So Far …

Part 1: OOAD Intro

Part 2: Inception

Part 3: Elaboration—Iteration 1

- Iteration 1—Basics
- Domain Models
- System Sequence Diagrams
- Operation Contracts
- Requirements to Design—Iteratively
- Logical Architecture and UML Package Diagrams
- On to Object Design
- UML Interaction Diagrams (Self Study)
- UML Class Diagrams (Self Study)
- GRASP: Designing Objects with Responsibilities
- Object Design Examples with GRASP
- Designing for Visibility
- Mapping Designs to Code

GRASP —Designing Objects with Responsibilities

Abdulkareem Alali

Ack Dale Haverstock

Based on Larman's Applying UML and Patterns Book, 3d

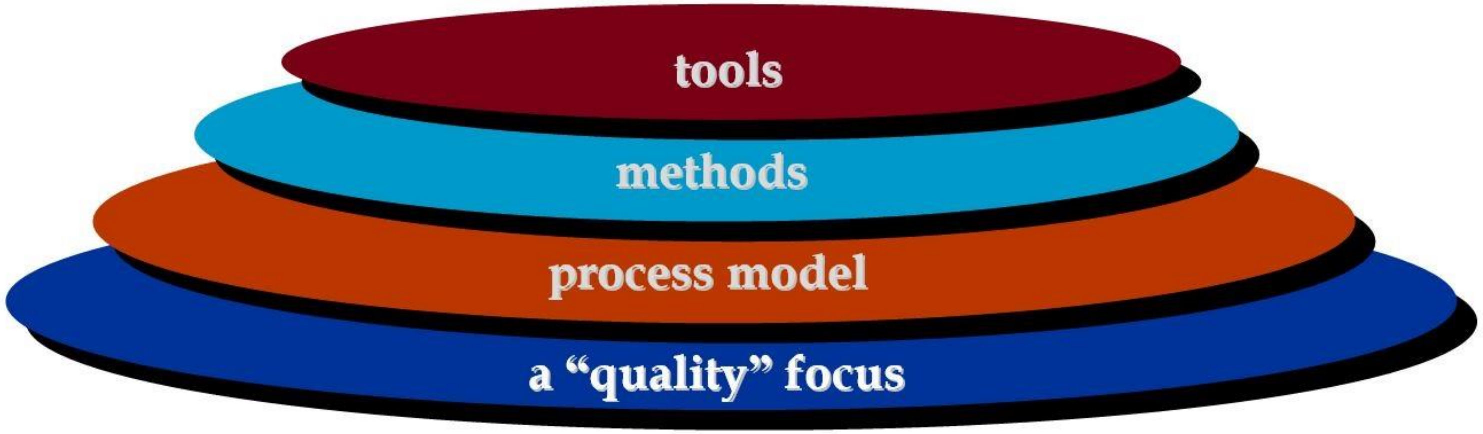
Understanding responsibilities is key to good object-oriented design

—Martin Fowler

The critical design tool for software development is a mind well educated in design principles. It is not the UML or any other technology

—Craig Larman

Layered Technology (Pressman)



Designing Objects with Responsibilities

Focus on **OOD**. Consider this:

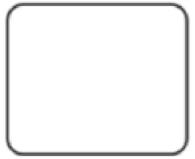
1. After identifying your requirements and
2. creating a domain model,
3. then add methods to the appropriate classes (**Responsibility**),
4. and define the messaging between the objects to fulfill the requirements (**Collaboration**)

Too vague!

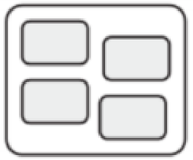
Good guidelines are needed. **GRASP, General Responsibility Assignment Software Patterns**.

ZOOM-OUT,-IN —What’s Software Design?

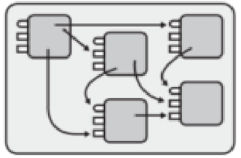
Design is **needed** at several different levels of detail in a system:



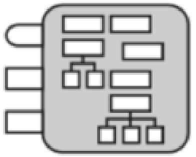
System



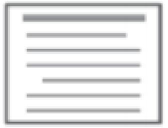
Subsystems or packages: user interface, data storage, application-level classes, graphics



Classes within Packages, class relationships, interface of each class: public methods



Attributes, private methods, inner classes . . .



Source code implementing methods

How to Design Object-Oriented?

Responsibility Assignment:

Deciding what methods **belong** where,

and how the **objects** should **interact**, and their **Role**

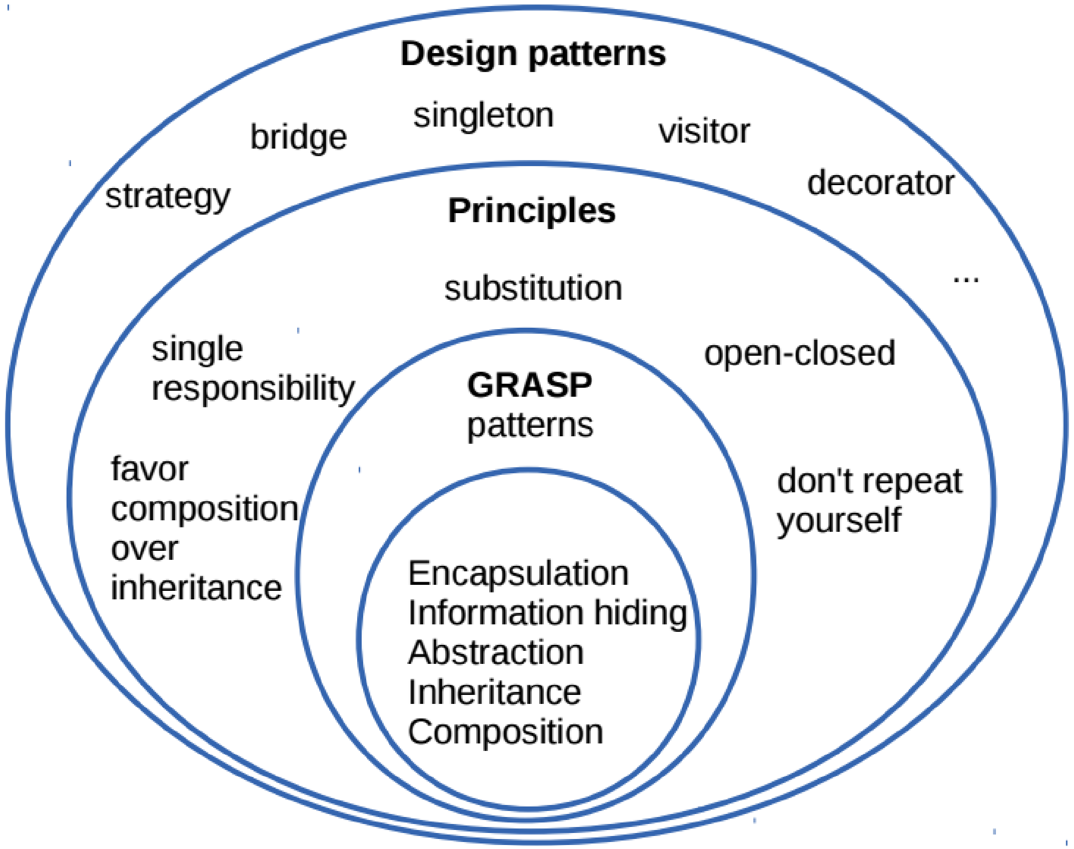
is terribly important and *anything but trivial*

How to Design Object-Oriented?

There's **no methodology** to get the best object-oriented design,

but there are

Principles, Patterns, Best Practices, and **heuristics**.



