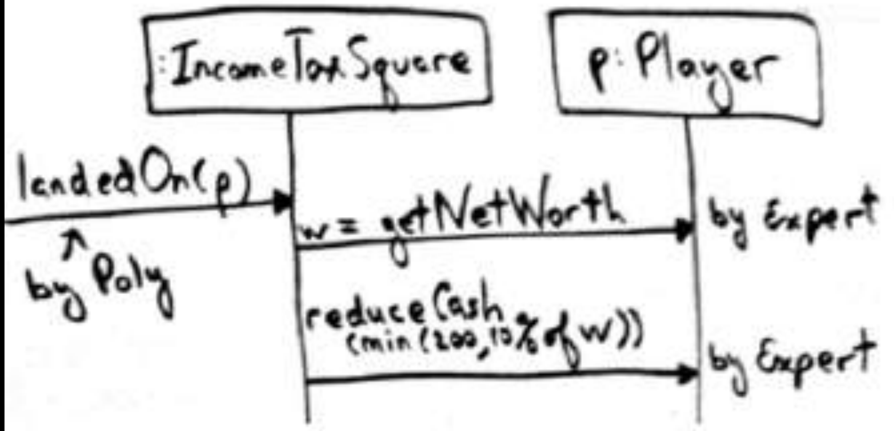
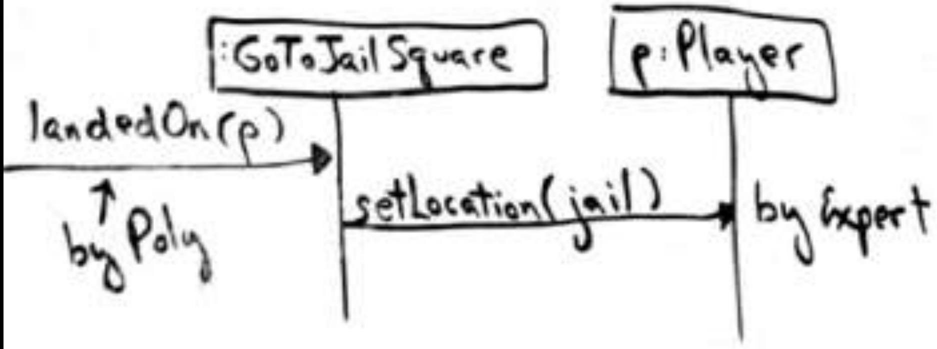


Figure 25.7. The *GoToJailSquare* case.



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?
- **Solution**
  - Assign a highly cohesive set of responsibilities to an artificial or convenience class that does not represent a problem domain concepts omething made up, to support high cohesion, low coupling, and reuse.

Pure Fabrication

- 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

Pure Fabrication

- NextGen Problem: Saving a Sale Object in a Database
  - 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.

Pure Fabrication

- NextGen Problem: Saving a Sale Object in a Database
  - 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.

Pure Fabrication

- Monopoly Problem: Handling the Dice

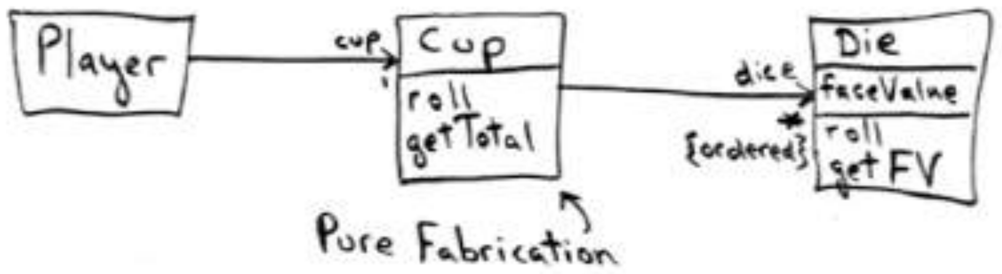
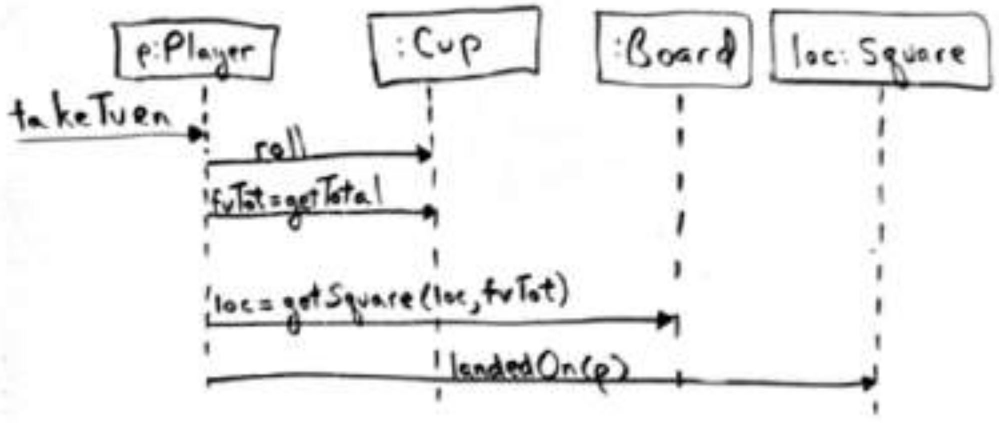


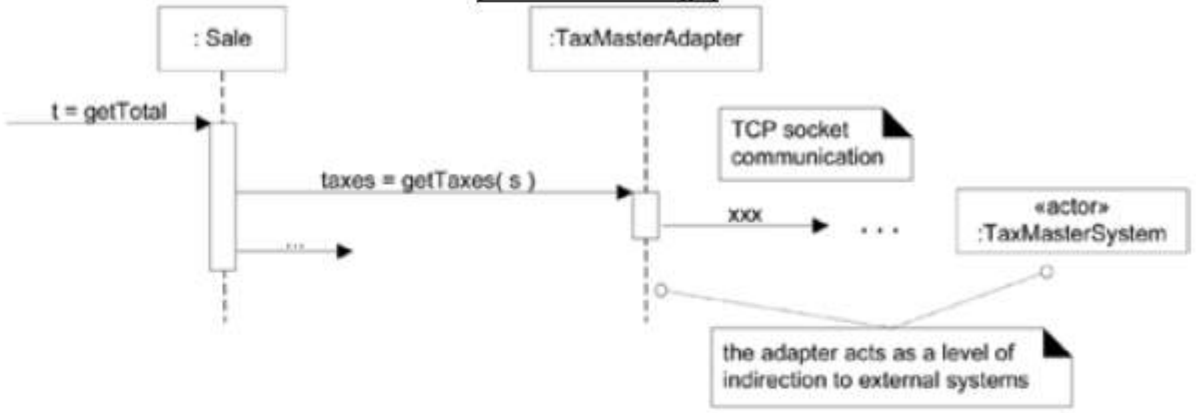
Figure 25.9. Using the *Cup* in the Monopoly game.



Pure Fabrication

Figure 25.10. Indirection via the adapter.

[View full size image]



Applying UML: Notice how the external TaxMaster remote service application is modeled in Figure 25.10: It's labeled with the «actor» keyword to indicate it's an external software component to our NextGen system.