5 Elaboration Iteration 1 Basics Third Week

庞雄文
Tel：18620638848
Wechat: augepang
QQ: 443121909

Contents

- Part3: Elaboration Iteration 1 Basics
  - UML Class Diagram
  - GRASP: Designing Objects with Responsibilities
  - Mapping Designs to Code

Do the right thing right

UML Class Diagram

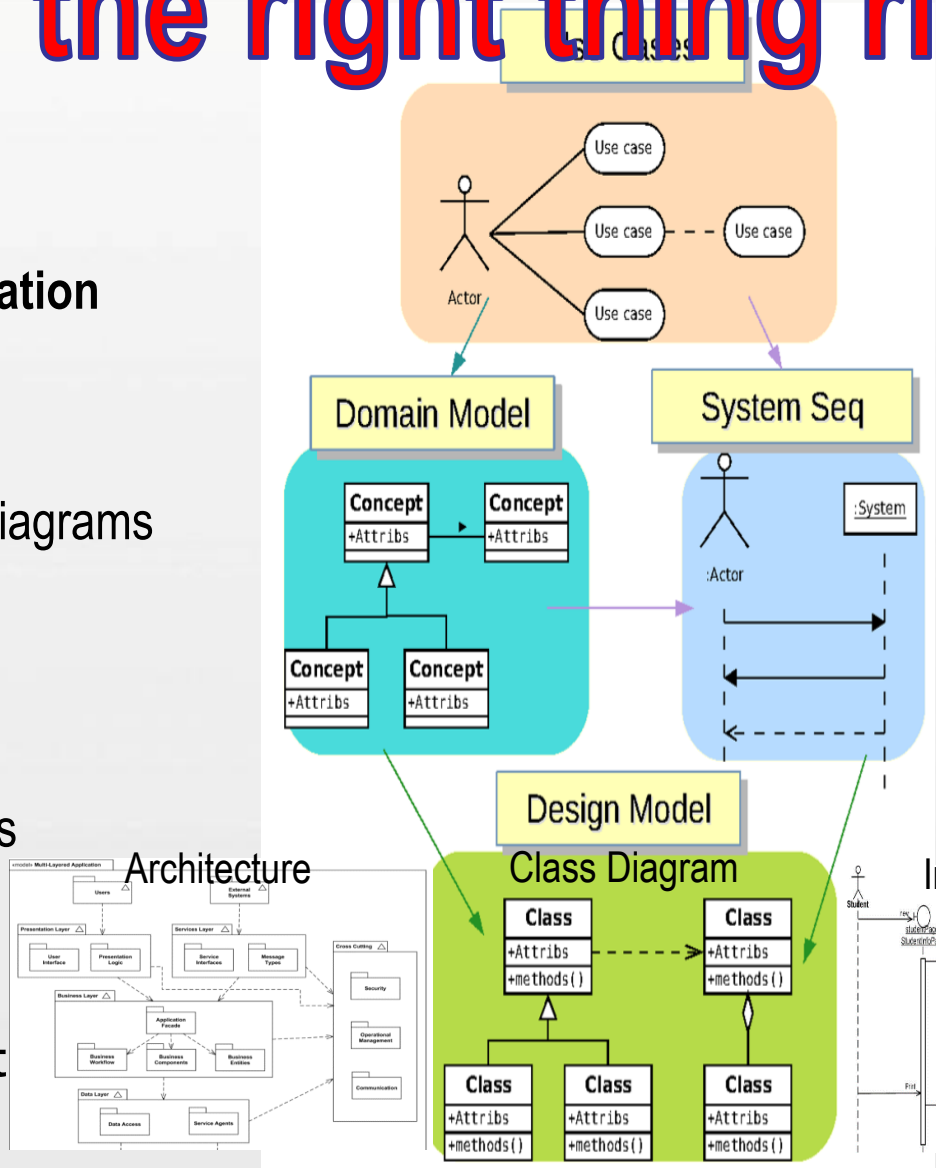
Do the right thing right

- Design road
  - We have described:
    - Use case,

      Supplementary pecification
    - OOA:
      - domain model
      - System sequence diagrams
      - operation contract
    - OOD
      - Architecture
      - Interaction Diagrams
  - We now describe

    Class Diagram
  - Afterwards, we go into t

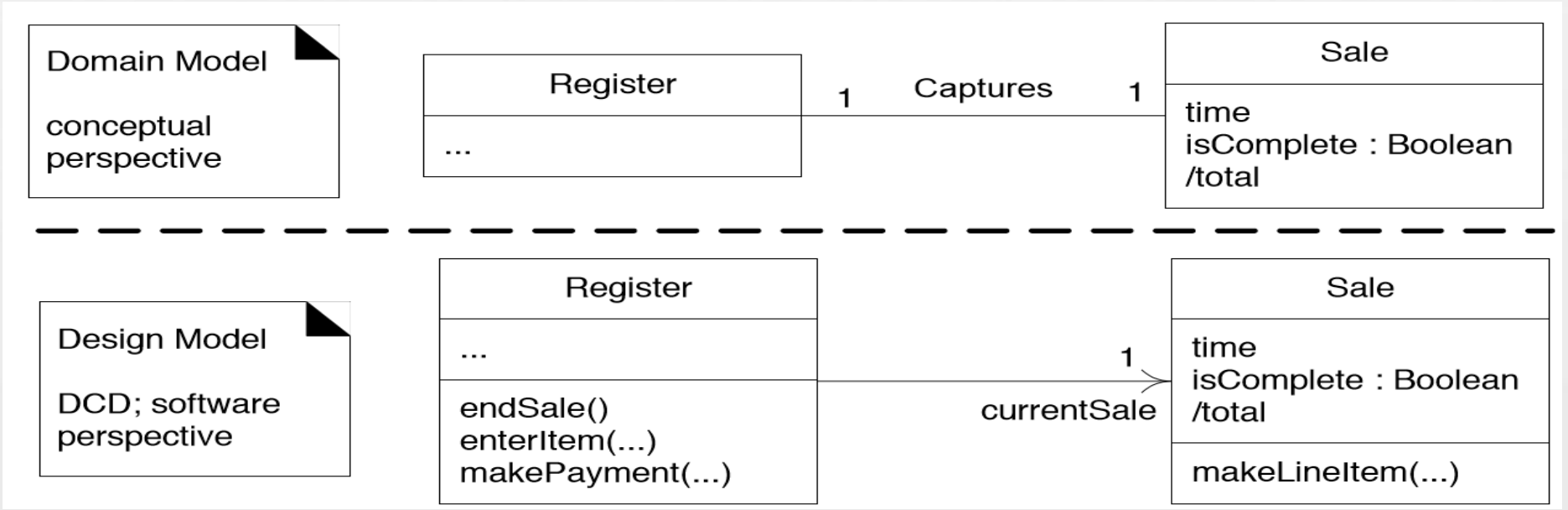
    GRASP



UML Class Diagram

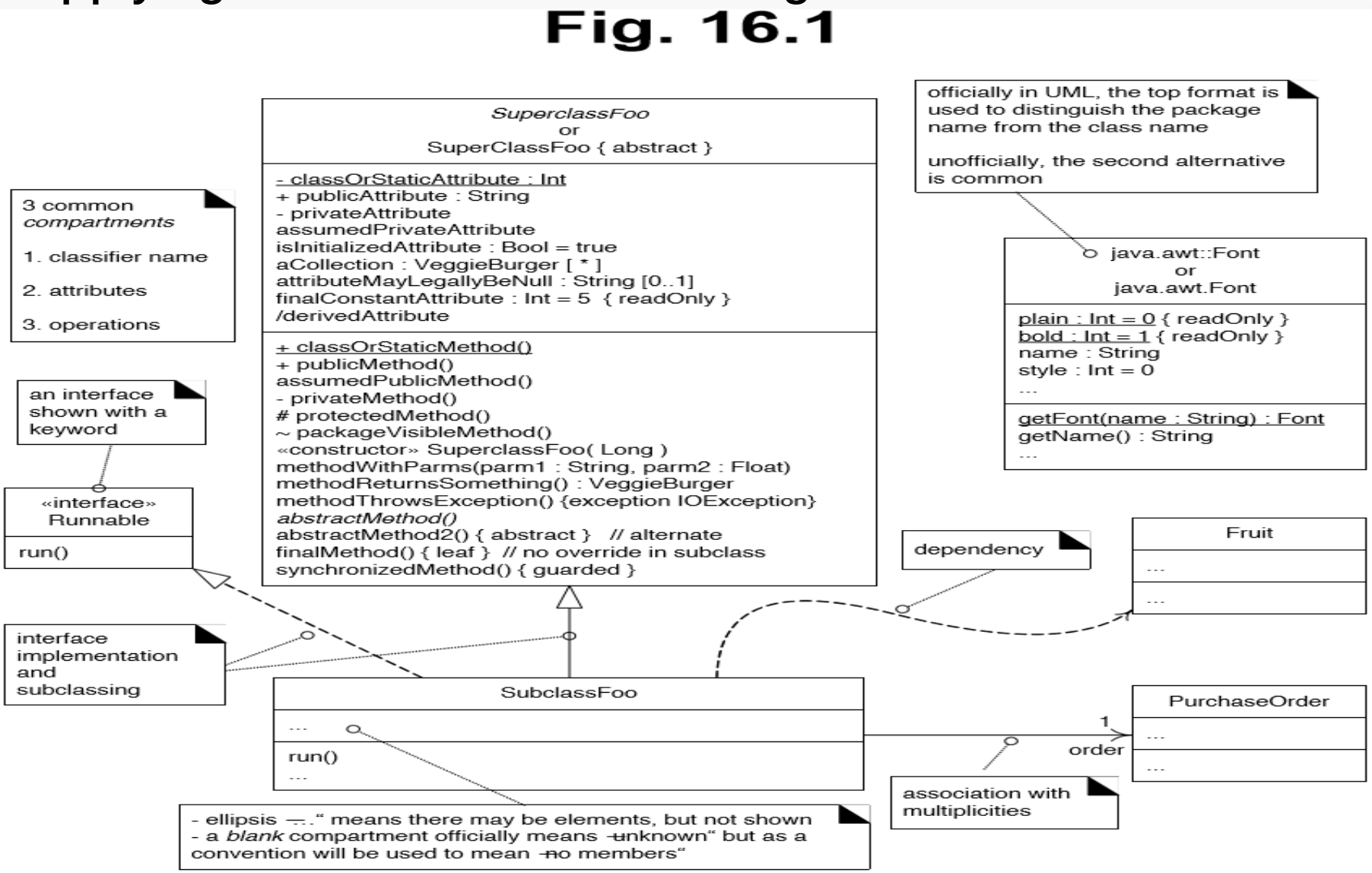
Class Diagram

- The UML includes class diagrams to illustrate classes, interfaces, and their associations.
- They are used for static object modeling
- **UML class diagrams in two perspectives**
  - Conceptual perspective
  - Software perspective - design class diagram



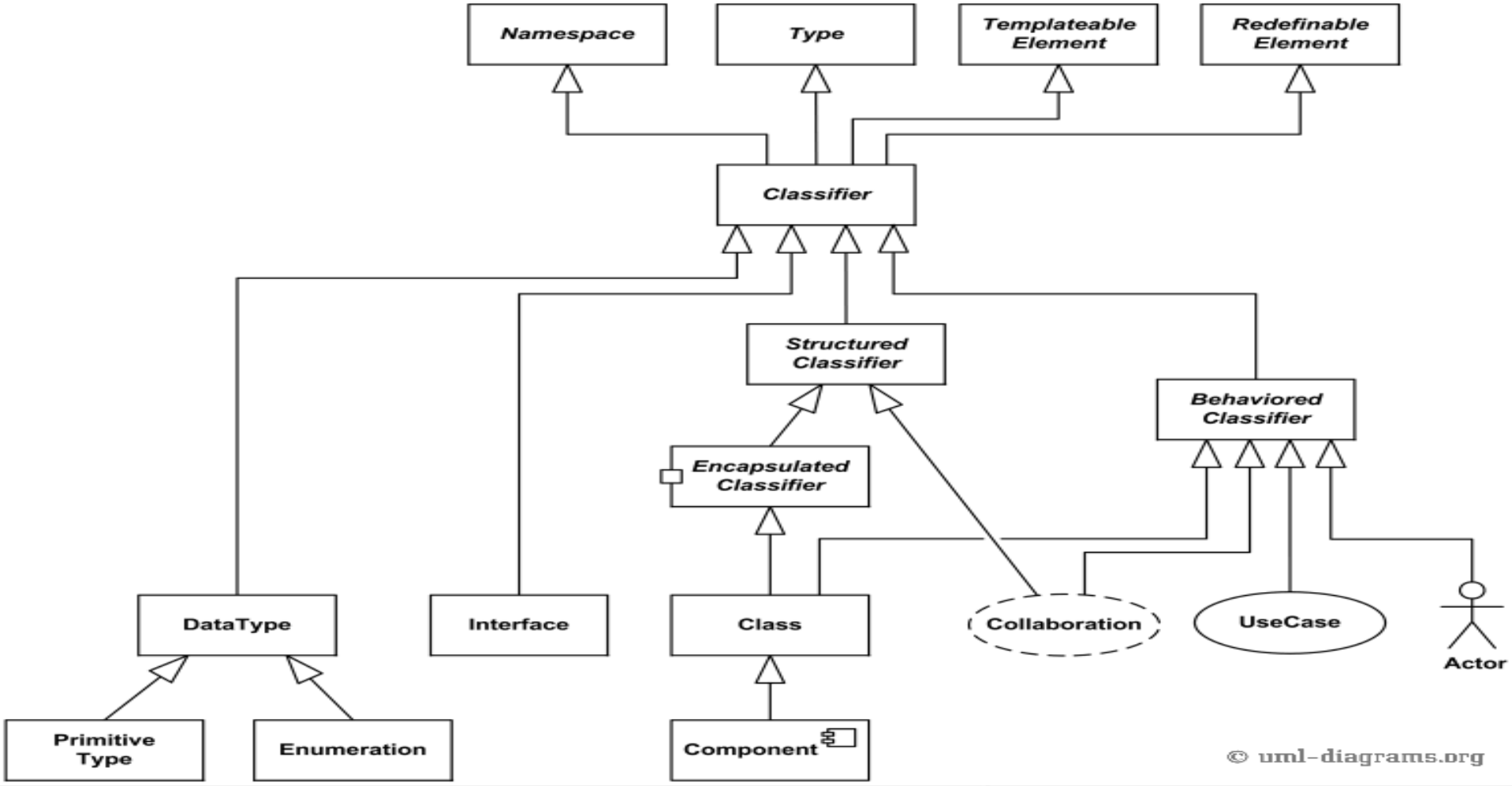
UML Class Diagram

- Applying UML: Common Class Diagram Notation



UML Class Diagram

- **UML Classifier**
  - a model element that describes behavioral and structure features



UML Class Diagram

UML Attributes: Attribute Text and Association Lines

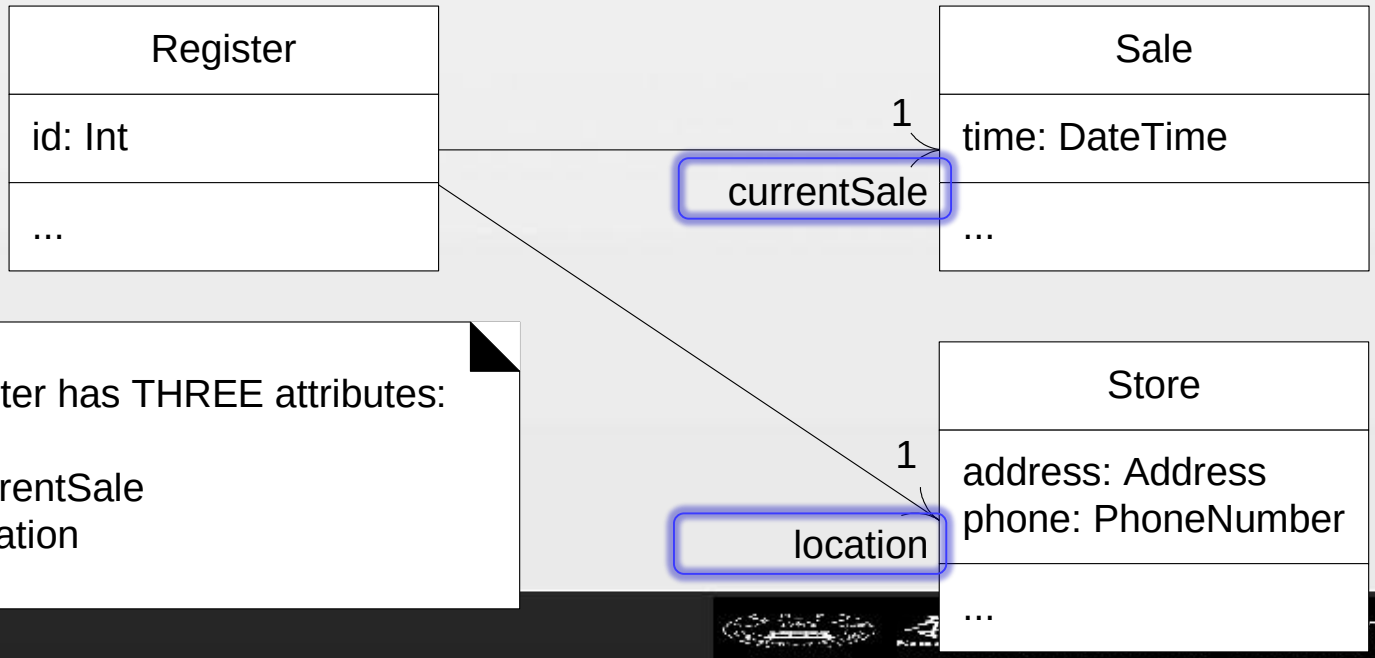
- **attribute text notation**
  - visibility /name : type multiplicity = default {property-string}
    - <visibility> ::='+'|'-'|'#'|'~'
    - 'readOnly' | 'ordered' | 'unordered' | 'unique' | 'nonunique' | 'seq' | 'sequence'|'id' ….
- **association line**
  - navigability arrow, role name (**only at the target end to show the attribute name**), multiplicity value, property string such as {ordered} or {ordered, List}
- both

UML Class Diagram

- UML Attributes: Attribute Text and Association Lines

Multiplicity	Option	Cardinality
0..0	0	Collection must be empty
0..1		No instances or one instance
1..1	1	Exactly one instance
0..*	*	Zero or more instances
1..*		At least one instance
5..5	5	Exactly 5 instances
m..n		At least m but no more than n instances

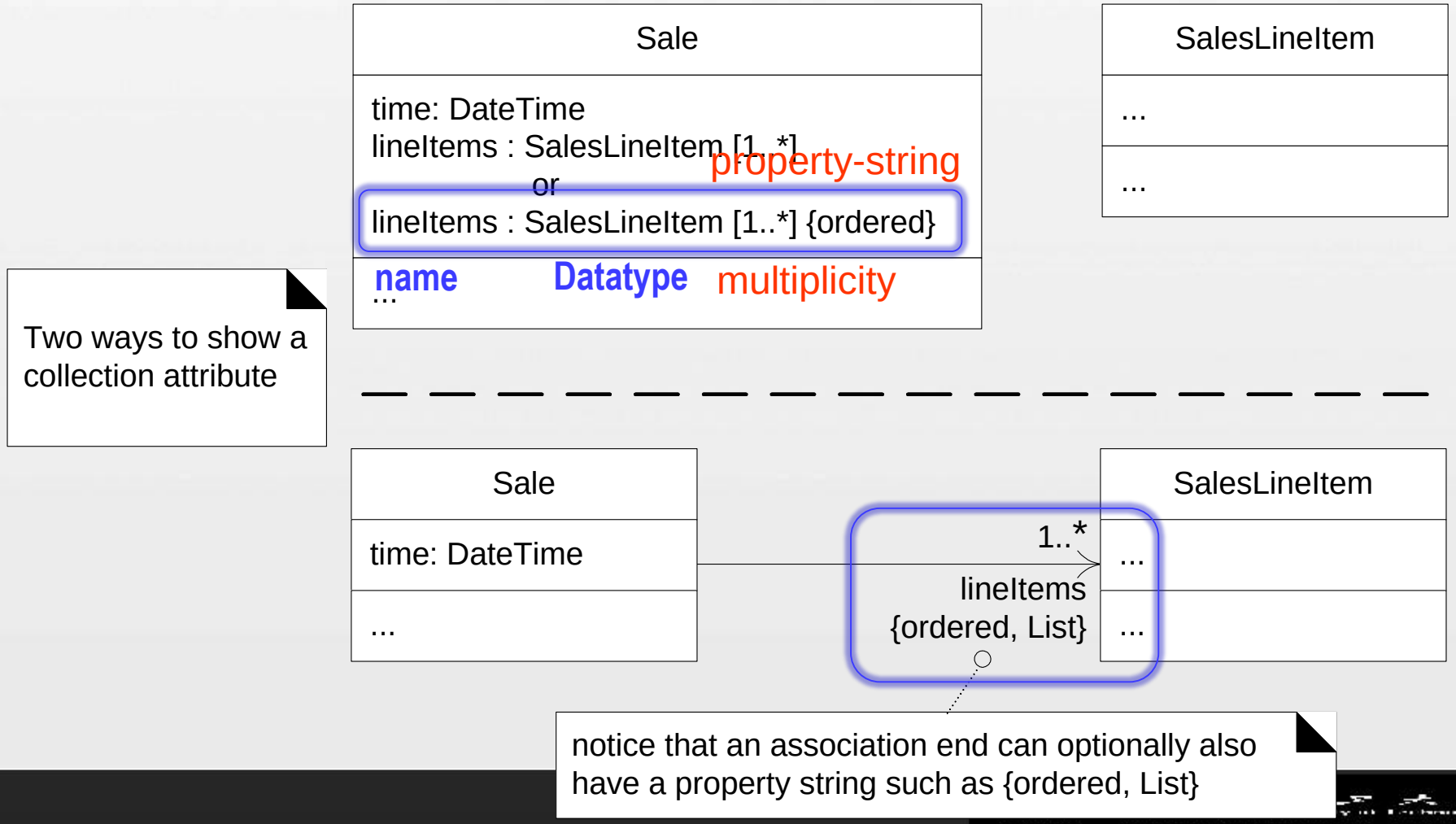
applying the guideline to show attributes as attribute text versus as association lines



Register has THREE attributes:
1. id
2. currentSale
3. location

UML Class Diagram

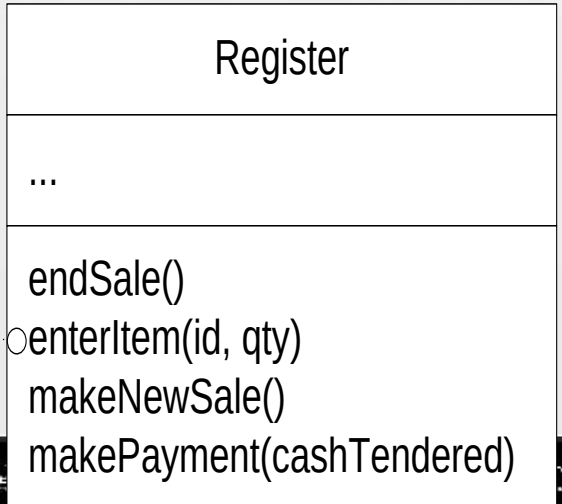
- UML Attributes: Attribute Text and Association Lines
  - How to Show Collection Attributes with Attribute Text and Association Lines?



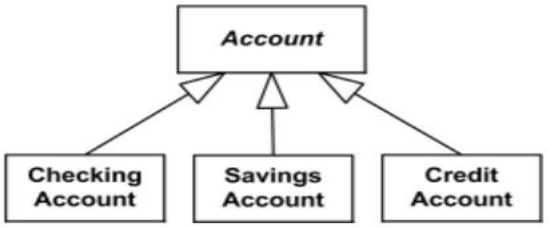
UML Class Diagram

- **Operations and Methods**
  - **Operation** is a **behavioral feature** that may be owned by an **interface**, **data type**, or **class**
    - **visibility name (parameter-list) : return-type {property-string}**
      - ‘query’|‘ordered’|‘unordered’| ‘unique’|‘nonunique’|‘seq’| ‘sequence’|….
  - **Method**
    - **Method** is the implementation of an operation. It specifies the algorithm or procedure associated with an operation.
      - in class diagrams, with a UML note symbol stereotyped with «method»

```
«method»
// pseudo-code or a specific language is OK
public void enterItem( id, qty )
{
  ProductDescription desc = catalog.getProductDescription(id);
  sale.makeLineItem(desc, qty);
}
```



UML Class Diagram



Generalization

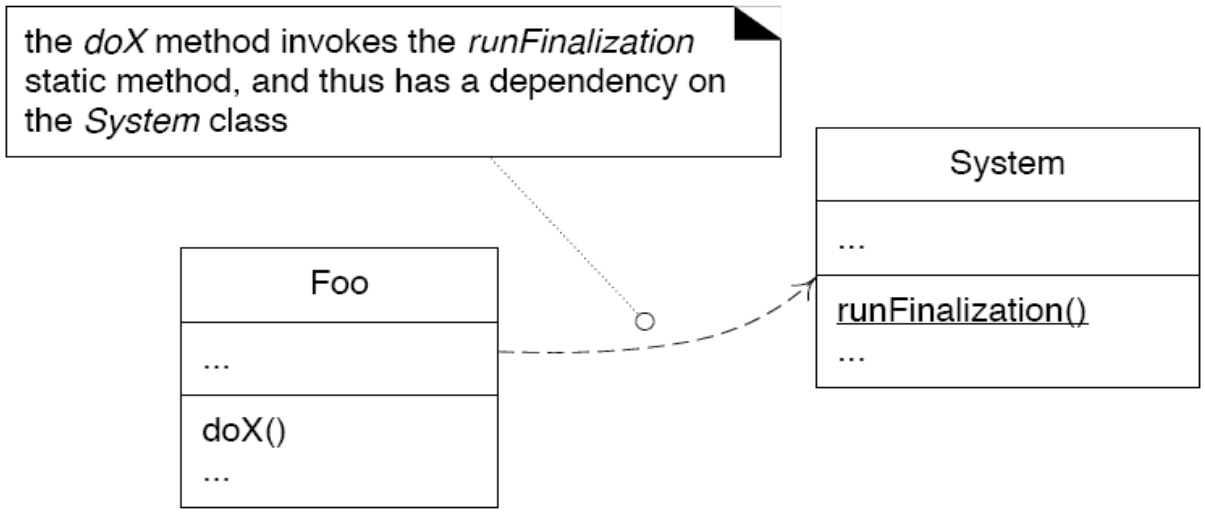
- A **generalization** is a binary taxonomic (i.e. related to classification) **directe**
  relationship between a more general **classifier** (superclass) and a more speci
  classifier (subclass)

- **Abstract Class**
  - **abstract class** does not have complete declaration and "typically" can not be instantiated
  - The name of an **abstract class** is shown in **italics**
- ***Abstract Operation***
  - operation without implementation
  - was marked as **{abstract}**.

