Agenda

- Designing with GRASP
- MVC and GRASP

What is object design

- In the analysis part you have
- Identified use cases and created use case descriptions to get the requirements
- Created and refined the domain concept model
- Now in order to make a piece of object design you
  - ❑Assign methods to software classes
  - ❑Design how the classes collaborate (i.e. send messages) in order to fulfill the functionality stated in the use cases.
- You have learned how to use sequence diagrams.

Responsibilities and Methods

Responsibilities are assigned to classes of objects during object design. E.g.,

- ***doing***
    - ***doing itself (like creating an object)***
    - ***initiating action in other objects***
    - ***controlling and coordinating action in other objects***
- ***knowing***
    - ***Knowing about private encapsulated data***
    - ***knowing about related objects***
    - ***knowing about things it can compute***
-

Responsibilities and Methods

Responsibility is not the same thing as a method, but methods are implemented to fulfill responsibilities.

• Responsibilities are implemented by means of methods that either act alone or **collaborate with other methods and objects.**

Central tasks in design are:

➢ Deciding what methods belong where so that you a**dd methods to the software classes, and define the messaging between the objects to fulfill the requirements.**

➢ How the objects should interact

Assigning responsibilities

Responsibilities are assigned to objects during object design while creating interaction diagrams.

➢Sequence diagrams

➢Collaboration diagrams.

➢Examples:

- "a *Sale is responsible for creating SalesLineItems" (a doing), or*
- "a *Sale is responsible for knowing its*
- total" (a knowing).

GRASP: Designing Objects with Responsibilities

GRASP

- Name chosen to suggest the importance of **grasp**ing fundamental principles to successfully design object-oriented software.
- **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns.
- Fundamental principles of object design and responsibility .
- Strictly speaking, these are not 'design patterns', rather fundamental principles of object design.
- GRASP patterns focus on one of the most important aspects of object design.
- assigning responsibilities to classes.
- GRASP patterns do not address architectural design.

Basic objectives of GRASP

Which class, in the general case is responsible for a task?

- Responsibilities can include behaviour, data storage, object creation and more
- As mentioned, they often fall into two categories:
  - Doing (creating object, initiating action in other objects, coordinating action in other objects)
  - Knowing (encapsulated data, related abject, what it can calculate)

Basic objectives of GRASP

- You want to assign a responsibility to a class
- You want to avoid or minimize additional dependencies
- You want to maximise cohesion and minimise coupling

(We will very soon define what these terms mean)

- You want to increase reuse and decrease maintenance
- You want to maximise understandability

Five GRASP patterns:

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

Creator pattern

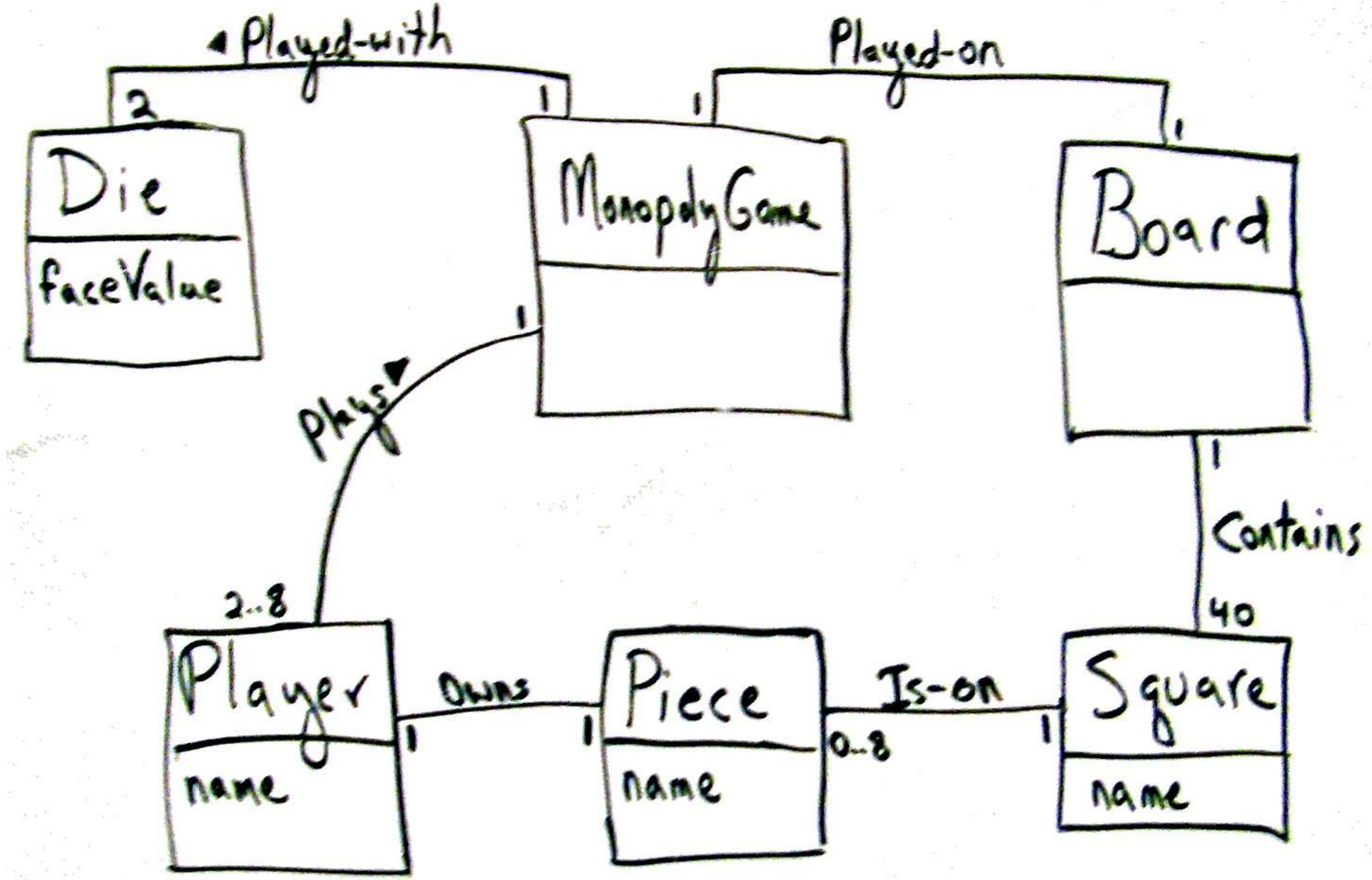
Name: **Creator**

Problem: Who creates an instance of any class say class 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
- ***If we have more than 1 class that satisfies the above condition for creating B, give responsibility to the class that aggregates B or contains B.***

Who creates the Squares?



