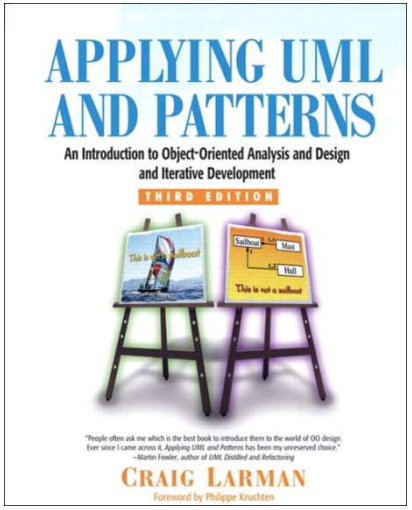
GRASP Principles

UA.DETI.PDS - 2022/23

José Luis Oliveira

Resources & Credits

- **Applying UML and Patterns**
  Craig Larman
  Chapters 16 & 22



- GRASP – General Responsibility Assignment Software Patterns Explained
  - http://www.kamilgrzybek.com/design/grasp-explained/

GRASP

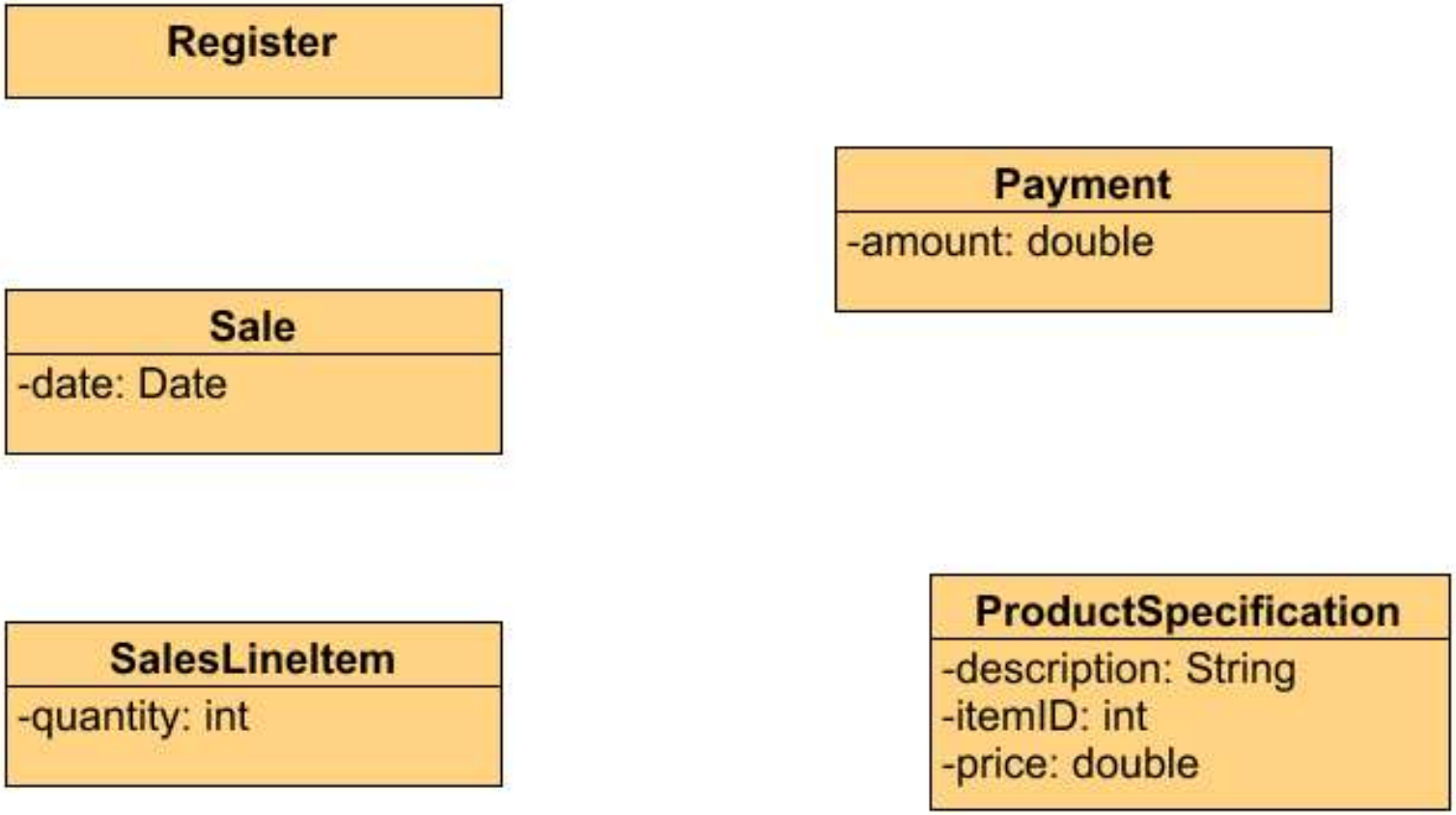
- General Responsibility Assignment Software Patterns
  - Name chosen to suggest the importance of grasping fundamental principles to successfully design object-oriented software
- Describe fundamental principles of object design and responsibility
- For instance …
  - You want to assign a responsibility to a class
  - You want to avoid or minimize additional dependencies
  - You want to maximise cohesion and minimise coupling
  - You want to increase reuse and decrease maintenance
  - You want to maximise understandability
  - …..etc.

Conducting example 1 - POS

- Point of Sale / Point of Sale Terminal:
  - Application for a shop, restaurant, etc. that registers sales.
  - Each sale is of one or more items of one or more product types, and happens at a certain date.

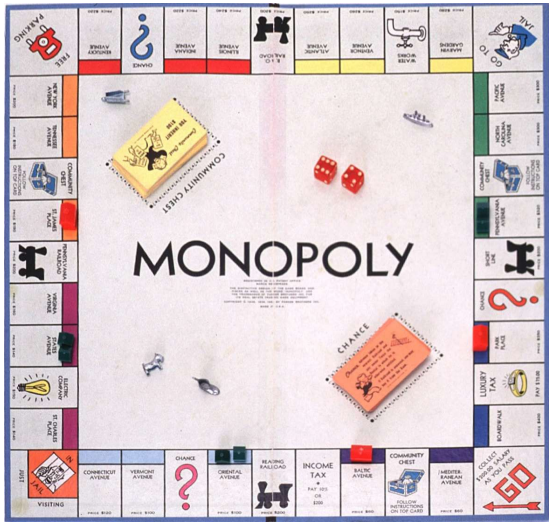
  - A product has a specification including a description, unitary price and identifier.
  - The application also registers payments (say, in cash) associated to sales.
  - A payment is for a certain amount, equal or greater that the total of the sale.

POS - A simple model



Conducting example 2: Monopoly

Die	MonopolyGame	Board
Player	Piece	Square



GRASP principles

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

GRASP principles

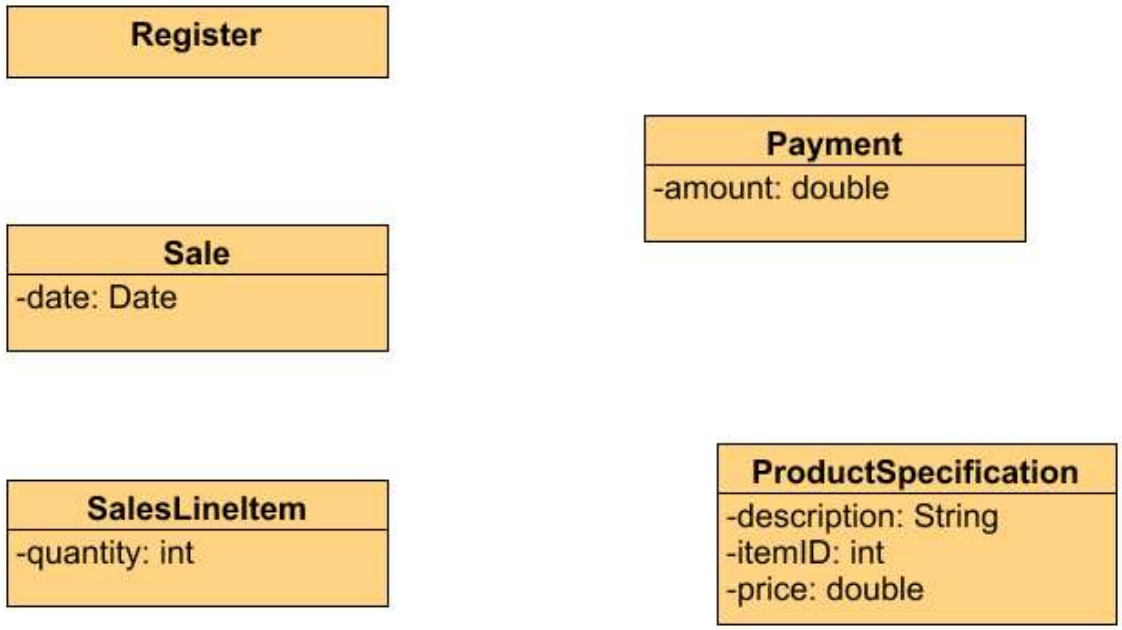
- Creator
- Information Expert
- Low Coupling
- High Cohesion
- Controller
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

Creator

- 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

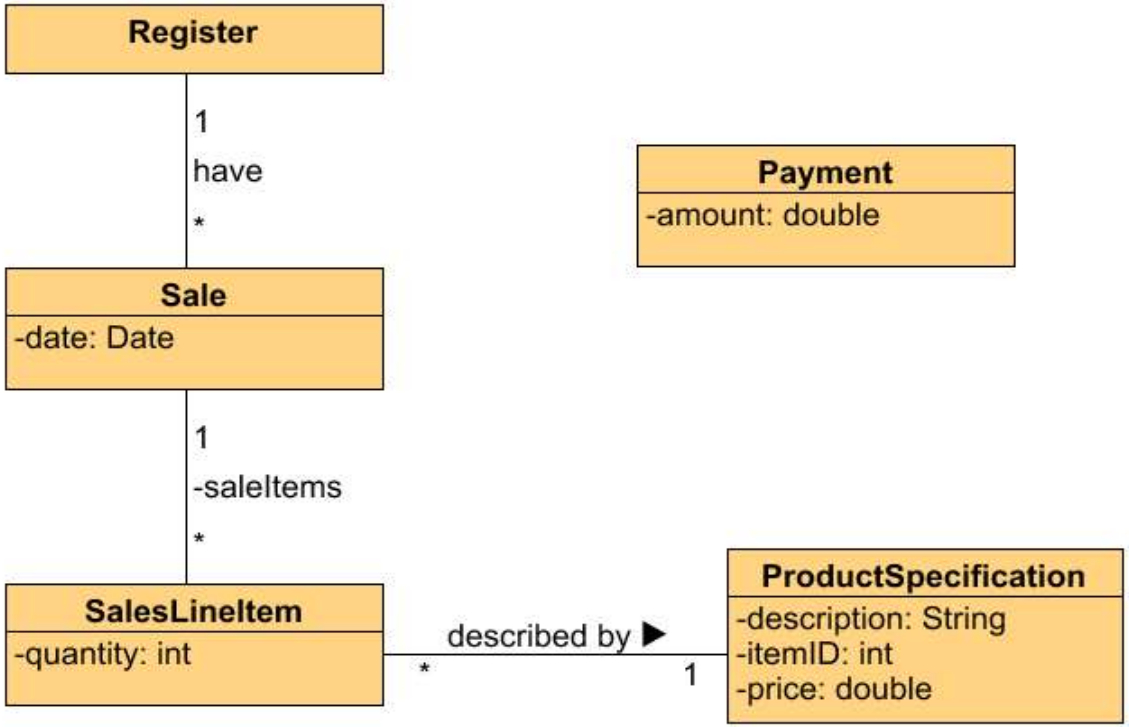
Creator: example

- Who is responsible for creating SalesLineItem objects, from an itemID and a quantity?