UML Class Diagram

Dependency

- indicates that a **client element** (of any kind, including classes, packages, use cases, and so on) has knowledge of another supplier element
  - having an attribute of the supplier type
  - sending a message to a supplier;
  - receiving a parameter of the supplier type
  - the supplier is a superclass or interface
- Dependency can be viewed as another version of coupling
- **is illustrated with a dashed arrow line from the client to supplier**

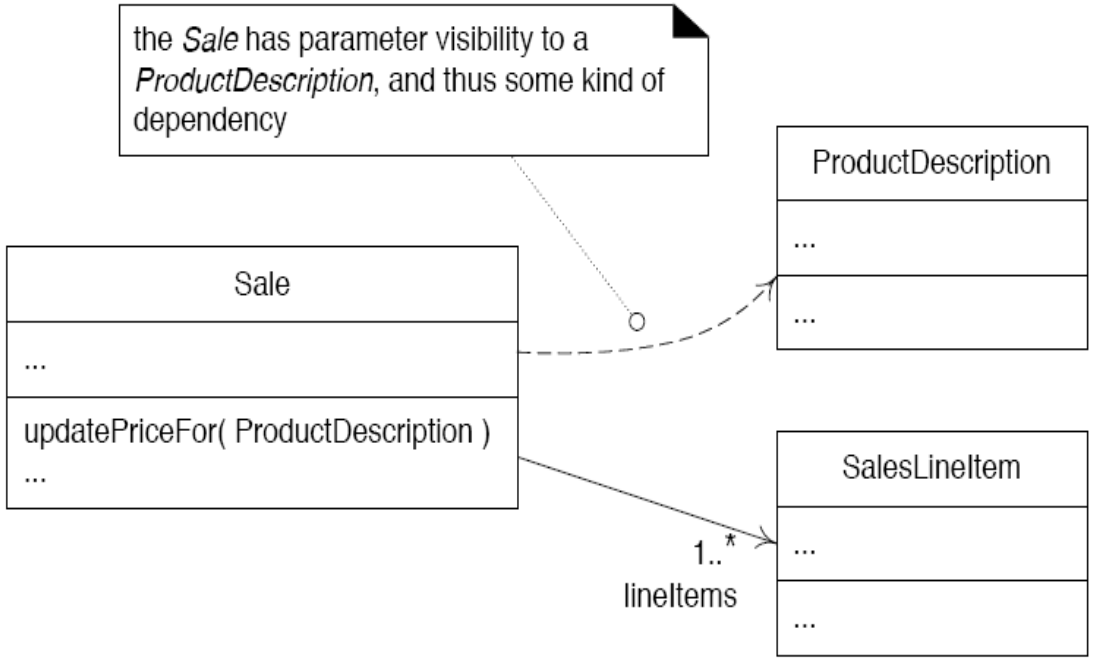


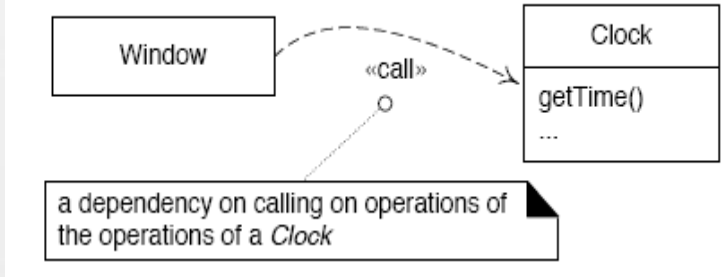
In class diagrams use the dependency line to depict global parameter variable, local variable and static-method (when a call is made to a static method of another class) dependency between objects

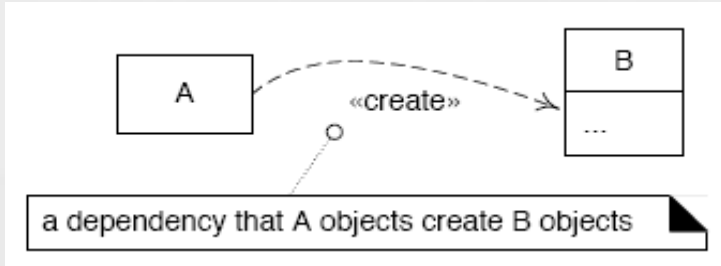
UML Class Diagram

■ Dependency. Example

```
public class Sale
{
public void updatePriceFor( ProductDescription description )
{ Money basePrice = description.getPrice(); //… }
// … }
```

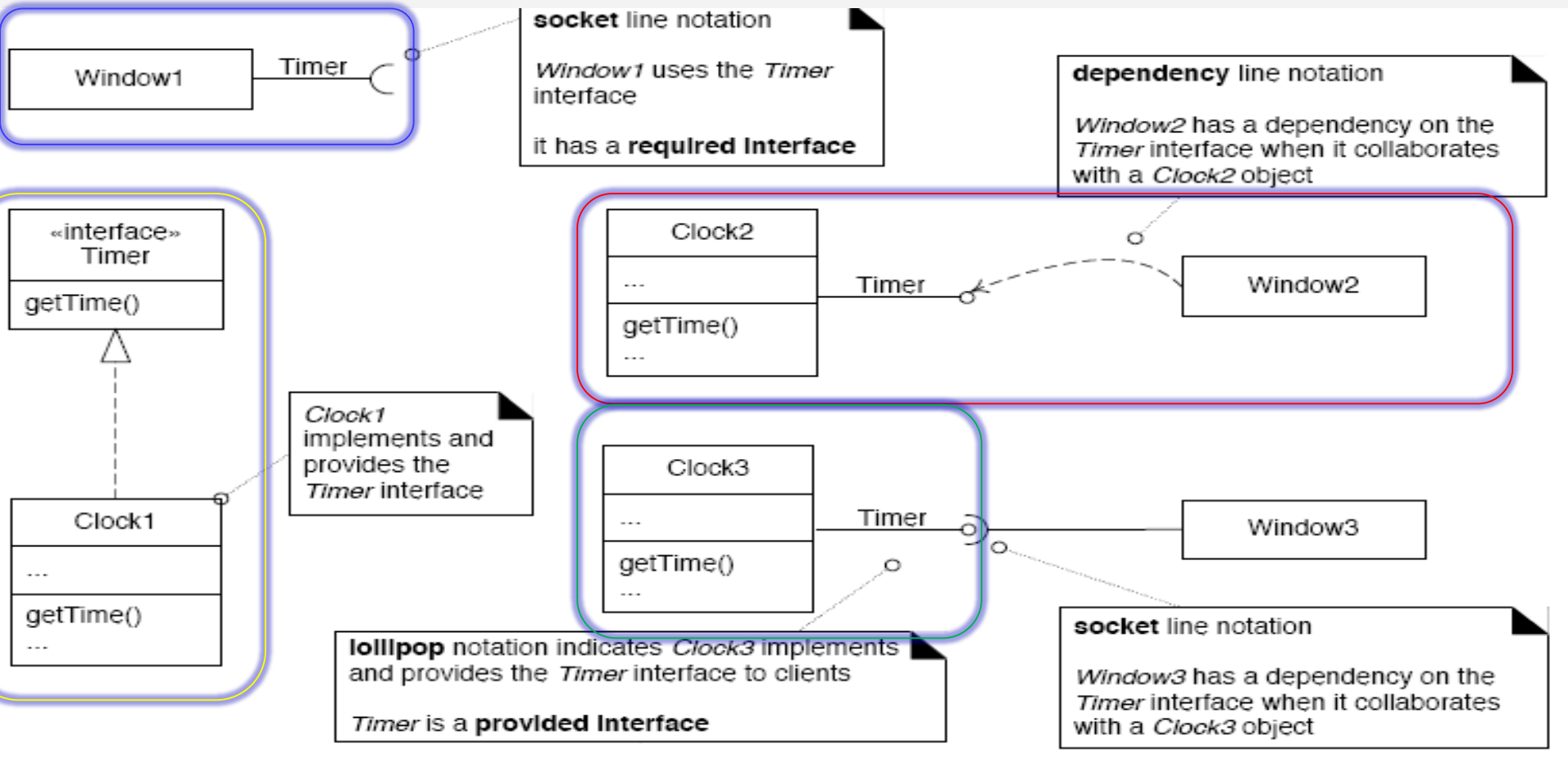




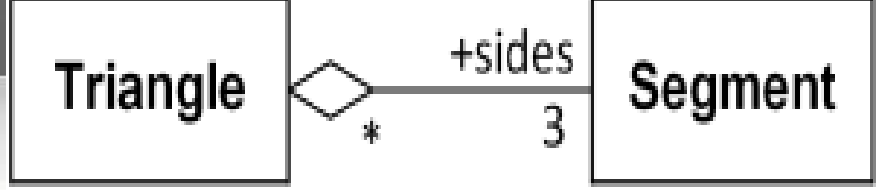


UML Class Diagram

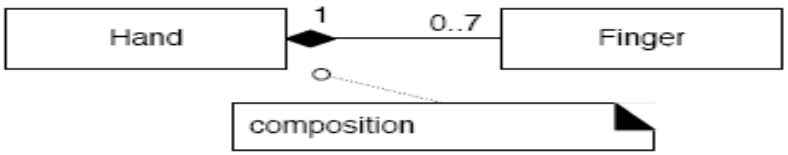
- **Interfaces**
  - An **interface** is a **classifier** that declares of a set of coherent public features and obligations. An interface specifies a **contract**
  - . interface implementation is formally called ***interface realization***

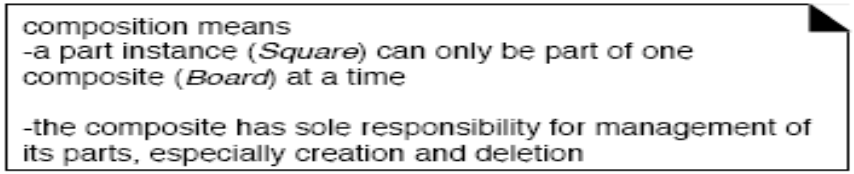


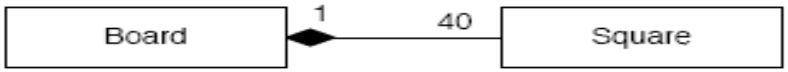
UML Class Diagram

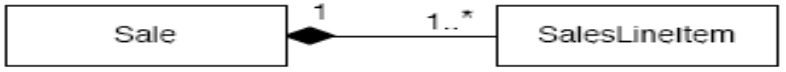


- **Composition Over Aggregation -- whole-part relationships**
  - **Aggregation (**Shared aggregation**)**
    - part instance is independent of the composite
    - depicted as association decorated with a *hollow diamond*
  - **Composition**
    - strong kind of whole-part aggregation
    - **A composition relationship implies**
      - part (such as a Square) belongs to only one composite (such as one Board) at a time
      - the part must always belong to a composite
      - the composite is responsible for the creation and deletion of its part
    - **filled diamond on an association line.**



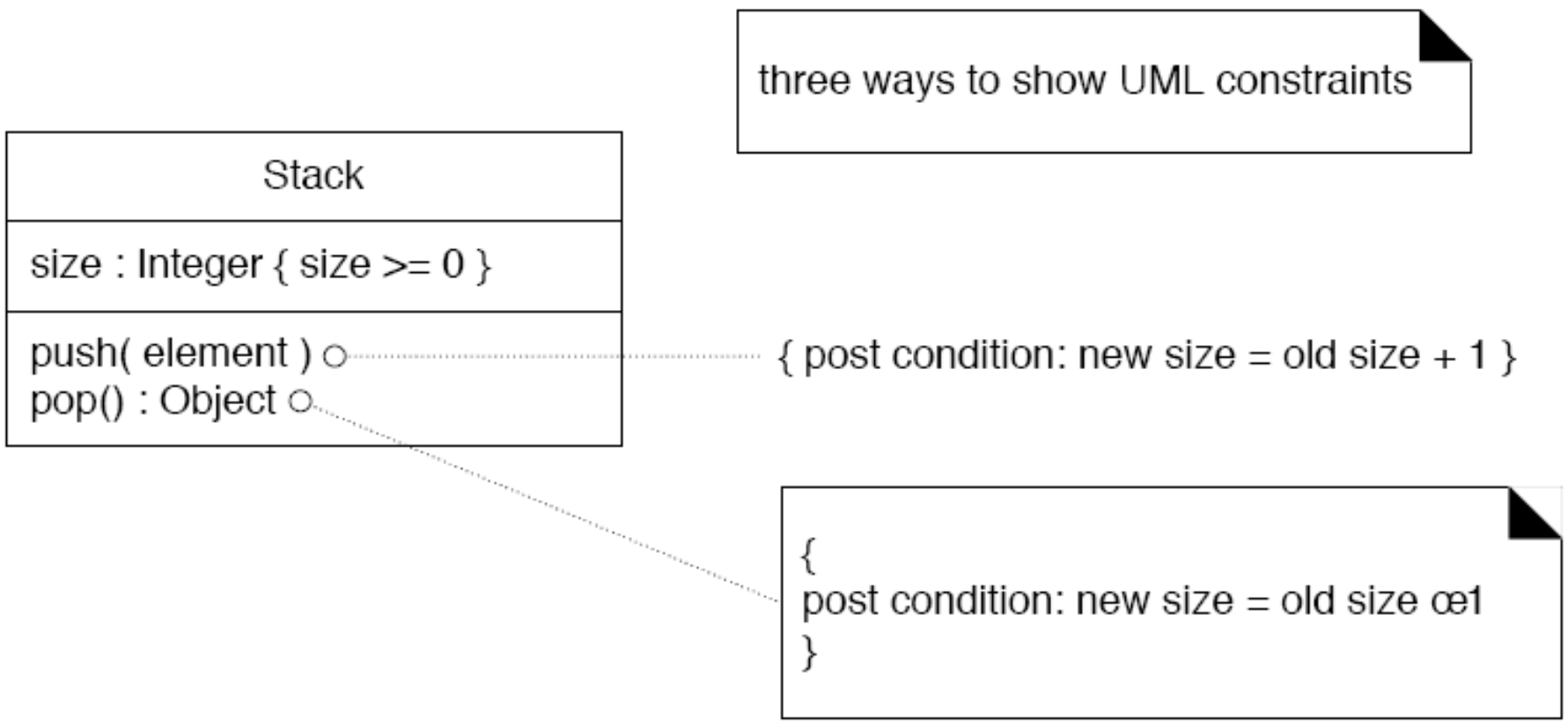






UML Class Diagram

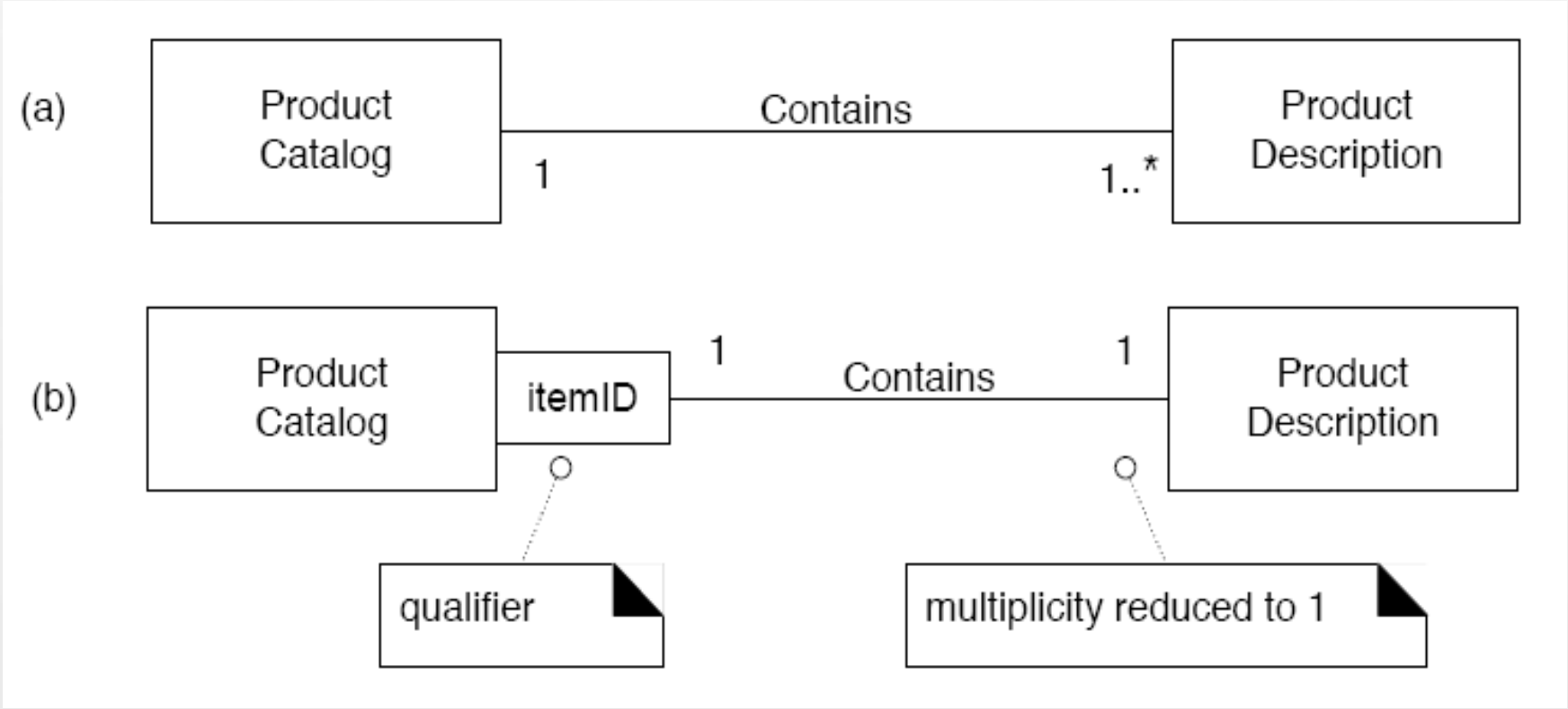
- **Constraints**
  - restriction or condition on a UML element.
  - visualized in text between braces;



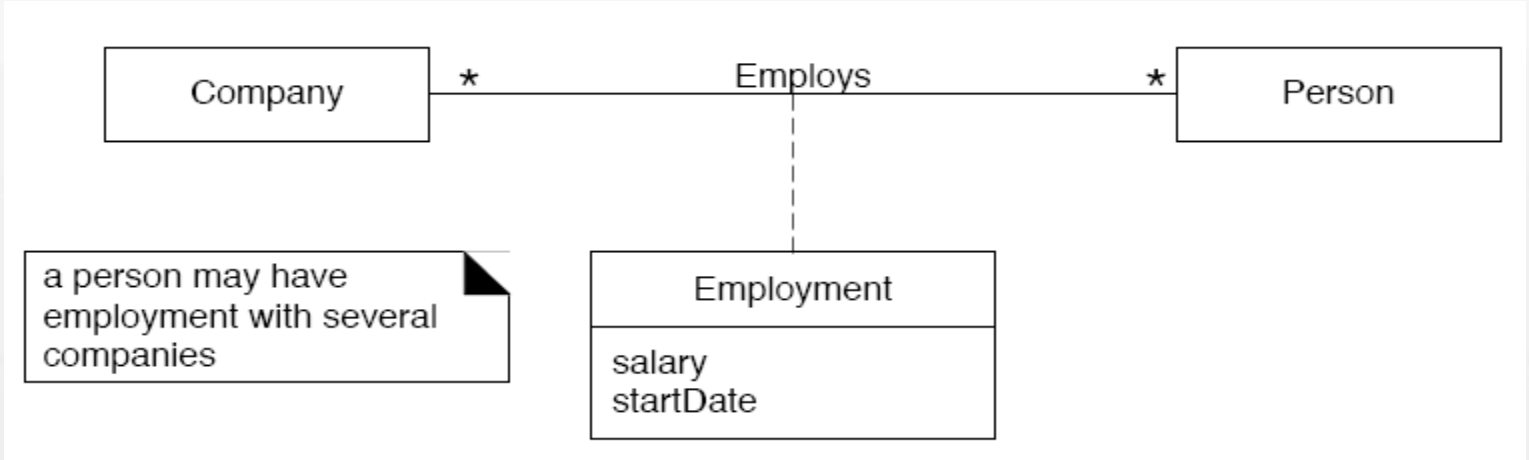
UML Class Diagram

Qualified Association

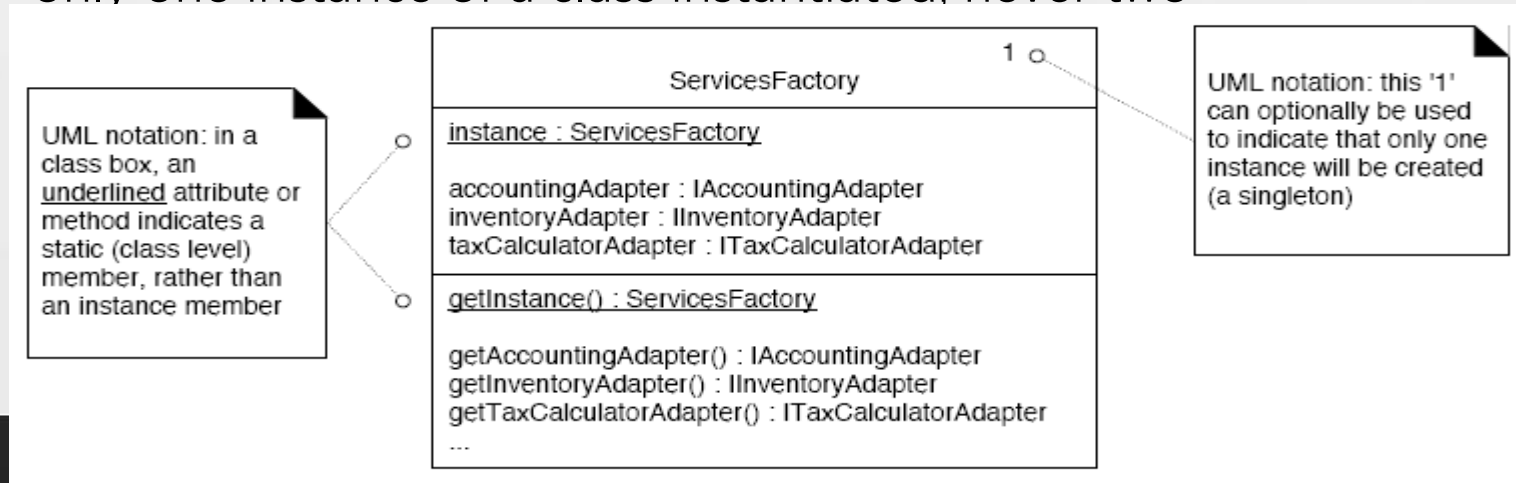
- qualified association has a **qualifier** that is used to **select an object** (or objects) from a larger set of related objects
- qualification **reduces the multiplicity**



UML Class Diagram

- **Association Class**
  - treat an association itself as a class
  - 

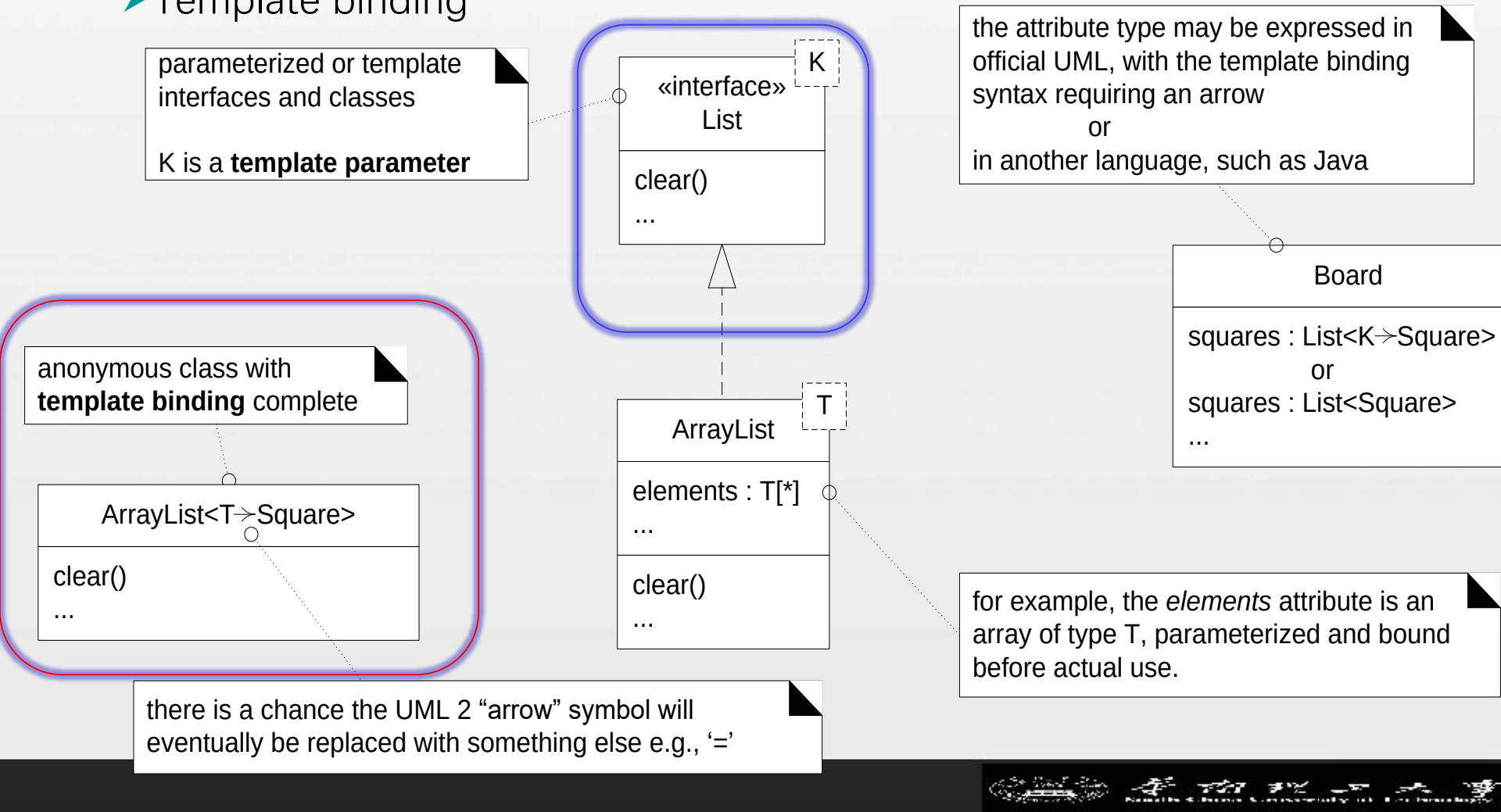
- **Singleton Classes**
  - only one instance of a class instantiated, never two