Why Follow Principles And Patterns? Empirical!

- They are best practices found after decades of experience by many developers
- To build (more) **change resistant designs**
- Learn [to design objects] from the *successes of others*, not from their failures

Read Symptoms of rotting design: Rigidity, fragility, immobility, viscosity in the article “Design Principles and Design Patterns by Robert C. Martin, 2000” .

Why Follow Principles And Patterns? A Story!

- *Design of many software applications begins as a **vital** image in the minds of its designers, at this stage it is **clean**, **elegant**, and **compelling***
- *But then something begins to **happen**, software starts to **rot***
- *At first it isn't so bad. An ugly wart here, a clumsy **hack** there. Yet, over time as the rotting **continues**,*
- *The ugly festering sores and boils **accumulate** until they **dominate** the design of the application*
- *The program becomes a **festering mass of code** that the developers find increasingly **hard to maintain***
- *Eventually, the sheer effort required to make even the **simplest of changes to the application becomes so high** that the engineers and front line managers **cry** for a **redesign** project*

Why Follow Principles And Patterns?

*What kind of **changes** cause designs to **rot**?*

*Changes that **introduce new and unplanned dependencies***

Each of the four symptoms rigidity, fragility, immobility, viscosity

*(**resistance to change**) is caused by **improper dependencies** between the software modules.*

*(…) the **dependencies** between modules in an application must be **managed**. This management consists of the creation of dependency **firewalls**. Across such firewalls, dependencies do **not propagate***

Object Oriented Design principles build such firewalls and manage module dependencies.

—R.C. Martin, 2000 .

POS, What Might We Have At This Point? Elaboration in Prog.

2-day requirements workshop is finished

Chief architect and business agree to implement and test some scenarios of “Process Sale” in the first three-week time-boxed iteration

3/20 (10%-20%) fully-dressed use cases-architecturally significant and of high business value (e.g. Process Sale) before starting to program

POS, What Might We Have At This Point? Elaboration in Prog.

Other artifacts have been started

Proof-of-concept: Programming experiments have resolved the show-stopper technical questions

- e.g. Java Swing UI will work on a touch screen

The chief architect has drawn some ideas for the large-scale logical architecture, using UML package diagrams

POS
—Recall Main Features

- Application for a shop, restaurant, etc. that registers sales
- Each sale is one or more items of one or more product types, at a certain date
- A product has a description, unitary price and identifier
- The application also registers payments associated to sales
- A payment is for a certain amount, equal or greater that the total of the sale

Point of Sale (POS)

OO Design Inputs

Use Cases

- Visible behavior that the software objects must ultimately support
- Objects are designed to "**realize**" (implement) the use cases

Supplementary Specification

- non-functional goals

System Sequence Diagrams (SSDs)

- System operation messages: **starting** messages on our interaction diagrams of collaborating objects

OO Design Inputs

Glossary/Data Dictionary

- UI layer parameters/data
- Database data
- Detailed item-specific logic or validation requirements
  - e.g. Legal formats and validation for product UPCs (universal product codes)

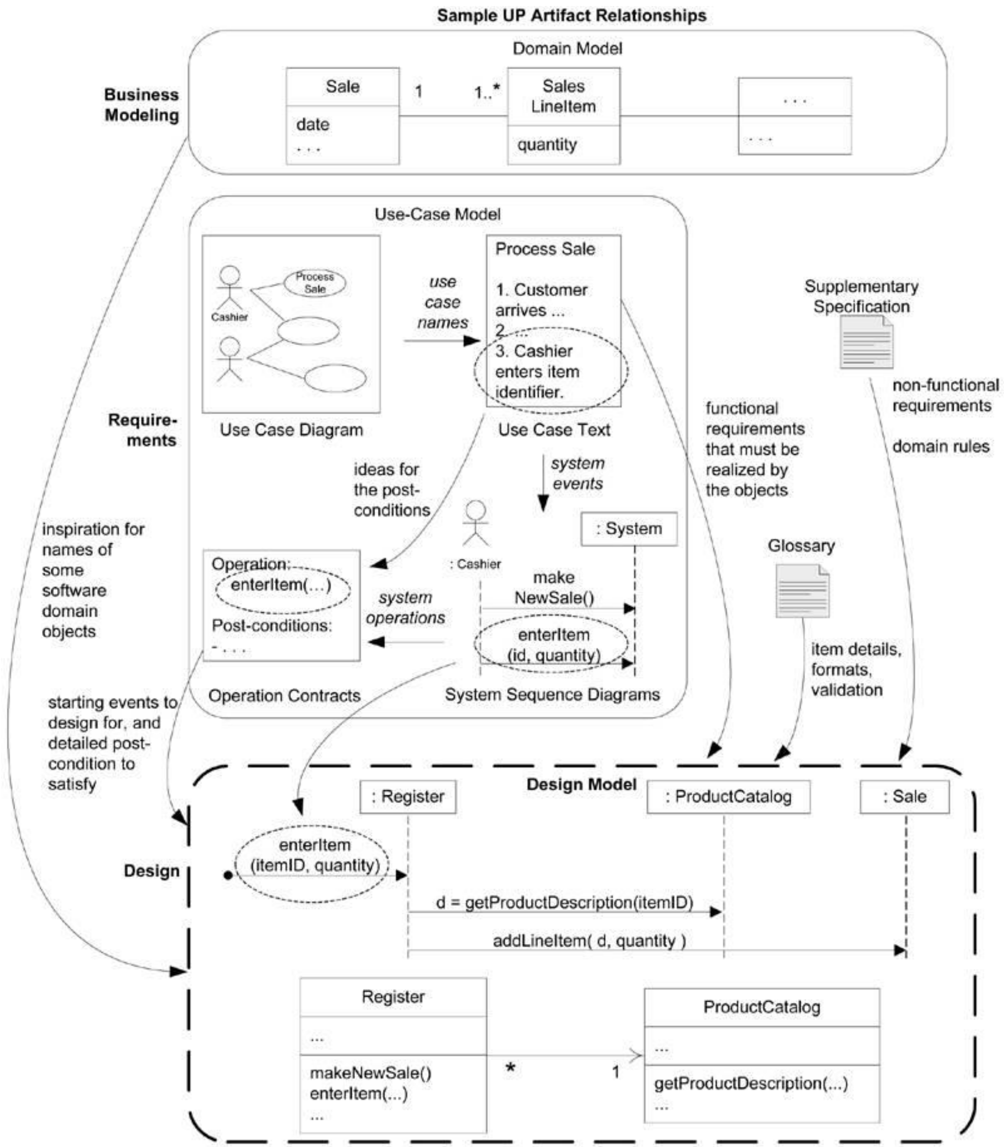
Operation Contracts

- Complement the use case text to clarify what the software objects must achieve in a system operation

Domain Model

- Suggests some names, relations and attributes of software domain objects in the domain layer of the software architecture

Artifacts Influence OOD



Analyst Hat is OFF
Designer-Modeler Hat is ON

Given inputs, developer could:

- Start some UML modeling for the object design

- Start immediately coding, ideally with test-first development (e.g. XP TDD) .