Who creates the square

- Shall we use
  - ❑Die?
  - ❑Player?
  - ❑MonopolyGame?
  - ❑Player?
  - ❑Piece?

- No! They don't appeal to our mental model of the domain.
- **Board is the right answer.**

How does Create pattern lead to this partial Sequence diagram?

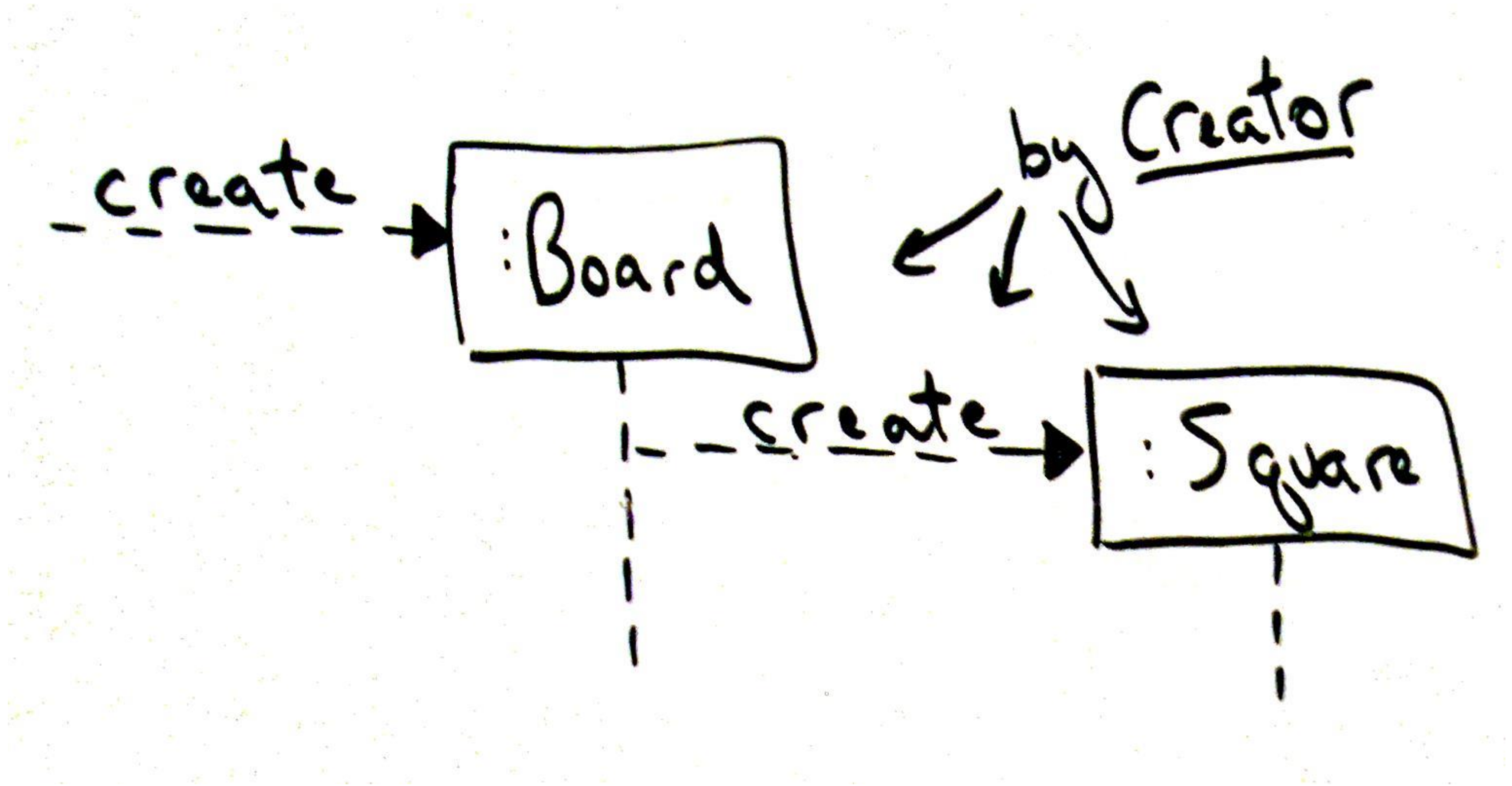


Figure 17.4, page 283

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