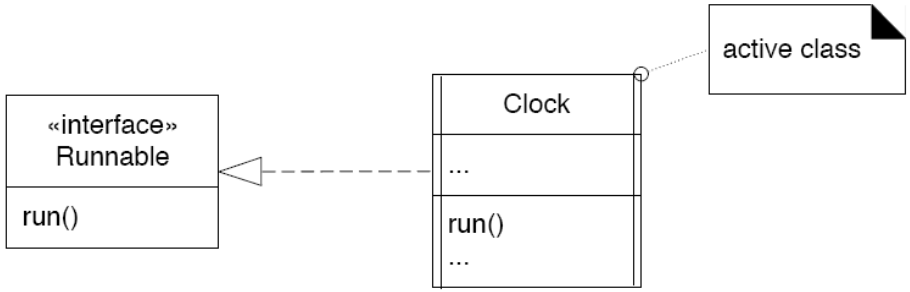
UML Class Diagram

Template Classes and Interfaces

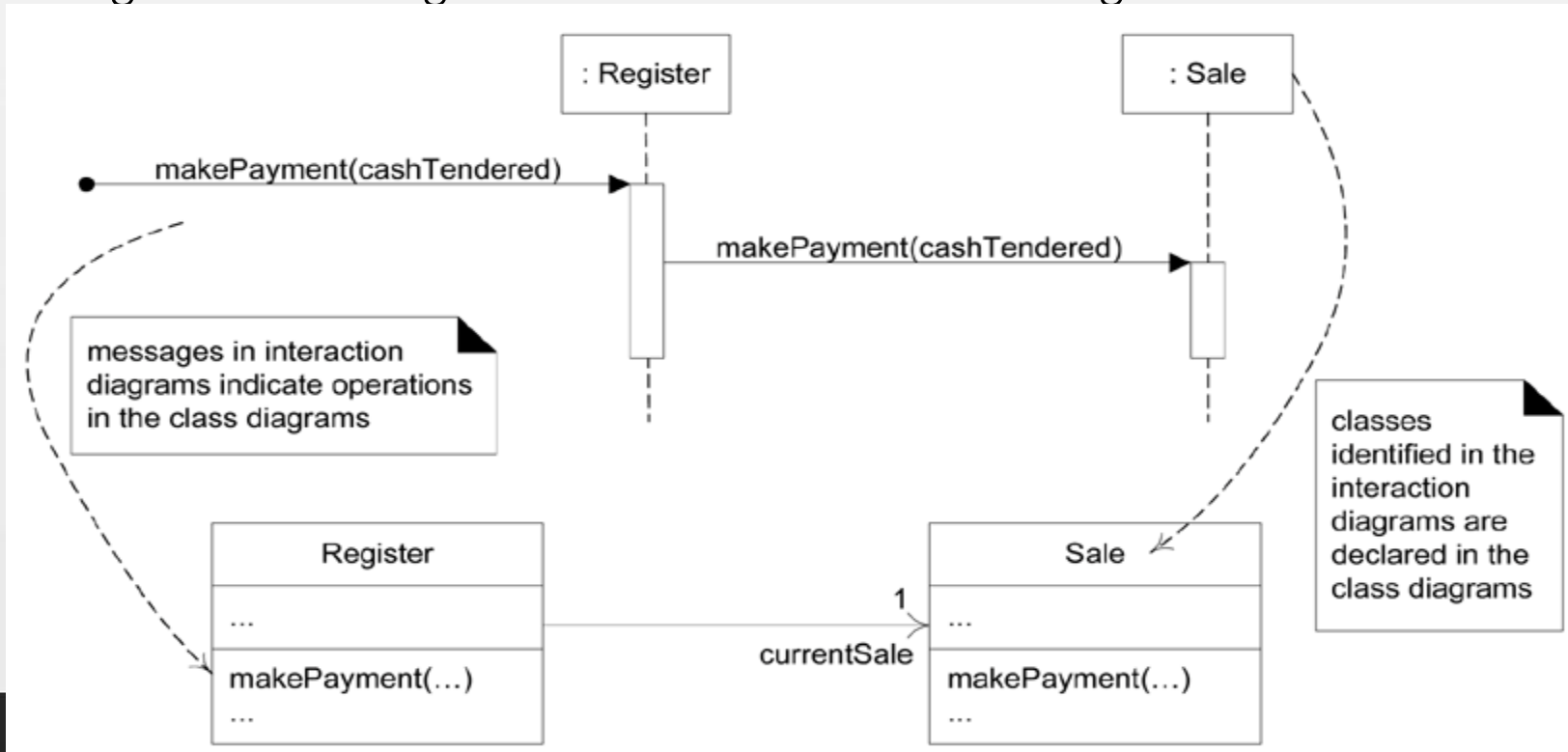
- templatized types
- Template binding



UML Class Diagram



- **Active** Class
  - **active object** runs on and controls its own thread of exec
  - be shown with double vertical lines on the left and right sides of the class box.
- **Relationship Between Interaction and Class Diagrams?**
  - class diagrams can be generated from interaction diagrams

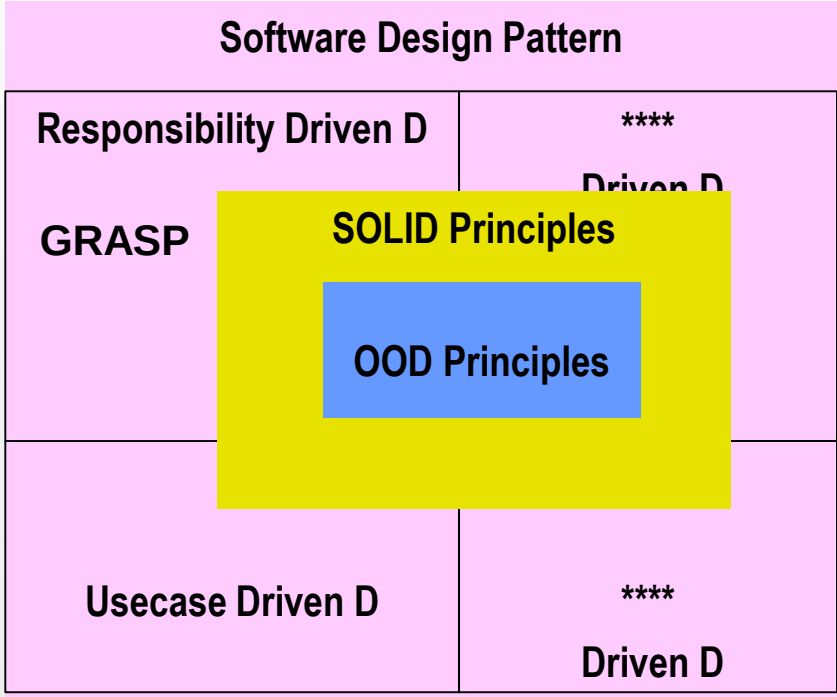


GRASP: Designing Objects with Responsibilities

- **OOD**
  - 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
  - class diagrams can be generated from interaction diagrams
  - **how objects should interact?**
- What we have and what we should do?
  - What wo have
    - use case text ,Supplementary Specification, Glossary,
    - Domain Model， system sequence diagrams , operation contracts
  - what we should do?
    - 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
    - ….

GRASP: Designing Objects with Responsibilities

- But, how?
  - OOD Principles?
  - SOLID Principle
  - Software Design
    - RDD--**Responsibility Driven** Design
      - GRASP
    - **DDD—Domain Driven Design**
    - **UDD- Use case Drive Design**
    - ***** DD-- ** Driven Design**



Although the last edition was released in 2004, according to me, the book is still up-to-date and explains perfectly how to design systems using object-oriented languages. It's hard to find the better book about this subject, believe me. It is not book about *UML*, but you can learn *UML* from it because it is good explained too. Must-have for each developer, period

GRASP: Designing Objects with Responsibilities

- **Responsibilities and Responsibility-Driven Design**
  - emphasizes modeling of objects' roles, responsibilities, and collaborations
- **Responsibilities**
  - **a contract or obligation of a classifier**
  - **two types: doing and knowing**
  - Doing
    - 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
    - knowing about private encapsulated data
    - knowing about related objects
    - knowing about things it can derive or calculate
  - **the granularity of the responsibility**
  - **Responsibility and method**

GRASP: Designing Objects with Responsibilities

- Responsibility can be
  - accomplished by a single object.
  - or a group of object collaboratively accomplish a responsibility.

- **Responsibility-Driven Design**
  - an application = a set of interacting objects
  - an object = an implementation of one or more roles
  - a role = a set of related responsibilities
  - a responsibility = an obligation to perform a task or know information
  - a collaboration = an interaction of objects or roles (or both)
  - a contract = an agreement outlining the terms of a collaboration

- **RDD is a Metaphor**
  - **Think of software objects as similar to people with responsibilities**

GRASP: Designing Objects with Responsibilities

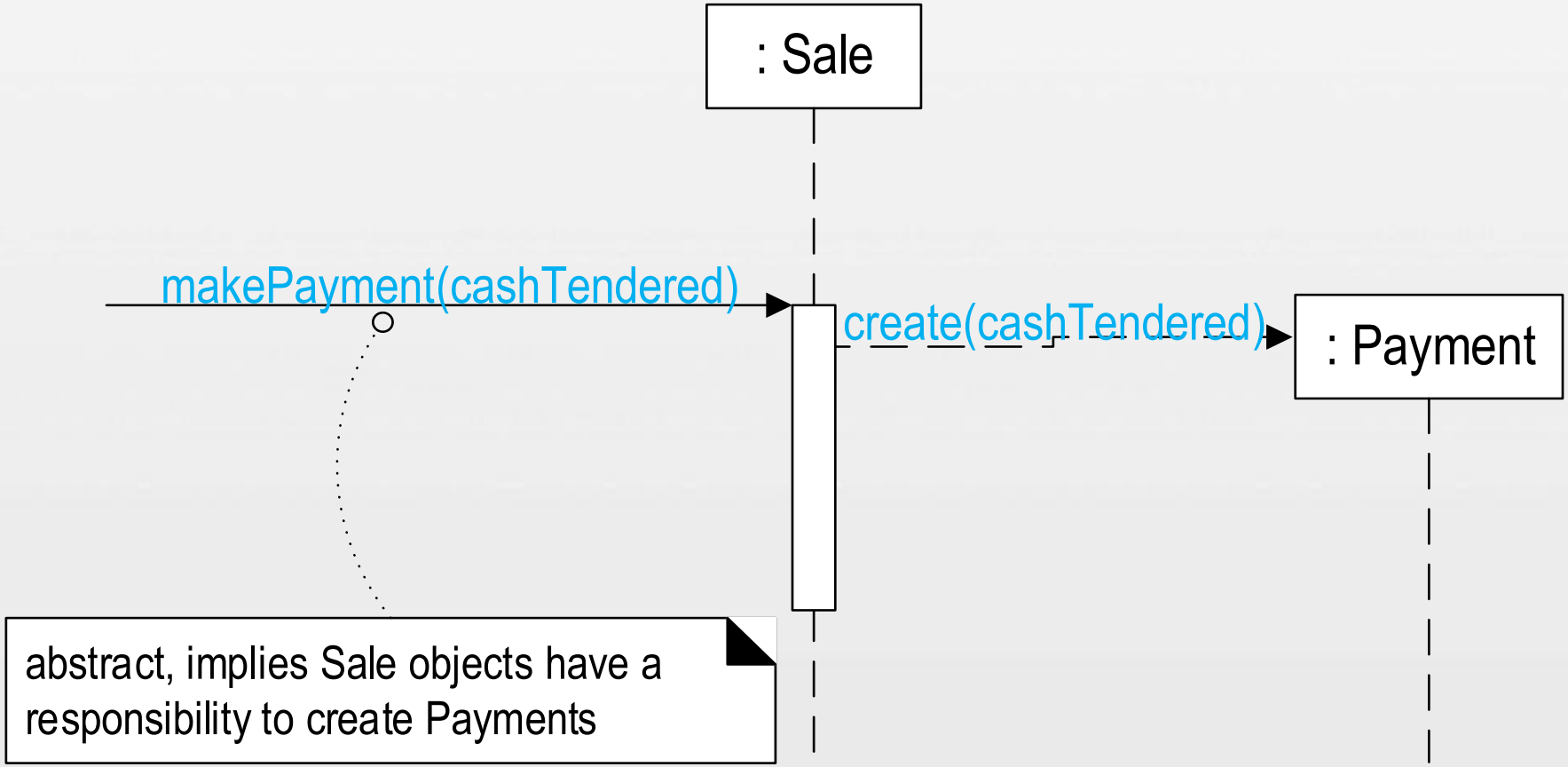
- **GRASP**
  - Acronym for **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns or Principles (GRASP)
  - Name chosen to suggest the importance of grasping fundamental principles to successfully design object-oriented software
  - Describe fundamental principles of object design and responsibility
  -
  - General principles, may be overruled by others

UNDERSTANDING HOW TO APPLY GRASP FOR OBJECT DESIGN IS A KEY GOAL OF THE BOOK.

GRASP: Designing Objects with Responsibilities

Responsibilities, GRASP, and UML Diagrams

- Within the UML, drawing interaction diagrams becomes the occasion for considering these responsibilities (realized as methods)



GRASP: Designing Objects with Responsibilities

Pattern?

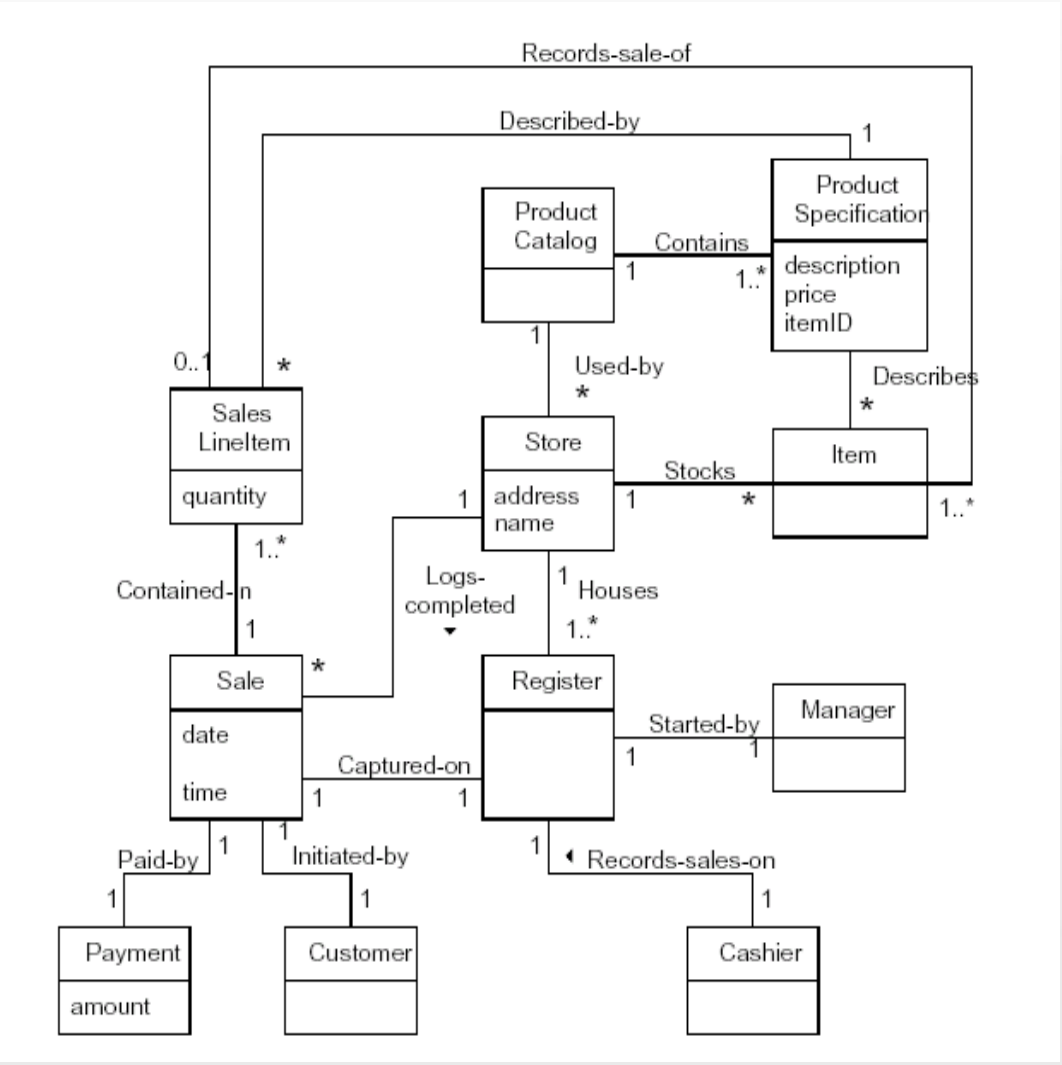
- pattern is a named description of a problem and solution that can be applied to new contexts

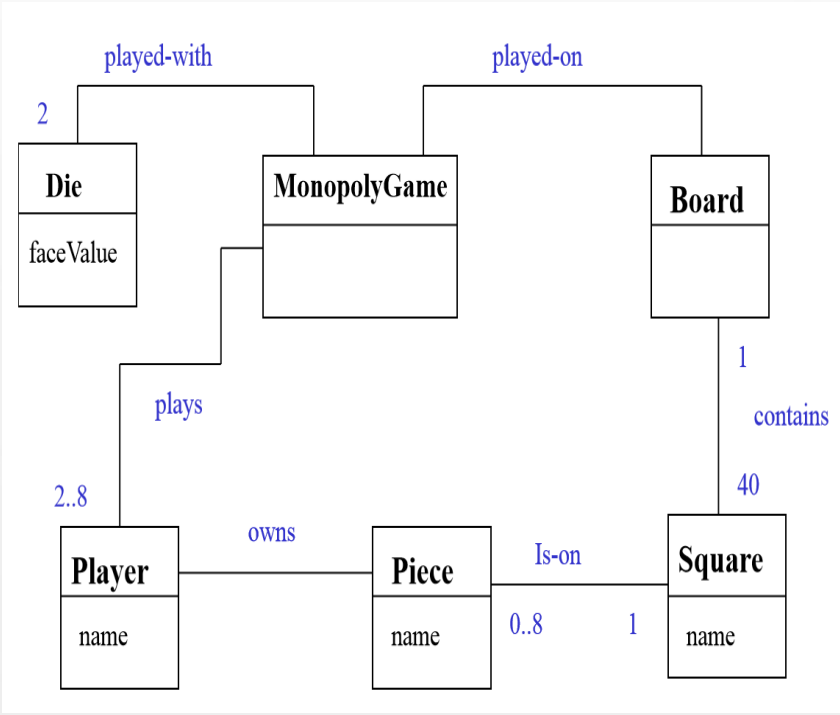
- What we have learned?
  - Architecture Pattern (Style)
  - GRASP
  - Design Pattern
  - **Collections of Principles, Patterns**

- How to describe pattern
  - Name
  - Problem
  - Solution

GRASP: Designing Objects with Responsibilities

- **Used Domain model in this chapter**

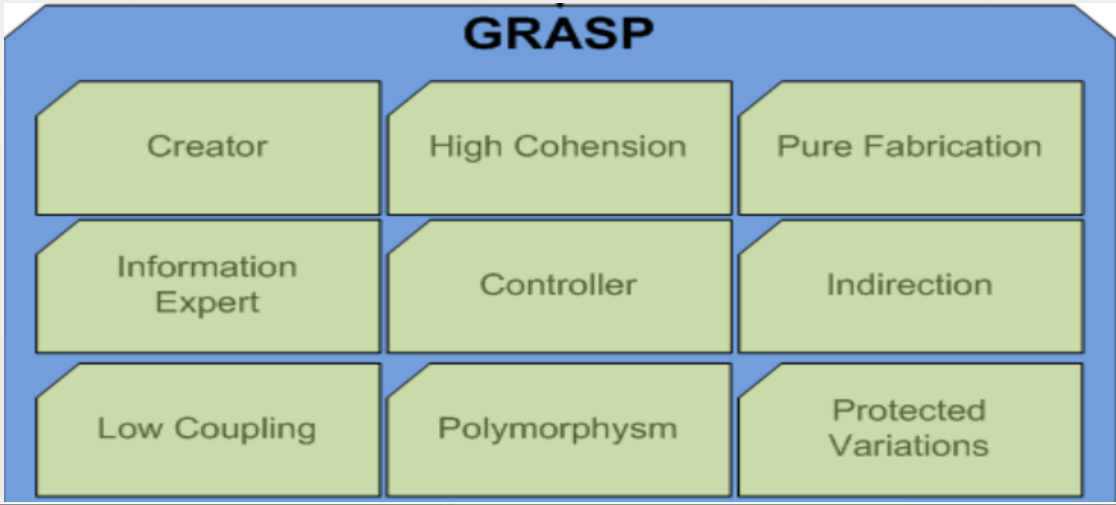




GRASP: Designing Objects with Responsibilities

GRASP

- Low coupling: How to support low dependency, low change impact, and increased reuse?
- High cohesion: How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?
- Creator: Who creates the Square object?
- Information expert : What is a general principle of assigning responsibilities to objects?
- Controller: What first object beyond the UI layer receives and coordinates ("controls") a system operation?
- Polymorphism
- Indirection
- Pure fabrication
- Protected variations



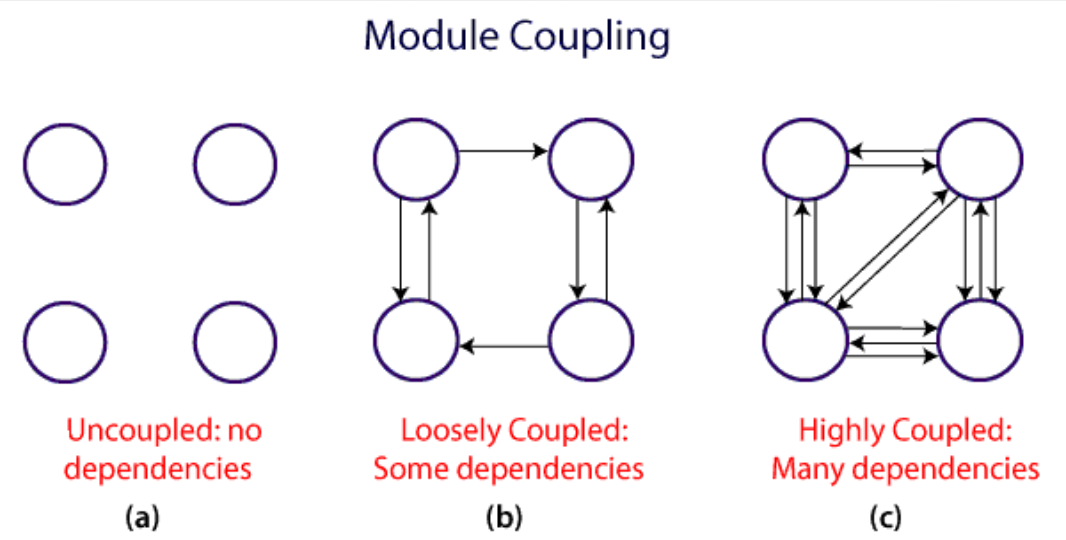
GRASP: Designing Objects with Responsibilities

Low Coupling

- One of the major GRASP principles is Low Coupling.
- Problem: **How to increase reuse and decrease the impact of change.**
- Solution: Assign responsibilities to minimize coupling

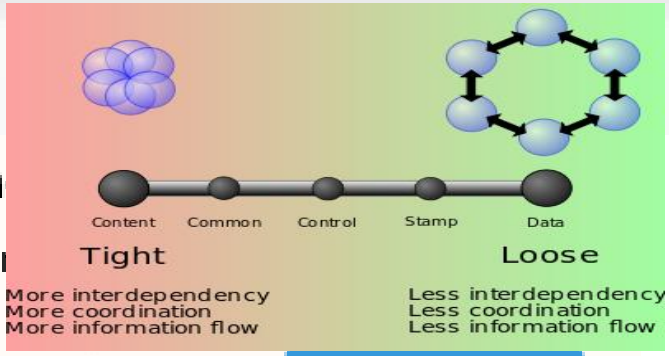
Coupling

- Coupling is a measure of how strongly one object is connected to, has knowledge of, or depends upon other objects. An object A that calls on the operations of object B has coupling to B's services. When object B changes, object A may be affected



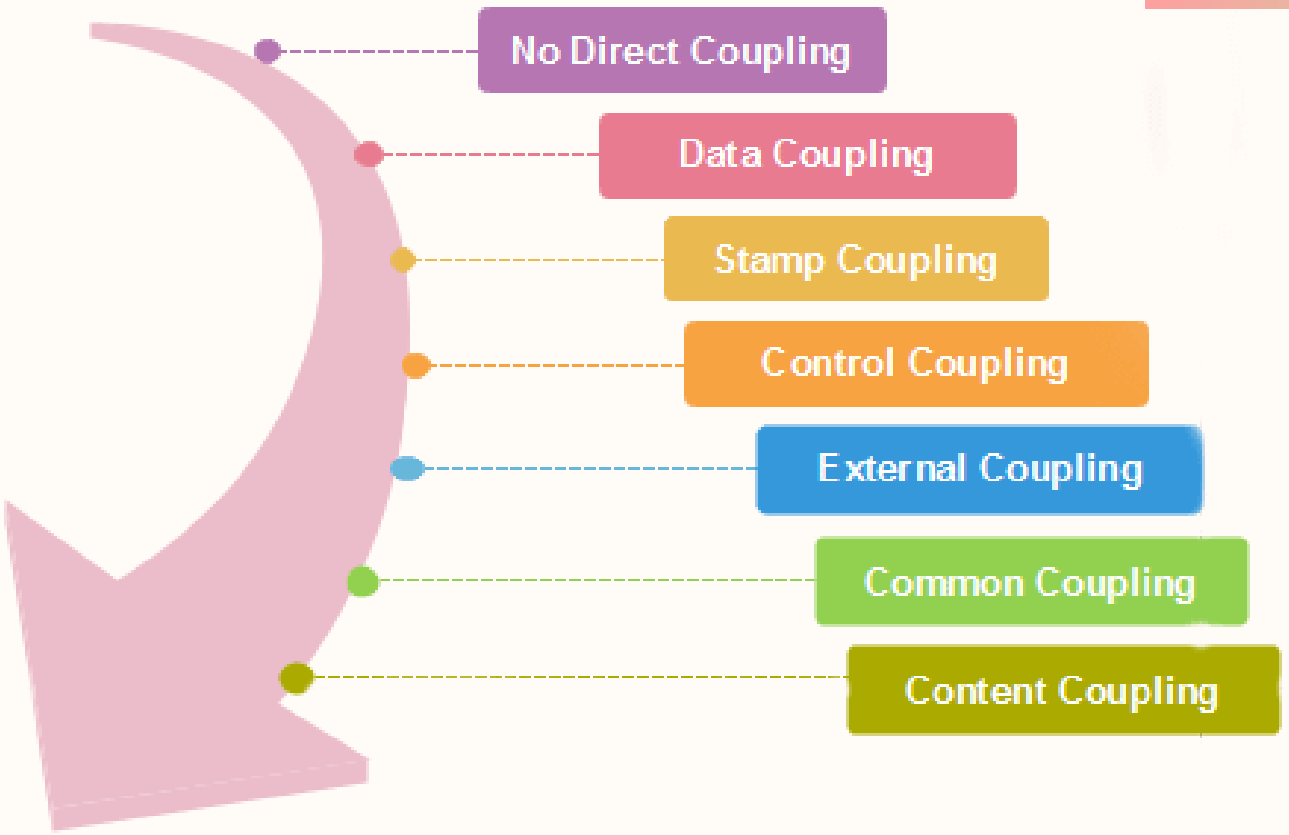
GRASP: Designing Objects with Responsibilities

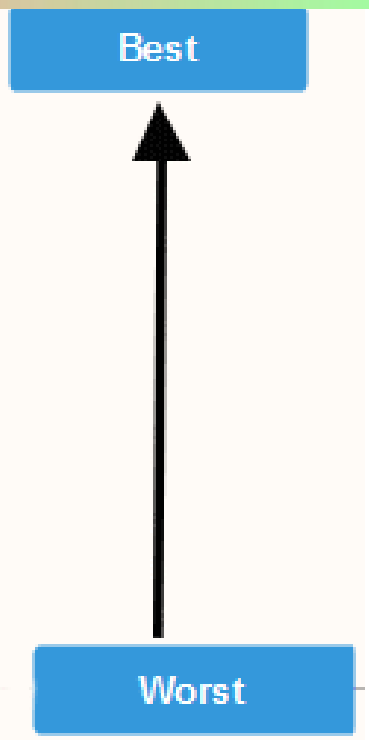
- Low Coupling



Types of Modules Couplin

There are various types of module Coupli

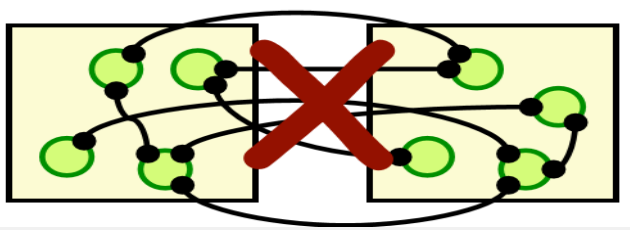




GRASP: Designing Objects with Responsibilities

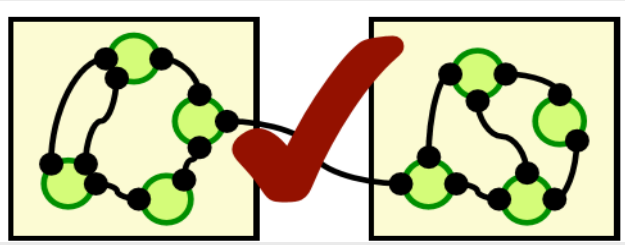
Low Coupling

- An element with low (or weak) coupling is not dependent on too many other elements (classes, subsystems, …)
  - "too many" is context-dependent



- A class with high (or strong) coupling relies on many other classes. Disadvantage:
  - Changes in related classes force local changes.
  - Such classes are harder to understand in isolation.
  - They are harder to reuse because its use requires the additional presence of the classes on which it is dependent.