Analyst Hat is OFF
Designer-Modeler Hat is ON

Creating UML is not the point, it is the visual modeling vs. Text (UCs)

UML developers draw both in **one modeling day**, interaction diagrams and complementary class diagrams (**dynamic** and **static** modeling)

During drawing/coding developers apply various OO design principles:

- **GRASP, GoF, SOLID** design patterns and principles

Responsibility-Driven Design (RDD):

Thinking about how to assign responsibilities to collaborating objects

Activities of Object Design —Example

- On the modeling day, perhaps the team works in small groups for 2-6 hours either at the walls or with software modeling tools,
- Doing different kinds of modeling for the difficult, creative parts of the design
- Modeling include UI, OO, and database modeling with UML drawings, UI prototyping tools, sketches, and so forth

Activities of Object Design —Example

- Next day, still early in the three-week time-boxed iteration, the **team stops modeling and puts on programmer hats** to avoid a waterfall mentality of over-modeling before programming

Responsibility Driven Design

A popular way of thinking about the design of software objects is in terms of

- **Responsibilities**,
- **Roles**, and
- **Collaborations**

In RDD objects are thought of as having responsibilities

Responsibility Driven Design

The UML defines responsibility as "**a contract or obligation of a classifier**”

A classifier is a mechanism that describes **structural** and **behavioral** features

Classifiers include classes, interfaces, subsystems, … (others)

Responsibilities are related to the **obligations** or **behavior** of an object in terms of its **role**

Two Types of Responsibility

1. Doing responsibilities of an object include:

- Doing something **itself**, such as **creating** an object or doing a **calculation**

- **Initiating action** in *other objects*

- **Controlling** or **coordinating** activities in other objects

- **Doing**: **Create**, **Calculate**, **Initiate**, **Coordinate**

Two Types of Responsibility

2. Knowing responsibilities of an object include:

- Knowing about **private**, **encapsulated** data
- Knowing about **related** objects
- Knowing about things it can **derive** or **calculate**

Types of Responsibility— POS Example

Doing

a **Sale** object should most likely be responsible for creating **SalesLineItems** (*Creator Pattern*)

Knowing

a **Sale** object is responsible for knowing its *total* (Expert Pattern)

Assigning Responsibilities

Start assigning responsibilities by clearly stating the responsibility

For a domain object, because of the attributes and associations it has, knowing responsibilities are often apparent

For example, if the domain model **Sale** concept has a time attribute, it is natural, due to LRG, a software **Sale** class knows its *time*

What is Responsibilities Anyway?

Ultimately responsibilities are realized with methods

Big responsibilities may ultimately require many classes and methods, e.g. provide access to a database

A responsibility is not the same thing as a method,

a responsibility is an abstraction (Responsibility ≠ Method)

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

Assigning Responsibilities —Analogy

Think of software **objects** as like **people with responsibilities** who ***collaborate*** *with other people to get work done*!

Responsibility Driven Design leads to viewing an OO design as a

community of collaborating and responsible objects

GRASP: A Methodical Approach to Basic OO Design

Understanding how to apply GRASP for object design is a key goal

Once you "**grasp**" the fundamentals, the specific

GRASP **terms or labels or names** (Information Expert, Creator, ..., as many as 9)

aren't that important!

Responsibilities, GRASP, and UML Diagrams *Connection*

A developer can think about

assigning responsibilities to objects while coding or while modeling

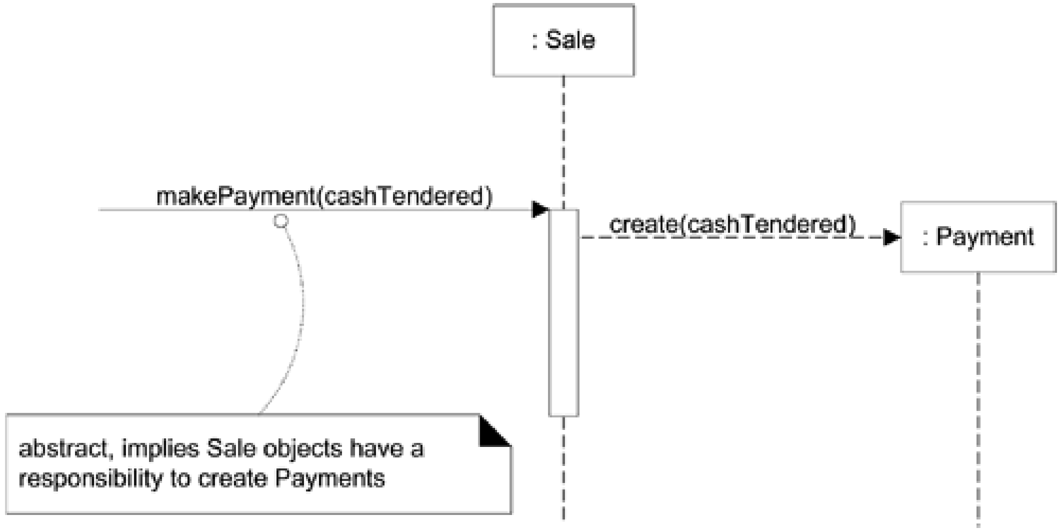
Drawing a UML interaction diagram decisions about responsibility assignments are made (realized as methods)

Fulfillment of **responsibility** will often require **collaboration** with other objects **(GRASP)**

Responsibilities, Methods are Related

Sale object have responsibility to create **Payment** using **makePayment message** and handled with a corresponding ***makePayment method***

The fulfillment of this responsibility requires collaboration to **create** the **Payment** object and invoke its constructor


: Sale
makePayment(cashTendered)
create(cashTendered)
: Payment
abstract, implies Sale objects have a responsibility to create Payments

Patterns

Experienced OO developers build up a toolbox of both General principles and idiomatic solutions that guide them in the creation of software

These **principles** and **idioms**, when **codified** in a structured format describing the problem and solution and named, are called patterns

No One Owns Patterns!

The "Gang of Four“ **GoF** design patterns book caused software developers to take note of design patterns in the mid-1990s . .

Larman's **GRASP** patterns/principles don't state new ideas, they name and codify widely used basic principles

SOLID +Others

To an OO design expert, the ideas underlying the GRASP patterns will appear fundamental and familiar

GRASP 5/9 —Monopoly as a Case Study

Pattern	**Problem**
Creator	Who should be responsible for creating a new instance of some class?
Information Expert	What is a general principle of assigning responsibilities to objects?
Low Coupling	How to support low dependency, low change impact, and increased reuse?
Controller	What first object beyond the UI layer receives and coordinates ("controls") a system operation?
High Cohesion	How to keep objects focused, understandable, and manageable, and, as a side effect, support low coupling?

Creating Objects

One of the first problems that needs consideration in OO design is: "*Who creates object X?*“ This is a **Doing** responsibility

Problem: *In the Monopoly case study, who (what object) creates a **Square** software object?*

*Any object could create a **Square**, but what would be best, and why? Should some arbitrary object create **Square** objects?*

Solution: (**Creator Pattern**) Assign class B the responsibility to create an instance of class A if B "contains" or compositely aggregates A