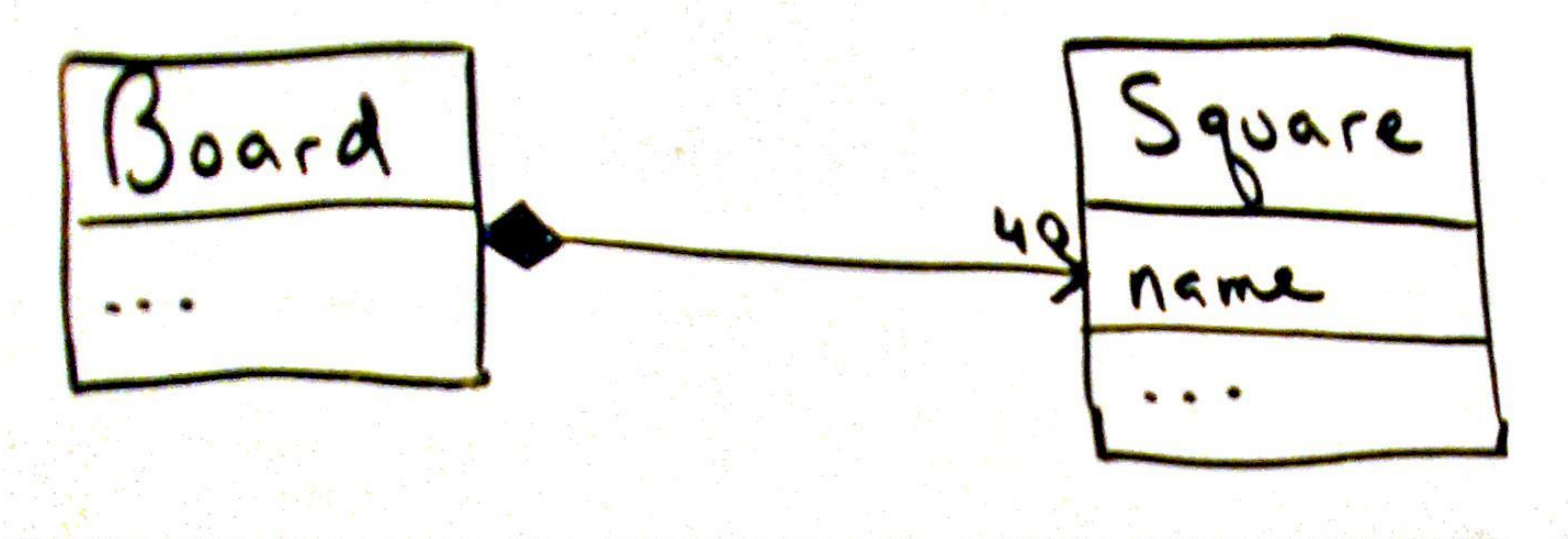


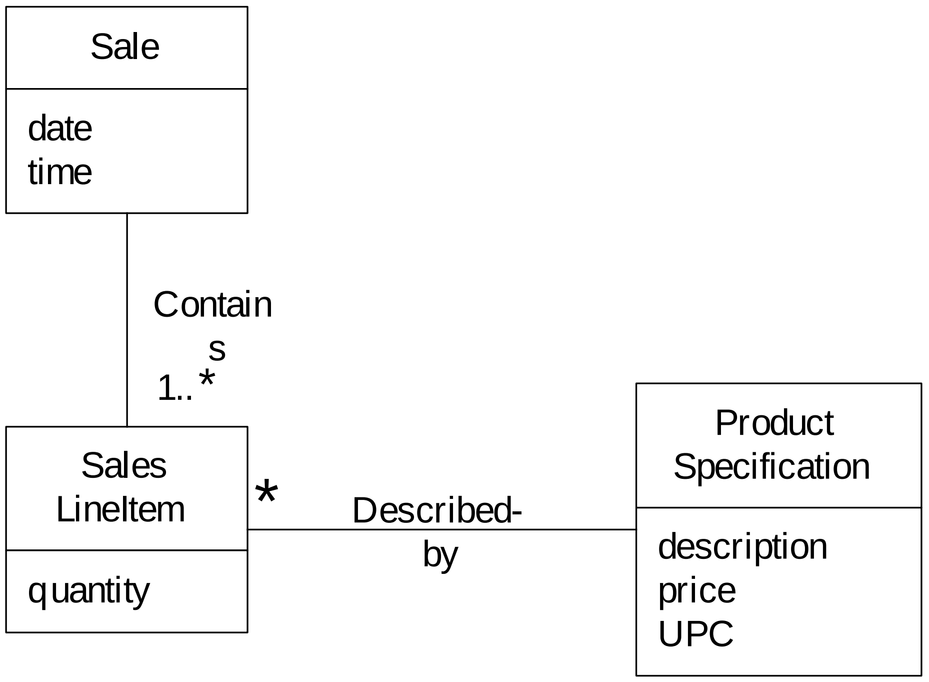
Figure 17.5 , page 283

Board has a composite aggregation relationship with *Square*

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

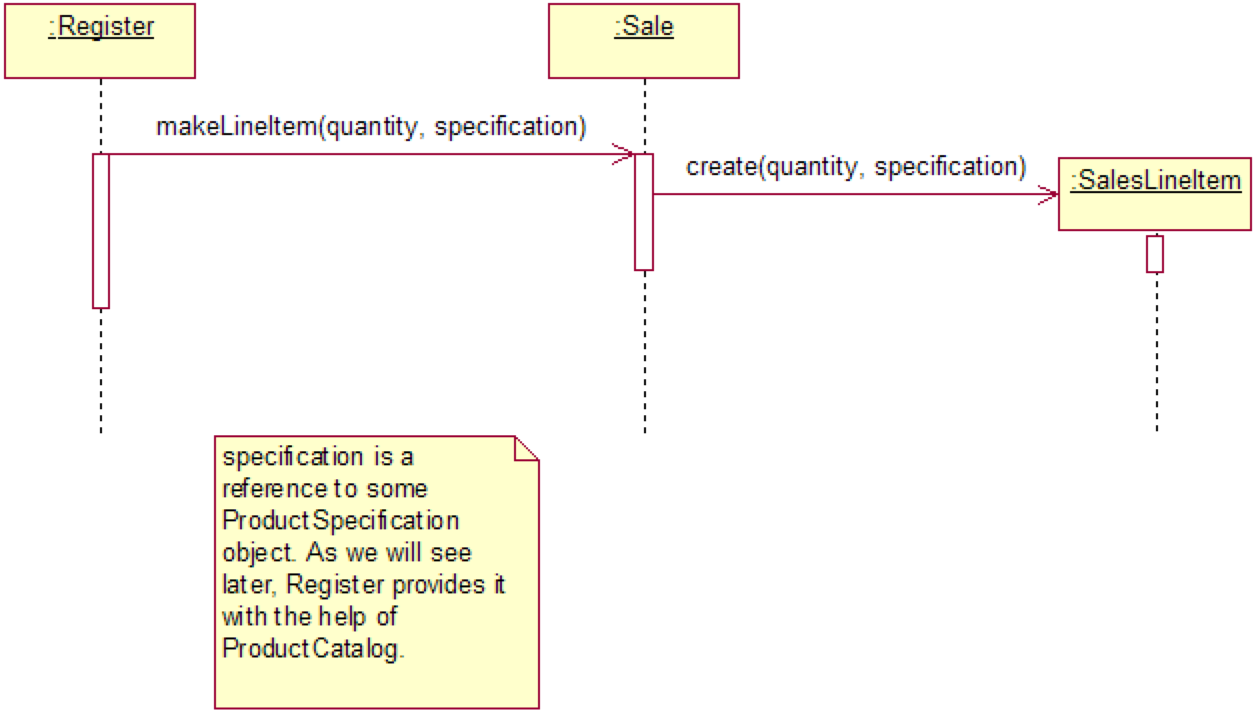
Creator: Another Example

Who should be responsible for the creation of a *SalesLineItem?*



Who should be responsible for the creation of a *SalesLineItem*?