- **Benefits of making classes independent of other classes**
  - changes are localized
  - easier to understand code
  - easier to reuse code

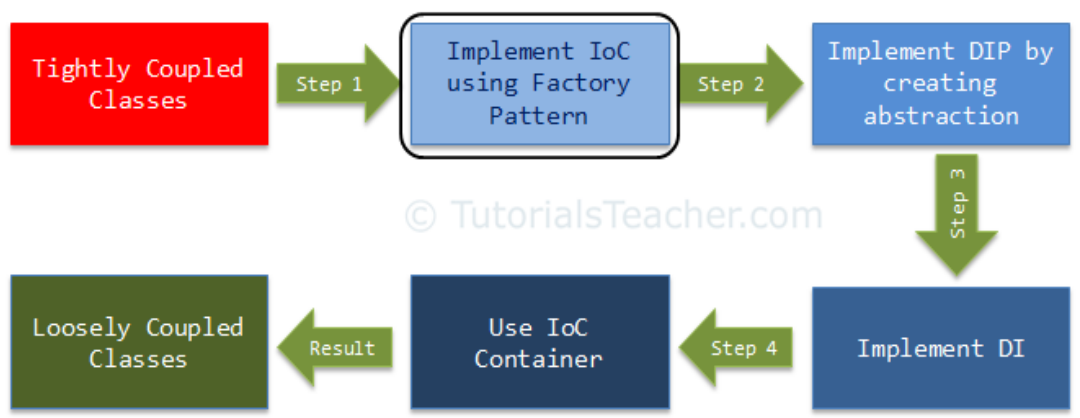


GRASP: Designing Objects with Responsibilities

Low Coupling

- Loose coupling is an architectural principle and design goal

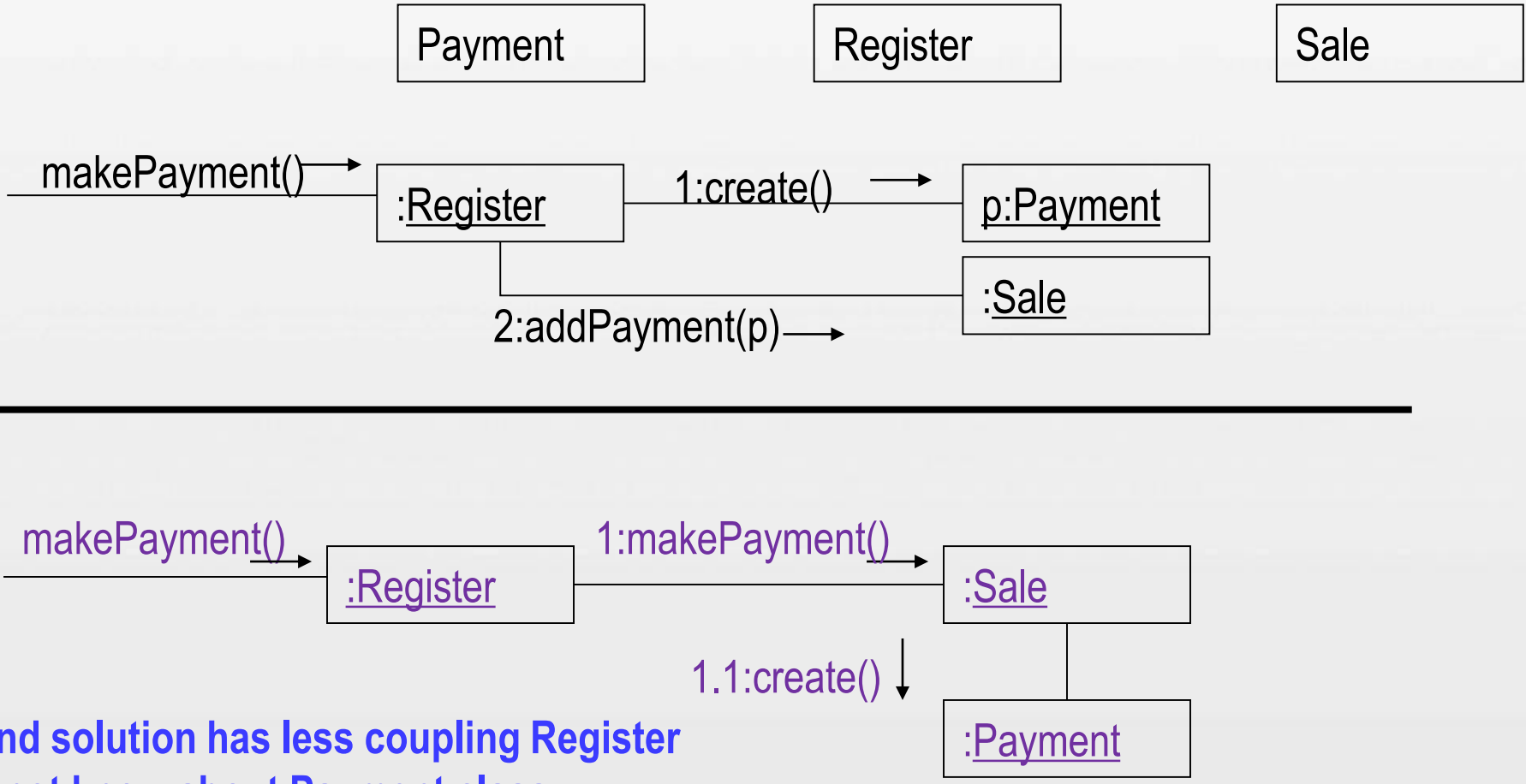
- **Strategies?**
  - Indirection: Don't access the other module directly but have another module do that. Such as Inversion of control(IOC)
  - Dependency Inversion:
  - Use lower form of coupling
  - Merge modules: when there is only one module, then there is no communication and thus no coupling
  - Hide information: Information which is hidden cannot be depended upon.
  - Encapsulate Change



GRASP: Designing Objects with Responsibilities

Low Coupling, example

- Create a Payment and associate it with the Sale. Which solution is better?



Second solution has less coupling Register does not know about Payment class

GRASP: Designing Objects with Responsibilities

Low Coupling

WHEN ARE TWO CLASSES COUPLED?

- Common forms of coupling from **TypeX** to **TypeY**:
  - TypeX has an attribute that refers to a TypeY instance.
  - A TypeX object calls on services of a TypeY object.
  - TypeX has a method that references an instance of TypeY (parameter, local variable, return type).
  - TypeX is a direct or indirect subclass of TypeY.
  - TypeY is an interface and TypeX implements that interface.

GRASP: Designing Objects with Responsibilities

Low Coupling: 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 a designer applies while evaluating all design decisions.

- Low Coupling supports the design of classes that are more independent
  - reduces the impact of change.

- Can't be considered in isolation from other patterns such as Expert and High Cohesion, needs to be included as one of several design principles that influence a choice in assigning a responsibility.

GRASP: Designing Objects with Responsibilities

Low Coupling: Discussion (continued)

- Sub-classing produces a particularly problematic form of high coupling
  - Dependence on implementation details of superclass
  - Prefer composition over inheritance
- cannot obtain an absolute measure of when coupling is too high
- Extremely low coupling may lead to a poor design
  - Few incohesive, bloated classes do all the work; all other classes are just data containers
- Contraindications: High coupling to very stable elements is usually not problematic

GRASP: Designing Objects with Responsibilities

Low Coupling: Pick Your Battles

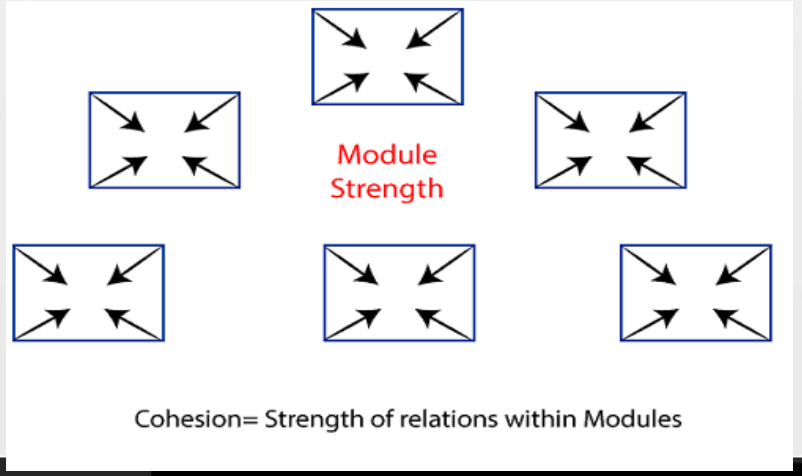
- It is not high coupling per se that is the problem; it is high coupling to elements that are unstable in some dimension, such as their interface, implementation, or mere presence

- Focus on the points of realistic high instability or evolution
  - 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

- Find the change and Encapsulate Change

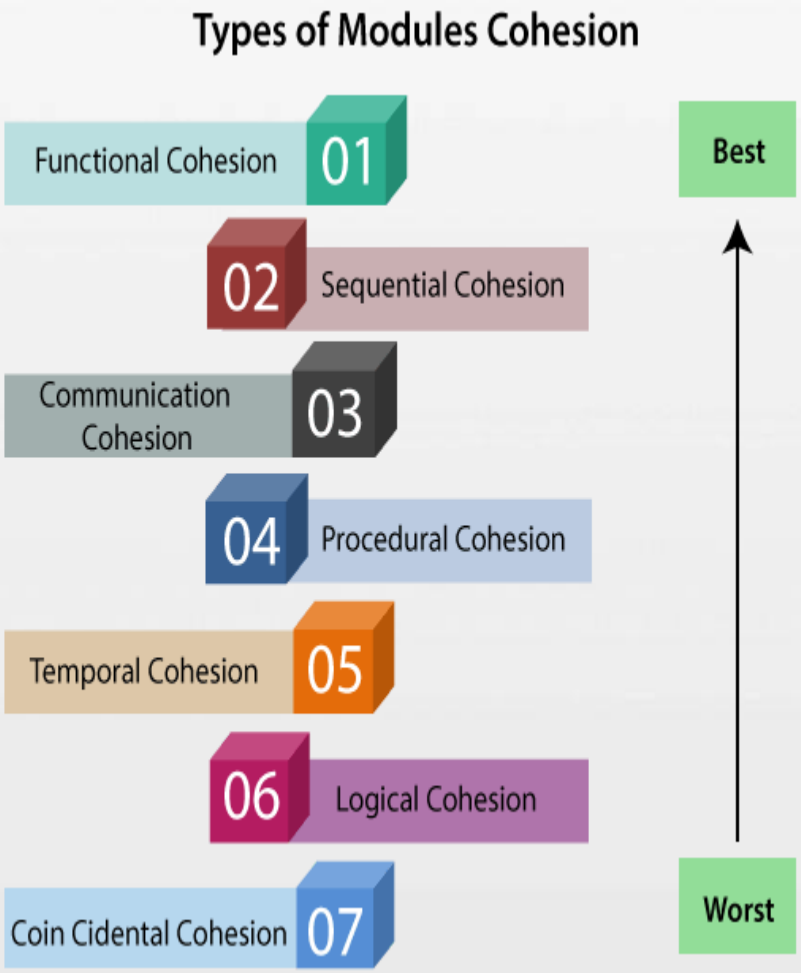
GRASP: Designing Objects with Responsibilities

- **High Cohesion**
  - One of the major GRASP principles is High Cohesion
  - Problem: How to keep objects focused, understandable, and manageable, and as a side effect, support Low Coupling?
  - Solution: Assign responsibilities so that cohesion remains high
- Cohesion
  - 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



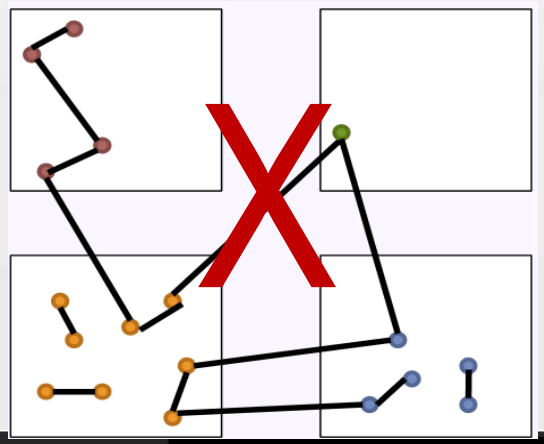
GRASP: Designing Objects with Responsibilities

■ High Cohesion



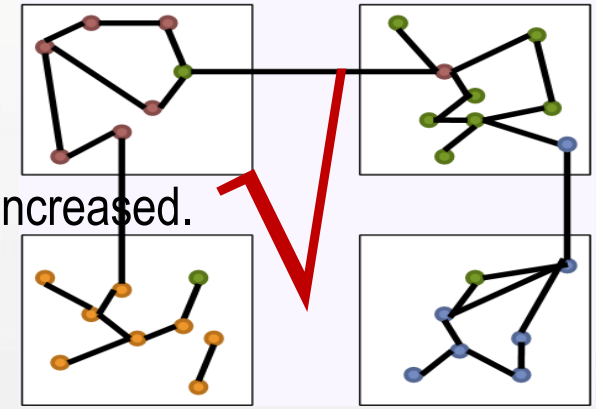
A class with low cohesion does many unrelated things or does too much work. Such classes are undesirable; they suffer from the following problems:

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



GRASP: Designing Objects with Responsibilities

High Cohesion

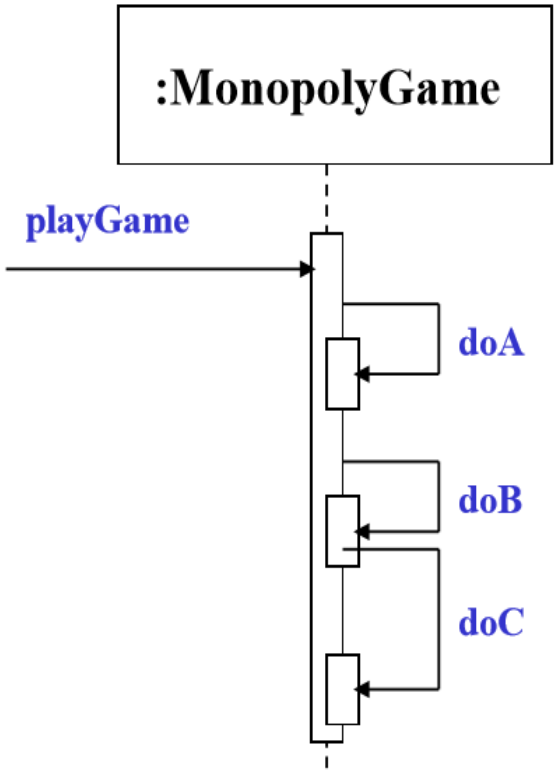


- benefits
  - Clarity and ease of comprehension of the design is increased.
  - Maintenance and enhancements are simplified.
  - Low coupling is often supported.
  - Reuse of fine-grained, highly related functionality is increased because a cohesive class can be used for a very specific purpose

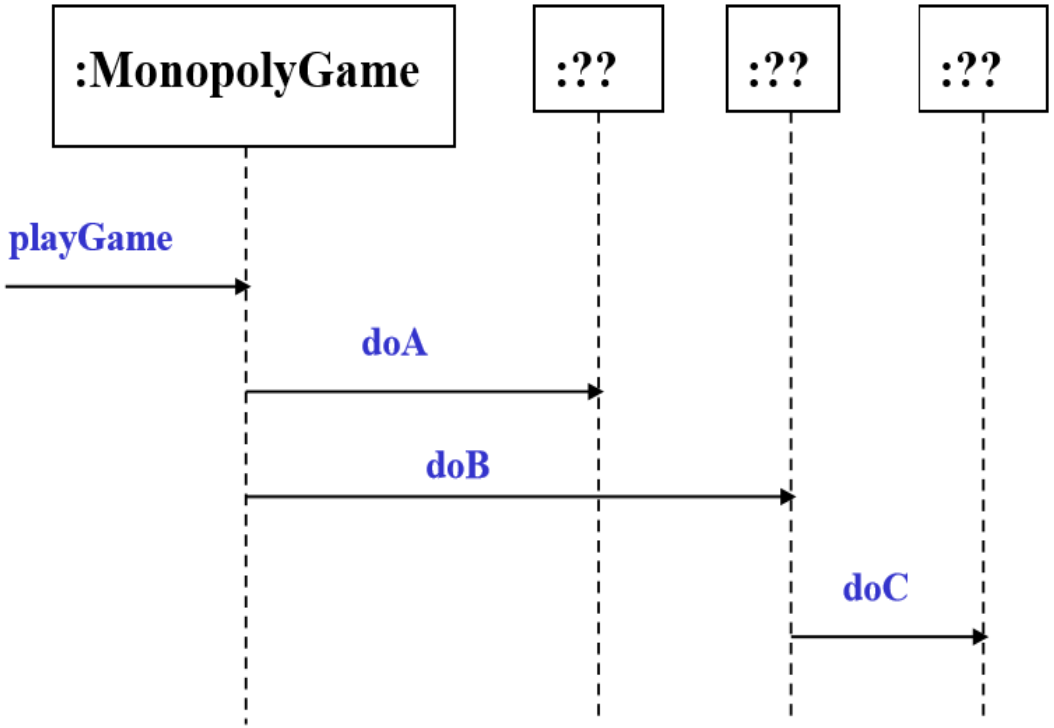
- Solutions:
  - **Assign a responsibility so that cohesion remains high. Use this to evaluate alternatives**

GRASP: Designing Objects with Responsibilities

- High Cohesion -- example1



Poor (low) Cohesion in the MonopolyGame object

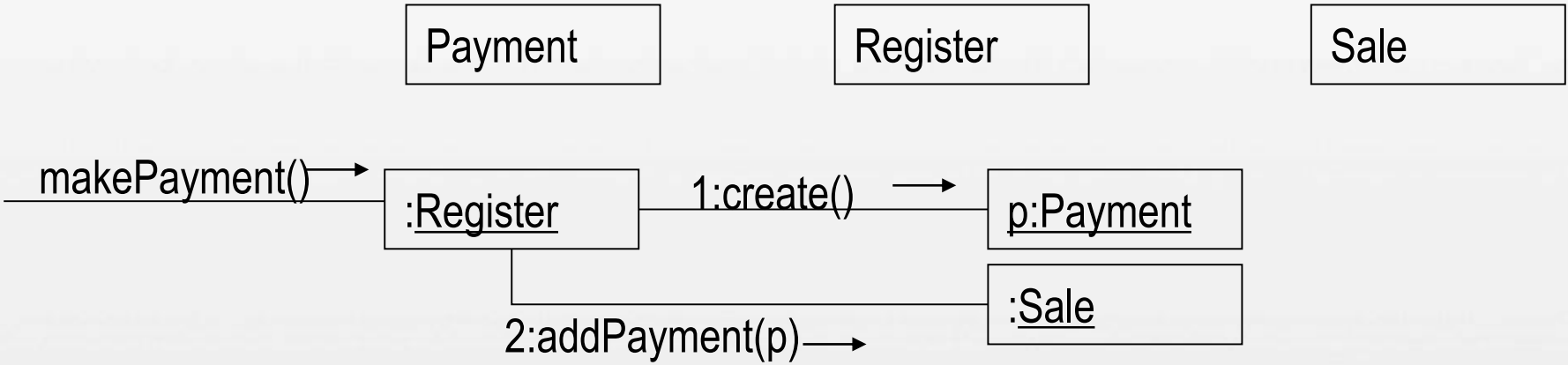


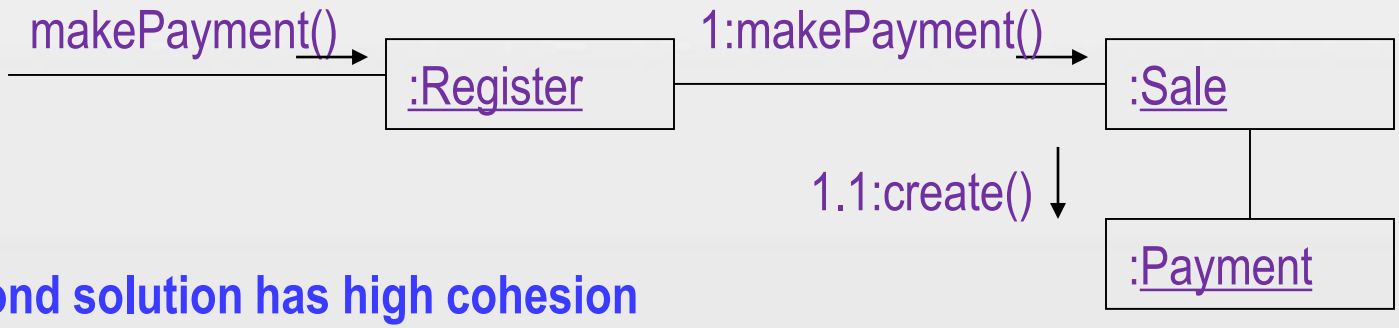
Better Design

GRASP: Designing Objects with Responsibilities

High Cohesion -- example2

- Create a Payment and associate it with the Sale. Which solution is better





Second solution has high cohesion

GRASP: Designing Objects with Responsibilities

High Cohesion: Discussion

- Scenarios:
  - Very Low Cohesion: A Class is solely responsible for many things in very different functional areas
  - Low Cohesion: A class has sole responsibility for a complex task in one functional area.
  - High Cohesion. A class has moderate responsibilities in one functional area and collaborates with other classes to fulfil tasks.
  - 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

- 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.**

GRASP: Designing Objects with Responsibilities

Creator

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

- **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 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.

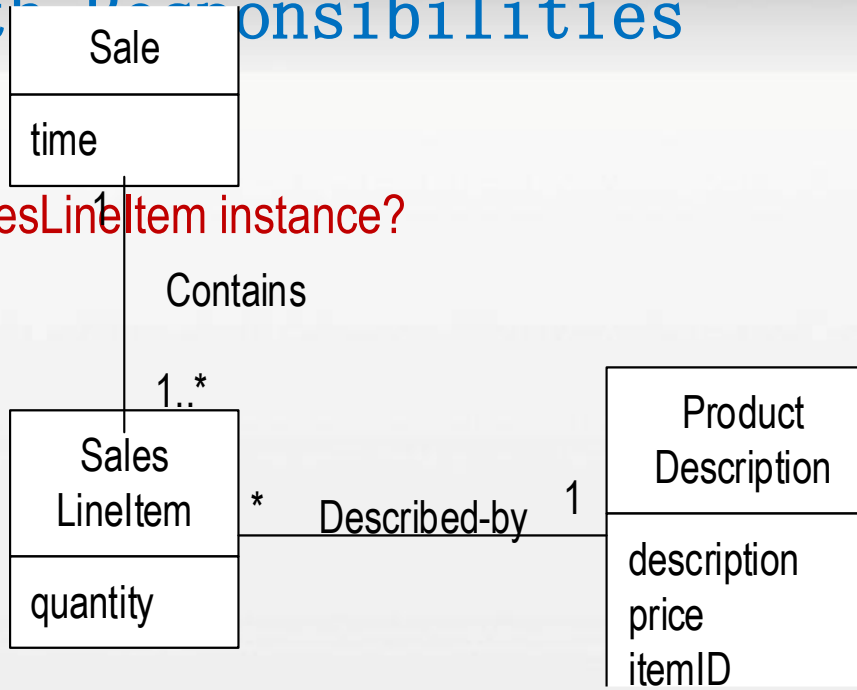
B is a creator of A. If more than one option applies, usually prefer a class B which aggregates or contains class A.

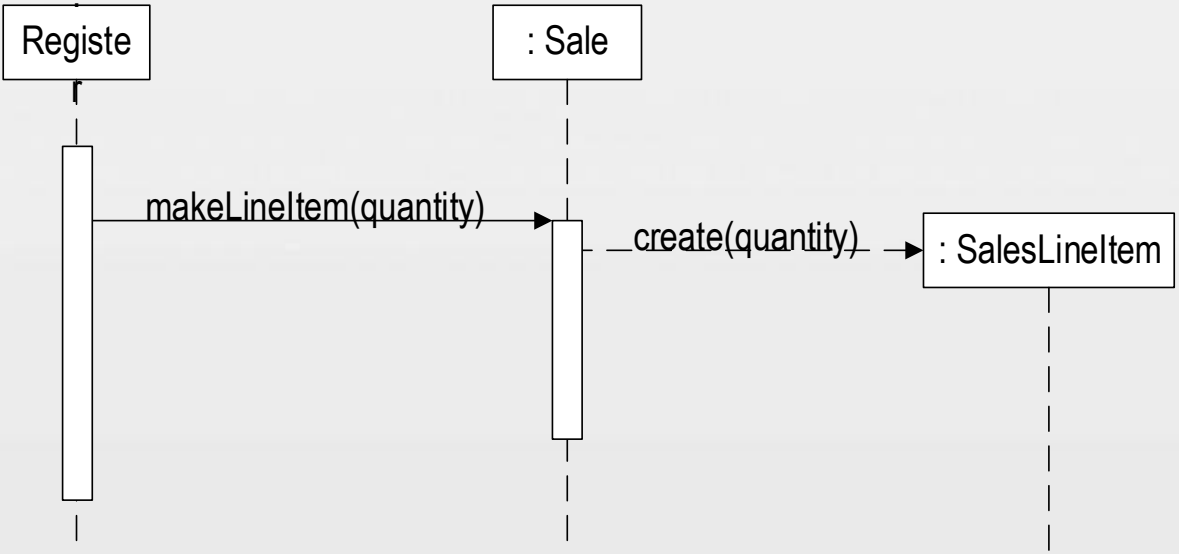
GRASP: Designing Objects with Responsibilities

■ Creator -- Example

➢who should be responsible for creating a SalesLineItem instance?