Creator: example

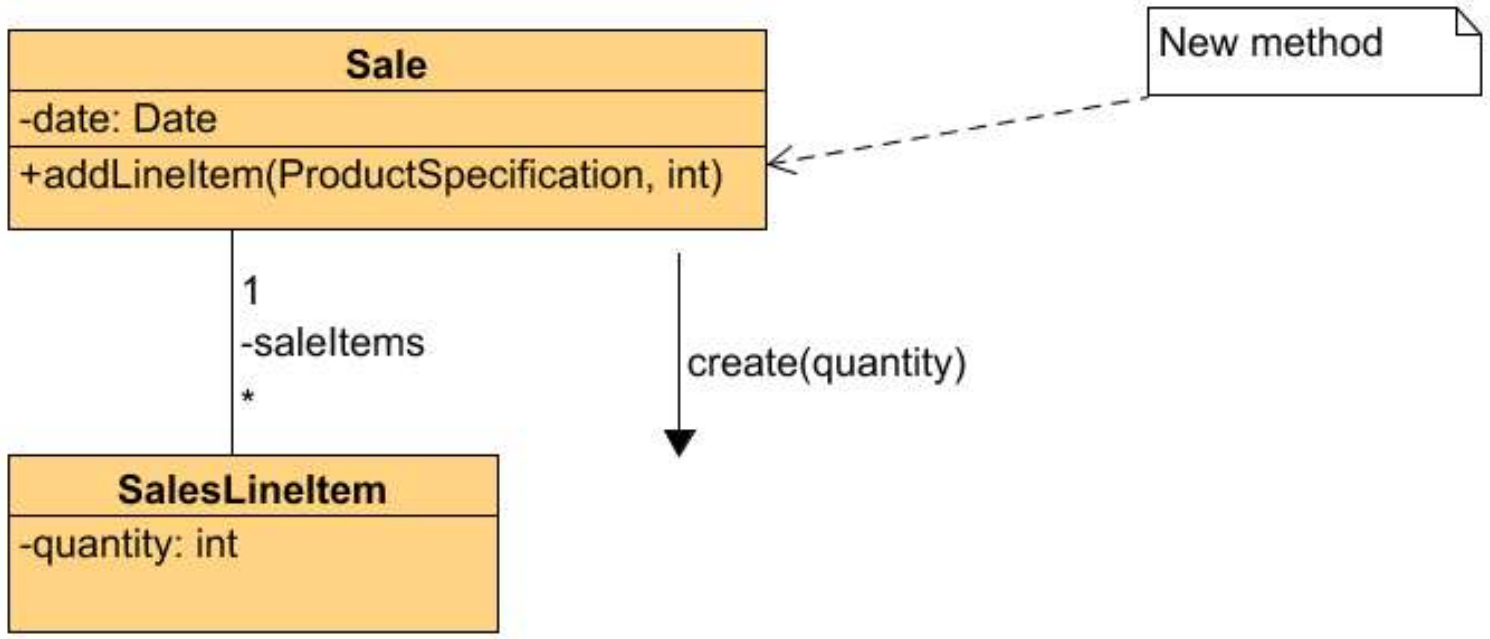
- Who is responsible for creating SalesLineItem objects, from an itemID and a quantity?
- Look for a class that aggregates or contains SalesLineItem objects.



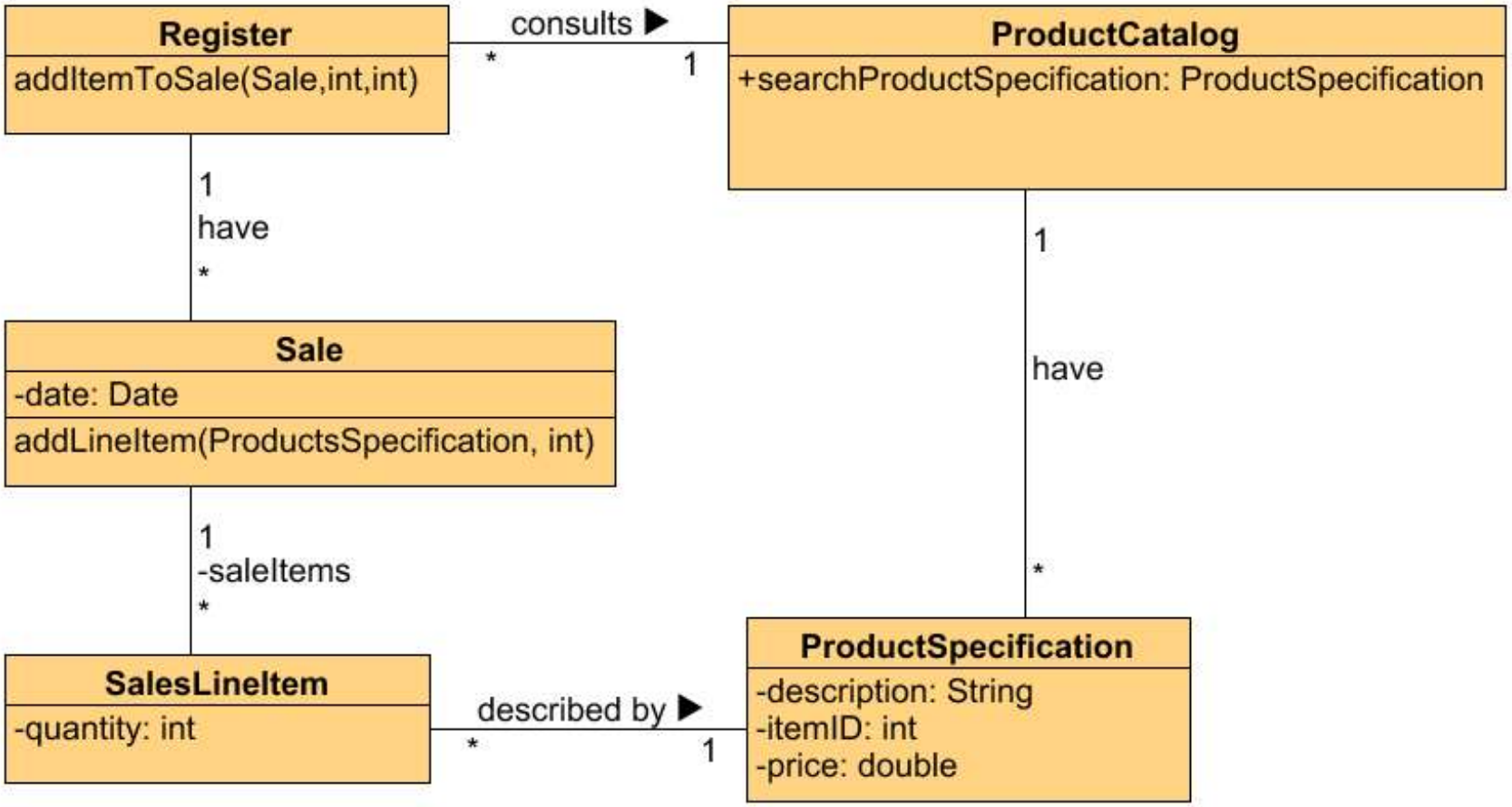
Creator: example

- Creator pattern suggests Sale.

- Collaboration diagram is



Creator: example

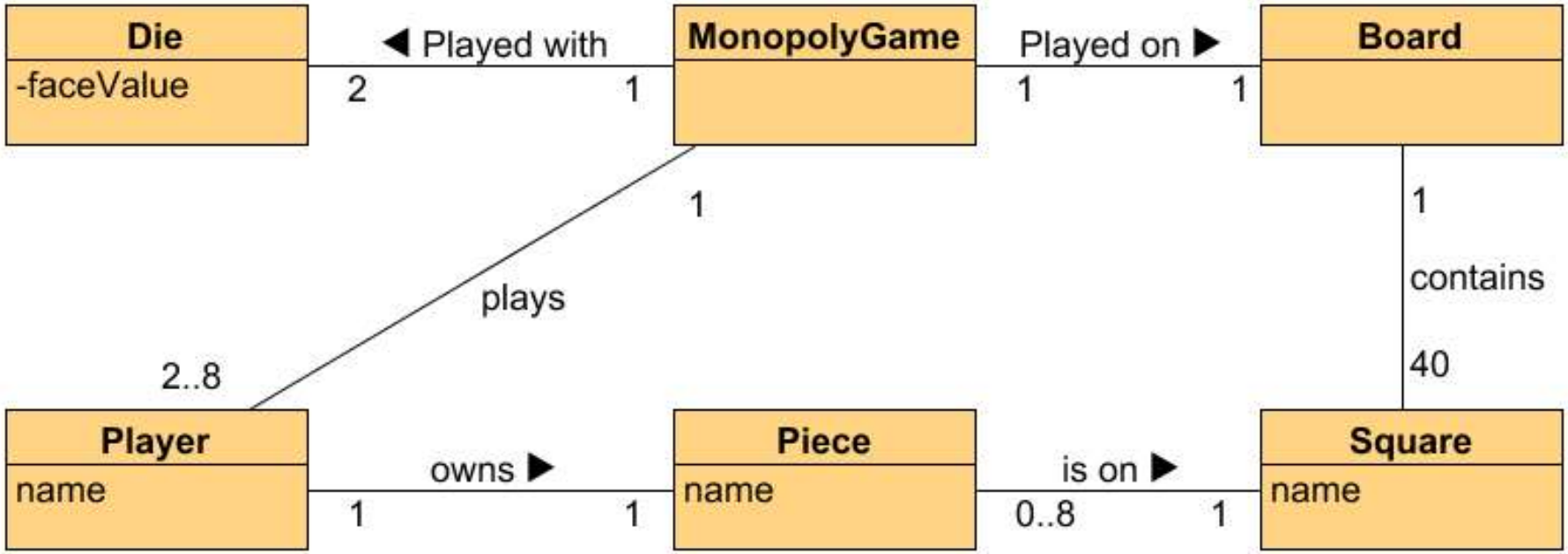


Creator: another example

- Who creates what?

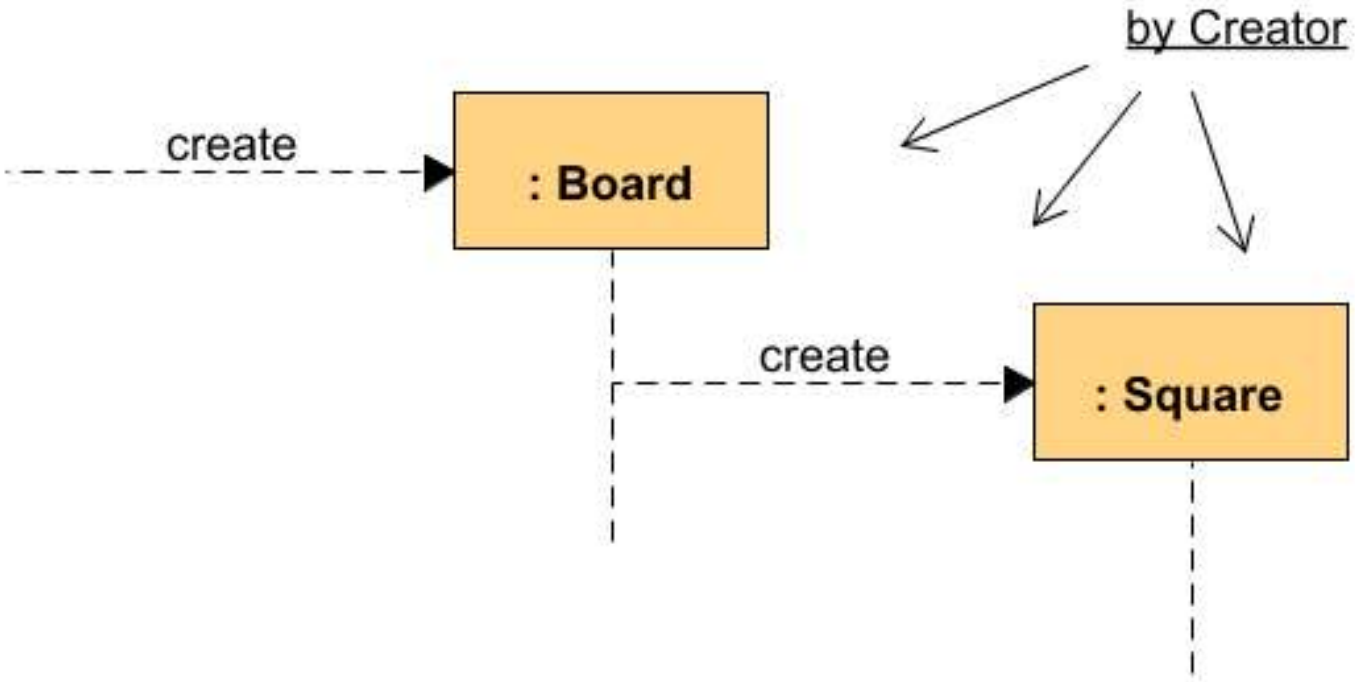
Creator: another example

- Who creates the Squares?



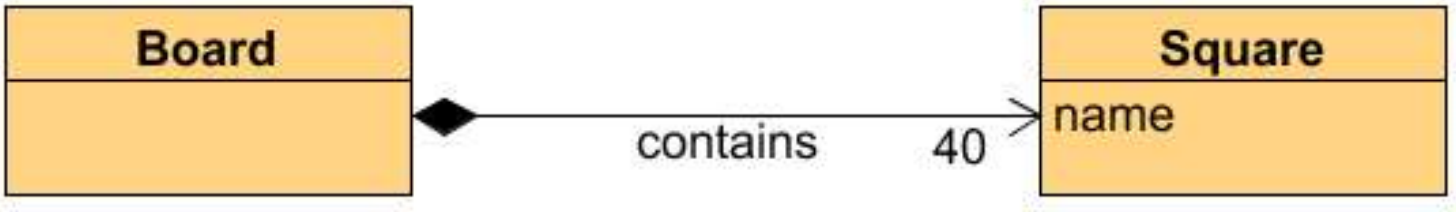
Creator pattern

- How does Create pattern lead to this partial Sequence diagram?



Creator pattern

- How does Create pattern develop this design class diagram?