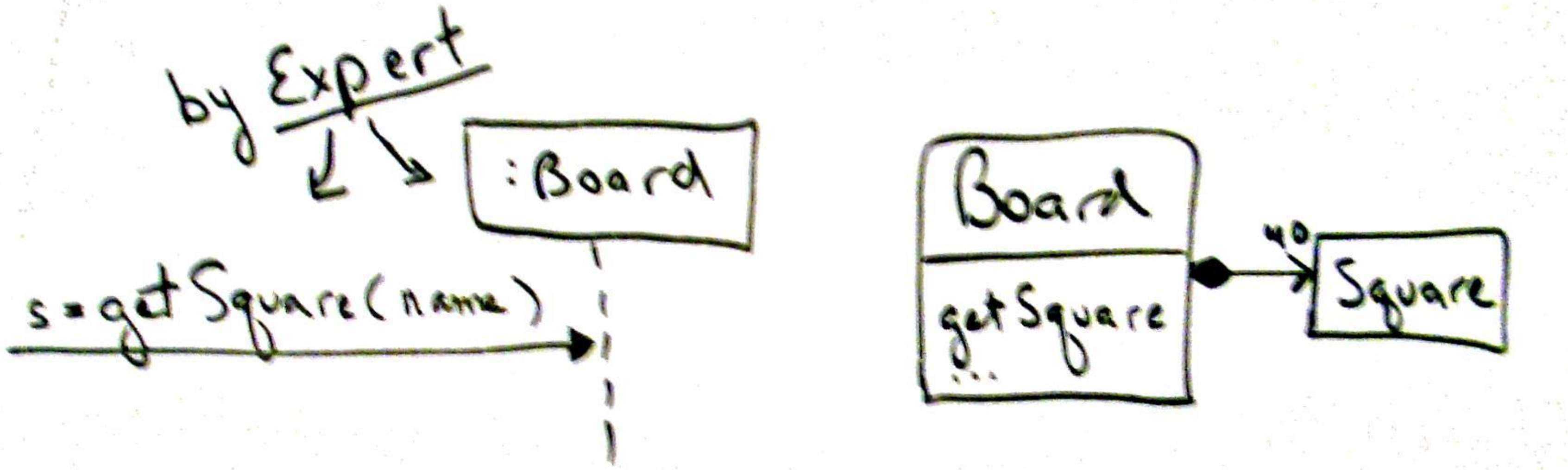
Answer : Sale
This assignment requires that a method *makeLineItem is defined in the Sale class.*


:Register
:Sale
makeLineItem(quantity, specification)
create(quantity, specification)
:SalesLineItem

specification is a reference to some ProductSpecification object. As we will see later, Register provides it with the help of ProductCatalog.

Information Expert pattern or principle

- **Problem: A system will have hundreds of classes. How do I begin to assign responsibilities to them?**
- **Solution:Assign responsibility to the *Information Expert–the class that has the information necessary to fulfill the responsibility.***
- E.g., Board has the information needed to get a Square



Mechanics

- **Step 1: Clearly state the responsibility**
- Step 2: Look for classes that have the information we need to fulfill the responsibility.
- Step 3:Domain Model or Design Model? See next slide
- Step 4:Sketch out some interaction diagrams.
- Step 5:Update the class diagram.

Question

- Do we look at the Domain Model or the Design Model to
- analyze the classes that have the information needed?
- Domain model illustrates conceptual classes, design model 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.

Ideas to remember

- Information Expert is a basic guiding principle used continuously in object design.
- The fulfillment of a responsibility often requires information that is spread across different classes of objects.
- This implies that there are many "partial" information experts who will collaborate in the task.
- Different objects will need to interact via messages to share the work.
- The Information Expert should be an early pattern considered in every design unless the design implies a controller or creation problem, or is contraindicated on a higher design level.

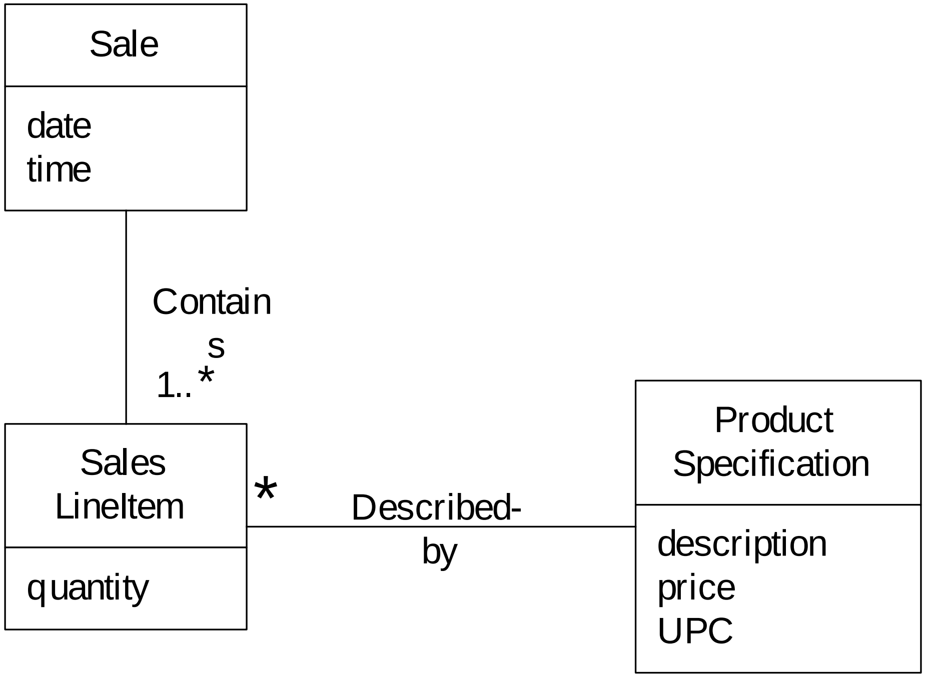
Contradictions

- In some situations a solution suggested by Expert is undesirable, usually because of problems in coupling and cohesion.
- For example, who should be responsible for saving a *Sale in a* database?
- If Sale is responsible, then each class has its own services to save itself in a database. The *Sale class must now contain* logic related to database handling, such as related to SQL and JDBC.
- This will raises its coupling and duplicate the logic. The design would violate a separation of concerns – a basic *architectural design goal.*
- Thus, even though by Expert there could be justification on object design level, it would result a poor architecture design.