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





GRASP: Designing Objects with Responsibilities

Creator

- creation requires significant complexity, it is advisable to delegate creation to a helper class called a Concrete Factory or an Abstract Factory rather than use the class suggested by Creator

- **Benefits:**
  - **Low coupling is supported**

- **Related Patterns or Principles**
  - Low Coupling
  - Concrete Factory and Abstract Factory
  - Whole-Part

GRASP: Designing Objects with Responsibilities

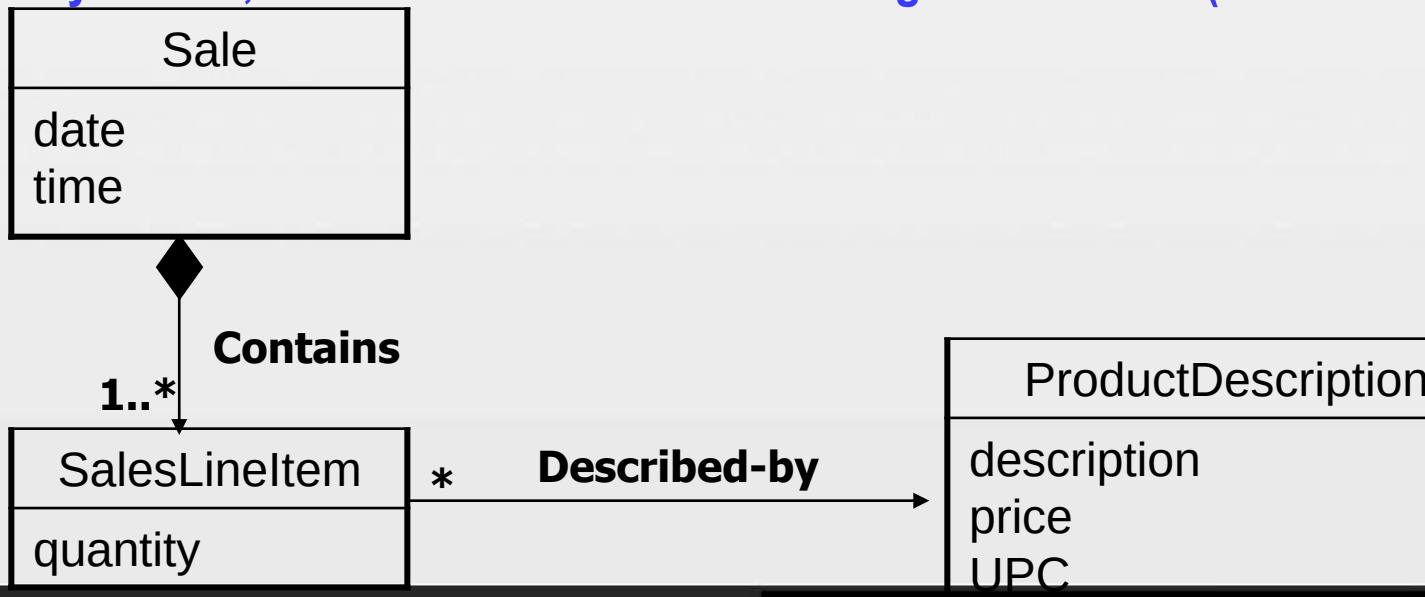
- **Information Expert**
  - Problem:
    - What is the basic principle by which to assign responsibilities/functionality to objects?
  - Solution:
    - **Assign a responsibility to a class that has the information needed to fulfil it**
    - **Information Expert is frequently used in the assignment of responsibilities; it is a basic guiding principle used continuously in object design**
  - Assign responsibility to expert

GRASP: Designing Objects with Responsibilities

- Information Expert -- example
  - Who should be responsible for knowing the grand total of a sale?

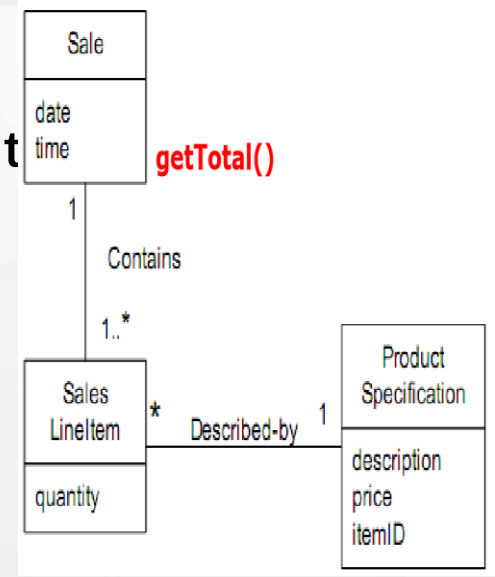
Start assigning responsibilities by clearly stating the responsibility.
Who should be responsible for knowing the grand total of a sale?

By Information Expert, we should look for that class of objects that has the information needed to determine the total.
Design Classes (Software Classes) instead of Conceptual Classes
If Design Classes do not yet exist, look in Domain Model for fitting abstractions (-> low representational gap)



GRASP: Designing Objects with Responsibilities

- **Information Expert -- example**
  - **Who should be responsible for knowing the grand t**
    - What information is needed to determine the grand total?
      - Line items and the sum of their subtotals •
    - Sale is the information expert for this responsibility.

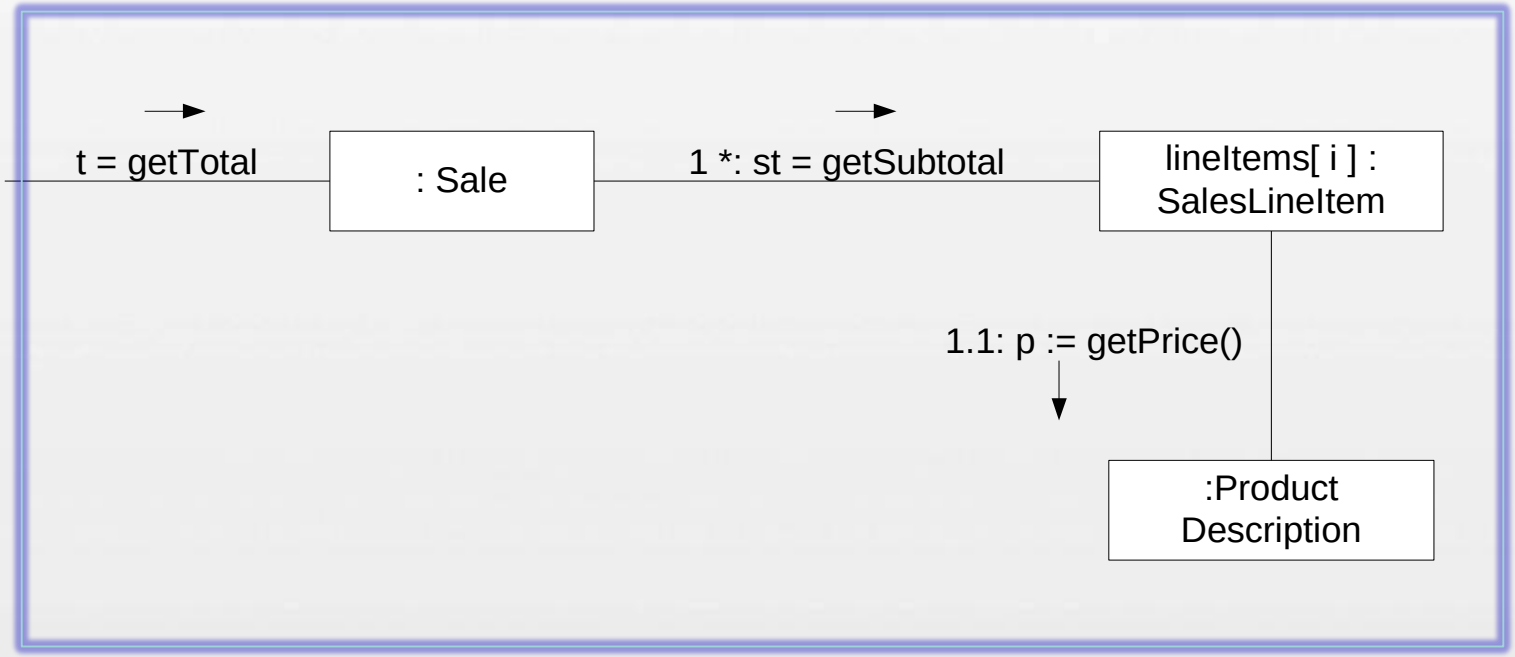


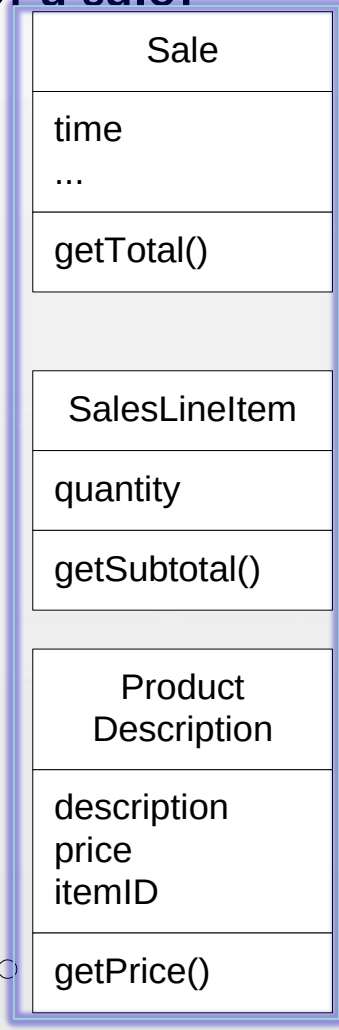
To fulfill the responsibility of knowing and answering the sale's total, three responsibilities were assigned to three design classes of objects

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

GRASP: Designing Objects with Responsibilities

- **Information Expert**
  - Who should be responsible for knowing the grand total of a sale?





New method

GRASP: Designing Objects with Responsibilities

- **Information Expert -- Do It Yourself**
  - Expert usually leads to designs where a software object does those operations that are normally done to the inanimate real-world thing it represents
    - a sale does not tell you its total; it is an inanimate thing
  - In OO design, all software objects are "alive" or "animated," and they can take on responsibilities and do things.
  - They do things related to the information they know.

GRASP: Designing Objects with Responsibilities

Information Expert -- Discussion

- **"partial" information experts?**
  - fulfillment of a responsibility often requires information that is spread across different classes of objects
  - many "partial" information experts will collaborate in the task

- **In some situations, a solution suggested by Expert is undesirable, usually because of problems in coupling and cohesion**
  - Example: Who is responsible for saving a sale in the database?
    - Adding this responsibility to Sale would distribute database logic over many classes
    - low cohesion, tight coupling
    - Conflict with separation of concerns

GRASP: Designing Objects with Responsibilities

- **Information Expert**
  - Benefits:
    - **Information encapsulation**
      - objects use their own information to fulfill tasks
      - supports low coupling
    - **Behavior is distributed across the classes that have the required information**
      - lightweight class definitions
      - High cohesion
  - Also Known As
    - "Place responsibilities with data," "That which knows, does," "Do It Myself," "Put Services with the Attributes They Work On."

GRASP: Designing Objects with Responsibilities

- **Information Expert**

```
class Foo {
public:
   void SetValue(int);
   int GetValue();
private:
   int value;
};
```

```
class Foo {
public:
   void SetValue(int);
   int GetValue();
   void Discount()
   {
      if (value > 100)
         value *= 0.8;
      else
         value *= 0.9
   }
private:
   int value;
};
```

```
Some other class ...

Foo *foo = new Foo();
...
int  v = foo->GetValue();
if (v >= 100)
   foo->SetValue(v * 0.8);
else
   foo->SetValue(v * 0.9);
...
        What's wrong?
```

Violate information expert pattern

```
Some other class ...

Foo *foo = new Foo();
...
foo->Discount();
...
        Better!!!
```

GRASP: Designing Objects with Responsibilities

Controller

Problem

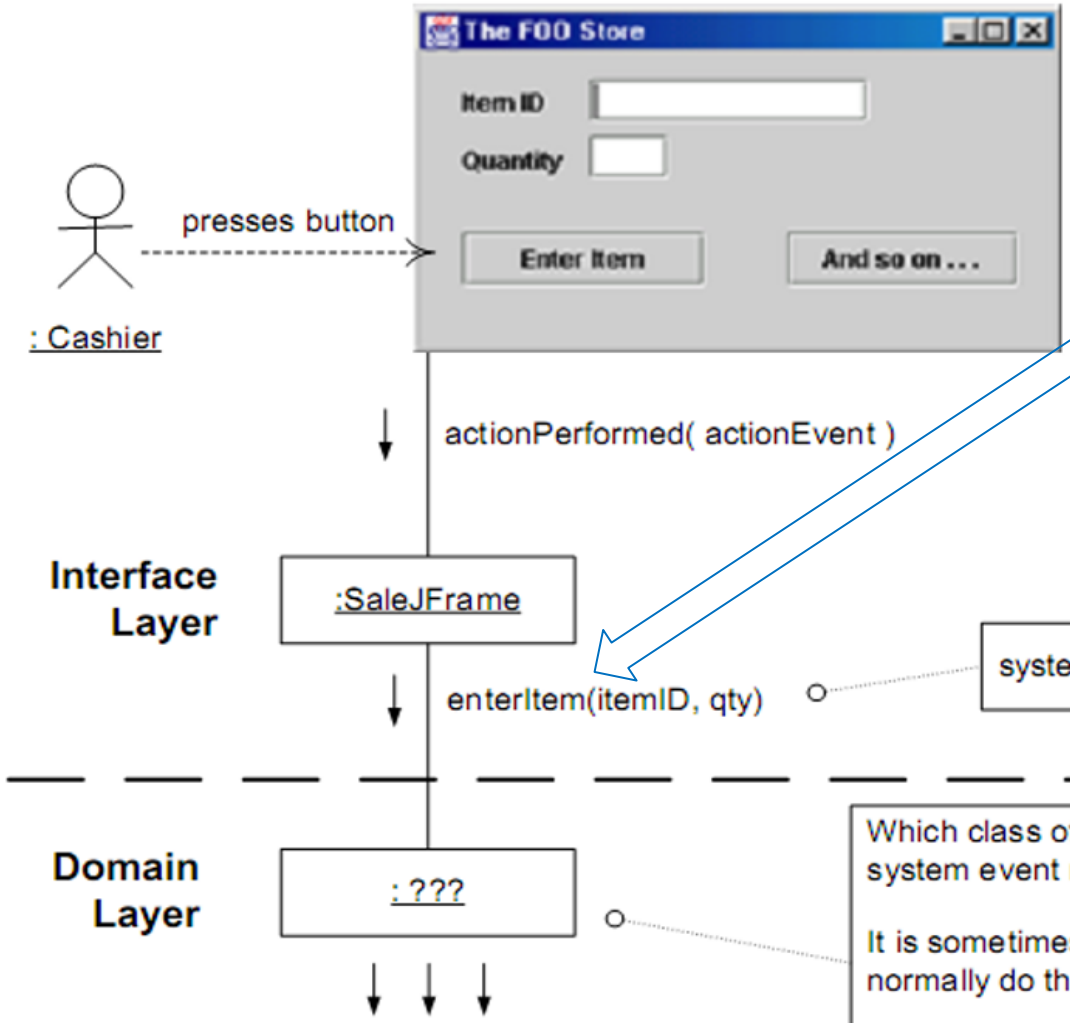
- **Who should be responsible for handling an input system event?**

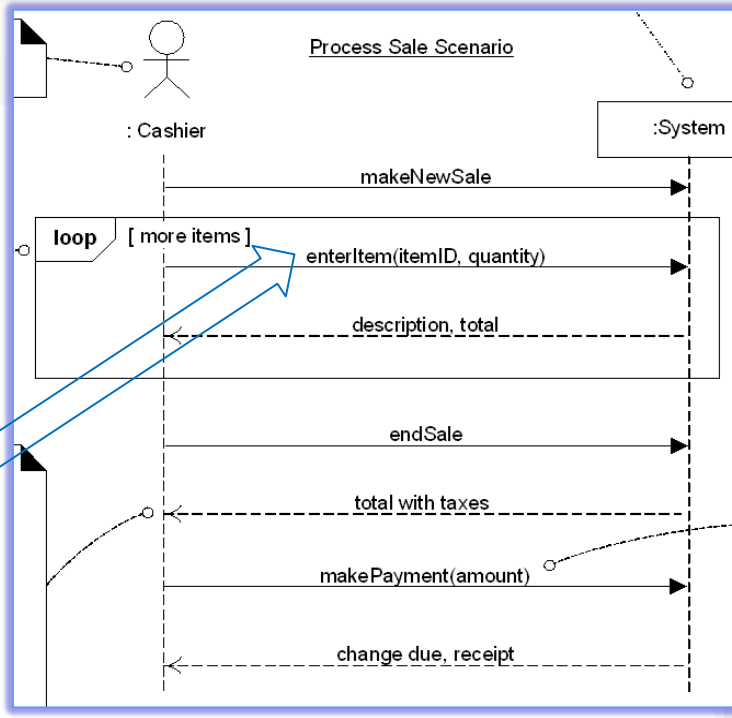
Solution:

- **A Controller is the first object beyond the UI later that is responsible for receiving and handling a system operation message**
- Assign the responsibility for receiving or handling a system event message to
  - a class representing the overall system, device, or subsystem (facade controller) , facade controller
  - or a use case scenario within which the system event occurs (use case controller)
    - » use the same controller class for all system events in the same use case scenario
    - » think in terms of sessions – instances of conversations with an actor

GRASP: Designing Objects with Responsibilities

- Controller -- example



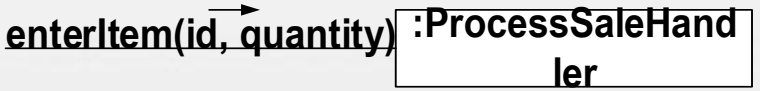


GRASP: Designing Objects with Responsibilities

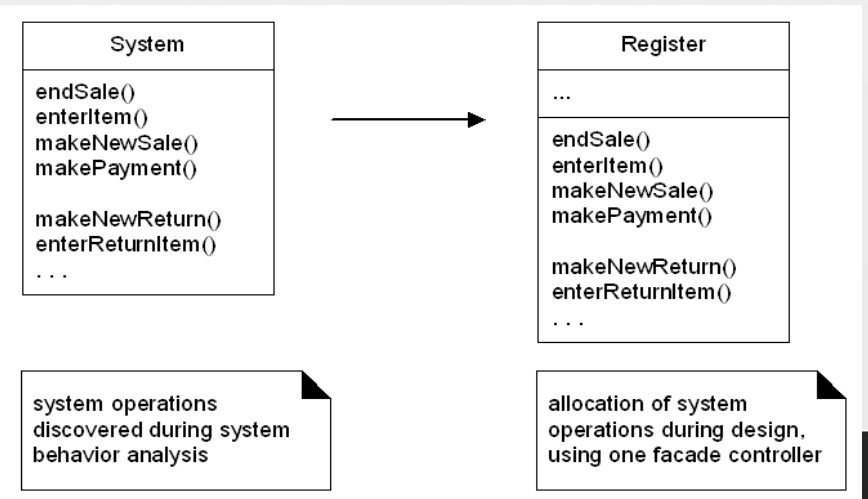
Controller

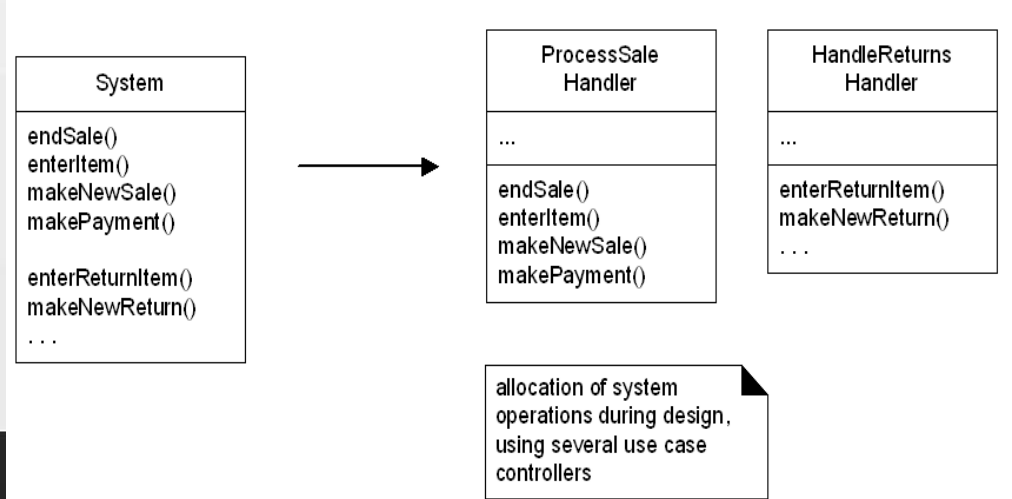
- Who should be the controller for system events such as enterItem and endSale? **By the Controller pattern, here are some choices:**
  - Register, POSSystem: represents the overall "system," device, or subsystem

  - ProcessSaleSession, ProcessSaleHandler: represents a receiver or handler of all system events of a use case scenario



- **the system operations identified during system behavior analysis are assigned to one or more controller classes**





GRASP: Designing Objects with Responsibilities

Controller --Discussion

- 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.

- Facade controllers are suitable when there are not "too many" system events

- A use case controller is an alternative to consider 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
  - **use different controller for each use case**

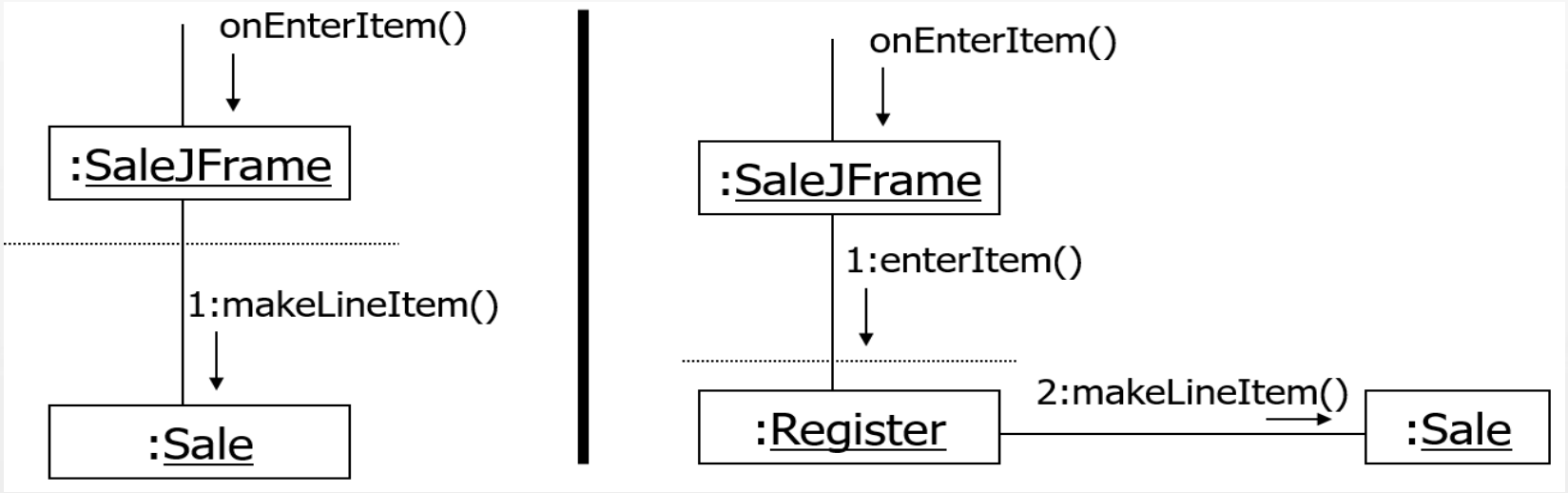
GRASP: Designing Objects with Responsibilities

- **Controller**
  - Benefits
    - **Increased potential for reuse, and pluggable interfaces**
      - No application logic in the GUI
    - **Dedicated place to place state that belongs to some use case**
      - E.g. operations must be performed in a specific order
  - Avoid bloated controllers!
    - **E.g. single controller for the whole system, low cohesion, lots of state in controller**
    - **Split into use case controllers, if applicable**

GRASP: Designing Objects with Responsibilities

Controller

- Interface layer does not handle system events

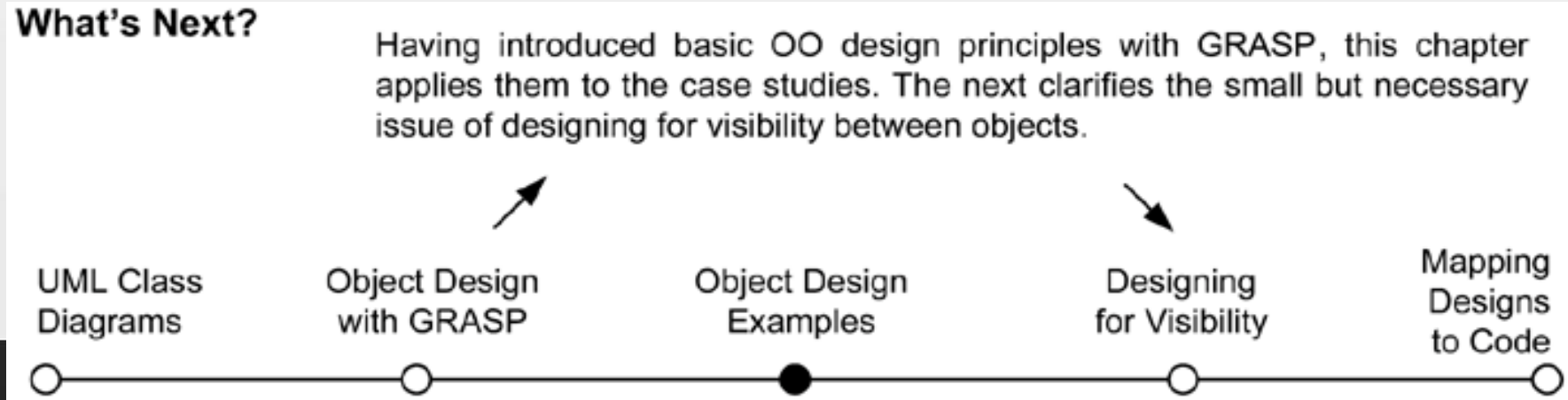


- Related Patterns
  - Command
  - Façade
  - Layer
  - Pure Fabrication

Object Design Examples with GRASP

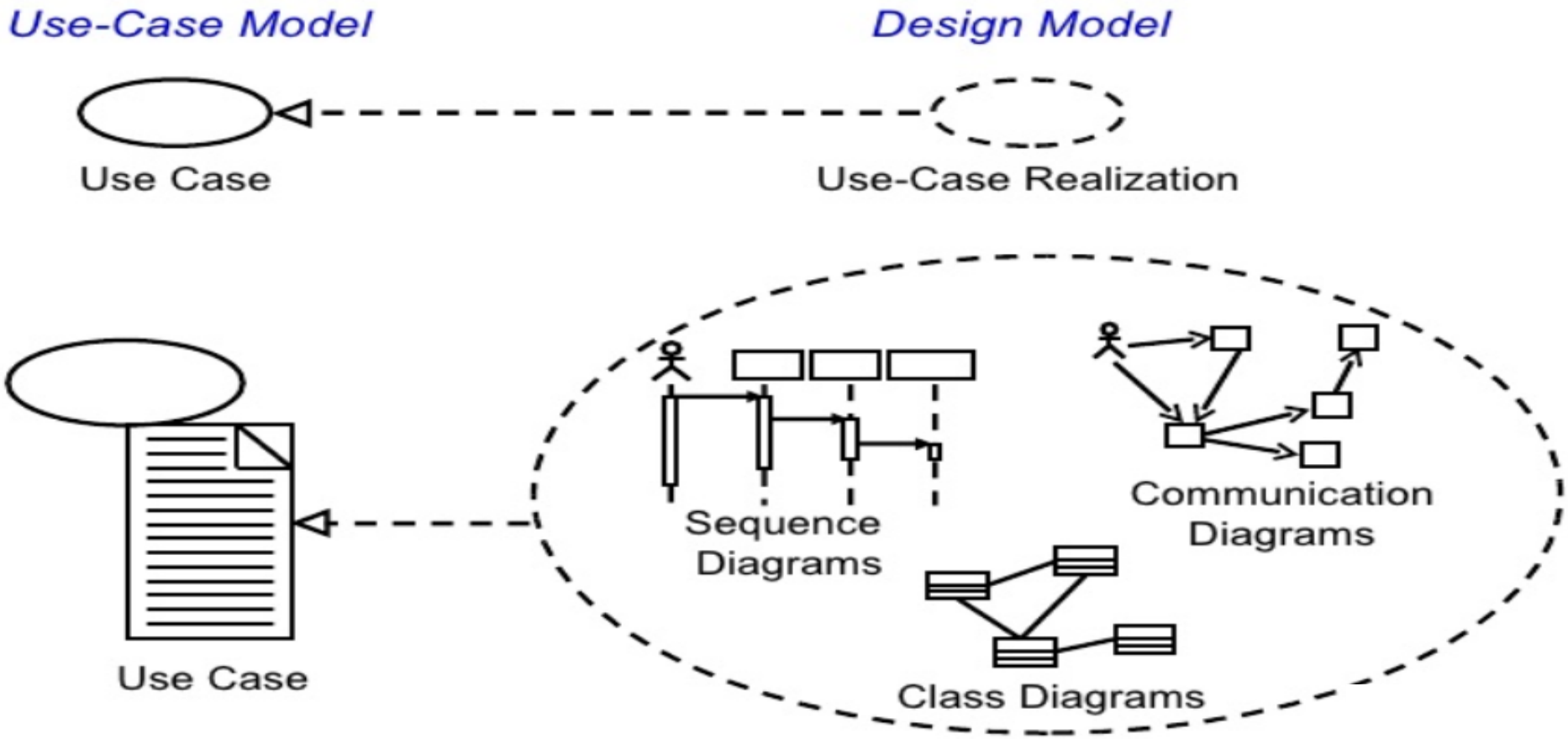
- Objective
  - Design use case realizations.
  - Apply GRASP to assign responsibilities to classes.
  - Apply UML to illustrate and think through the design of objects.