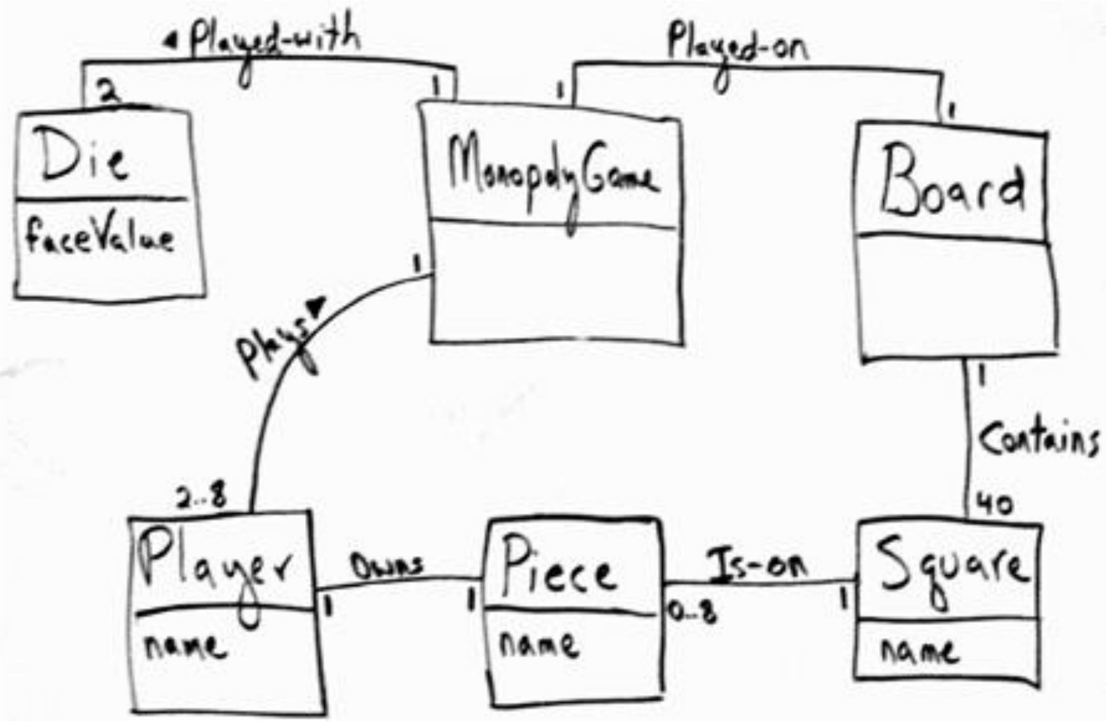
1. Creator Pattern

Since we are just starting the OO design, we haven't defined any software classes yet

We can look at the domain model for inspiration for our software objects

This applies low representational gap principle

Monopoly Iteration-1 —Domain Model



Creating Monopoly Squares

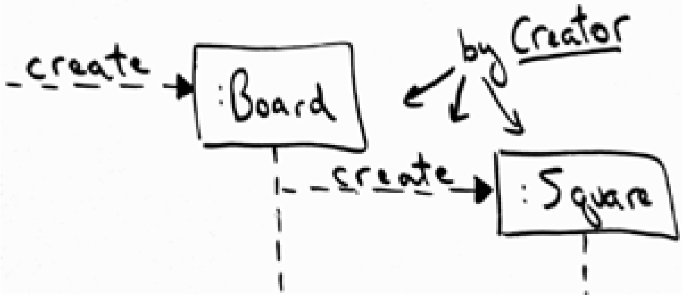
*What object should create the **Square** objects?*

According to the Creator Pattern, since the **Board** object contains **Square** objects the **Board** object should create Square objects

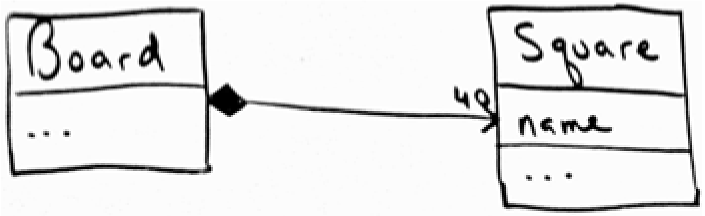
In parallel we can model the **Square** object creation and the static and dynamic design model

Creating Monopoly Squares

Applying the Creator Pattern in a dynamic model



Board has a composite aggregation association with **Squares,** Static model



2. Information Expert Pattern

Information Expert pattern is the most basic responsibility assignment principles in object design

Suppose objects need to be able to reference a particular **Square**, by name

Problem: "*Who should be responsible for knowing a **Square**, given a name?*“

As with Creator, any object could fulfill this responsibility, *but what would many OO developers choose? And why?*

Solution: As with the creation problem, most OO developers choose the **Board** object. But why?

Information Expert Pattern

Solution: (**Information Expert**) Assign a responsibility to the class that has the information needed to fulfill it

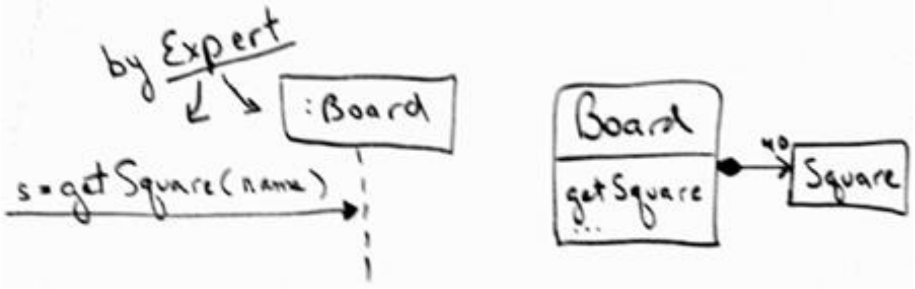
As has been previously decided, a software **Board** will aggregate all the **Square** objects

Thus, the **Board** object has the information necessary to fulfill this responsibility

Information Expert Pattern

The ***getSquare*** operation will be assigned to the **Board** object (**Knowing**)

In this case this responsibility assignment may seem obvious and trivial, but in other cases it is not



Applying Expert

3. Low Coupling Pattern

Low Coupling is a GRASP pattern/principle/guideline

The Low Coupling pattern is used to **evaluate alternatives**

All other things being equal,

we should prefer a design whose

coupling is lower than the alternatives

Low Coupling Principle

The Low Coupling GRASP principle explains why Expert is a useful, core principle of OO design

Question: *Why **Board** over some other class, what are the benefits of using **Board**?*

The previous use of the Information Expert pattern gives low coupling

Low Coupling Pattern— Poor Design

Why not assign ***getSquare*** to **Dog** (i.e., some arbitrary other class **Dog**, a random pick of any class)?

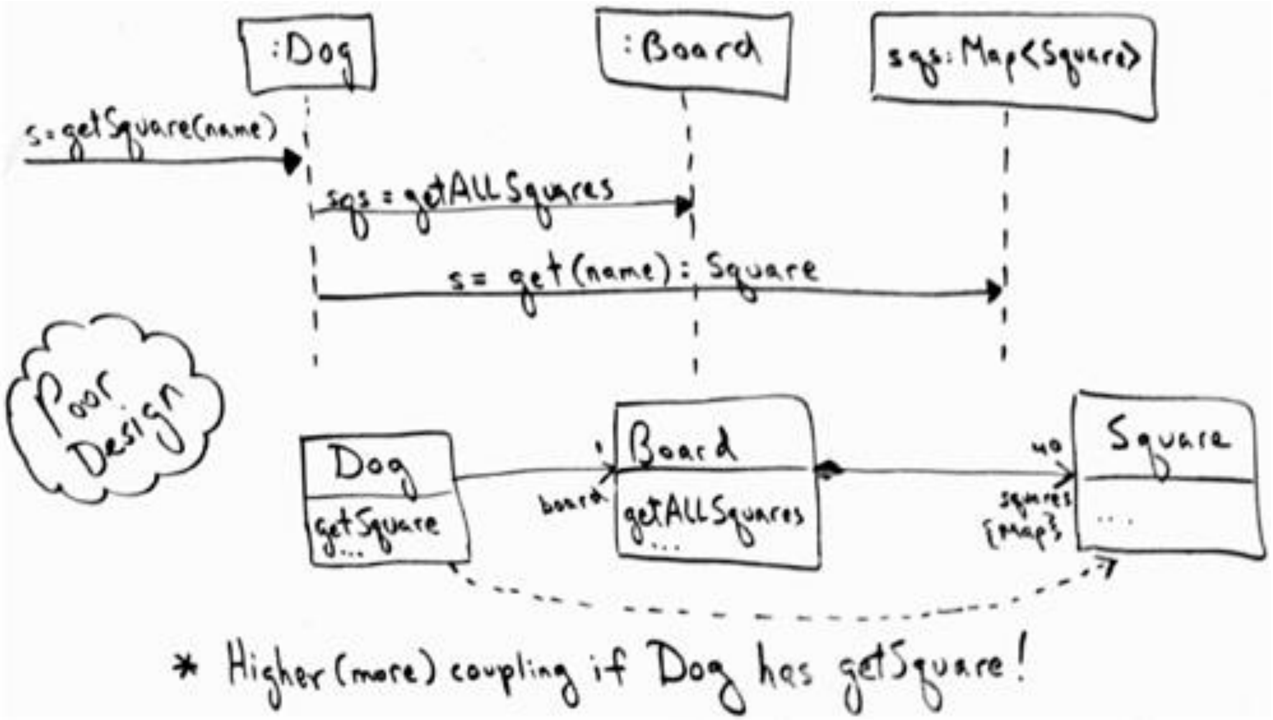
Consider the impact in terms of low coupling