Information Expert : Example

Who is responsible for knowing the grand total of a sale in a typical Point of Sale application?

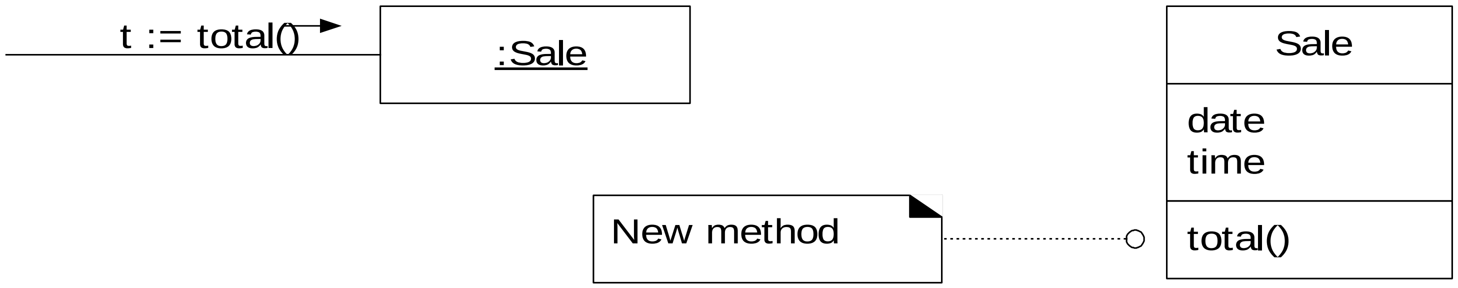
- To compute the grand total we need the total of a SalesLineItem.
- So we have to decide who calculates the total of a SalesLineItem.



Expert : Example

Need all *SalesLineItem* instances and their subtotals. Only *Sale* knows this, so *Sale* is the information expert.

Hence



Expert : Example

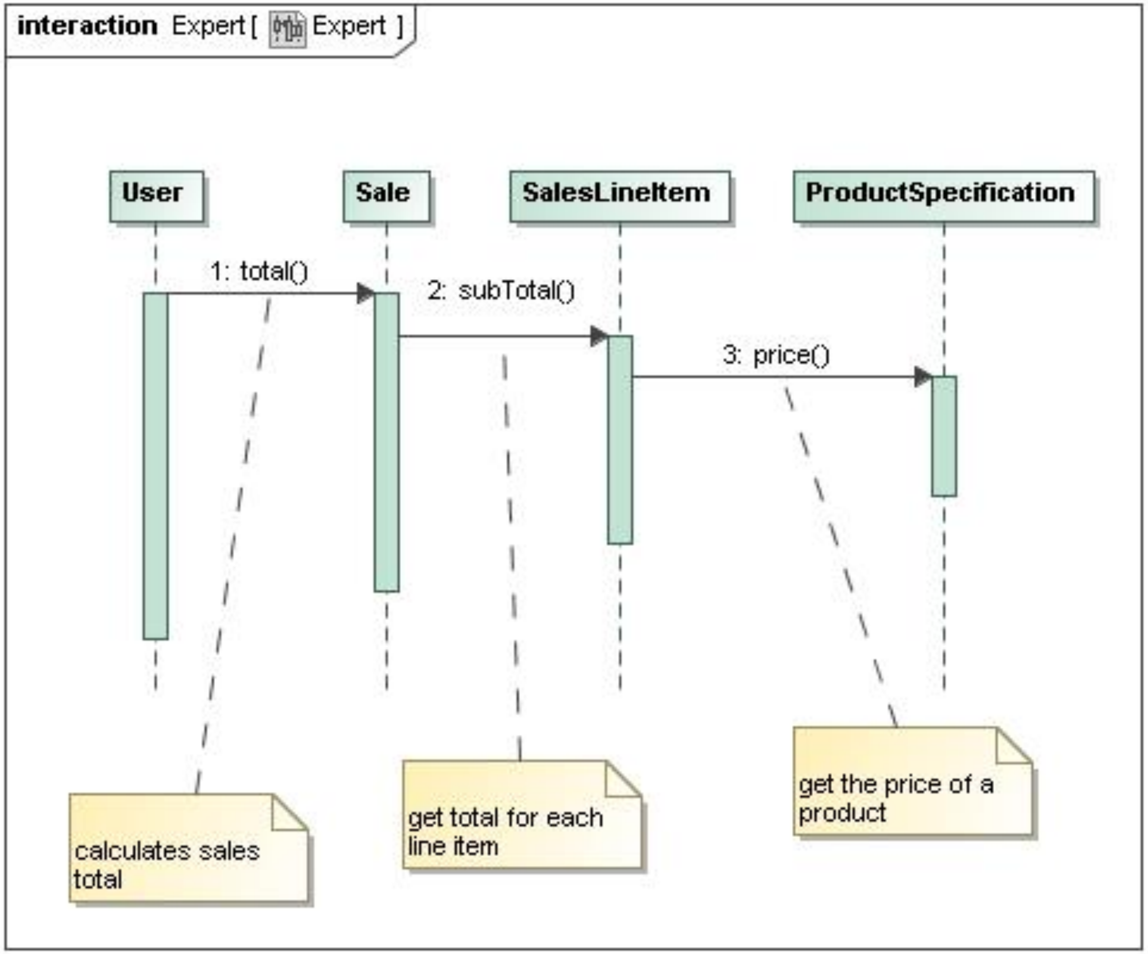
Hence responsibilities assigned to the 3 classes are as folllows

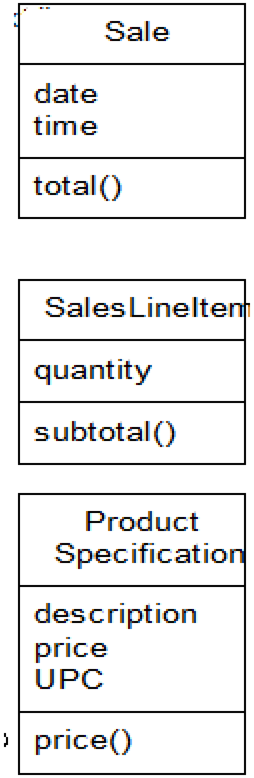
Class	Responsibility
Sale	knows sale total
SalesLineItem	knows line item subtotal
ProductSpecification	knows product price

Expert : Example

Subtotals are needed for each line item(multiply quantity by price).