- Introduction
  - applies OO design principles and the UML to the case studies
  - The assignment of responsibilities and design of collaborations are very important and creative steps during design, both while diagraming and while coding



Object Design Examples with GRASP

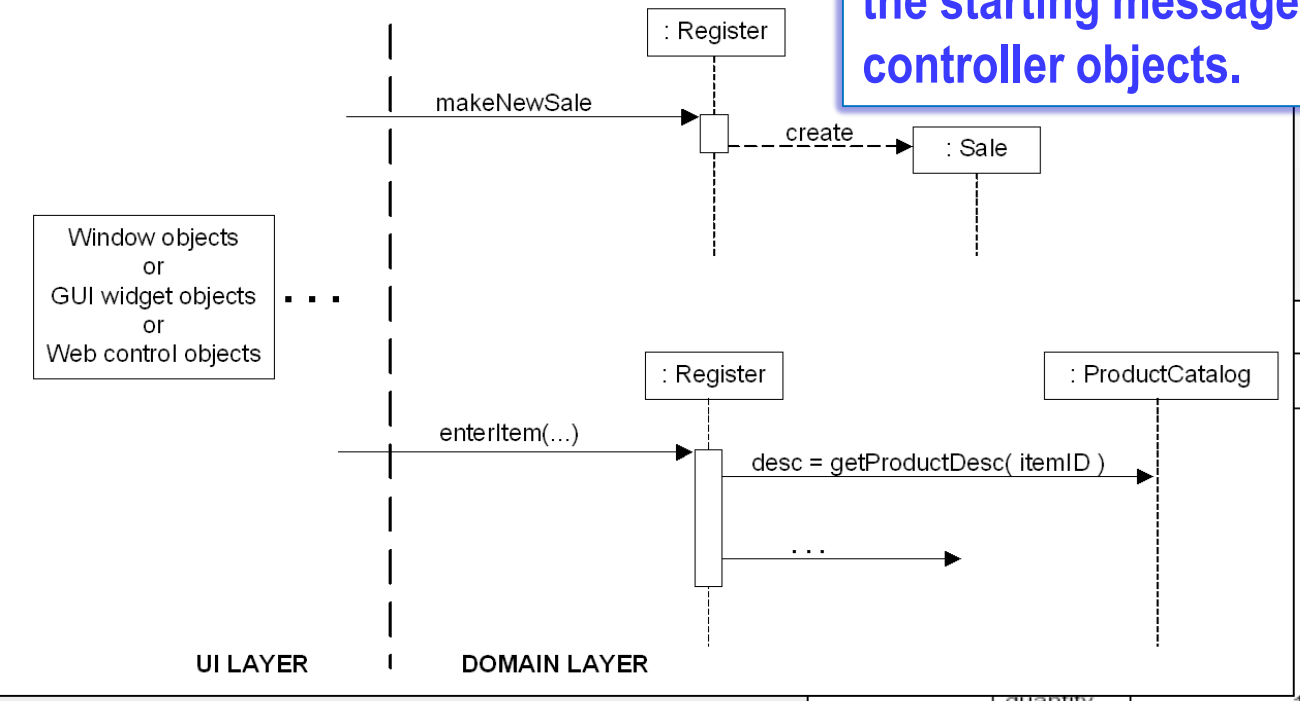
- **Use Case Realization?**
  - A use-case realization describes how a particular use case is realized within the Design Model, in terms of collaborating objects

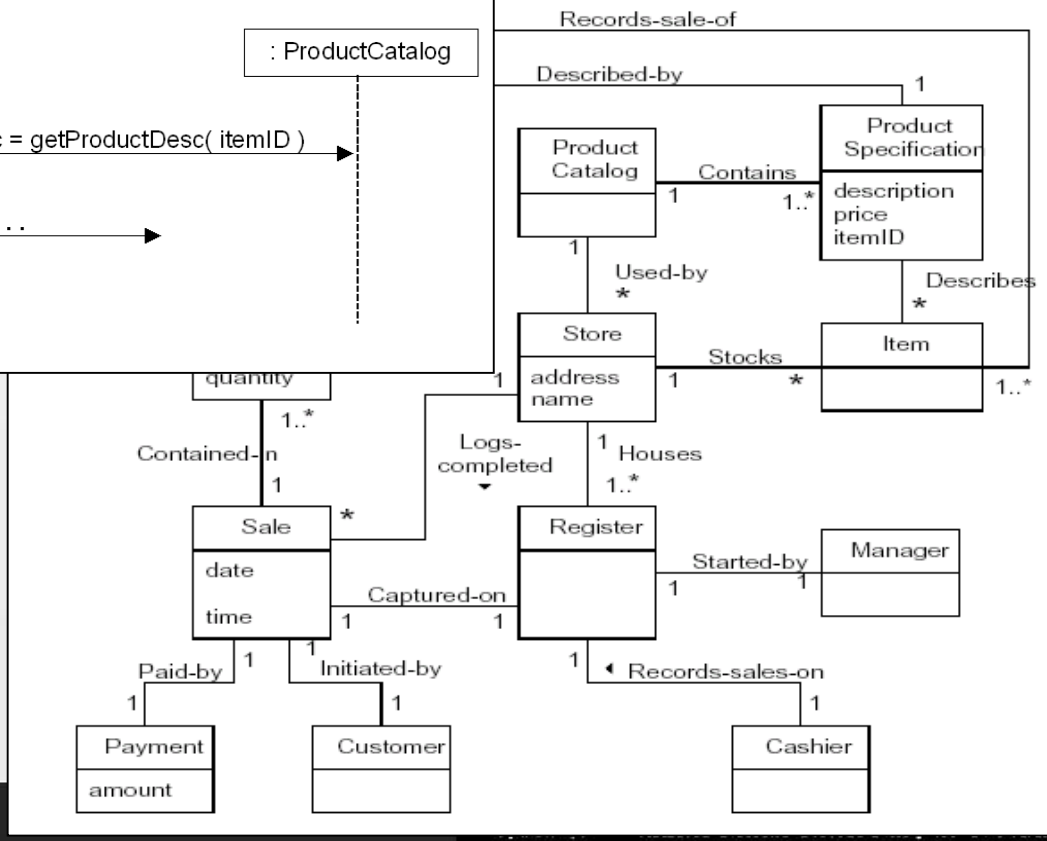


Object Design Examples with GRASP

- Use Case Realization?

The system operations in the SSDs are used as the starting messages into the domain layer controller objects.

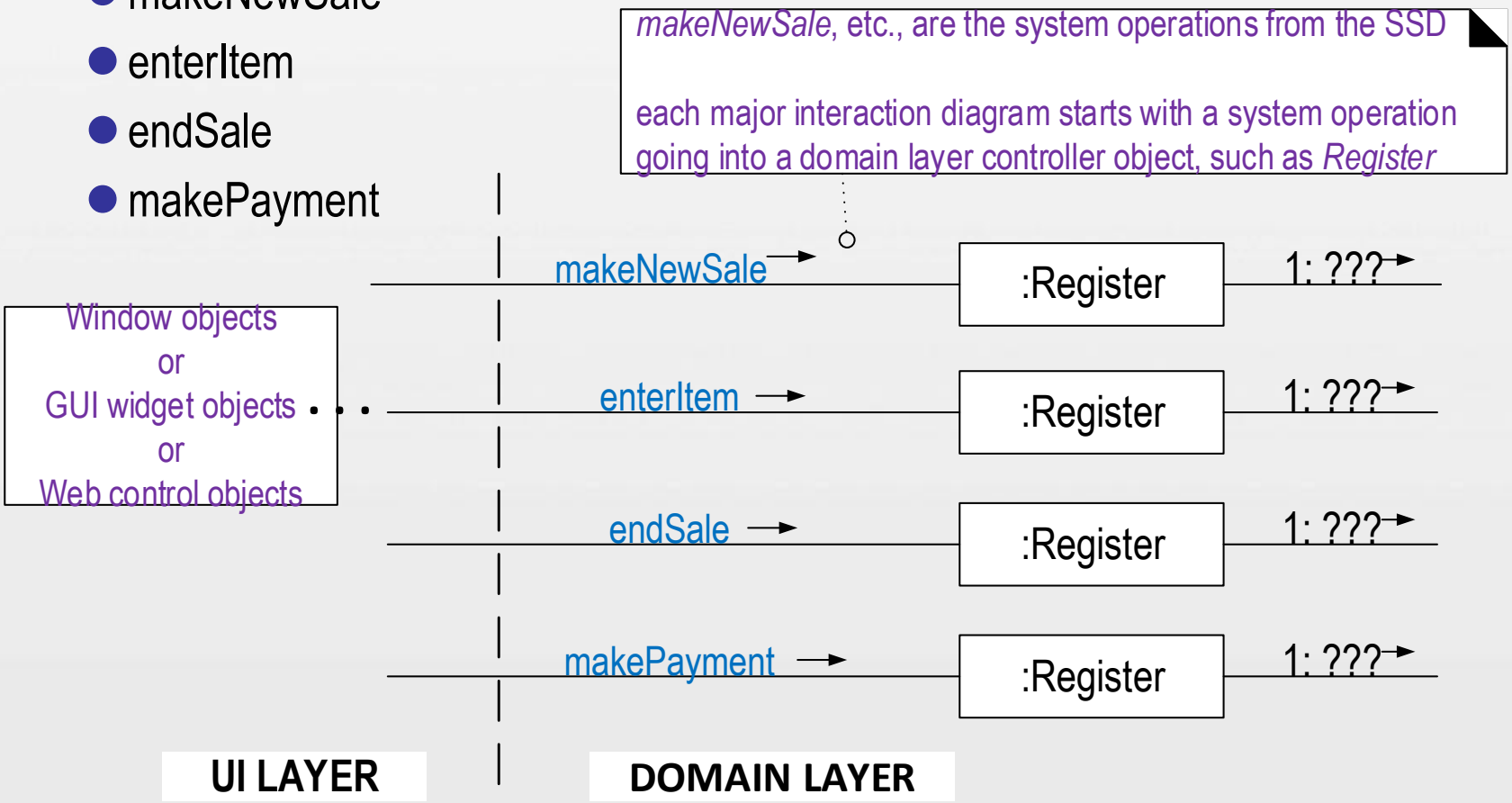




Object Design Examples with GRASP

Use Case Realizations for the NextGen Iteration

- considering scenarios and system operations identified on the SSDs of the Process Sale use case:
  - makeNewSale
  - enterItem
  - endSale
  - makePayment



Object Design Examples with GRASP

How to Design makeNewSale

Contract CO1: makeNewSale

Contract CO1: makeNewSale

Postconditions:
- A Sale instance s was created (instance creation).
- s was associated with the Register (association formed).
- Attributes of s were initialized

Choosing the Controller Class

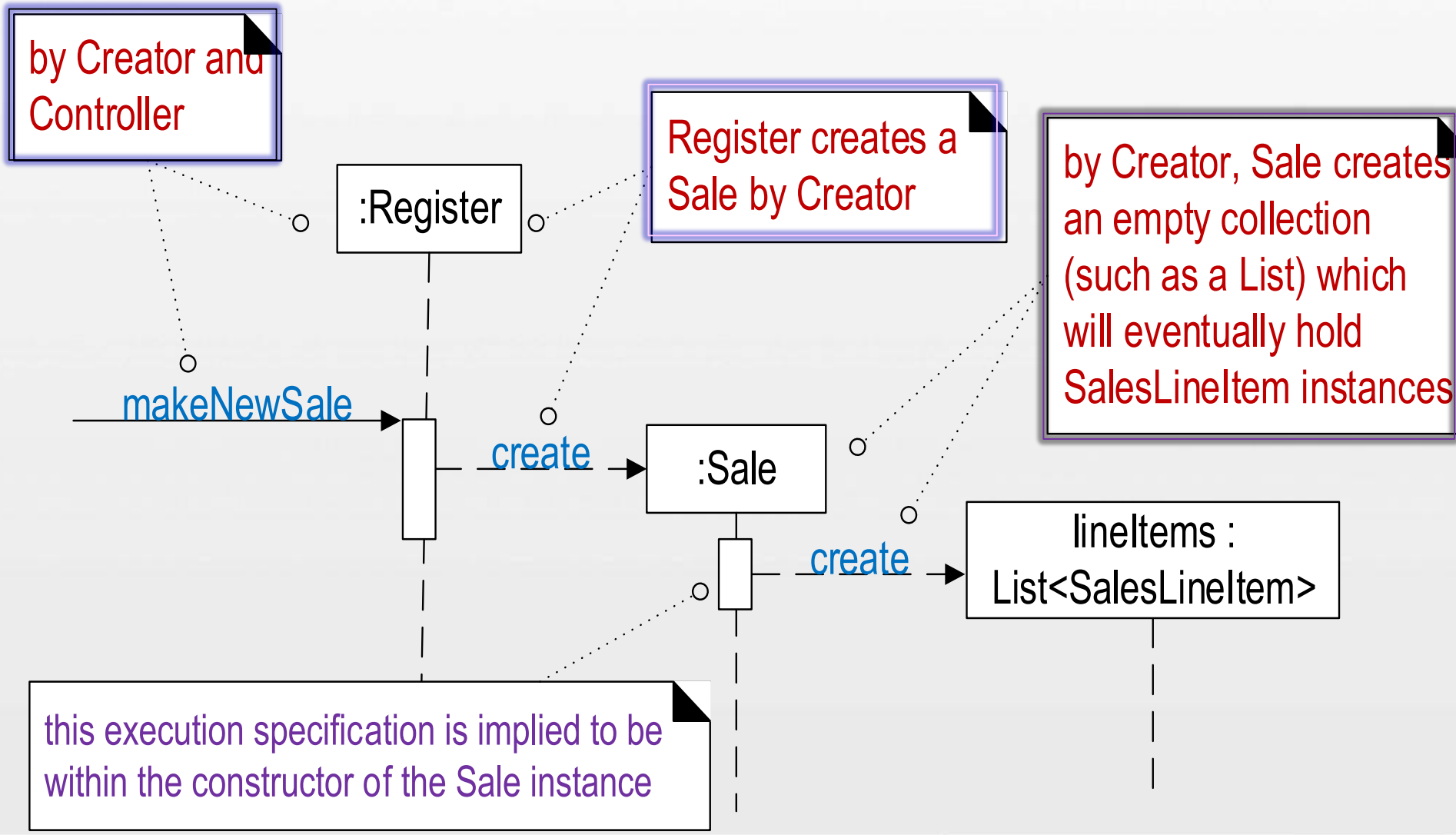
- Facade controller or Use case Controllers?

Creating a New Sale

- Creator pattern suggests assigning the responsibility for creation to a class that aggregates, contains, or records the object to be created
- **Domain Model reveals that a Register may be thought of as recording a Sale ---- Register is concept class**
- **So, Register is a reasonable candidate for creating a Sale – Register is design class here.**
- **when the Sale is created, it must create an empty collection (such as a Java List) to record all the future SalesLineItem**

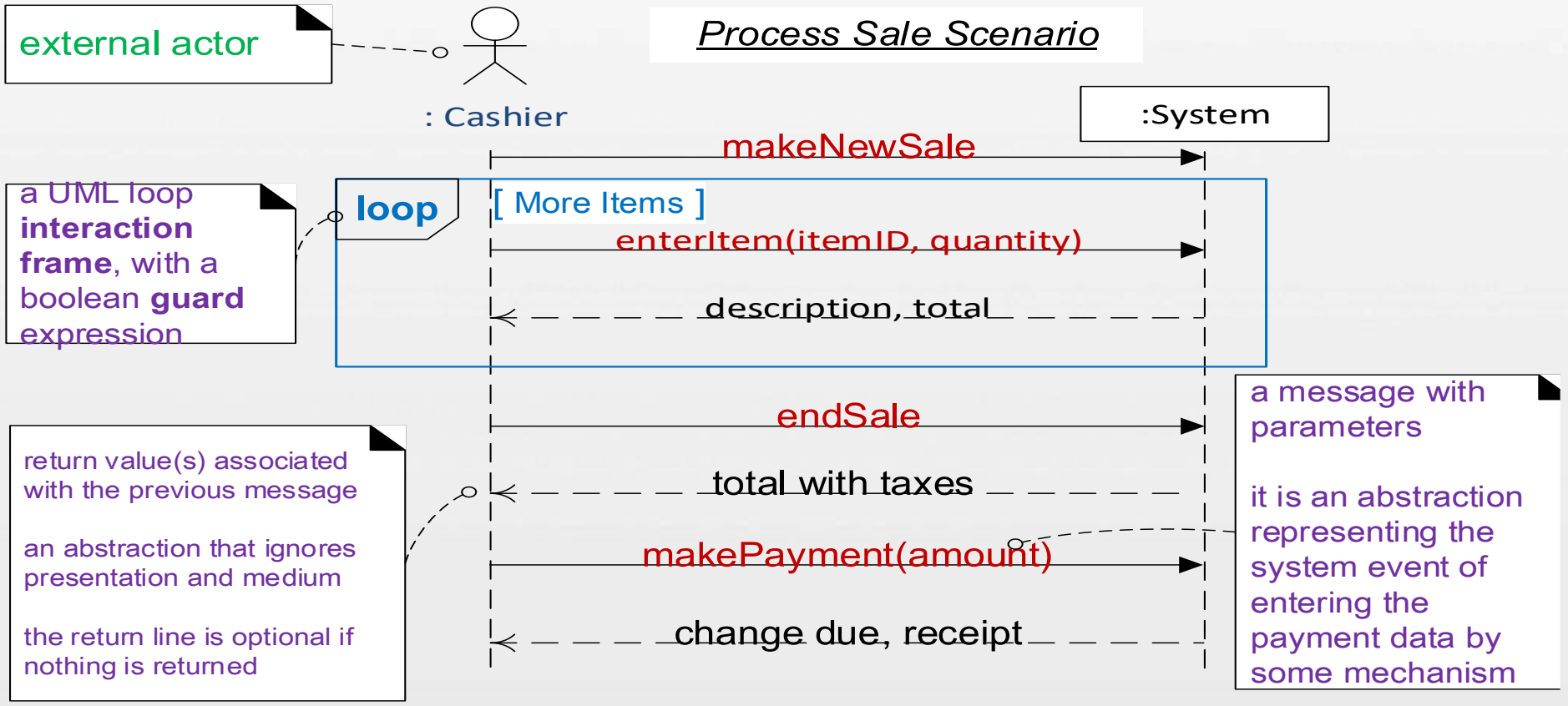
Object Design Examples with GRASP

- How to Design makeNewSale



Object Design Examples with GRASP

- How to Design enterItem?



Object Design Examples with GRASP

How to Design enterItem?

Contract CO2: enterItem	
Operation:	enterItem(itemID : ItemID, quantity : integer)
Postconditions:	- A SalesLineItem instance sli was created (instance creation). - sli was associated with the current Sale (association formed). - sli.quantity became quantity (attribute modification). - sli was associated with a ProductDescription, based on itemID match (association formed).

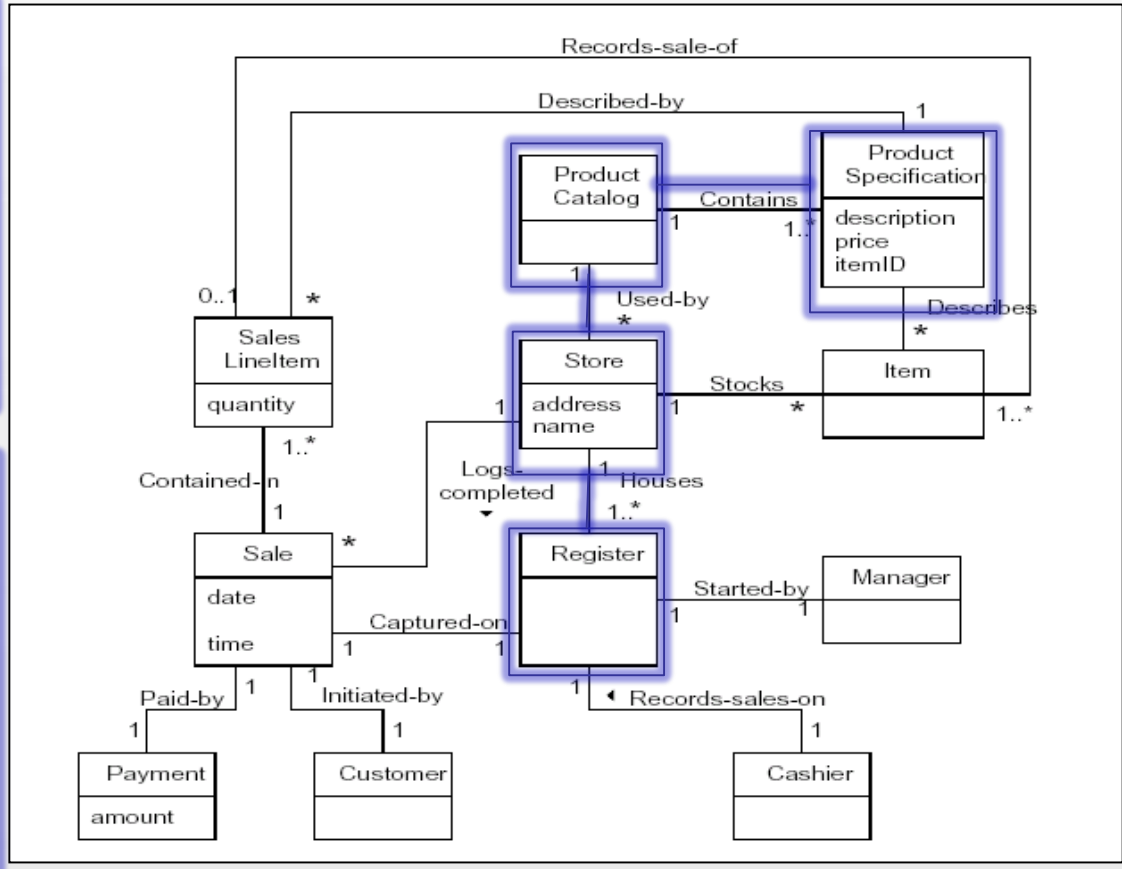
- **Choosing the Controller Class**
  - Facade controller or Use case Controllers?
- **Creating a New SalesLineItem**
  - Domain Model reveals that a Sale contains SalesLineItem objects
  - software Sale may similarly contain software SalesLineItem. By Creator, a software Sale is an appropriate candidate to create a SalesLineItem
- **Finding a ProductDescription**
  - **Who should be responsible for knowing a ProductDescription, based on an itemID match?**
  - **Information expert -- the principal pattern**

Object Design Examples with GRASP

- **How to Design enterItem?**
  - ➢ **Who knows about all the ProductDescription objects?**

Analyzing the Domain Model reveals that the ProductCatalog contains all the ProductDescriptions

by Information Expert ProductCatalog getProductDescription message



Who send the getProductDescription message to the ProductCatalog?

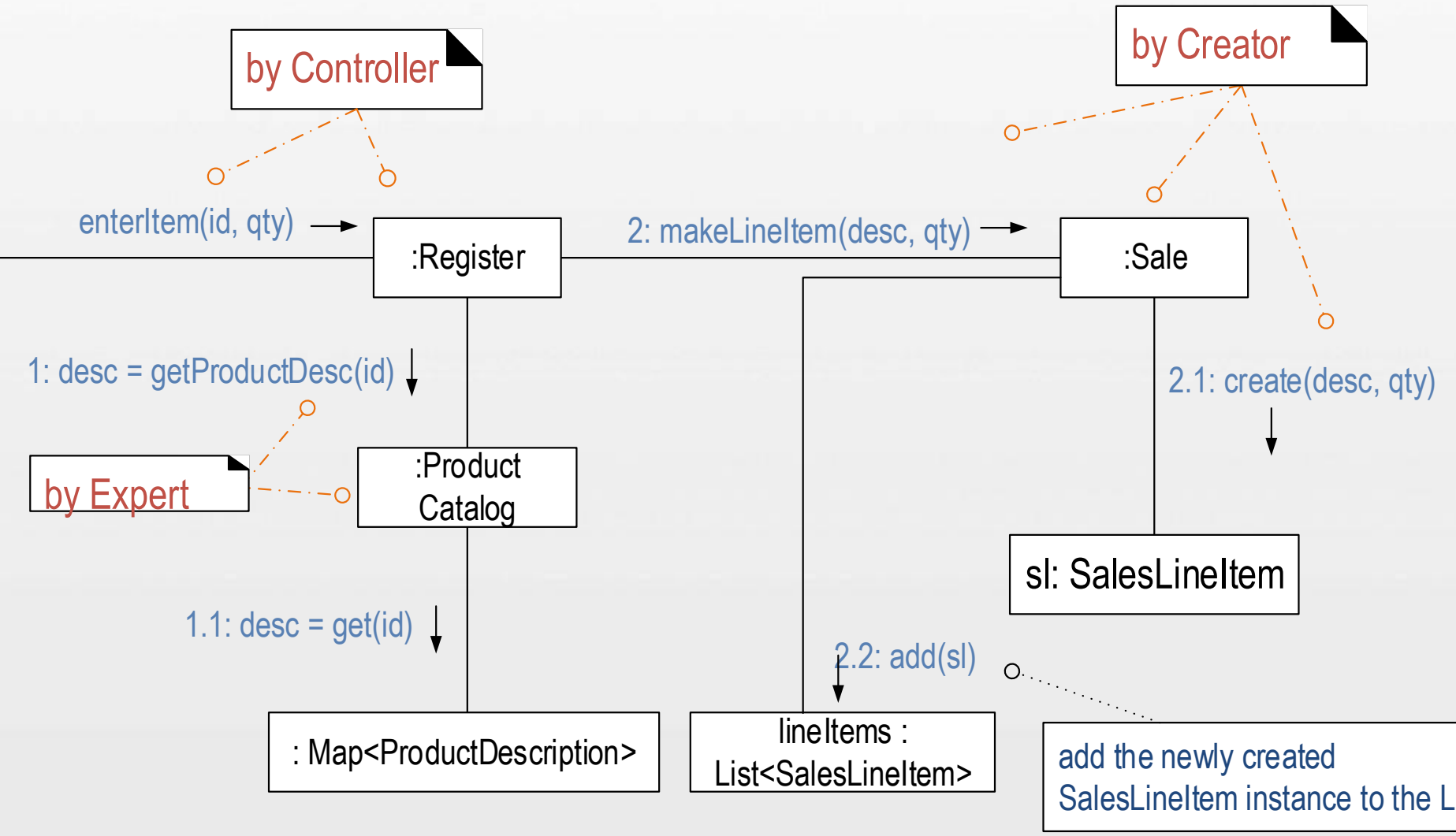
assume that the Register has a permanent reference to the ProductCatalog

Register

ProductDescriptions are stored in a relational database and retrieved on demand, how to ?