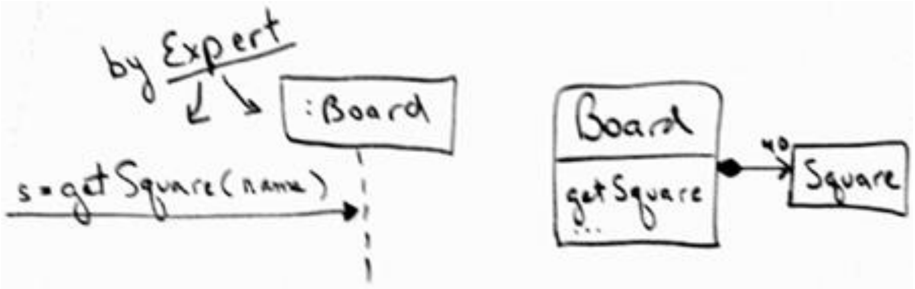
If a **Dog** has ***getSquare*** it must collaborate with the **Board** to get the collection of all the Squares in the **Board**

Squares are probably stored in a Map or a Dictionary collection object, which allows retrieval by a key

Then, the **Dog** can access and return one Square by the key name

Evaluating The Effect Of Coupling On This Design





UI Events, What Should Be Done With Them?

In a simple layered architecture, there is a UI layer and a domain layer, among others

Actors generate UI events, such as clicking on a button with a mouse, to play the game

The UI software objects must then react to the mouse click event and ultimately cause the game to play

UI Events, What Should Be Done With Them?

According to the **Model-View Separation Principle**:

UI objects should not contain application or "business" logic

Once the UI objects pick up the mouse event they need to **delegate** (forward the task to another object) the request to domain objects in the domain layer

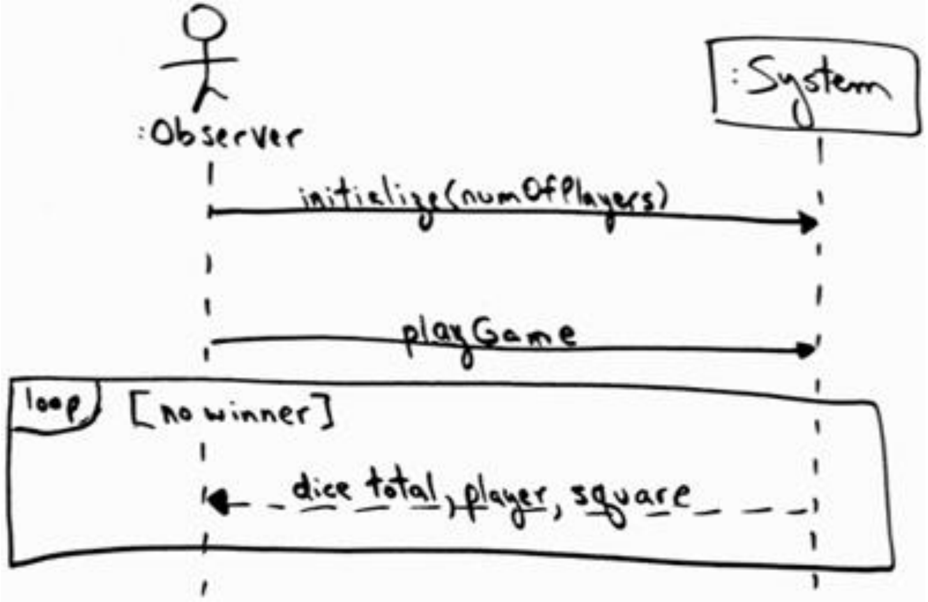
UI Event Question

Question: *What first object after or beyond the UI layer should receive the*

event/message from the UI layer?

Recall the **SSD** for the Monopoly game. Note the ***playGame*** operation.

SSD for the Monopoly game. Note the *playGame* operation

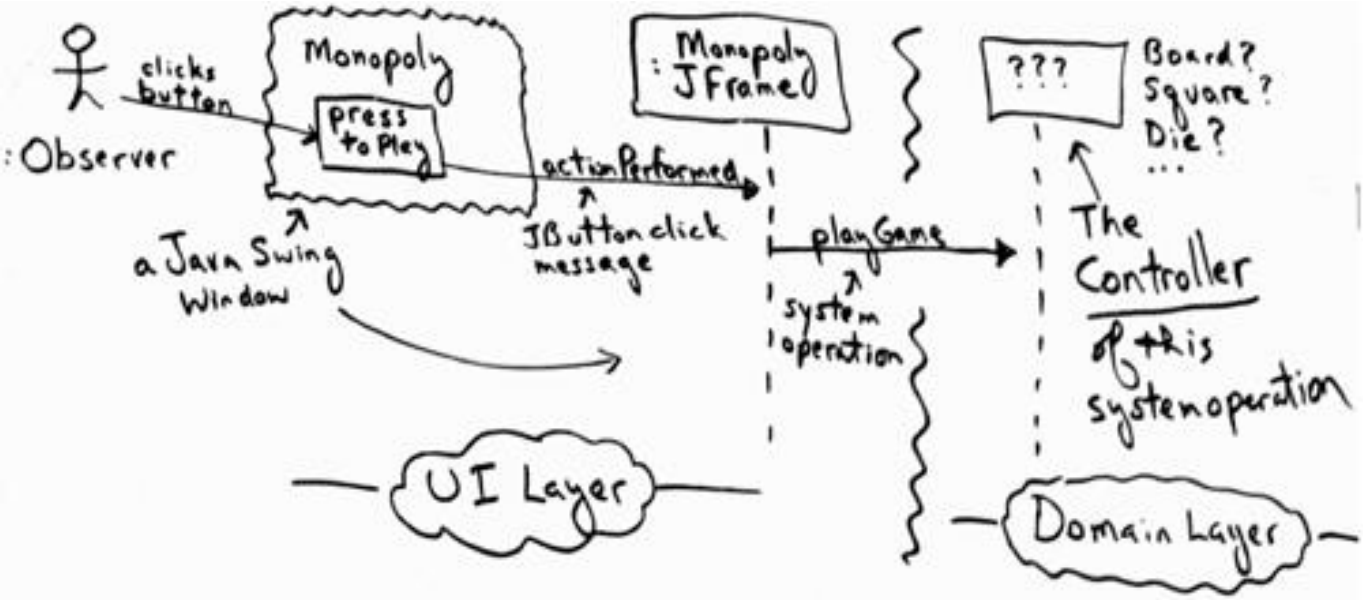


4. Controller Pattern

How is the UI layer ***connected*** *to the application logic layer?*

Problem: *What first object beyond the UI layer receives and coordinates ("controls") a system operation?*