By Expert, *SalesLineItem* is expert, knows quantity and has association with *ProductSpecification* which knows price.





Updated domain model

Advantages of using information expert

- Information expert has the effect of having a class with high cohesion.
  - ***Cohesion*** – the degree to which the information and responsibilities of a class are related to each other
- Cohesion is improved since the information needed for a responsibility is closely related to the responsibility itself
- Maintain encapsulation of information.
  - Classes use their own info to fulfill tasks
- Promotes low coupling (we will discuss coupling shortly)
- Promotes highly cohesive classes .
- (Caution) Can cause a class to become excessively complex.

Summary of Information expert

- Information encapsulation is maintained, since objects use their own information to fulfill tasks.
- This usually supports low coupling.
- Behavior is distributed across the classes that have the required information,
- thus encouraging cohesive "lightweight" class definitions that are easier to understand and maintain.

Coupling

- See http://msdn.microsoft.com/en-us/magazine/cc947917.aspx
- Much of software design involves the ongoing question,
  - where should this code go? (where to assign a responsibility?)
- Find the best way to organize to code to make it easier to write, easier to understand, and easier to change later.
- Three specific things to aim for:
  - ➢ Keep things that have to change together as close together in the code as possible.
  - ➢ Allow unrelated things in the code to change independently.
  - ➢ Minimize duplication in the code.

➢ Coupling among classes or subsystems is a measure of how interconnected those classes or subsystems are. Tight coupling means that related classes have to know internal details of each other, changes ripple through the system, and the system is potentially harder to understand.

Example of tightly coupled code taken from web page cited above

```
public class BusinessLogicClass { public void DoSomething() { // get some configuration
   int threshold = int.Parse(ConfigurationManager.AppSettings["threshold"]);
   String connectionString = ConfigurationManager.AppSettings["connectionString"];
   String sql = @"select * from things  // specify your retrieval condition
                        size > ";
    sql += threshold;
    using (SqlConnection connection = new SqlConnection(connectionString)) {
       connection.Open();
       SqlCommand command = new SqlCommand(sql, connection);
   using (SqlDataReader reader =command.ExecuteReader()) {
            while (reader.Read()) {
                string name = reader["Name"].toString();
                string destination = reader["destination"].toString();
               // do some business logic in here
                doSomeBusinessLogic(name, destination, connection);
} } } } }
```

Example (Cont’d)

Problem:

Our business logic code is intertwined with data-access concerns and configuration settings.

So what is the problem?

- The code is hard to understand because of the way the different concerns are intertwined.
- Any changes in data-access strategy, database structure, or configuration strategies will ripple through the business logic code as well because it's all in one code file.
- This business logic knows too much about the underlying infrastructure.
- We can't reuse the business logic code independent of the specific database structure or without the existence of the AppSettings keys.
- We also can't reuse the data-access functionality embedded in the BusinessLogicClass.
- What if we want to repurpose this business logic for usage against data entered directly into an Excel spreadsheet by analysts?
- What if we want to test or debug the business logic by itself? We can't do any of that because the business logic is tightly coupled to the data-access code.
- The business logic would be a lot easier to change if we could isolate it from the other concerns.

Our goals

- Make the code easier to read.
- Make our classes easier to consume by other developers by hiding the ugly inner workings of our classes behind well-designed APIs.
- Isolate potential changes to a small area of code.
- Reuse classes in completely new contexts.

Code smells

- It's good to know how to do the right things when designing new code
- It might be even more important to recognize when your existing code or design has developed problems.
- "code smell" is a tool that you can utilize to spot potential problems in code.
- (reminder) A code smell is a sign that something may be wrong in your code.
- It doesn't mean that you need to rip out your existing code and throw it away on the spot, but you definitely need to take a closer look at the code that gives off the offending "smell."
- Many, if not most, of the commonly described code smells are signs of poor cohesion or harmful tight coupling.

Reminder: resolving code smells help us decrease coupling

- **Divergent Changes** A single class that has to be changed in different ways for different reasons. This smell is a sign that the class is not cohesive. You might refactor this class to extract distinct responsibilities into new classes.
- **Feature Envy** A method in ClassA seems way too interested in the workings and data fields of ClassB. The feature envy from ClassA to ClassB is an indication of tight coupling from ClassA to ClassB. The usual fix is to try moving the functionality of the interested method in ClassA to ClassB, which is already closer to most of the data involved in the task.
- **Shotgun Surgery** A certain type of change in the system repeatedly leads to making lots of small changes to a group of classes. Shotgun surgery generally implies that a single logical idea or function is spread out over multiple classes. Try to fix this by pulling all the parts of the code that have to change together into a single cohesive class.