- Board has a composite aggregation relationship with Square
  - I.e., Board contains a collection of Squares

Discussion of Creator pattern

- Promotes low coupling by making instances of a class responsible for creating objects they need to reference

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

Contraindications or caveats

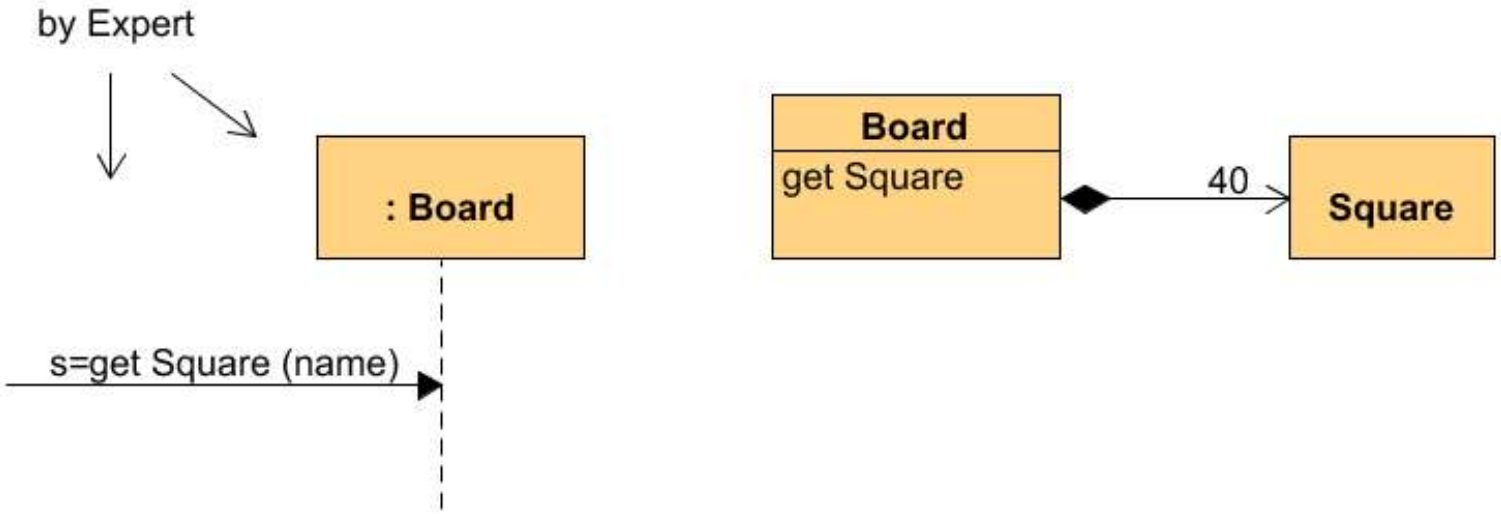
- Creation may require significant complexity:
  - recycling instances for performance reasons
  - conditionally creating instances from a family of similar classes
- In these instances, other patterns are available…
  - We'll learn about Factory and other patterns later…

GRASP

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

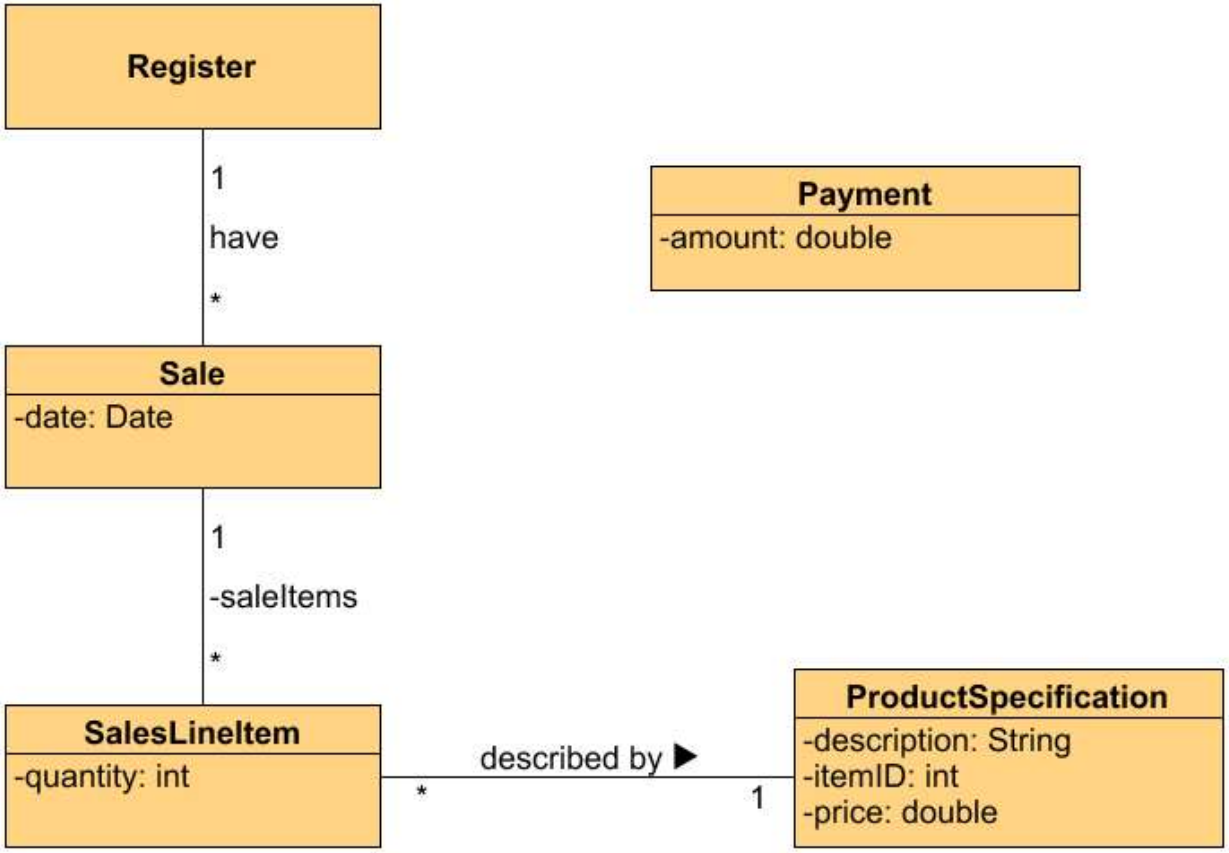
Information Expert principle

- 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*



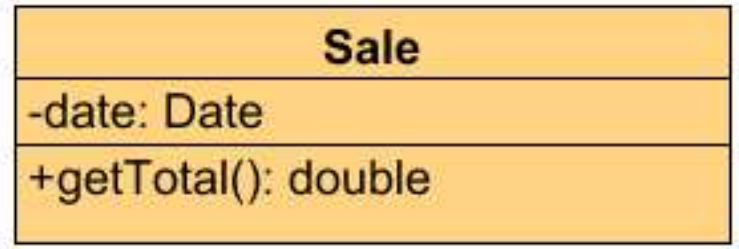
Information Expert: another example

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

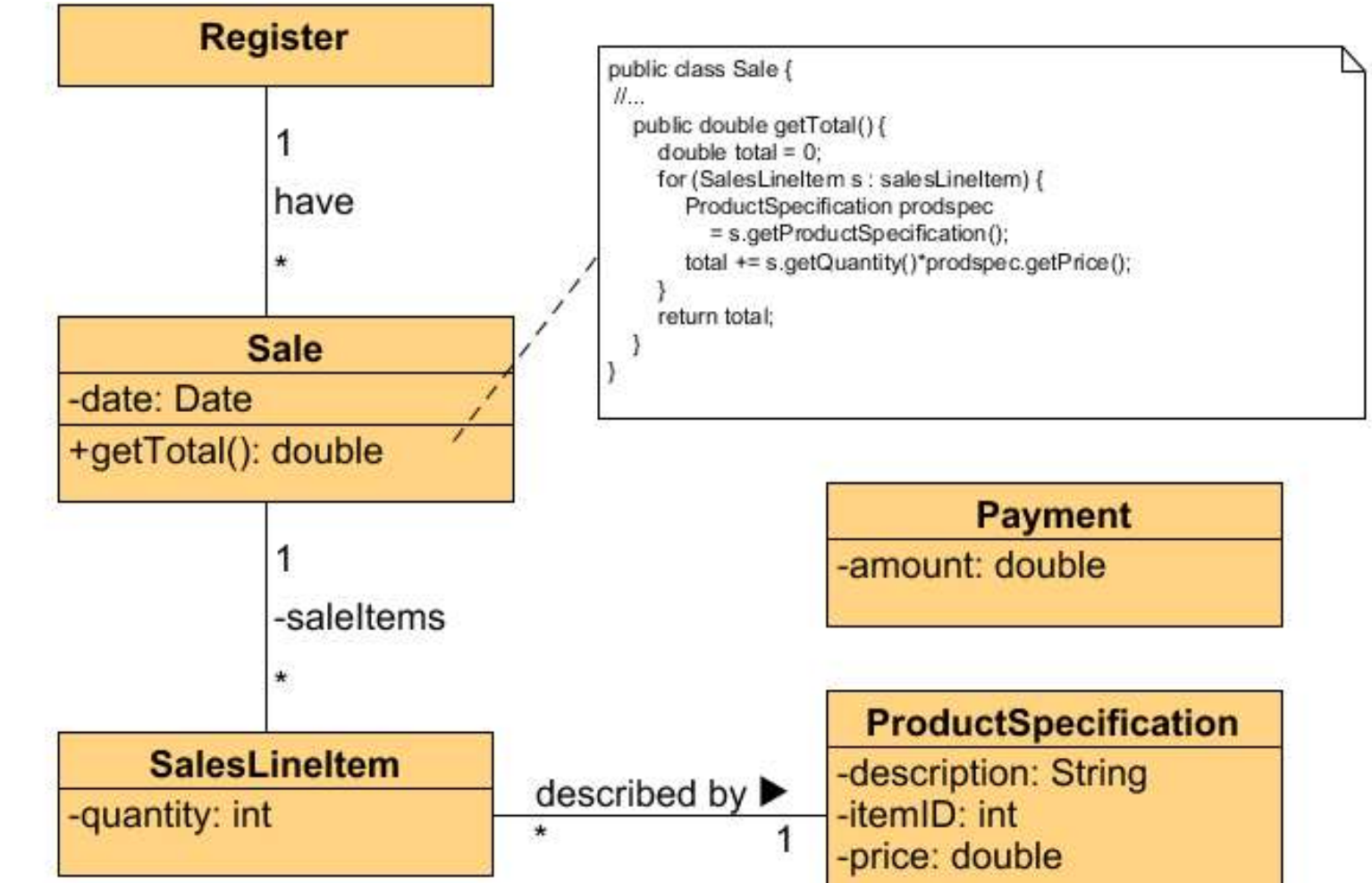


Information Expert: example

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



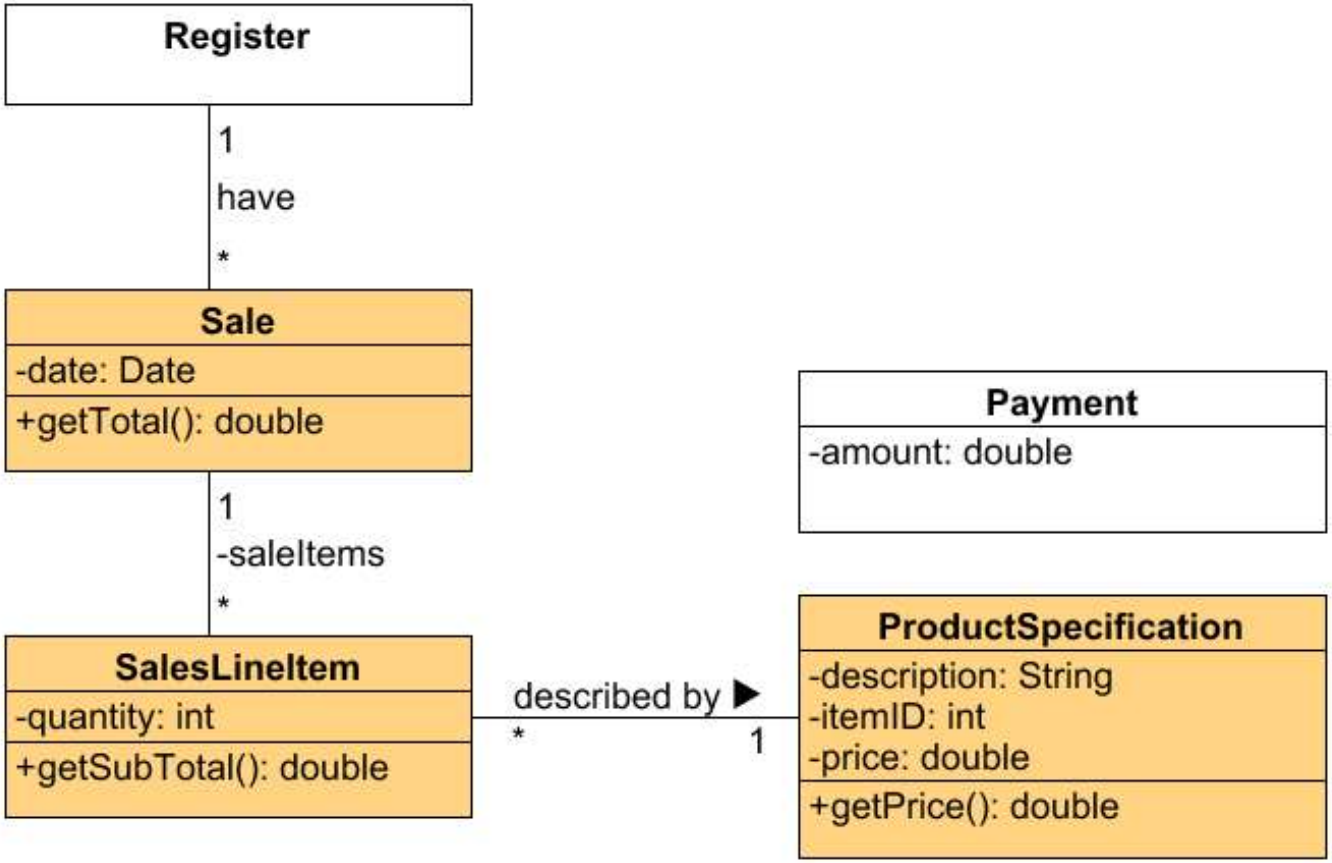
Information Expert: example



but … is this right?