Who is the Controller for the ***playGame*** *system operation?*



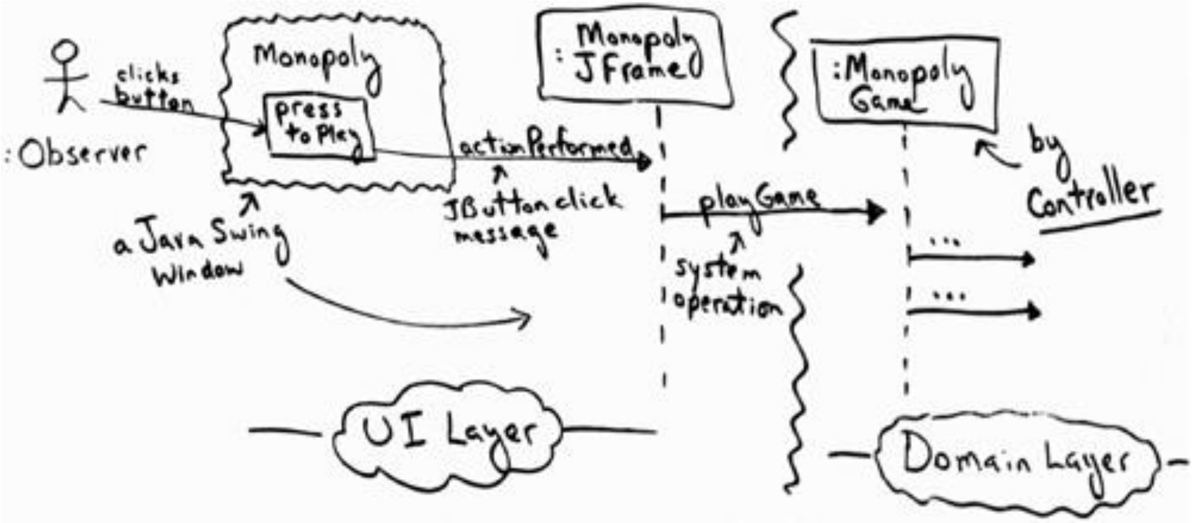
Controller Pattern

Should the ***Board*** *be the first object to receive the* ***playGame*** *message from the UI layer? Or something else?*

Solution(**Controller Pattern**): Assign the responsibility to an object that represents the **use case** or **session**

MonopolyGame is reasonable choice in case there are only a few system operations (More on the trade-offs when we discuss High Cohesion)

Applying Controller pattern —MonopolyGame

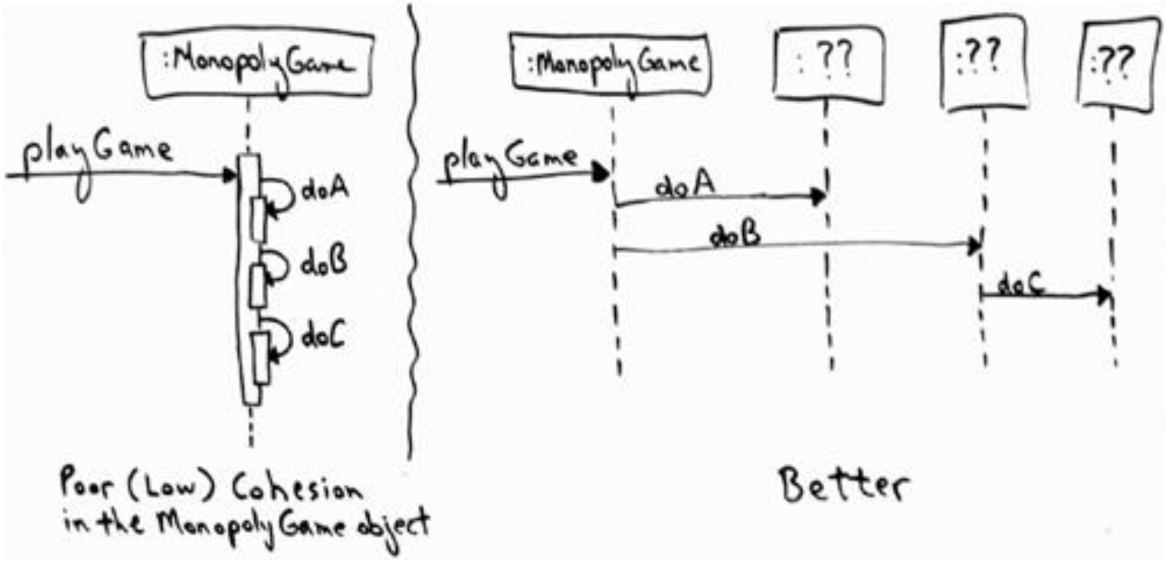


Connecting the UI layer to the domain layer of software objects

The MonopolyGame Object

At this point consider some possibilities for the **MonopolyGame** object

It can do most of the work itself, or delegate.



Contrasting the level of cohesion in different designs

5. High Cohesion Pattern

High Cohesion is a GRASP pattern/principle/guideline

A measure of how strongly related each piece of a classifier to the intended functionality of the classifier

The High Cohesion pattern is used to

evaluate alternatives

All other things being equal, we should prefer a design whose

cohesion is higher than the alternatives

Applying GRASP to Object Design

GRASP stands for **G**eneral **R**esponsibility **A**ssignment **S**oftware **P**atterns

The name GRASP was chosen by Larman to suggest the importance of **grasping** the GRASP principles to successfully design object-oriented software

Understanding + **apply** GRASP, while coding or drawing interaction and class diagrams

Developers new to object technology needs to master these basic principles as quickly as possible

GRASP forms a foundation for designing OO systems

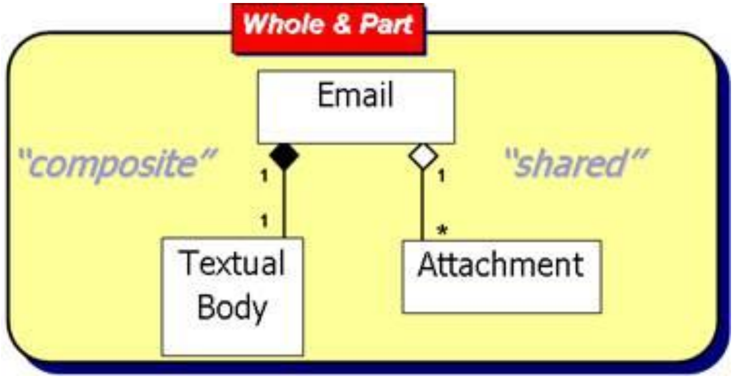
There Are Nine GRASP Patterns

1. Creator
2. Information Expert
3. Low Coupling
4. Controller
5. High Cohesion
6. Indirection
7. Pure Fabrication
8. Polymorphism
9. Protected Variations

1. 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 the following is true (the more the better): 1. B contains or aggregates A 2. B has the initializing data for A (when A is created) 3. B records A 4. B closely uses A

Aggregate Relationship



Creator

Object creation is a common task

The basic intent is to find a creator that needs to be connected to the created object

Choosing this object supports low coupling

Creator

Sometimes a creator can be identified by looking for a class that has the initializing data that will be passed in during creation