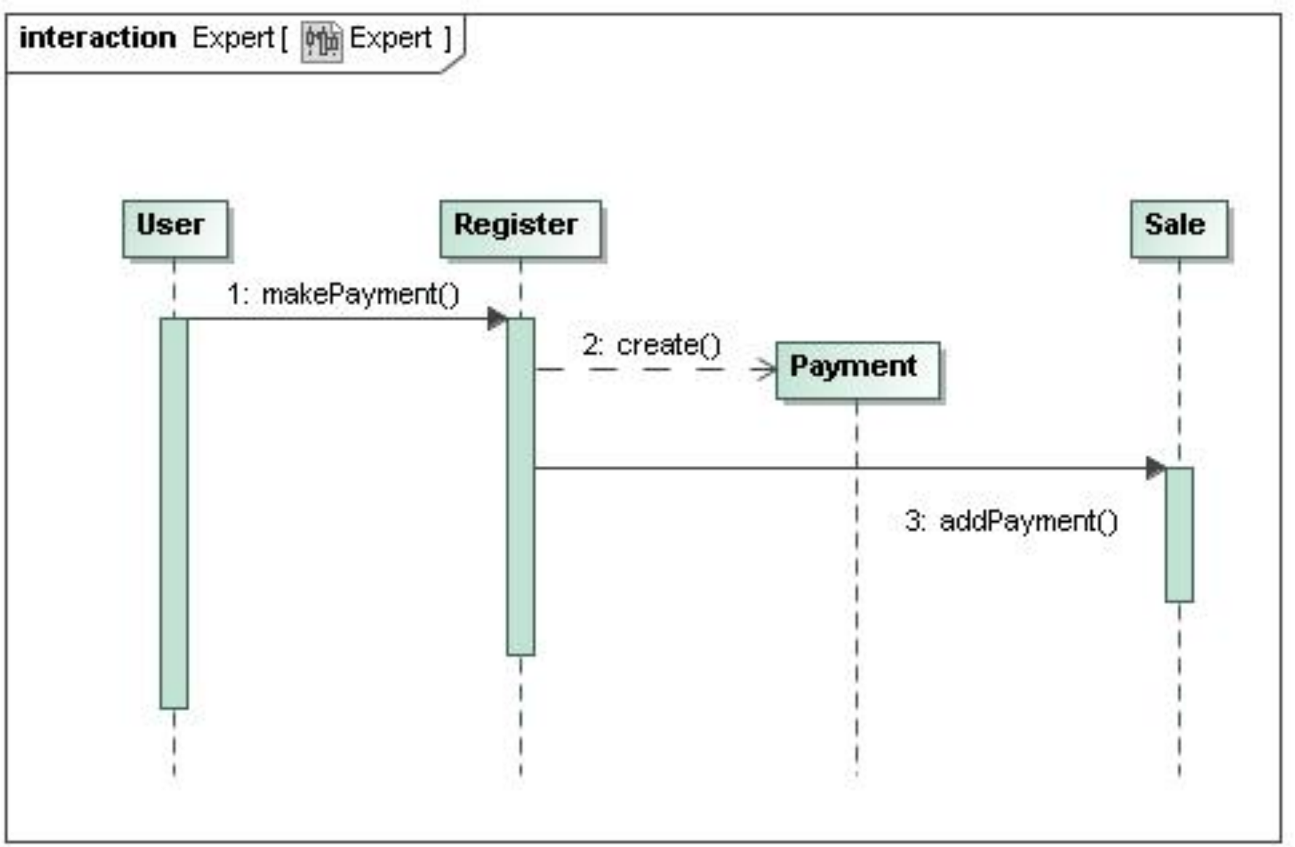
Summarize concept of Coupling

- **Coupling** refers to connectedness.
- **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.
- A class, for example, with high (or strong) coupling relies on many other classes.
- Tight coupling means that related classes have to know internal details of each other, changes ripple through the system, and the system is potentially harder to understand.

Example

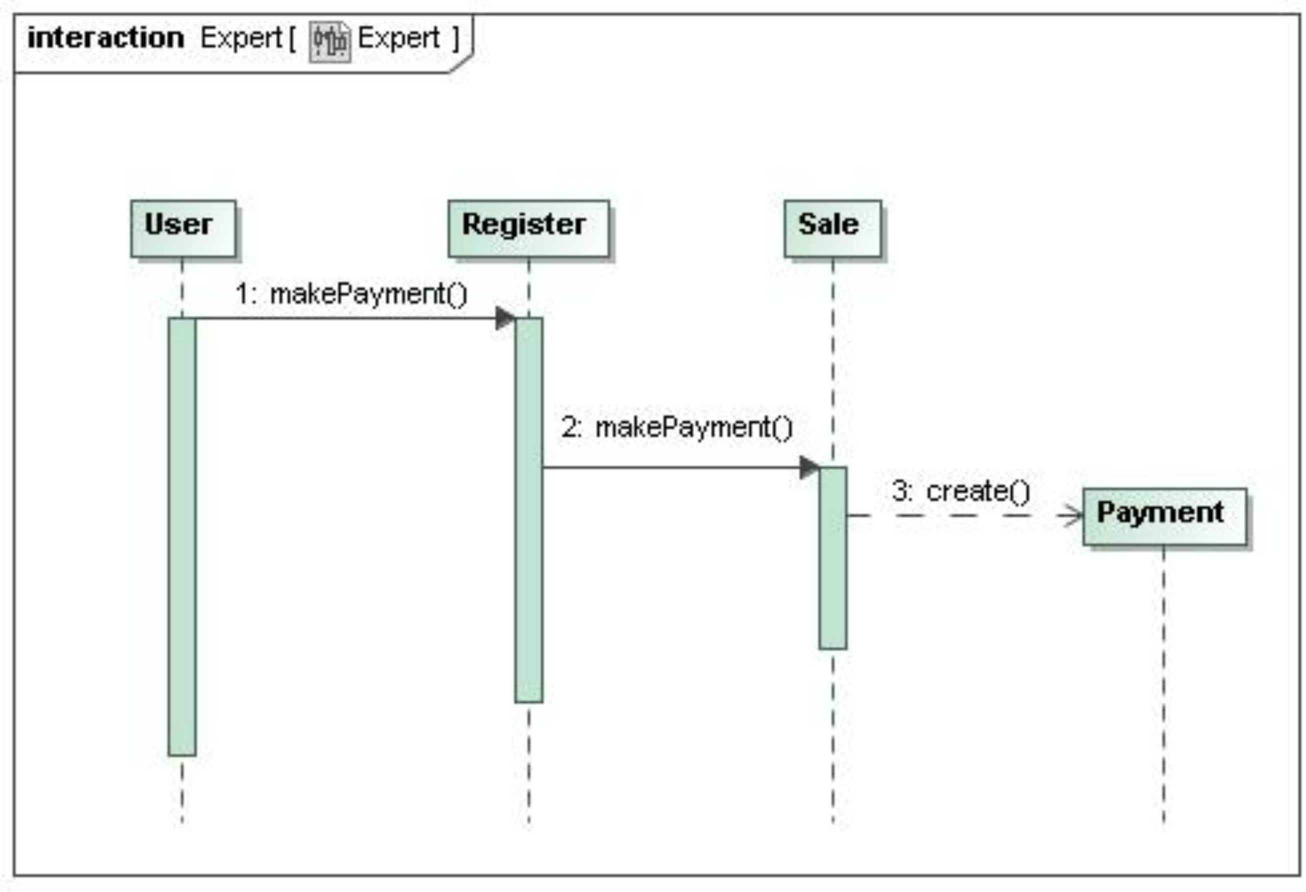
- Suppose at a departmental store, the user (The clerk at the teller) wishes to create a sale object, get payment for the sale and record details of the sale and the payment.
- The domain mode includes a Register to record these details.
- Since Register records the sale, creator pattern suggests the Register should be responsible for payment and the details of the sale.