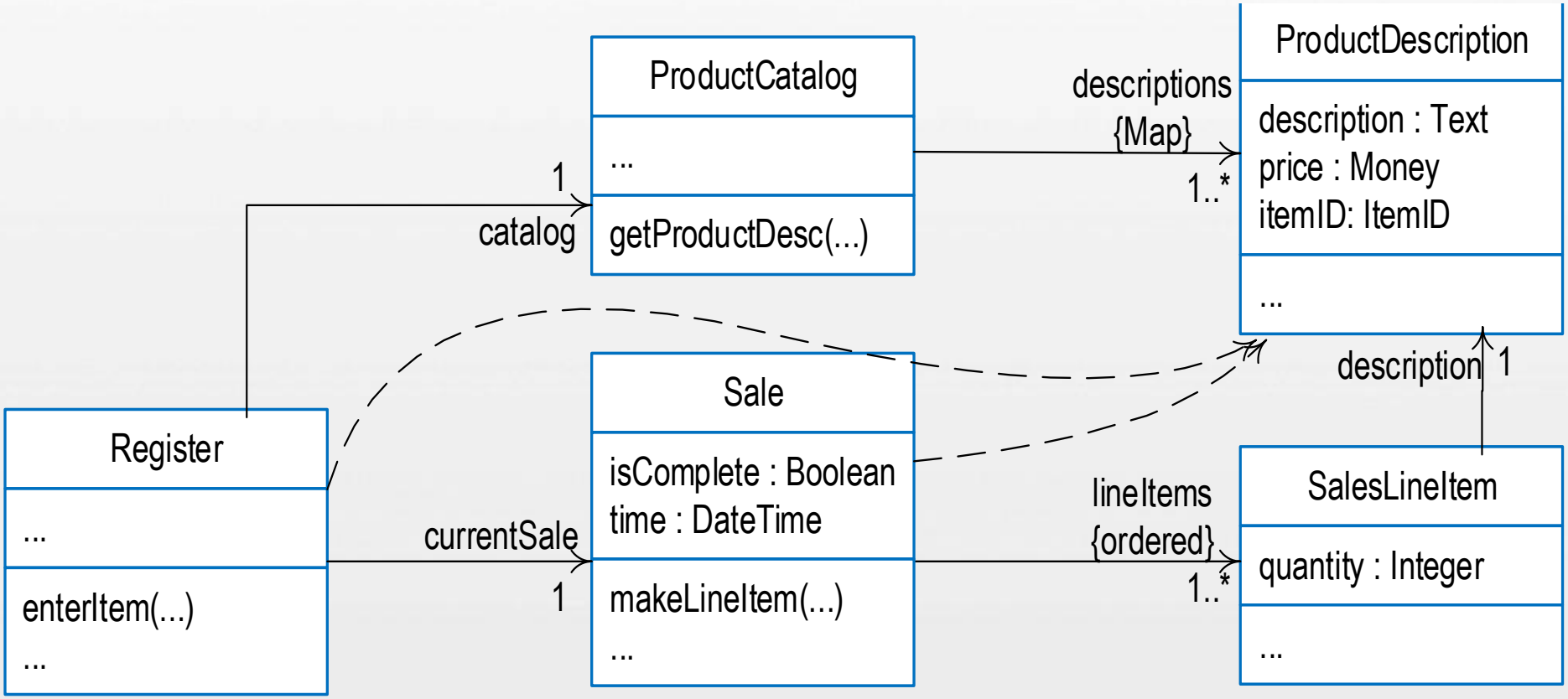
Object Design Examples with GRASP

- How to Design enterItem?



Object Design Examples with GRASP

- How to Design enterItem?




```
Register::enterItem(id, qty){
        ProductDescription desc = ProductCatelog.getInstance().get(id);
        currentSale.makeLineItem(desc,qty);
}
```

Object Design Examples with GRASP

- **How to Design endSale?**
  - **Postconditions:**
    - Sale.isComplete became true (attribute modification).
  - **1 Choosing the Controller Class**
    - Same as the previous operation.. **Register**
  - **2 Setting the Sale.isComplete Attribute?**
    - Who should be responsible for setting the isComplete attribute of the Sale to true?
    - By Expert, the Sale itself。 the Register send a becomeComplete message to the Sale to set it to true
  - **3 Calculating the Sale Total.**
    - sale total is the sum of the subtotals of all the sales line-items.
    - sales line-item subtotal := line-item quantity * product description price

Object Design Examples with GRASP

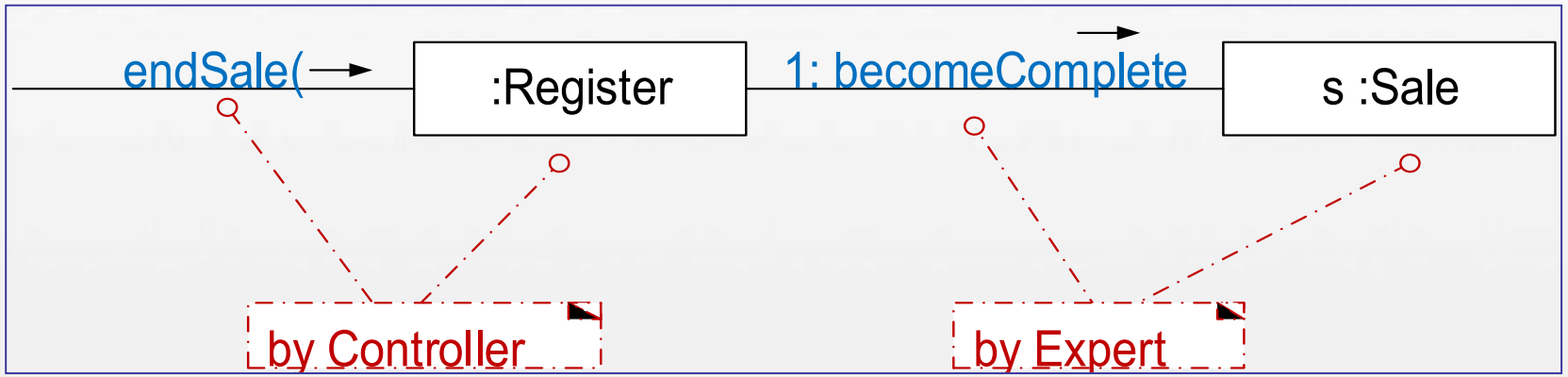
- **How to Design endSale?**
  - **3 Calculating the Sale Total.**

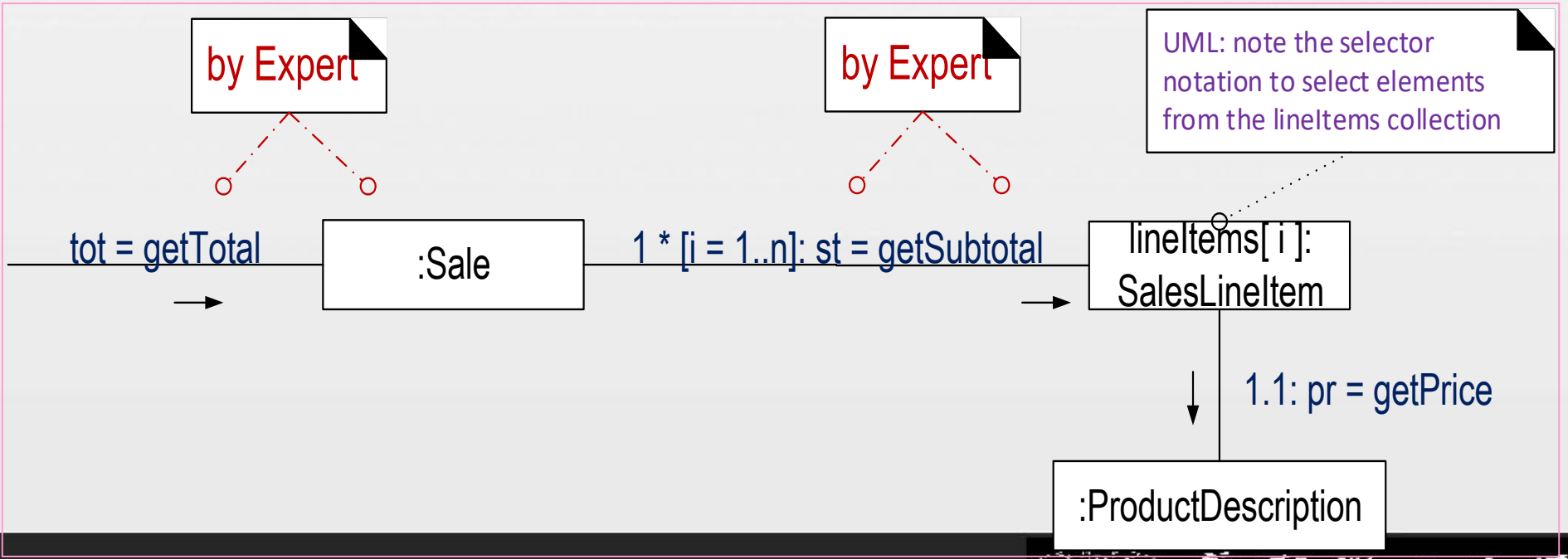
Information Required for Sale Total	Information Expert
ProductDescription.price	ProductDescription
SalesLineItem.quantity	SalesLineItem
all the SalesLineItems in the current Sale	Sale

- Who should be responsible for calculating the Sale total?
  - By expert, sale itself. getTotal method
- For a Sale to calculate its total, it needs the subtotal for each SalesLineItem, who?
  - By expert, SalesLineItem itself. getSubTotal method
- For the SalesLineItem to calculate its subtotal, it needs the price of the ProductDescription, who?
  - By expert, ProductDescription itself. getPrice method

Object Design Examples with GRASP

- How to Design endSale?





Object Design Examples with GRASP

How to Design makePayment?

Postconditions:
- A Payment instance p was created (instance creation).
- p.amountTendered became amount (attribute modification).
- p was associated with the current Sale (association formed).
- The current Sale was associated with the Store (association formed); (to add it to the historical log of completed sales).

➢ **1 Choosing the Controller Class**

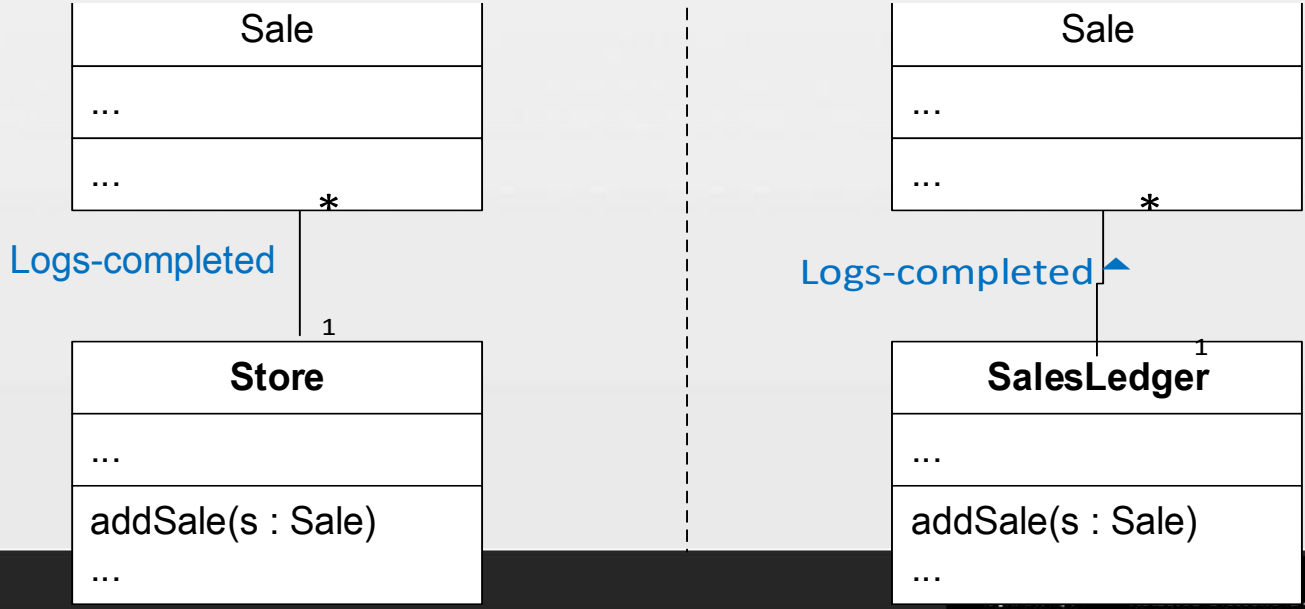
- Same as the previous operation.. **Register**

➢ **2 Creating the Payment,**

- Who records, aggregates, most closely uses, or contains a Payment?—Creator
  - Register records payment.
  - Sale object closely use a Payment
- **When there are alternative design choices, take a closer look at the cohesion and coupling**
  - The second design has low coupling

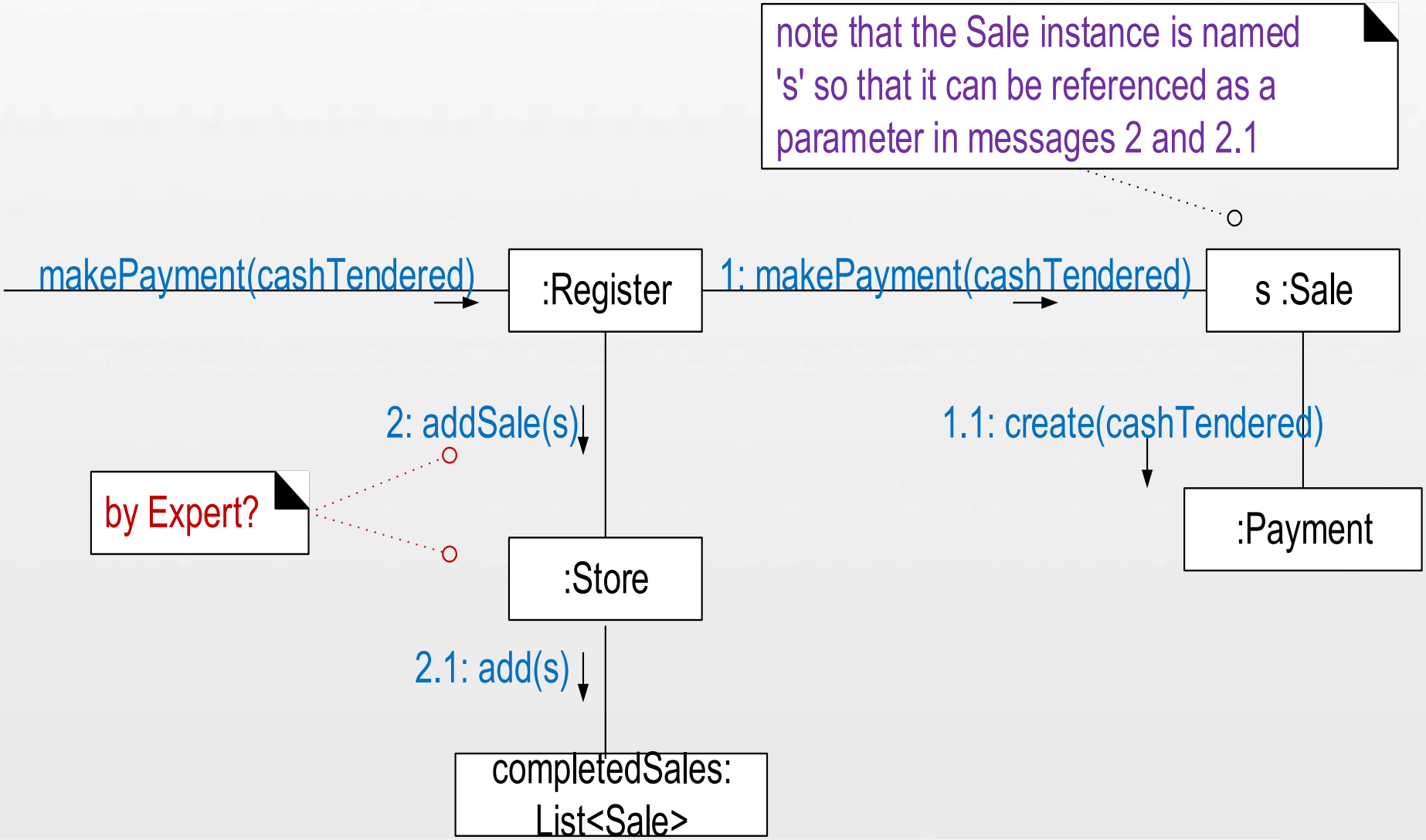
Object Design Examples with GRASP

- **How to Design makePayment?**
  - 3. Logging a Sale
    - Information Expert should be an early pattern considered unless the problem is a controller or creation problem
    - **Who is responsible for knowing all the logged sales and doing the logging**
      - Sale? -- information expert – low cohesion for sale class
      - Store? From domain model
      - OtherClass? Such as SalesLedger



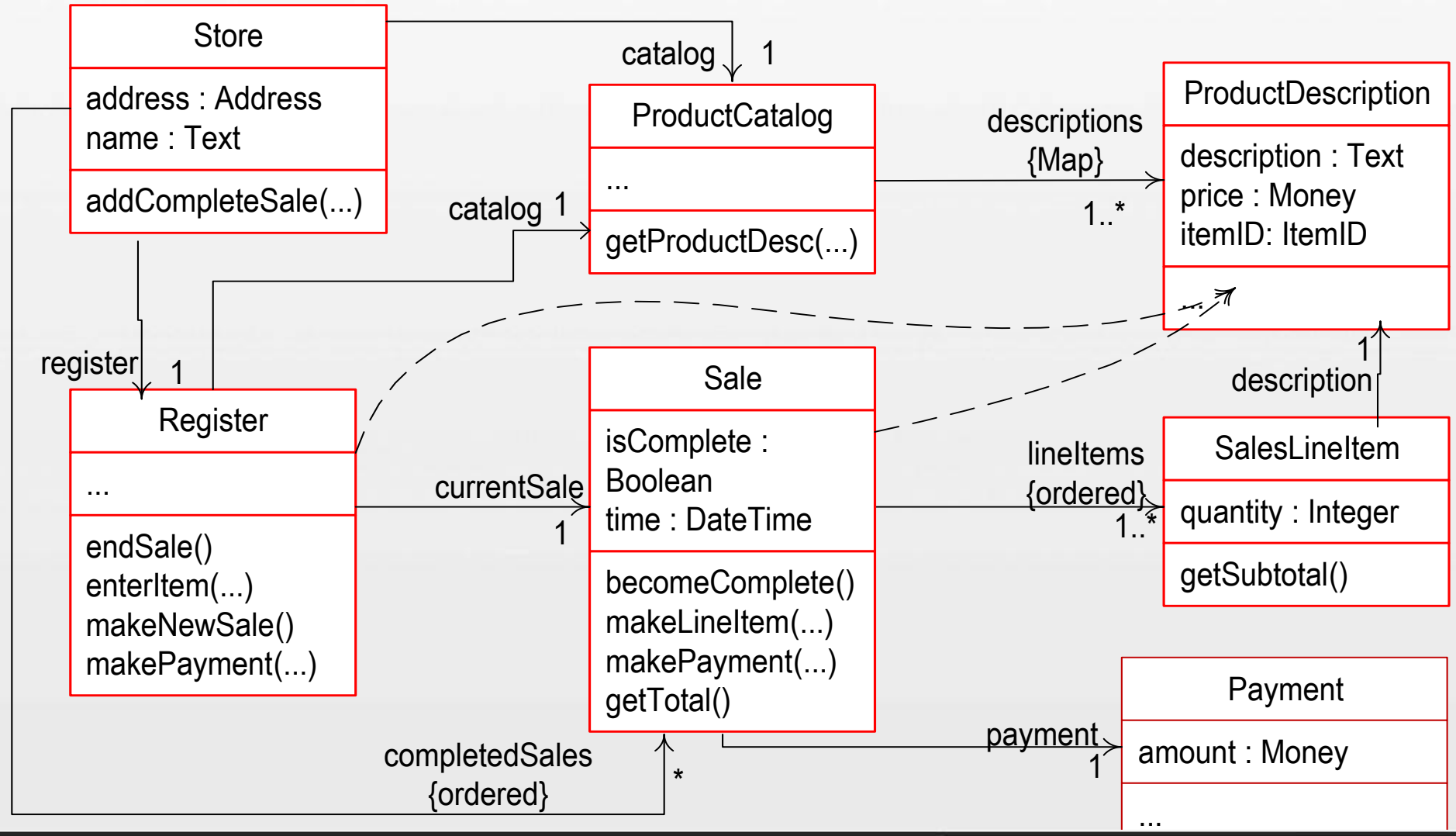
Object Design Examples with GRASP

- How to Design makePayment?



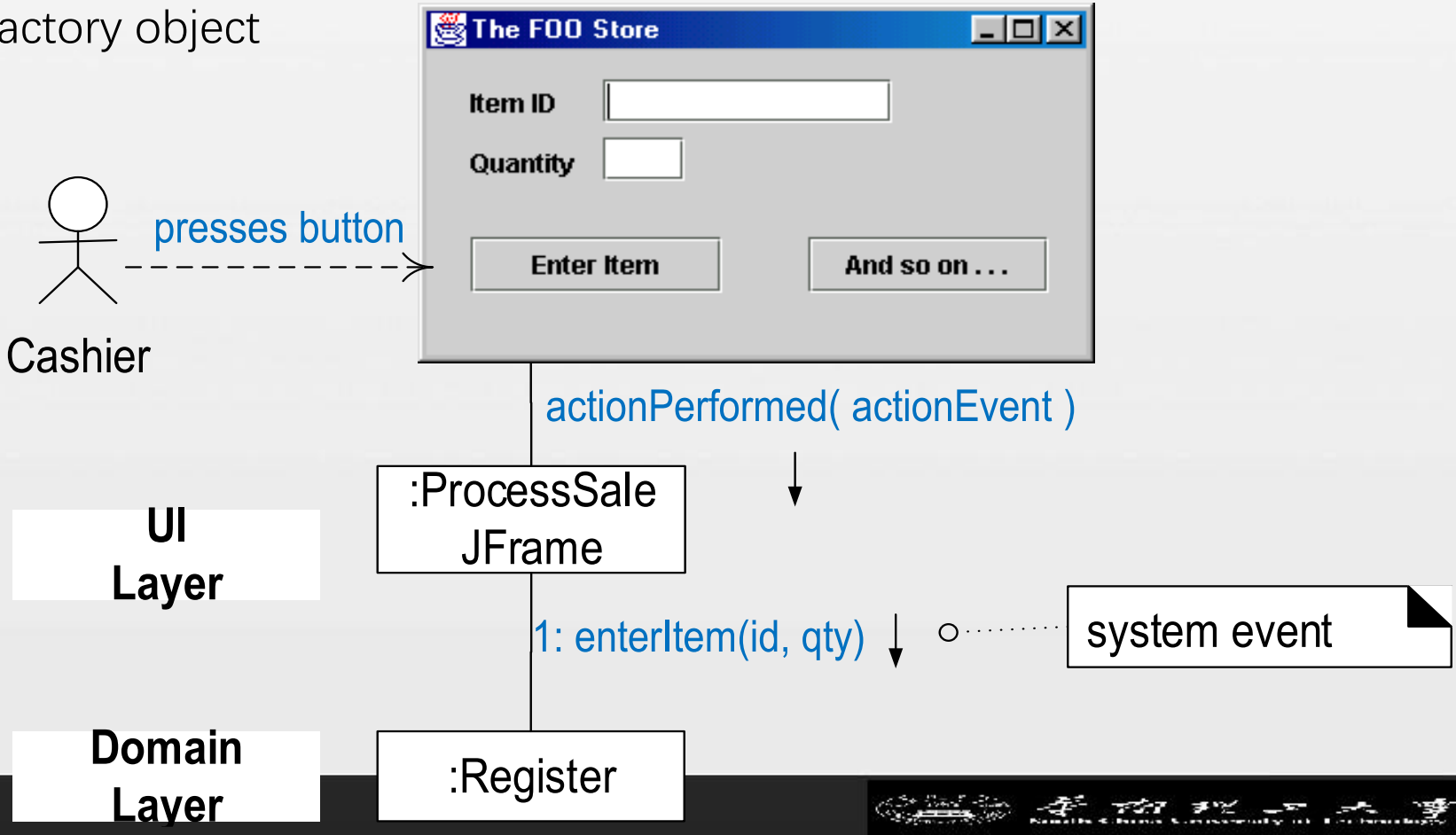
Object Design Examples with GRASP

- The Final NextGen DCD for Iteration-1



Object Design Examples with GRASP

- **Connect the UI Layer to the Domain Layer?**
  - An initializer object creates both a UI and a domain object and passes the domain object to the UI.
  - A UI object retrieves the domain object from a well-known source, such as a factory object



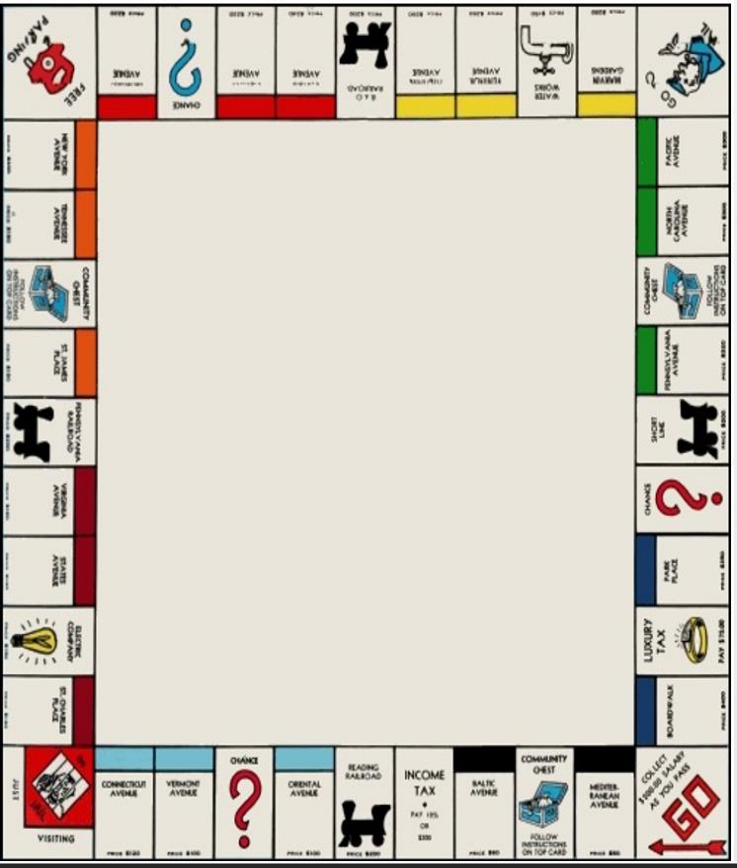
Object Design Examples with GRASP

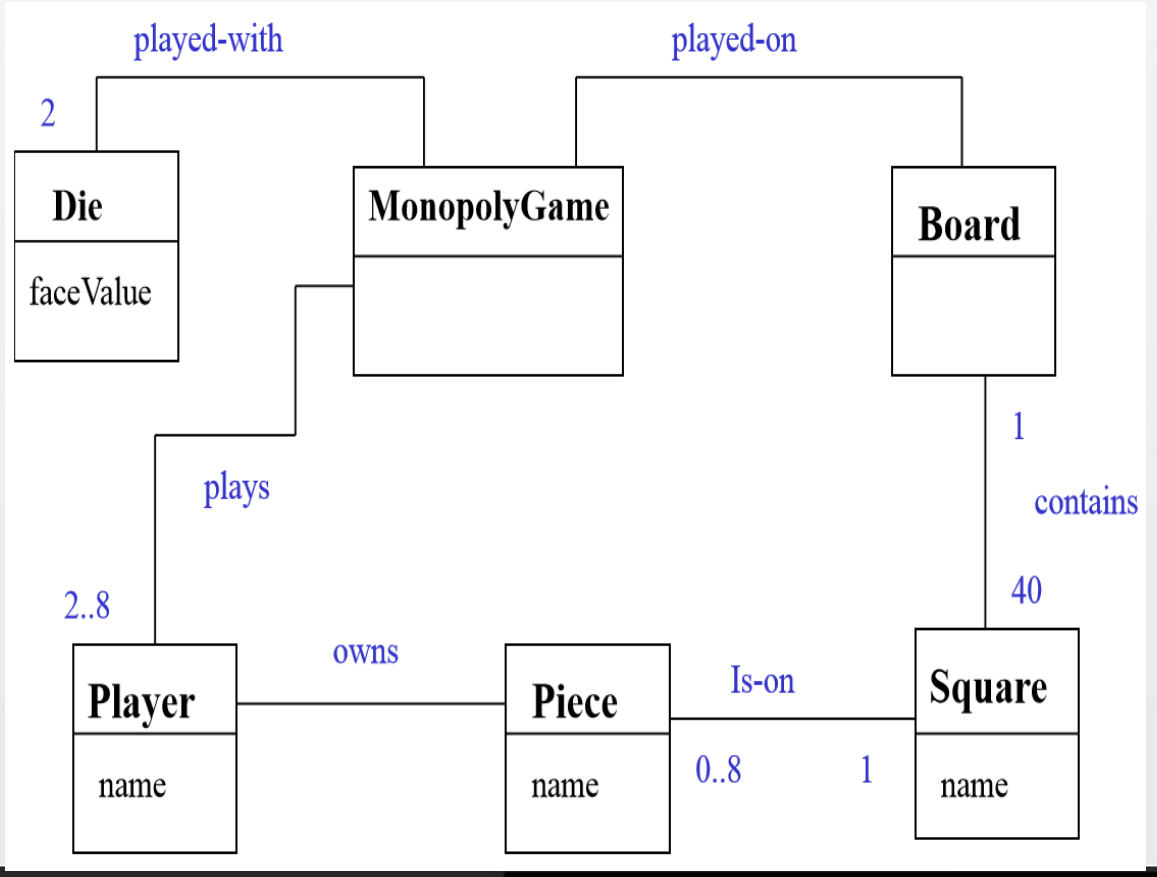
- **Initialization and the 'Start Up' Use Case**
  - When to Create the Initialization Design?
    - Do the initialization design last.
  - How do Applications Start Up?
    - **create an initial domain object**
      - Store or Register? Store
    - passes the domain object to the UI

```
public class Main
{
   public static void main(String[] args)
   {
            Store store = new Store();
            Register register = store.getRegister();
            ProcessSaleJFrame frame = new ProcessSampleJFrame(register);
            .....
   }
}
```

Object Design Examples with GRASP

- **Use Case Realizations for the Monopoly Iteration**
  - **Why another example?**
    - it isn't a business application. But the logic especially in later iterations is quite complex, with rich OO design problems to solve.