Information Expert: example

- But subtotals are needed for each line item.
  - By Expert, *SalesLineItem* is expert, knows quantity and has association with *ProductSpecification* which knows price.



Information Expert: example

```
class Register {
   List<Sale> sales = new ArrayList<>();
   //...
   public void addItemToSale(Sale sale, int itemID, int quantity) {
        ProductSpecification prodSpec =
                ProductCatalog.searchproductSpecification(itemID);
     sale.addLineItem(prodSpec, quantity);
   }
}

class Sale {
   List<SalesLineItem> salesLineItem = new ArrayList<>();
   //...
   public void addLineItem(ProductSpecification prodSpec, int quantity) {
        salesLineItem.add(new SalesLineItem(prodSpec, quantity);
   }
}
```

Information Expert: example

- Hence responsibilities assign to the 3 classes.

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

- Fulfillment of a responsibility may require information spread across different classes, each expert on its own data.
  - Real world analogy: workers in a business, bureaucracy, military. "Don't do anything you can push off to someone else".

Benefits and Contraindications

- Facilitates information encapsulation
  - Classes use their own info to fulfill tasks - highly cohesive classes
  - Code easier to understand just by reading it
- Promotes low coupling
  - *Sale* doesn't depend on *ProductSpecification*

But:

- Can cause a class to become excessively complex
  - e.g. who is responsible to save *Sale* in a database? Sale is the information expert, but with this decision, then each class has its own services to save itself in a database.
  - This needs another kind of separation – domain and persistence

GRASP

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

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. Try to avoid one class to have to know about many others.

  - changes are localised
  - easier to understand
  - easier to reuse

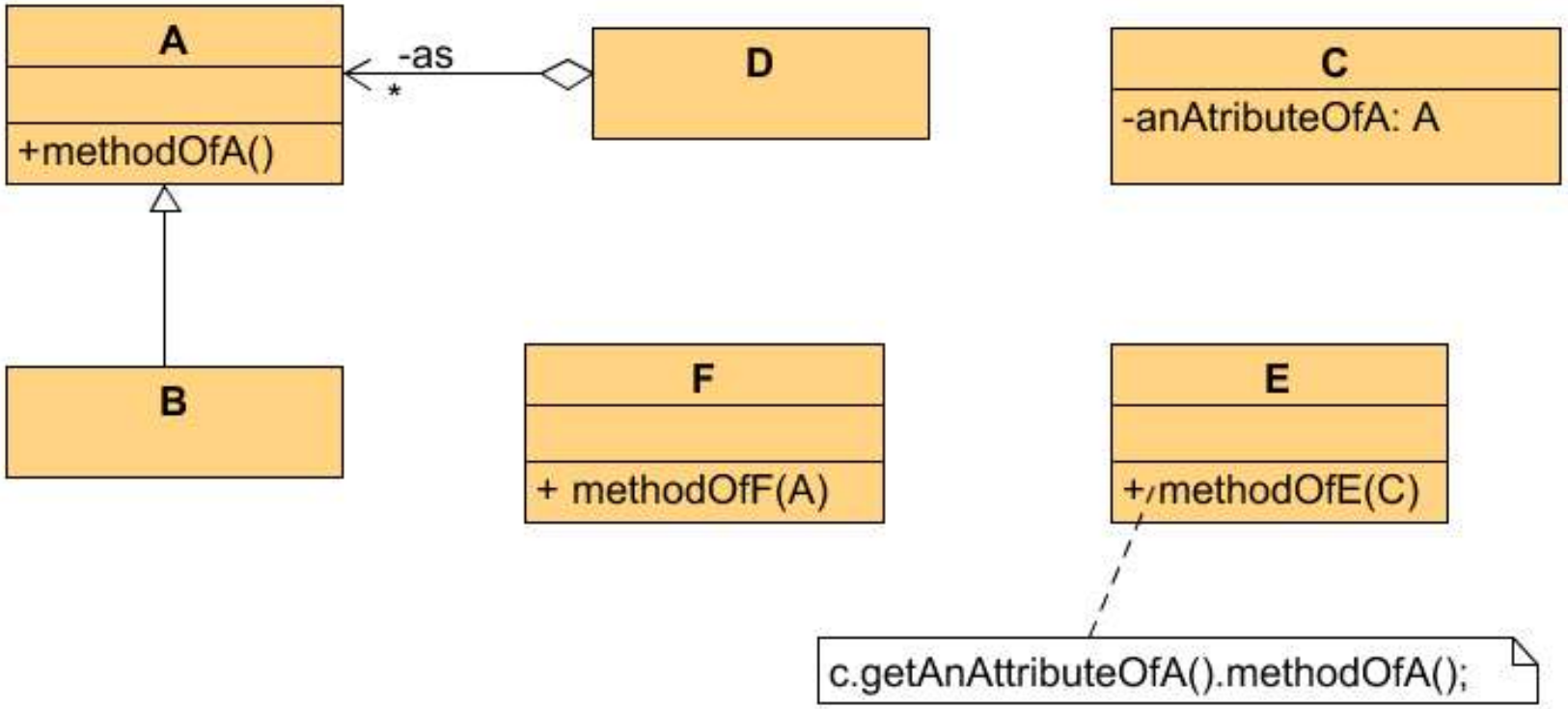
Low Coupling pattern

- Coupling measures of how strongly a class is connected, depends, relies on, or has knowledge of objects of other classes.
- Classes with strong coupling
  - suffer from changes in related classes
  - are harder to understand and maintain
  - are more difficult to reuse
- But coupling is necessary if we want classes to exchange messages!
  - The problem is too much of it and/or too unstable classes.

Entities coupling

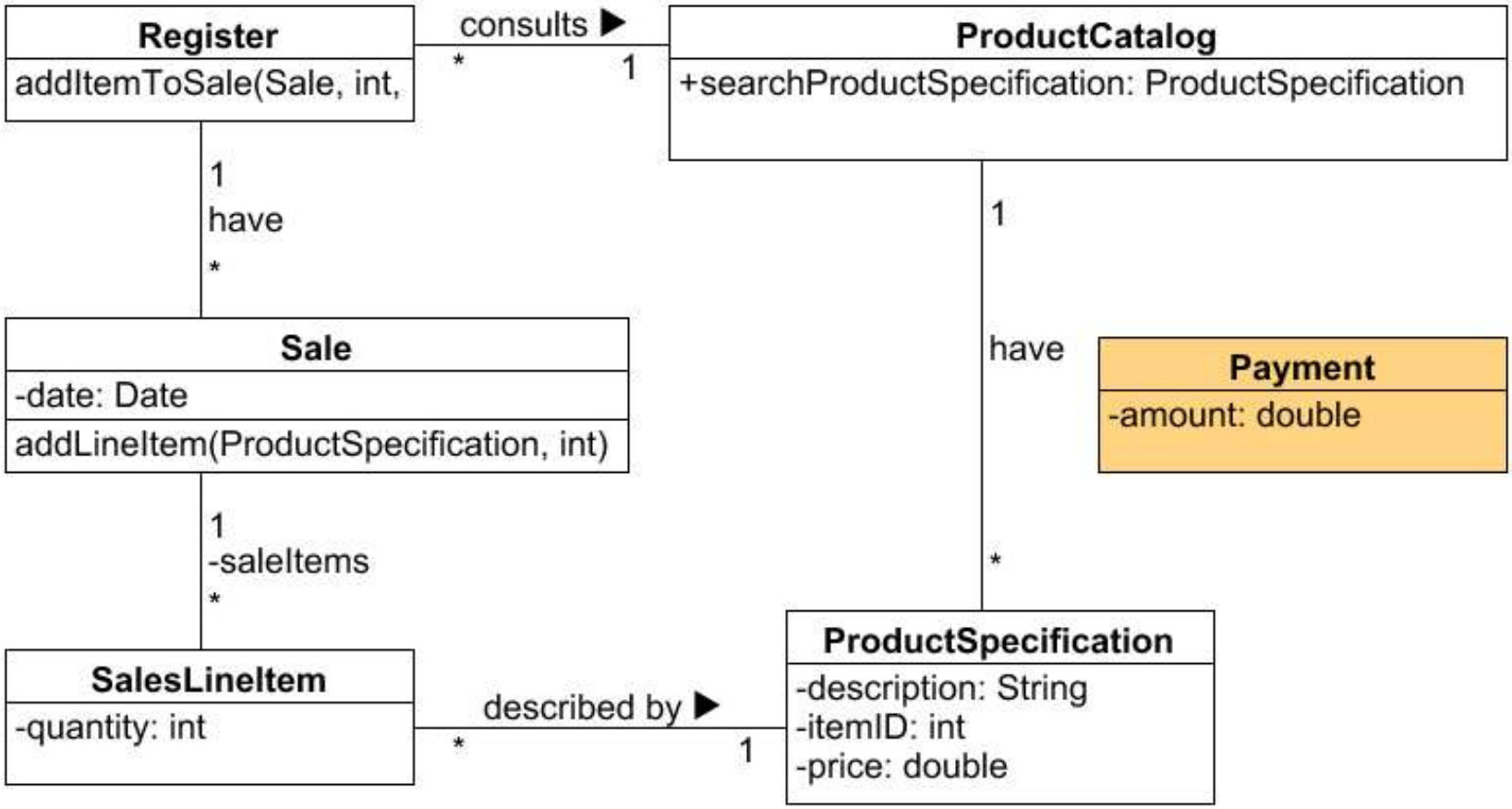
- In object-oriented languages, common forms of coupling from TypeX to TypeY include:
  - TypeX has an attribute (data member or instance variable) that refers to a TypeY instance, or TypeY itself.
  - TypeX has a method which references an instance of TypeY, or TypeY itself, by any means. These typically include a parameter or local variable of type TypeY, or the object returned from a message being an instance of TypeY.
  - TypeX is a direct or indirect subclass of TypeY.
  - TypeY is an interface, and TypeX implements that interface.

Entities coupling



Low Coupling: example

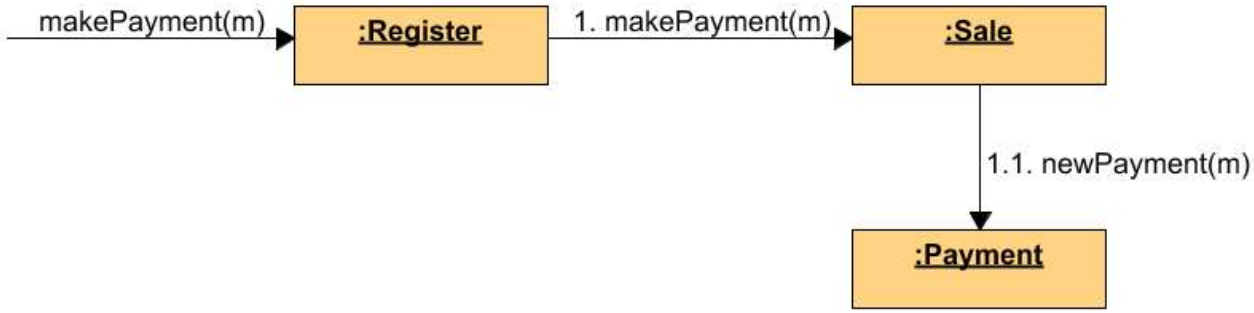
- Who has responsibility to create a payment?