Version 1 of adding a payment to Sale



- Here create() return an object p of class Payment.
- addPayment has the object p as an argument since the payment object must be updated with the details of the sale.
- In this version Register does all the work

Version 2 of adding a payment to Sale



Sale creates a Payment – as opposed to Register creating it.
Sale must know about payment so why don't we decouple Payment from Register? This reduces coupling of Register.
Does this conflict with creator principle we talked about?

Why high coupling is bad

- 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.

Why low coupling is desirable?

- A change in one area of an application will require less changes throughout the entire application. In the long run, this could alleviate a lot of time, effort, and cost associated with modifying and adding new features to an application.
- Our goal is to design for low coupling, so that changes in one element (sub-system, system, class, etc.) will limit changes to other elements.
- Low coupling supports increased reuse.
- Taken to the extreme, what if we were to design a set of classes with no coupling. Is this possible?
- We can't avoid coupling, but we want to make sure we understand the implications of introducing it and/or the tradeoffs of reducing it.

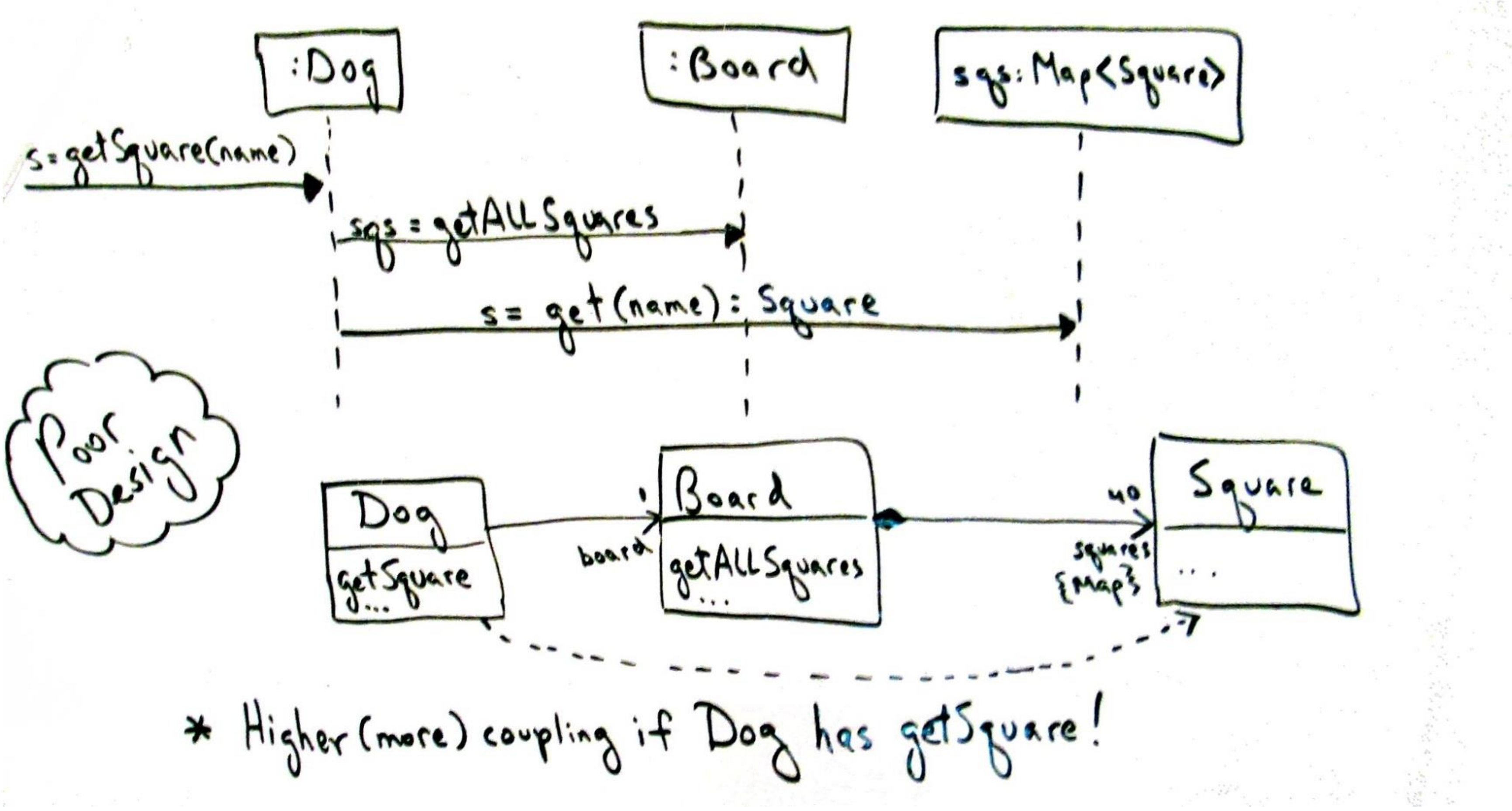
Low Coupling Pattern

- Problem: How to reduce the impact of change and encourage reuse?
- Solution: Assign a responsibility so that coupling (linking classes) remains low.
- Advantages
  - ❑ Classes are easier to maintain
  - ❑ Easier to reuse by hiding the ugly inner workings of our classes behind well-designed APIs.
  - ❑ Changes are localised
- Isolate potential changes to a small area of code.

Common forms of coupling

- *TypeX is coupled to TypeY if:*
  - *TypeX has an attribute (or instance variable) of TypeY.*
  - A *TypeX object calls on services of a TypeY object.*
  - *TypeX has a method that references an instance of TypeY. 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.*
- A subclass is strongly coupled to its superclass. The decision to derive from a superclass needs to be carefully considered since it is such a strong form of coupling.

Why does the following design violate Low Coupling?



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

Problems with the couplings between dog and square

- Both Dog and Board must both know about Square objects
- A solution where only Board knows about Square is better because the overall coupling is lower.
- Idea in a nutshell:
- In general, if you need to assign a new responsibility to an object, first look to assign the responsibility to objects that are already information experts on class X. This will keep coupling low.
- Giving responsibility anywhere else will increase coupling since more information has to be shared or moved (The square in the map collection has to be shared with the dog object, away from their home in the Board object).

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)