This is an example of the Information Expert pattern

If creation has significant complexity, it can be delegated to a helper class, *Abstract Factory*, *Factory Method*

Creator

Benefits:

- Low coupling is supported

Related Patterns or Principles:

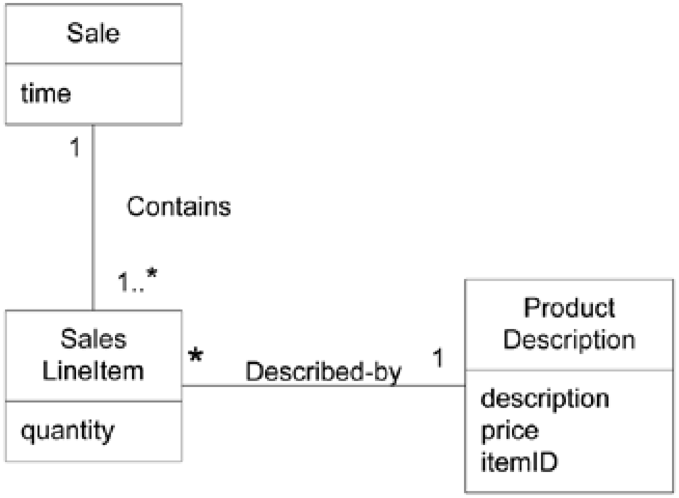
- Low coupling, a created class is likely to be **visible** to the creator class due to considerations that motivated the choice as creator
- *Factory Method*, *Abstract Factory*

Creator –POS

*In the POS application, who should be responsible for creating a **SalesLineItem** instance?*

Using the Creator principle, we should look for a class that aggregates and contains **SalesLineItem** instances

Partial Domain Model

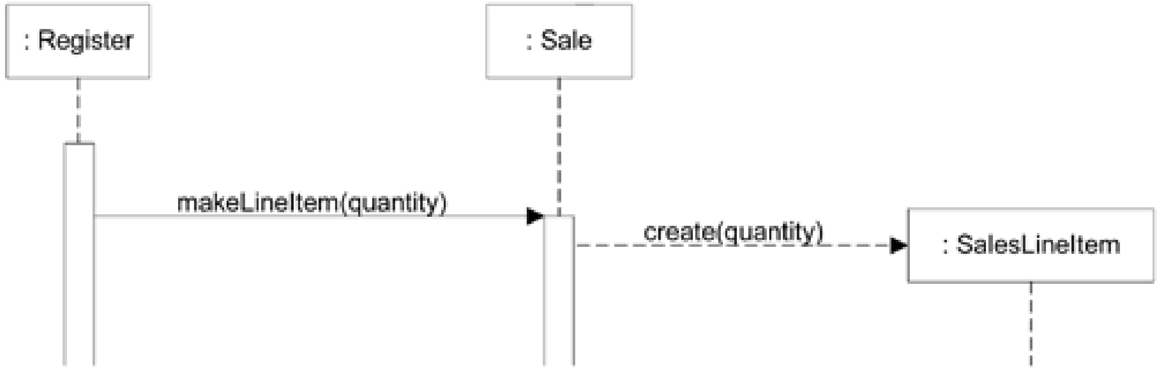


Creator –POS

Sale contains many **SalesLineItem** objects

Creator pattern recommends that **Sale** is a good candidate to have the responsibility of creating **SalesLineItem** instances

Creating a SalesLineItem



Creator —POS

This assignment of responsibilities requires that a ***makeLineItem*** method be defined in **Sale**

Note that the context in which this was considered and decided was while drawing an interaction diagram

Responsibility assignment realized by a method addition

2. Information Expert

Problem	**What is a general principle of assigning responsibilities to objects?**
Solution	Assign a responsibility to the information expert - the class that has the information necessary to fulfill the responsibility

Information Expert

Guiding principle, it expresses the **common intuition**

Objects should do things related to the information they have

Information expert reflects the real world,

e.g. in business who should be responsible of the profit-and-loss statement?

Answer: The person who has access to the data to create it, the chief financial officer possibly

Information Expert

Expert usually leads to designs where a software object does those operations that are normally done to the real-world (**LRG**)

Information will often be **spread** across classes and collaboration will be necessary

Information Expert

Problems in Cohesion or Coupling occur

For example, saving a **Sale** to a database, *should the **Sale** save itself?*

Probably not, see *Pure Fabrication*.

Information Expert

Benefits:

- Information Encapsulation is maintained since objects use their own information to fulfill tasks
- **Behavior is distributed across the classes** that have the information encouraging more **cohesive**, **lightweight classes** that are easier to understand and maintain

Related Patterns or Principles:

- *Low Coupling*
- *High Cohesion*

Information Expert —POS

In the POS application some class will need to know the grand total of a sale

Question: *To assigning a responsibility Who should be responsible for knowing the grand total of a sale?*

Answer: The Information Expert suggests that we should look for a class has the information needed to determine the total

Information Expert —POS

Question: *Do we look in the Domain Model or the Design Model to analyze the classes that have the information needed?*

Recall: The Domain Model illustrates conceptual classes of the real-world domain, and the Design Model illustrates software classes

Answer: If there are relevant classes in the Design Model, look there first, if not, look in the Domain Model and attempt to use what is present to inspire the creation of corresponding design classes

Information Expert —POS

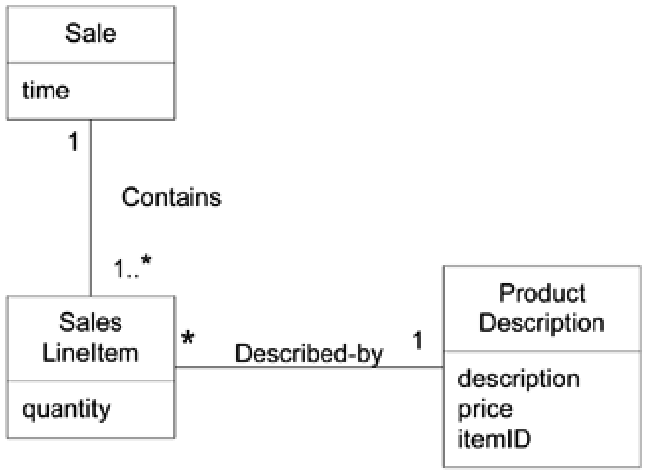
We look to the Domain Model for information experts and see the real-world **Sale** as a candidate

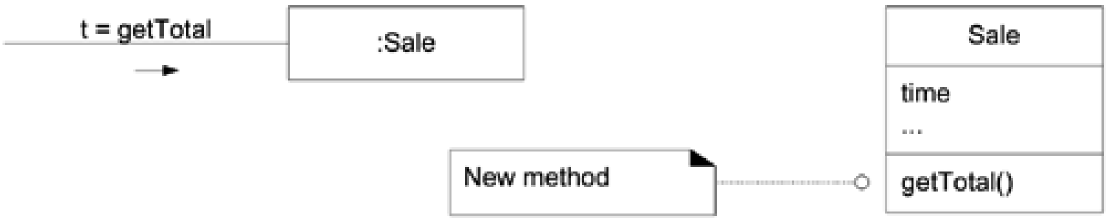
We add a **Sale** software class to the Design Model, if we haven't already

Give it responsibility of **knowing** its total with method named, perhaps, ***getTotal***

Also, this approach gives a low representational gap

Associations of Sale (Domain Model)