Low Coupling: example

- Two possibilities:
  - Register

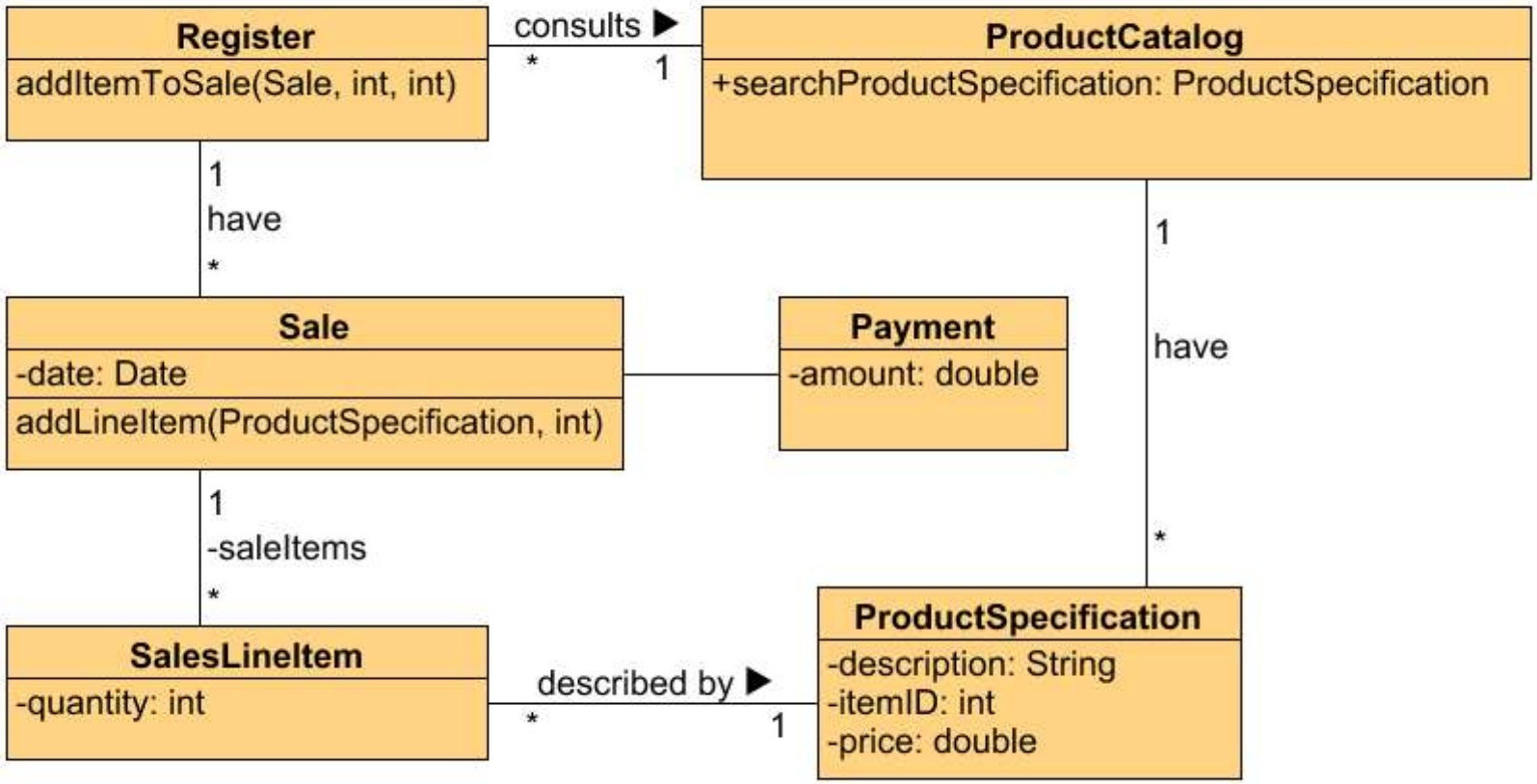
  - Sale



  - Low coupling suggests *Sale* because *Sale* must be coupled to *Payment* anyway (*Sale* knows its *total*).

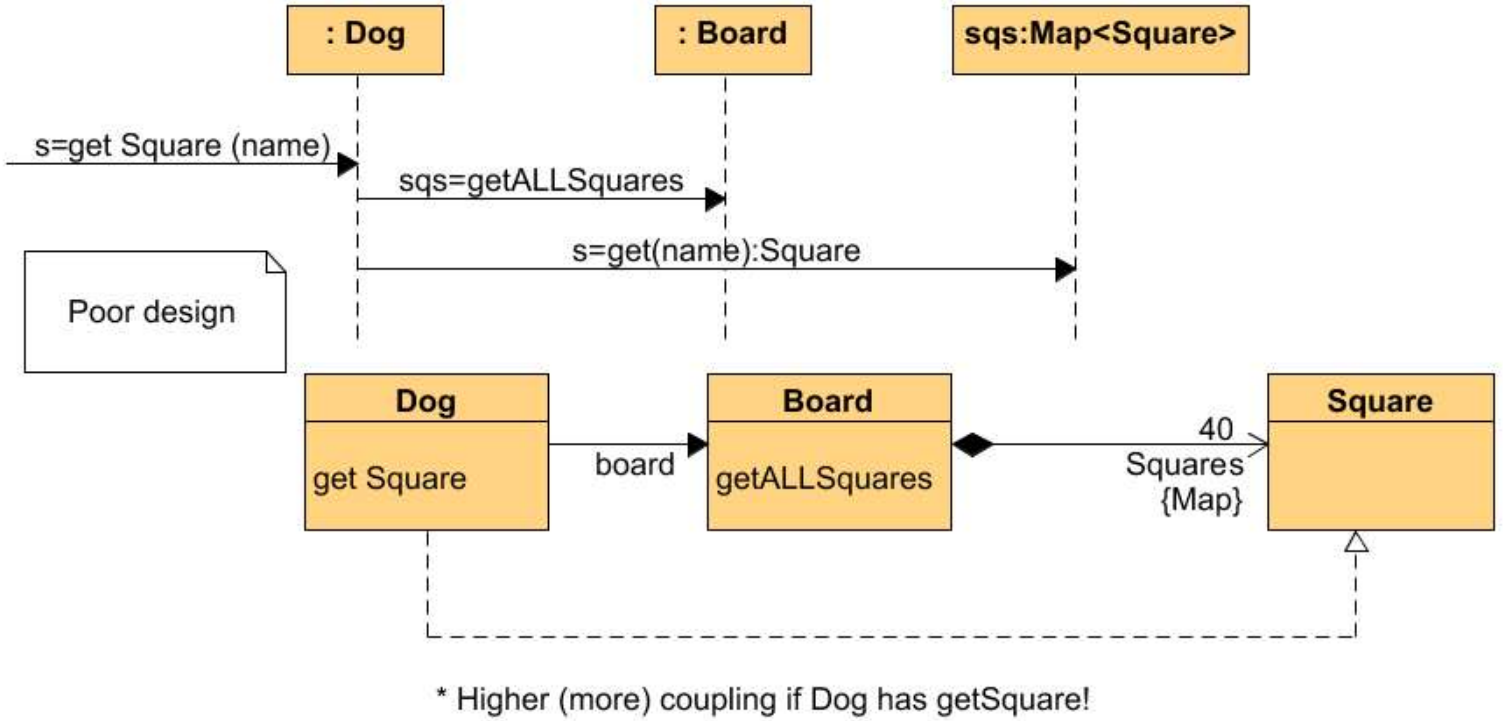
Low Coupling: example

- Who should own the method *getBalance()* that computes payment amount - total of sale?



Low Coupling: monopoly

- Why does the following design violate Low Coupling?



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

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:

- A higher coupling to stable classes is not a big issue
  - e.g., libraries and well-tested classes

GRASP

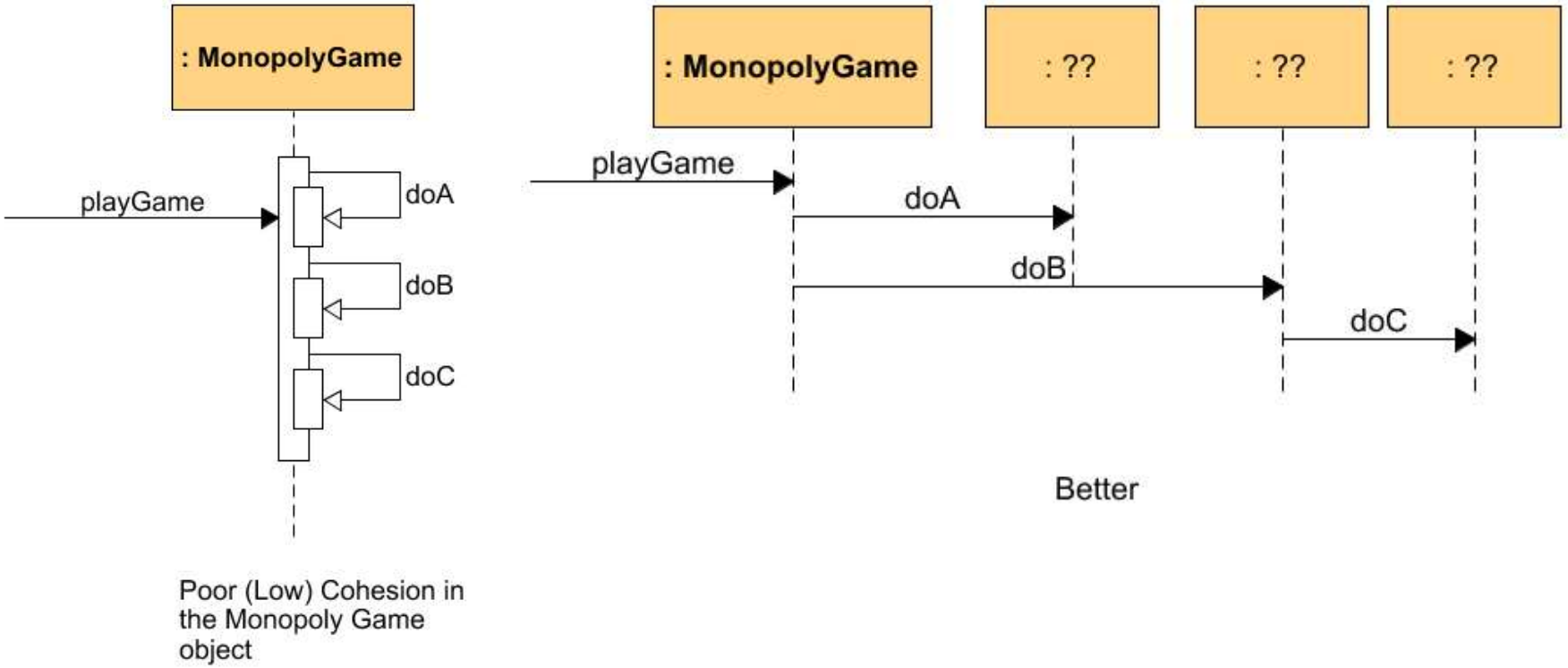
- Creator
- Information Expert
- Low Coupling
- High Cohesion
- Controller
- Polymorphism
- Pure Fabrication
- Indirection
- Protected Variations

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.

High Cohesion

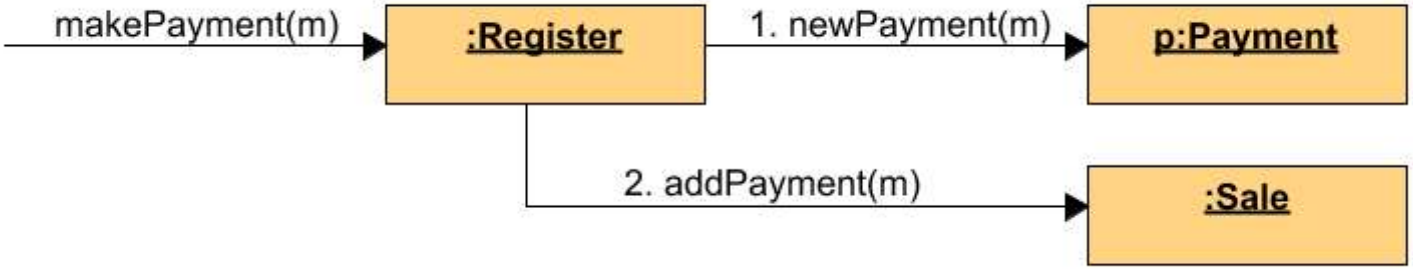
- How does the design on right promote high cohesion?