Object Design Examples with GRASP

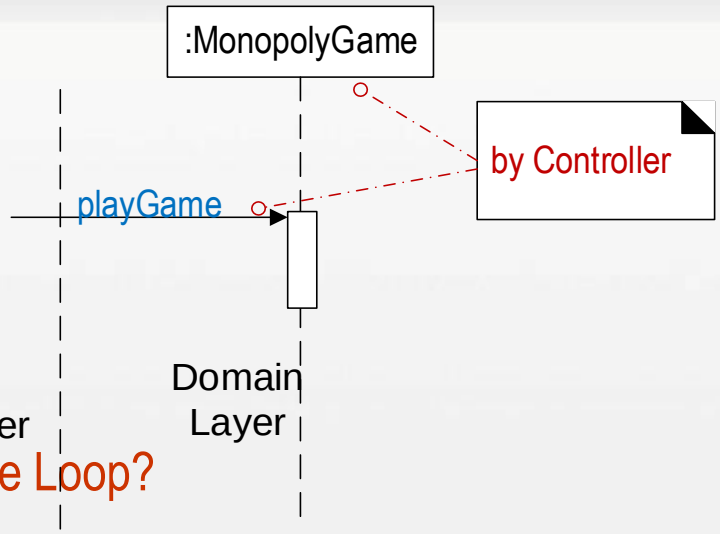
Use Case Realizations for the Monopoly Iteration

- First Iteration of the Monopoly Game
  - In Iteration 1 – there is no winner. The rules of the game are not yet incorporated into the design. Iteration 1 is merely concerned with the mechanics of having a player move a piece around the Board, landing on one of the 40 Squares each turn
- Definition
  - turn – a player rolling the dice and moving one piece
  - round – all players taking one turn
- the game loop algorithm:

```
for N rounds
        for each player p
                p takes a turn
```

Object Design Examples with GRASP

- **How to Design playGame?**
  - **Choosing the Controller Class**
    - Facade controller or Use case Controllers?
    - **MonopolyGame**
  - **The Game-Loop Algorithm**
    - 1 Who is Responsible for Controlling the Game Loop?
      - is a doing responsibility
      - Apply information expert pattern, What information is needed for the responsibility?“

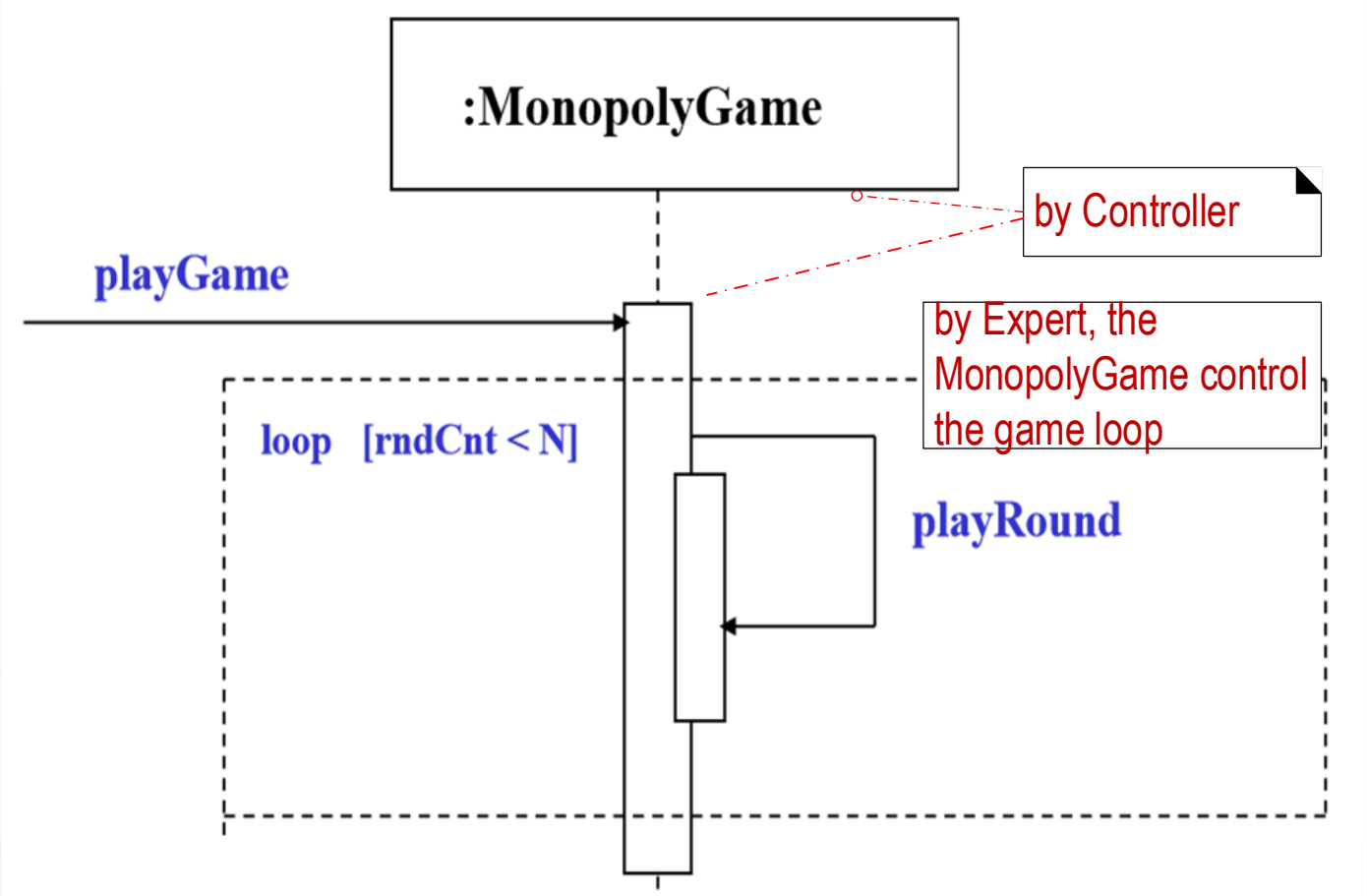


Information Needed	Who Has the Information?
the current round count	No object has it yet, but by LRG, assigning this to the MonopolyGame object is justifiable.
all the players (so that each can be used in taking a turn)	Taking inspiration from the domain model, MonopolyGame is a good candidate.

  - MonopolyGame is a justifiable choice to control the game loop and coordinate the playing of each round.

Object Design Examples with GRASP

- How to Design playGame?



Object Design Examples with GRASP

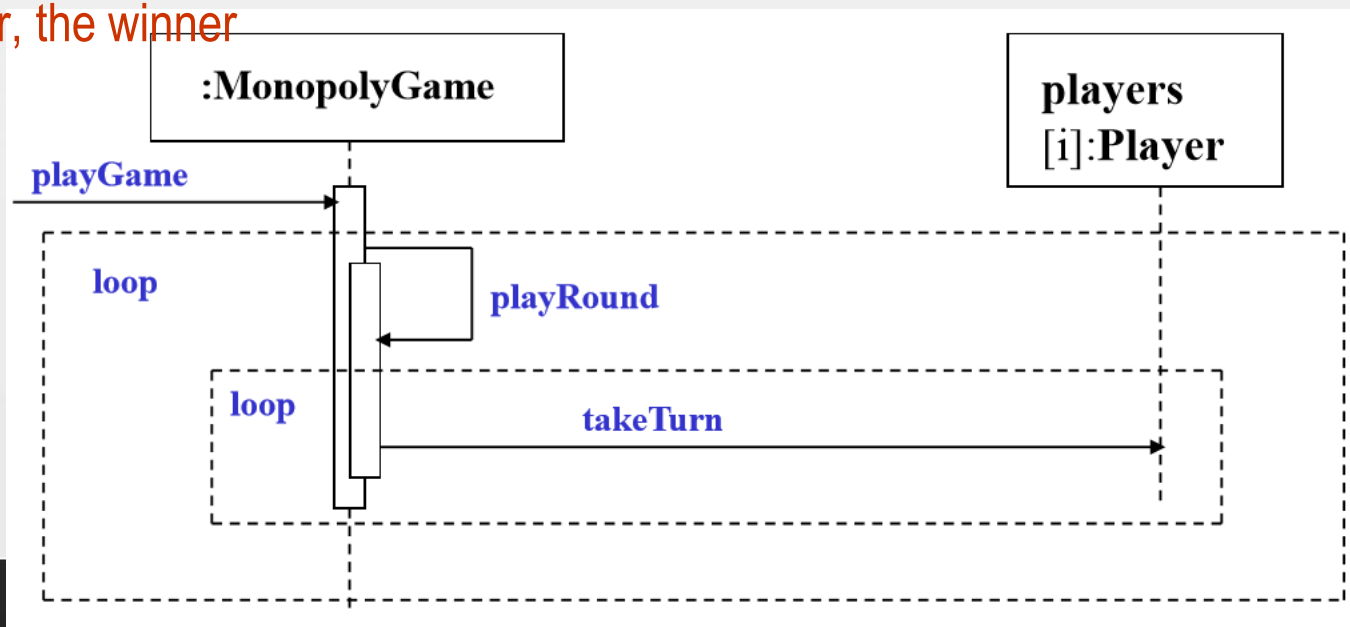
- **How to Design playGame?**
  - **The Game-Loop Algorithm**
    - 2 Who Takes a Turn?
      - is a doing responsibility. Again, Expert applies
      - What information is needed for the responsibility?

Information Needed	Who Has the Information?
current location of the player (to know the starting point of a move)	Taking inspiration from the domain model, a Piece knows its Square and a Player knows its Piece. Therefore, a Player software object could know its location by LRG.
the two Die objects (to roll them and calculate their total)	Taking inspiration from the domain model, MonopolyGame is a candidate since we think of the dice as being part of the game.
all the squaresthe square organization (to be able to move to the correct new square)	By LRG, Board is a good candidate

      - three partial information experts for the "take a turn" responsibility: Player, MonopolyGame, and Board? How to select?

Object Design Examples with GRASP

- How to Design playGame?
  - The Game-Loop Algorithm
    - 2 Who Takes a Turn? **three partial information experts, how to select?**
      - Guideline: place the responsibility in the dominant information expert
      - Guideline: consider the coupling and cohesion impact of each, and choose the best
      - Guideline: consider probable future evolution of the software objects and the impact in terms of Information Expert, cohesion, and coupling
      - Player, the winner

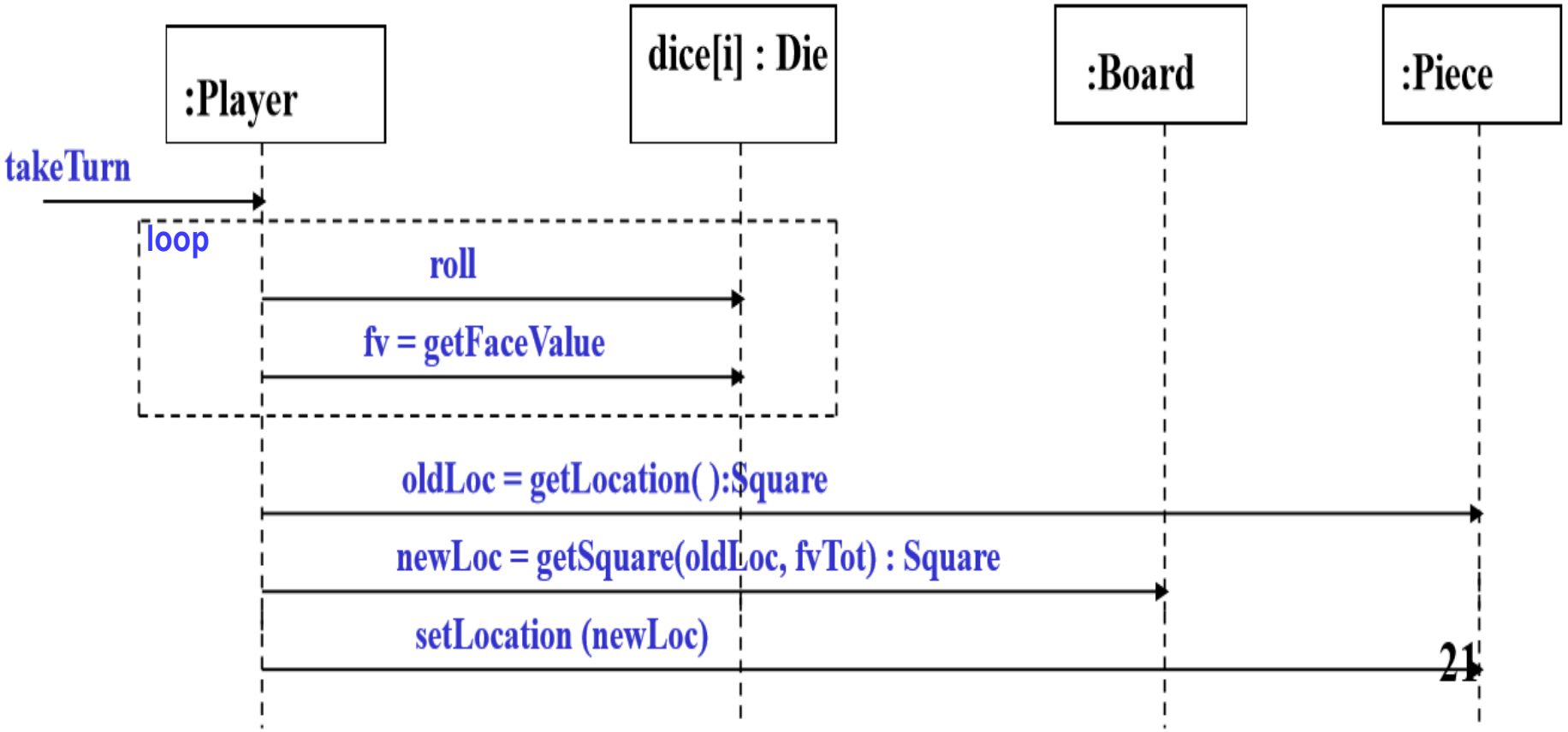


Object Design Examples with GRASP

- **How to Design playGame?**
  - The Game-Loop Algorithm
    - **How to Taking a Turn?**
      - 1 Calculating a random number between 2 and 12
        - » Calculating a new face value means changing information in Die, so by Expert, Die should be able to roll itself (generate a random number) and answer its face value
      - 2 Determining the location of the new square
        - » The new square location problem: Since the Board knows all its Squares, it should be responsible for finding a new square location, given an old square location and some offset (the dice total)
      - 3 Moving the player's piece from the old location to the new square
        - » The piece movement problem: By LRG it is reasonable for a Player to know its Piece, and a Piece its Square location (or even for a Player to directly know its Square location). By Expert, a Piece will set its new location, but may receive that new location from its Player

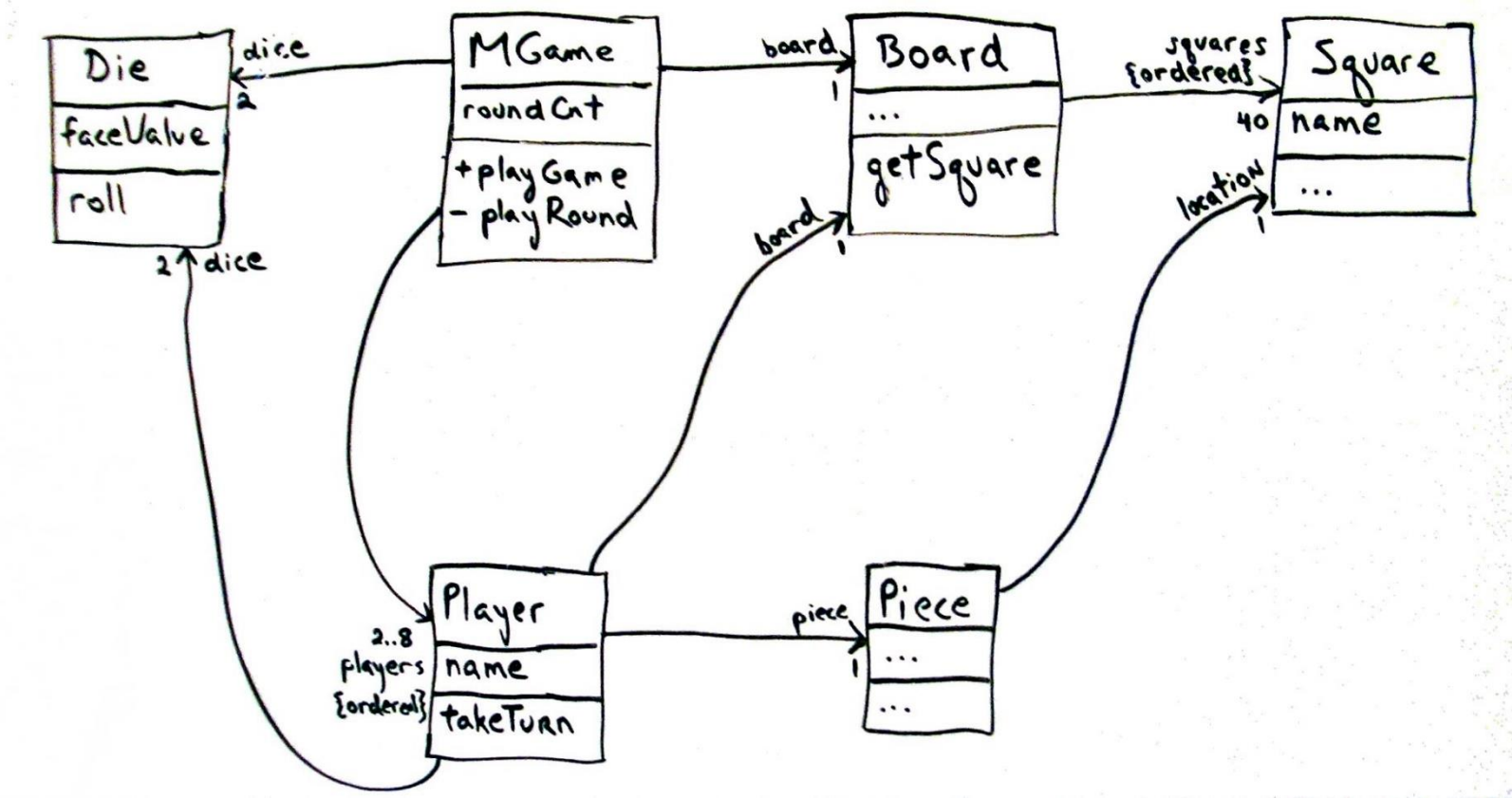
Object Design Examples with GRASP

- How to Design playGame?
  - The Game-Loop Algorithm
    - How to Taking a Turn?



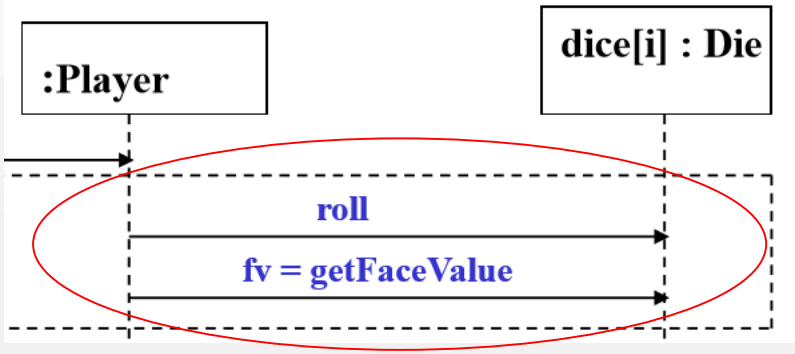
Object Design Examples with GRASP

- How to Design playGame?
  - The Game-Loop Algorithm
    - How to Taking a Turn?
      - Who Coordinates All This? Palyer
  - The Final Design of playGame



Object Design Examples with GRASP

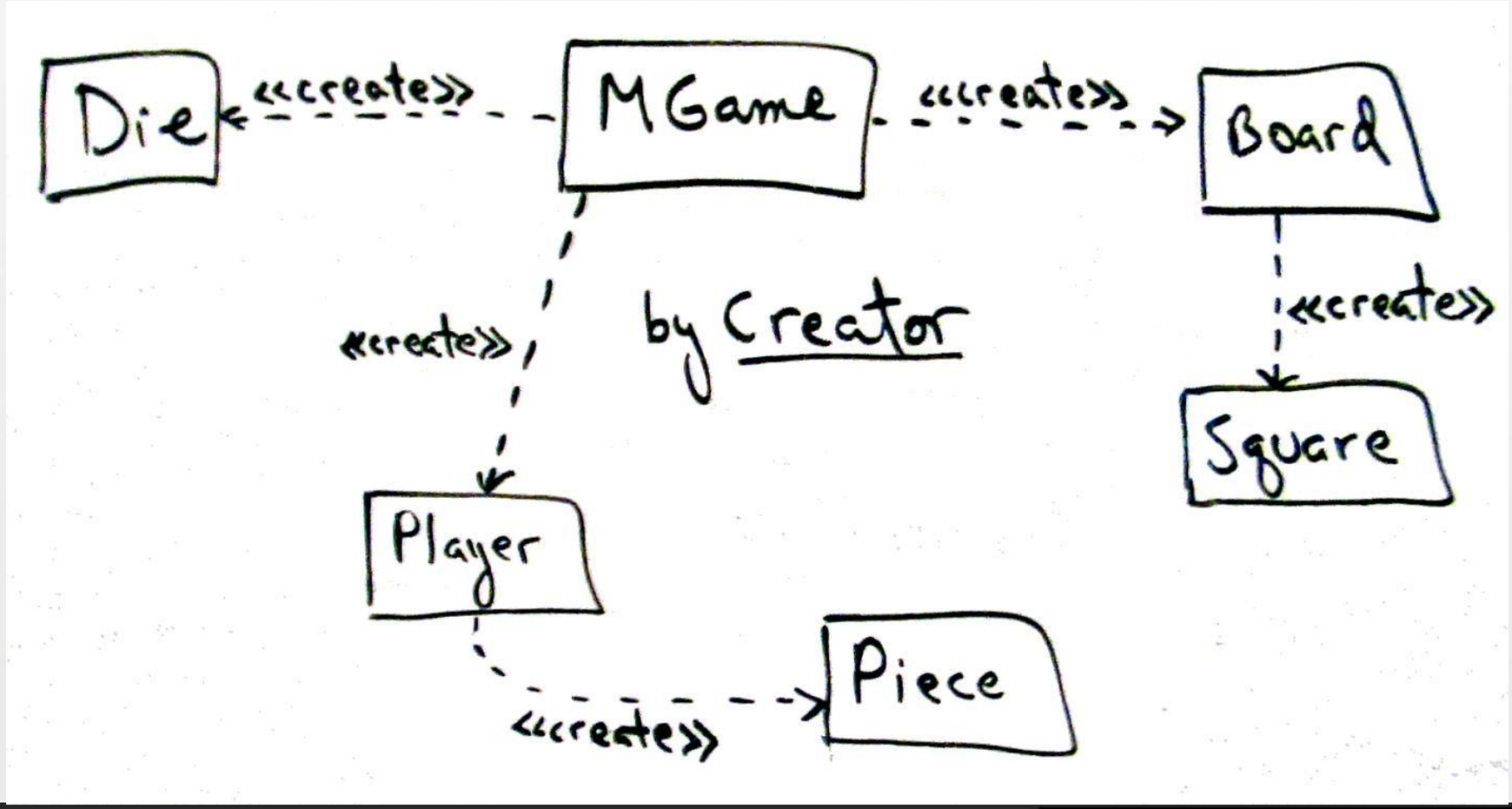
- The Command-Query Separation Principle，CQS
  - Why not make roll method returns the new faceValue?



  - violates the Command-Query Separation Principle, (CQS)
    - every method should either be
      - a command method that performs an action (updating, coordinating, …), often has side effects such as changing the state of objects, and is void (no return value); or
      - a query that returns data to the caller and has no side effects,it should not permanently change the state of any objects
    - The constant member function in class
  - Principle of Least Surprise

Object Design Examples with GRASP

- Initialization and the 'Start Up' Use Case
  - choose a suitable root object
    - MonopolyGame



Object Design Examples with GRASP

- **Iterative and Evolutionary Object Design**
  - **Elaboration**
    - use case realizations may be created for the most architecturally significant or risky scenarios of the design
    - do interaction diagrams for the key use case realizations