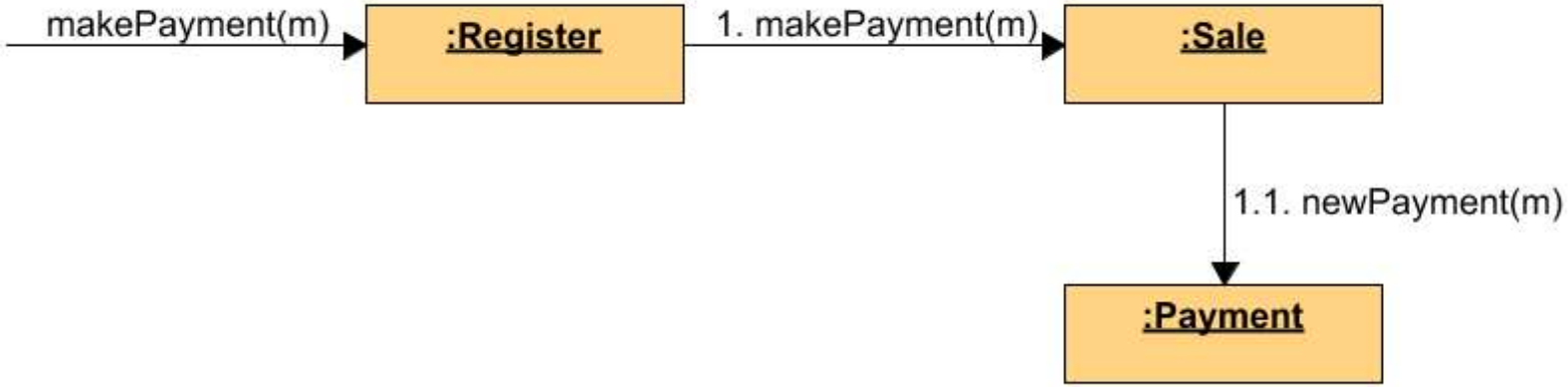
- Delegate responsibility & coordinate work

High Cohesion

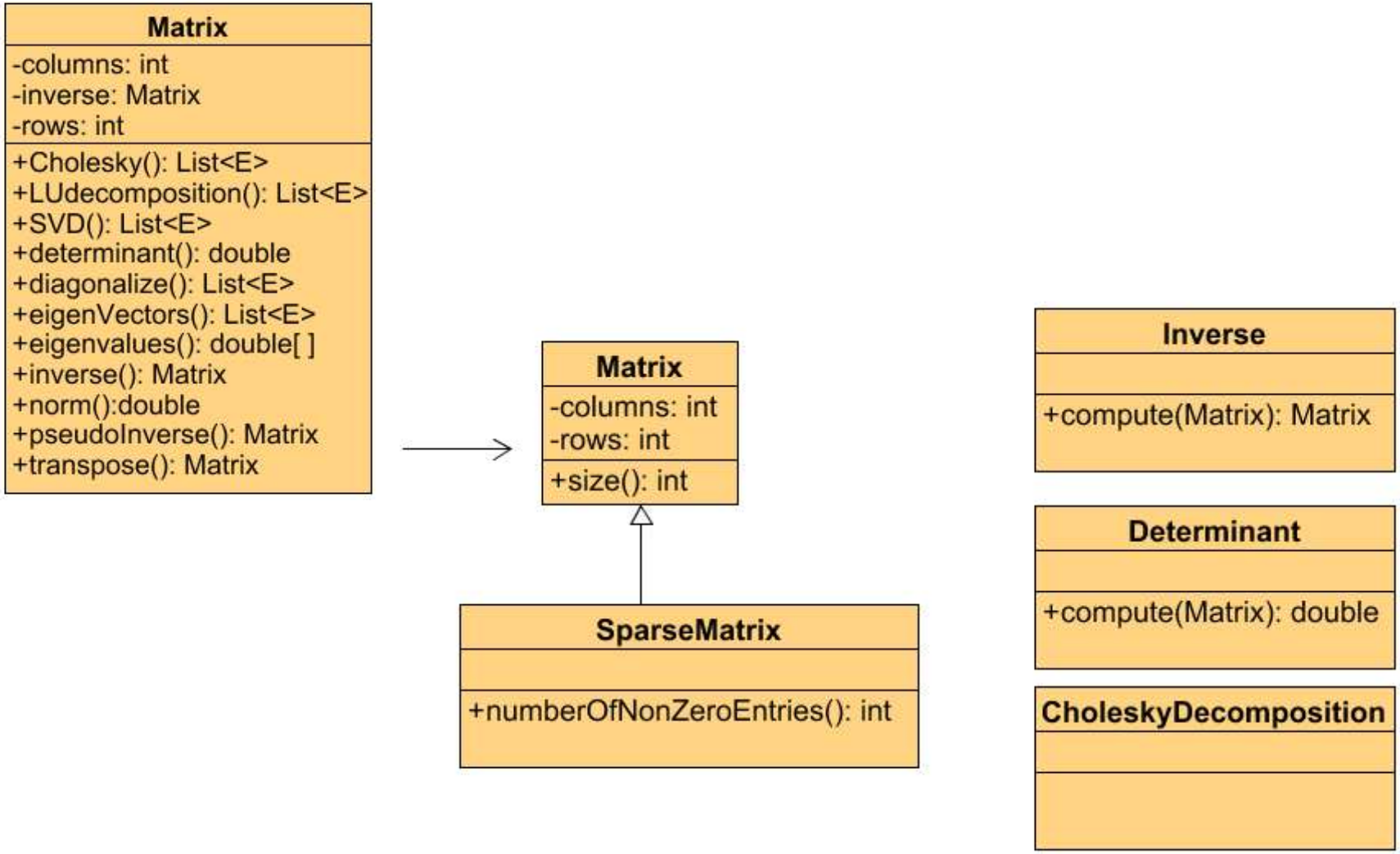
- *Register* would take on more and more responsibilities and become less cohesive.



- Giving responsibility to *Sale* supports higher cohesion in *Register*, as well as low coupling.

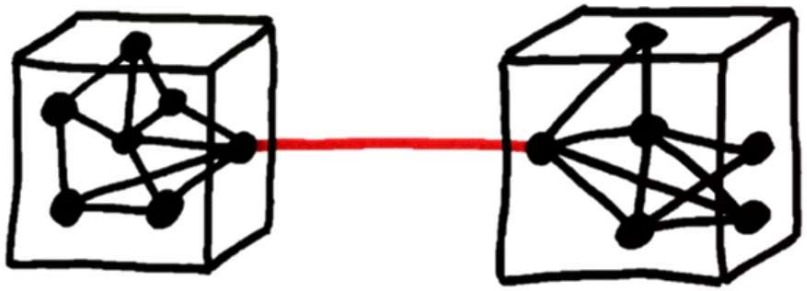


High Cohesion



Benefits & Contraindications

- Understandability, maintainability
- Complements Low Coupling



But:

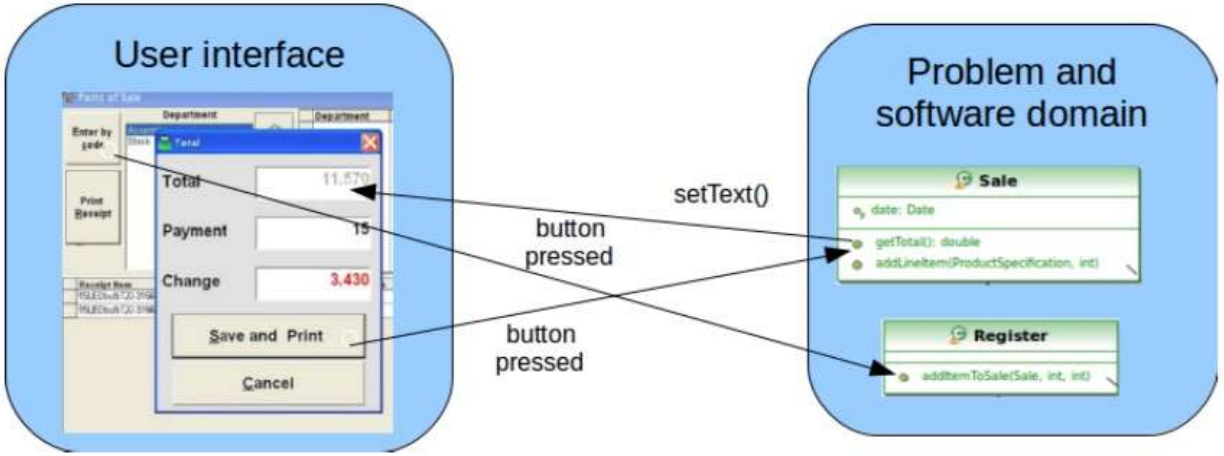
- Sometimes desirable to create less cohesive server objects
  - that provide an interface for many operations, due to performance needs associated with remote objects and remote communication

GRASP

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

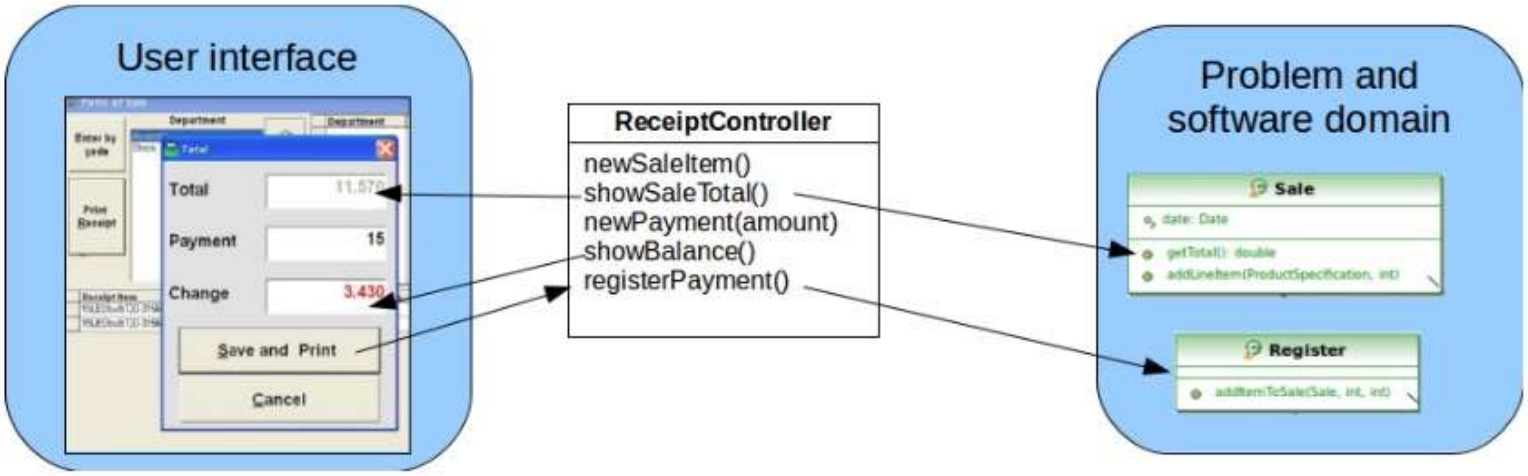
Controller

- Name: Controller
  - *(more on Model-View-Controller architecture)*
- Problem: Who should be responsible for UI events?



Controller

- Solution:
  - If a program receive events from external sources other than its GUI, add an event class to decouple the event source(s) from the objects that actually handle the events.

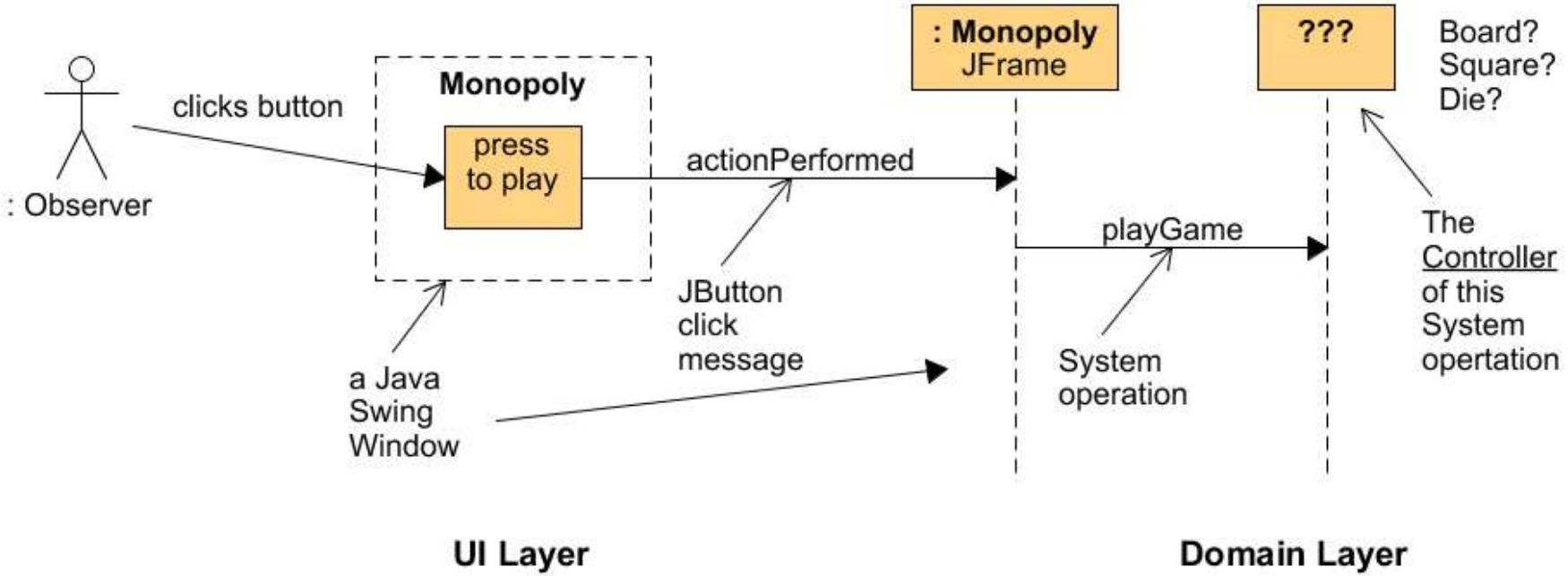


Controller

- Assign the responsibility for handling a system event message to a class representing one of these choices:
  1. The business or overall "system"(a façade controller).
  2. An artificial class, Pure Fabrication representing the use case (a use case controller).

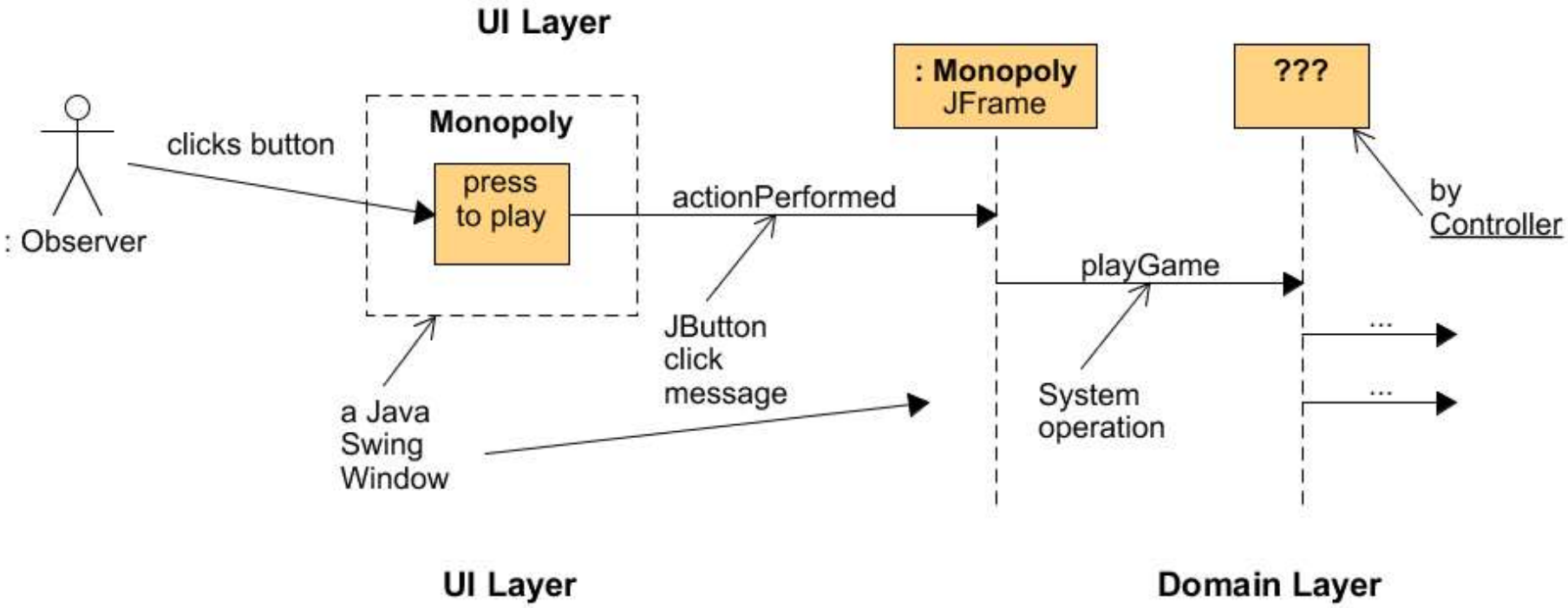
Controller

- Who is the controller of playGame operation?



Controller

- Separation between logical and UI views



Benefits & Contraindications

- Increased potential for reuse
  - Using a controller object keeps external event sources and internal event handlers independent of each other' s type and behaviour.
  - either the UI classes or the problem/software domain classes can change without affecting the other side.
- Controller just forwards
  - event handling requests
  - output requests
- Reason about the states of the use case
  - Ensure that the system operations occurs in legal sequence, or to be able to reason about the current state of activity and operations within the use case.

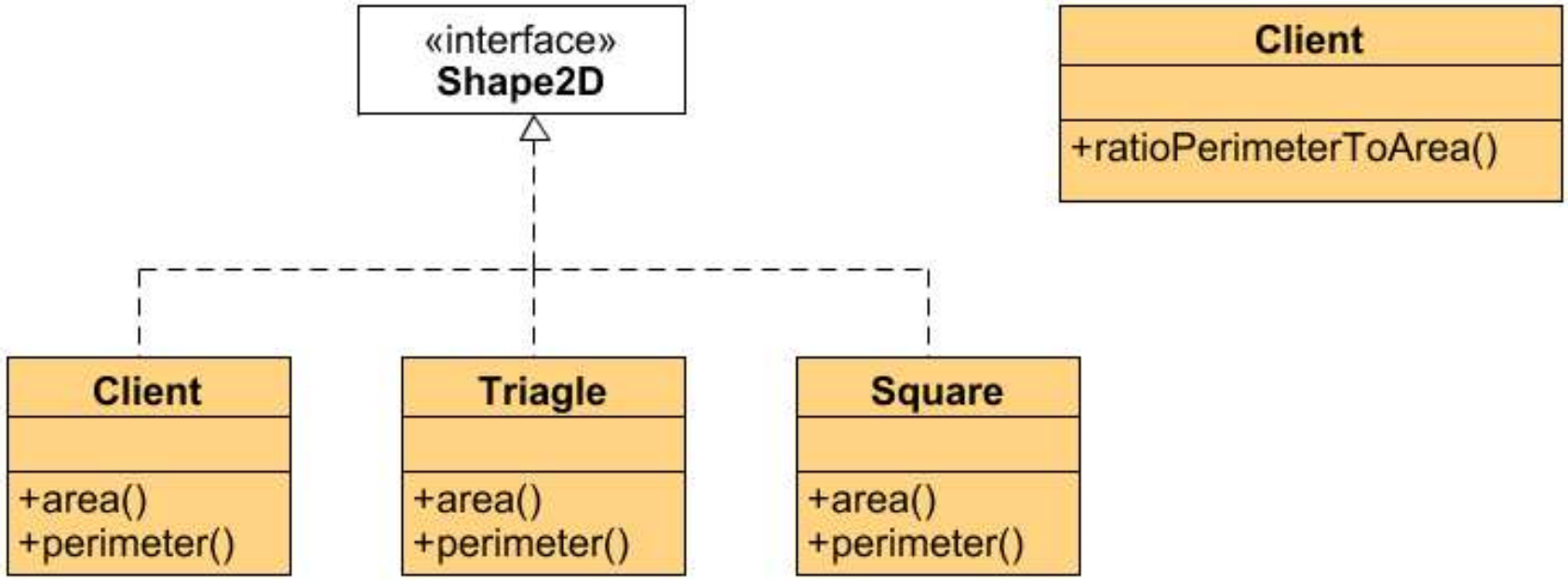
GRASP

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

Polymorphism

- Problem:
  - How to handle behavior based on type (i.e., class) but not with an if-then-else or switch statement involving the class name or a tag attribute?
- Solution:
  - When alternate behaviours are selected based on the type of an object, use polymorphic method call to select the behaviour, rather than using if statement to test the type.
  - Polymorphic methods: giving the same name to (different) services in different classes. Services are implemented by methods.

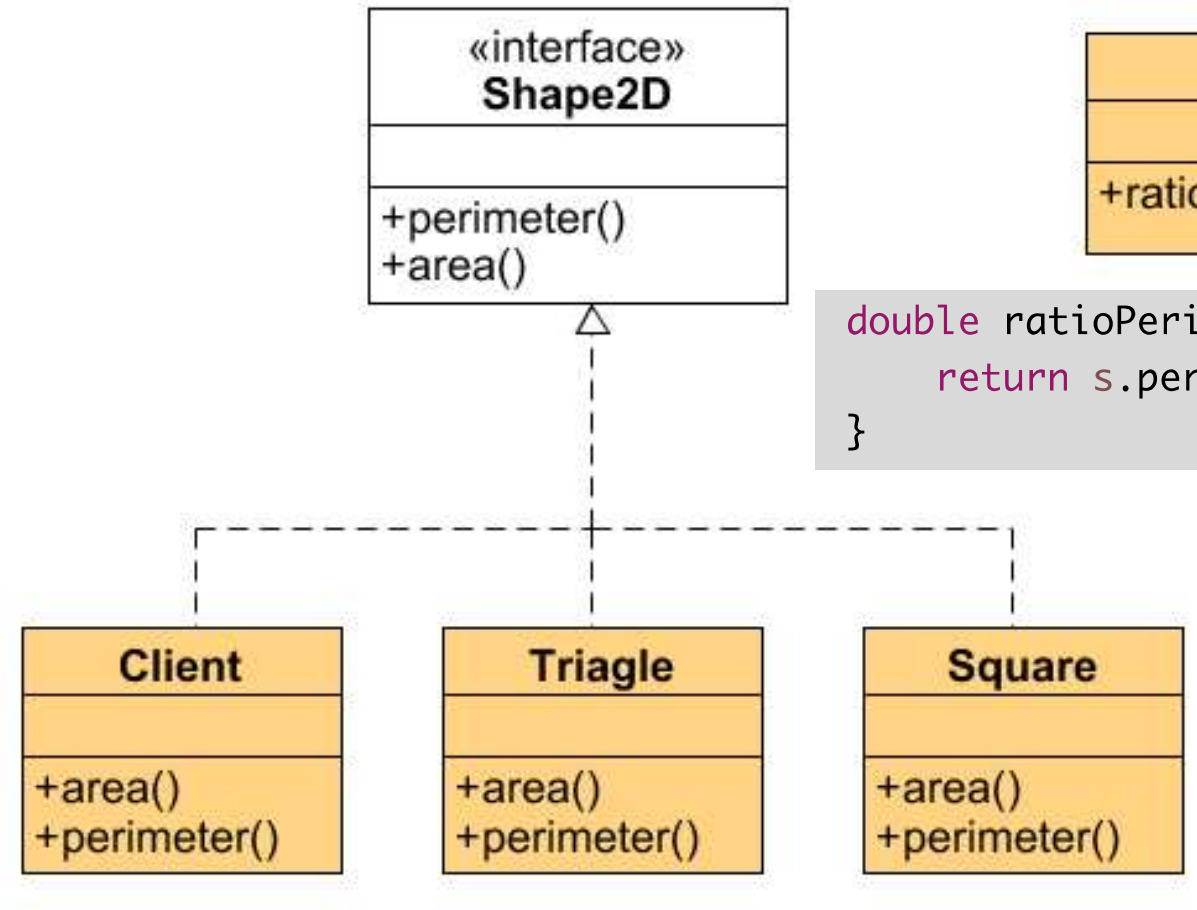
Example


«interface»
Shape2D
Client
+ratioPerimeterToArea()
Client
+area()
+perimeter()
Triagle
+area()
+perimeter()
Square
+area()
+perimeter()

Example

```
double ratioPerimeterToArea(Shape2D s) {
   double ratio = 0.0;
   if (s instanceof Triangle) {
     // or String name = s.getClass().getName();
     // if (name=="Triangle") {
        Triangle t = (Triangle) s;
        ratio = t.perimeter()/t.area();
    } else if (s instanceof Circle) {
        Circle c = (Circle) s;
        ratio = c.perimeter()/c.area();
    } else if (s instanceof Square) {
        Square sq = (Square) s;
        ratio = sq.perimeter()/sq.area();
    }
    return ratio;
}
```

Example - Polymorphism




```
Client
+ratioPerimeterToArea()
```

```
double ratioPerimeterToArea(Shape2D s) {
    return s.perimeter()/s.area();
}
```

Benefits & Contraindications

- Easier and more reliable than using explicit selection logic.
- Easier to add additional behaviours later on.

- If polymorphism is not used, and instead the code tests the type of the object, then that section of code will grow as more types are added to the system.
  - This section of code becomes more coupled (i.e., it knows about more types) and less cohesive (i.e., it is doing too much).

GRASP

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

Pure Fabrication

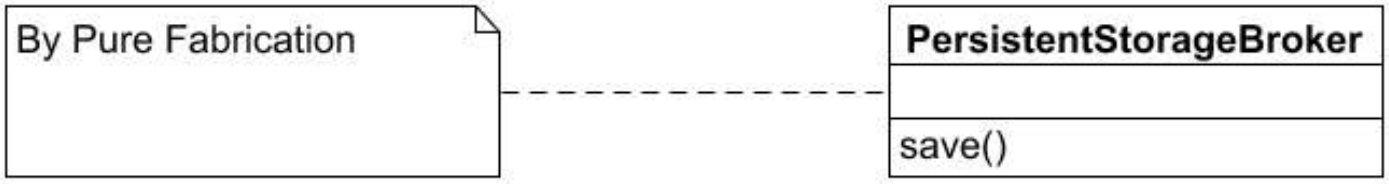
- Problem:
  - What object should have a responsibility when no class of the problem domain may take it without violating High Cohesion and Low Coupling?
  - Not all responsibilities fit into domain classes, like persistence, network communications, user interaction, etc.

- Solution:
  - Assign a highly cohesive set of responsibilities to an artificial class that does not represent anything in the problem domain.

Example

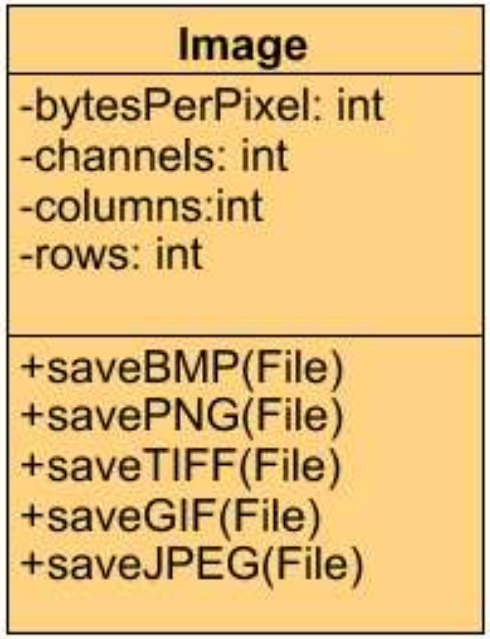
- Suppose, in the point of sale example, that we need to save Sale instances in a relational database.
  - By Expert, there is some justification to assign this responsibility to Sale class.
- However..
  - The task requires a relatively large number of supporting database-oriented operations and the Sale class becomes not cohesive.
  - The sale class has to be coupled to the relational database increasing its coupling.
  - Saving objects in a relational database is a very general task for which many classes need support. Placing these responsibilities in the Sale class suggests there is going to be poor reuse.

Pure Fabrication: example

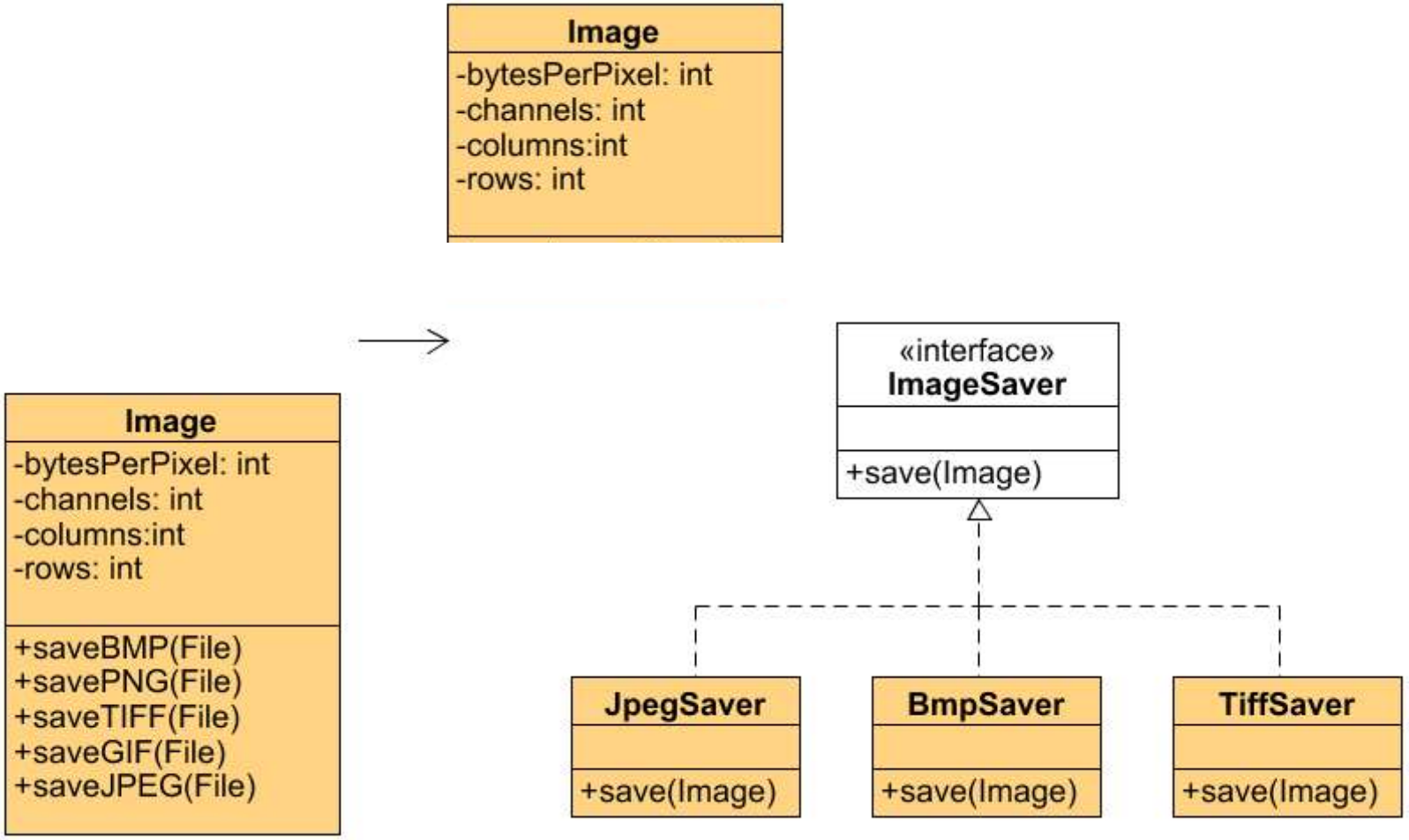


- The Sale remains well design, with high cohesion and low coupling
- The PersistentStorageBroker class is itself relatively cohesive
- The PersistentStorageBroker class is a very generic and reusable object

Pure Fabrication: Another example



Pure Fabrication: Another example



Benefits & Contraindications

- High cohesion is supported because responsibilities are factored into a class that only focuses on a very specific set of related tasks.

- Reuse potential may be increased because of the presence of fine grained Pure Fabrication classes.

GRASP

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

Indirection

- Problem:
  - How to avoid direct coupling?
  - How to de-couple objects so that Low coupling is supported, and reuse potential remains high?
- Solution:
  - Assign the responsibility to an intermediate object to mediate between other components or services, so that they are not directly coupled.

Indirection

- Indirection can be categorized into the following main groups:
  - Behavioural Extension.
  - Interface Modification.
  - Technology Encapsulation.
  - Complexity Encapsulation.

Example: PersistentStorageBroker

- The Pure fabrication example
  - de-coupling the Sale from the relational database services through the introduction of a PersistentStorageBroker is also an example of assigning responsibilities to support Indirection.
  - The PersistentStorageBroker acts as a intermediary between the Sale and database

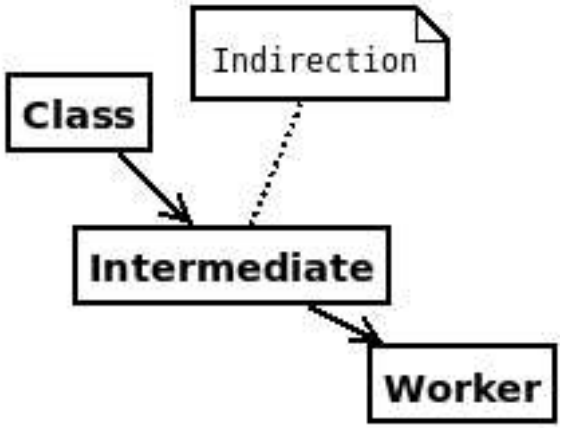
Indirection: example

- Assume that :
  - A point-of-sale terminal application needs to setup a specific communication channel in order to transmit credit payment request
  - The operating system provides a low-level function call API for doing so.

- How to?
  - A class called CreditAuthorizationService is responsible for talking to the communication equipment

Benefits & Contraindications

- Low coupling
- Promotes reusability



GRASP

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

Protected Variations

- Problem:
  - How to design objects, subsystems, and systems so that variations or instabilities in some elements do not have an undesirable impact on other elements?
- Solution:
  - Identify points of predicted variation or instability, assign responsibilities to create a stable interface around them.

- Protected Variations is a fundamental design principle which is the foundation for many design patterns

Protected Variations

- Mechanisms motivated by Protected Variations:
  - Core PV mechanisms: data encapsulation, interfaces, polymorphism, indirection, standards
  - Data-driven designs: style sheets, property files, other mechanisms for reading in configuration data at run time
  - Service lookup including naming services (Java's JNDI) or traders (Java's Jini or UDDI for web services)
  - Interpreter-driven designs
  - Reflective or meta-level designs
  - Uniform access – language support for uniform access to methods and data
  - Liskov Substitution Principle (LSP) - *more following*
  - Structure-hiding designs (Law of Demeter – "don't talk to strangers") - *more following*

Liskov Substitution Principle (LSP)

- Due to Barbara Liskov, Turing Award 2008
- LSP: a subclass B of A should be substitutable for superclass A, i.e., B should be a true subtype of A
- Reasoning at the specification level
  - B should not remove methods from A
  - For each B.m, which "substitutes" A.m, B.m's specification is stronger than A.m's specification
  - Client: A a; … a.m(int x,int y);
  - Call a.m can bind to B's m and B's m should not surprise client

Classic Example

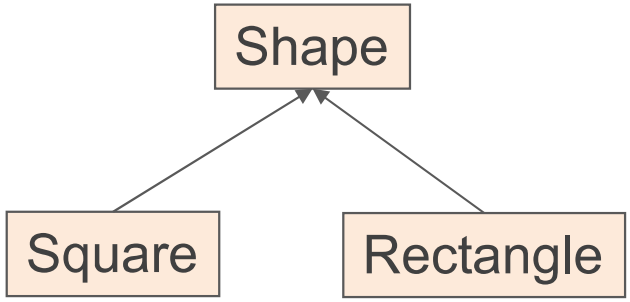
- Every Square is-a Rectangle?

- Thus,
  - class Square extends Rectangle { … }

- But is a Square a true subtype of Rectangle?
  - In other words, is Square substitutable for Rectangle in clients expecting a Rectangle?

Every Square is-a Rectangle?

- Square is not a true subtype of Rectangle
  - Rectangles are expected to have height and width that can change independently
  - Squares violate that expectation. Surprise clients
- And the opposite? Is Rectangle a true subtype of Square?
  - No. Squares are expected to have equal height and width. Rectangles violate this expectation
- One solution:
  - make them unrelated



Law of Demeter (Don't talk to strangers)

- Problem:
  - How to avoid knowing about the structure of indirect objects?
- Solution:
  - If two classes have no other reason to be directly aware of each other or otherwise coupled, then the two classes should not directly interact.
    - e.g., in A don't do getB().getC().methodOfC()
  - Within a method, messages should only be sent to the following objects:
    - The this object (or self)
    - A parameter of the method
    - An attribute of self
    - An element of a collection which is an attribute of self
    - An object created within the method

Law of Demeter: Example

```
class Company {
    Collection<Department> departments = new ArrayList<>();
}
class Department {
    private Employee manager;
    public Employee getManager() {
        return manager;
}
class Employee {
    private double salary;
    public double getSalary() {
        return salary;

    }
}
```

Now *Company* needs to have the total of amount spend with Managers' salary. How?

Law of Demeter: Example

- Don't:

```
// within Company
for (Department dept : departments) {
     System.out.println( dept.getManager().getSalary() );
     // now Company depends on Employee
}
```

- Do:

```
class Department { //...
    double getManagerSalary() {
        return getManager().getSalary();
    }
}
// within Company
for (Department dept : departments) {
    System.out.println( dept.getManagerSalary() );
}
```

Benefits & Contraindications

- Keeps coupling between classes low and makes a design more robust
- Adds a small amount of overhead in the form of indirect method calls

Others - SOLID principles

- Single responsibility
  - "*every class should have a single responsibility, and that responsibility should be entirely encapsulated by the class*" (Robert Martin)
- Open/closed (OCP)
  - "*software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification*" (Bertrand Meyer)
- Liskov substitution (LSP)
- Interface segregation
  - "*no client should be forced to depend on methods it does not use*" (similar to High Cohesion of GRASP)
- Dependency inversion
  - "High-level modules should not depend on low-level modules. Both should depend on abstractions"

Others principles / jargons

- Minimalism
  - Keep it simple, stupid (KISS)
  - Worse is better (Less is more)
  - You aren't gonna need it (YAGNI)
  - Principle of good enough (POGE)
  - Quick-and-dirty
- Don't repeat yourself (DRY)
  - *Cut and paste of code is evil.*
- Inversion of control (IoC)

- … and many others

Summary

- Skillful assignment of responsibilities is extremely important in object-oriented design
- Patterns are named problem/solution pairs that codify good advice and principles related to assignment of responsibilities
- GRASP identifies several principles:
  - Creator, Information Expert, Controller, Low Coupling, High Cohesion, Polymorphism, Pure Fabrication, Indirection, Protected Variations