Partial Interaction, Class Diagrams Design Model

Information Expert —POS

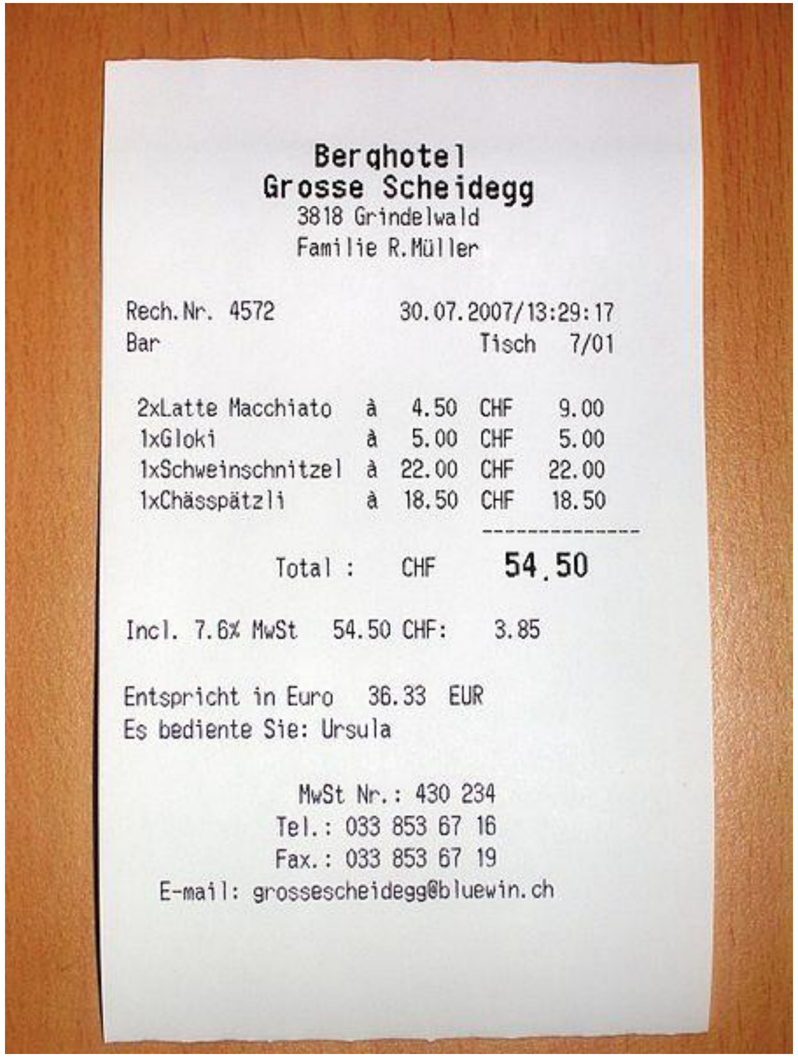
What information is needed to determine the grand total?

It is necessary to know all the **SalesLineItem** instances of a sale and the sum of their subtotals

A **Sale** instance contains the **SaleLineItems** and so by the guideline of Information Expert

Sale is a suitable class for this responsibility

A Receipt!



Berghotel
Grosse Scheidegg
3818 Grindelwald
Familie R.Müller

Rech.Nr. 4572 30.07.2007/13:29:17
Bar Tisch 7/01

2xLatte Macchiato	à	4.50	CHF	9.00
1xGloki	à	5.00	CHF	5.00
1xSchweinschnitzel	à	22.00	CHF	22.00
1xChässpätzli	à	18.50	CHF	18.50

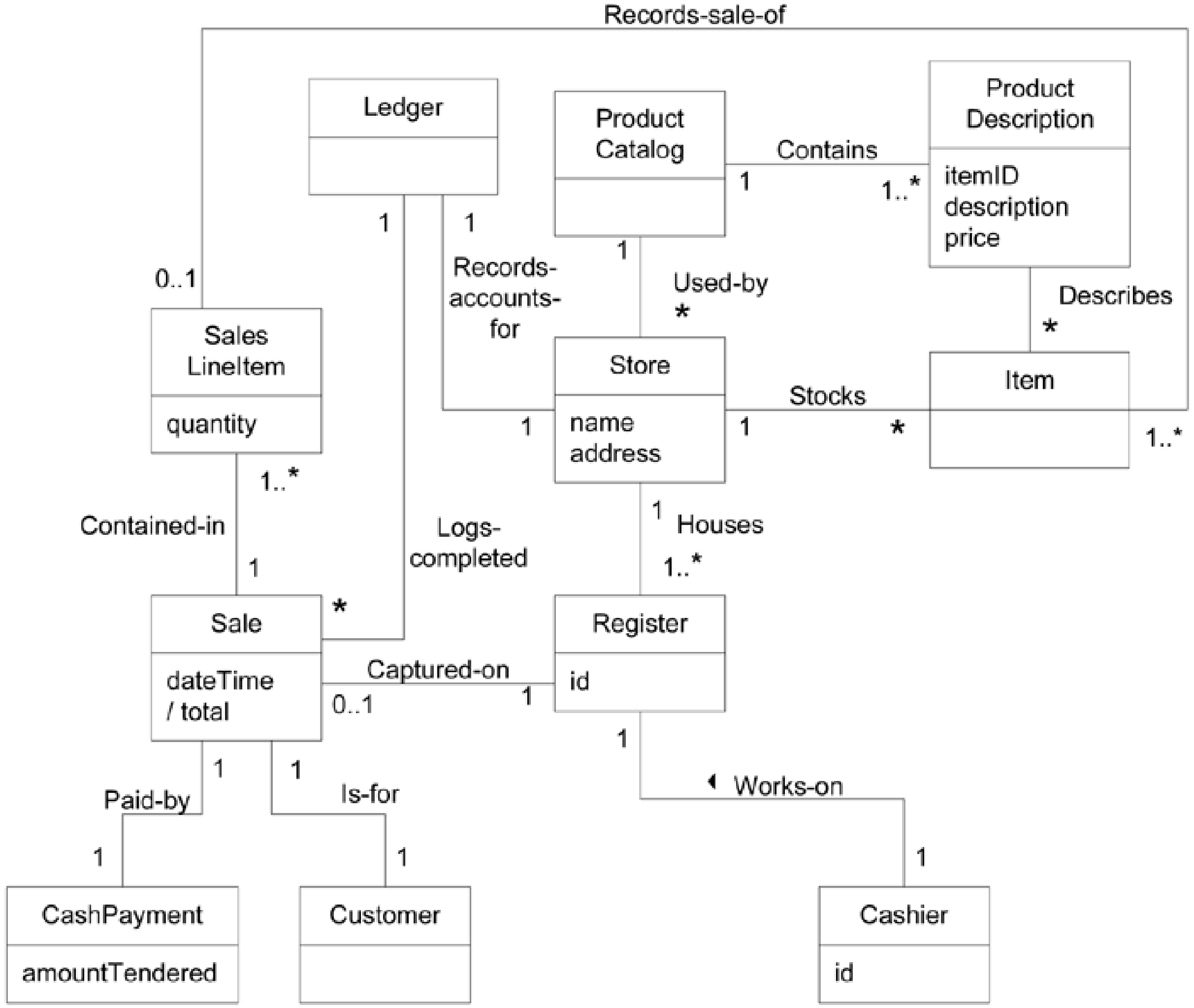
Total : CHF **54.50**

Incl. 7.6% MwSt 54.50 CHF: 3.85

Entspricht in Euro 36.33 EUR
Es bediente Sie: Ursula

MwSt Nr.: 430 234
Tel.: 033 853 67 16
Fax.: 033 853 67 19
E-mail: grossescheidegg@bluewin.ch

POS Partial Domain Model



Information Expert —POS

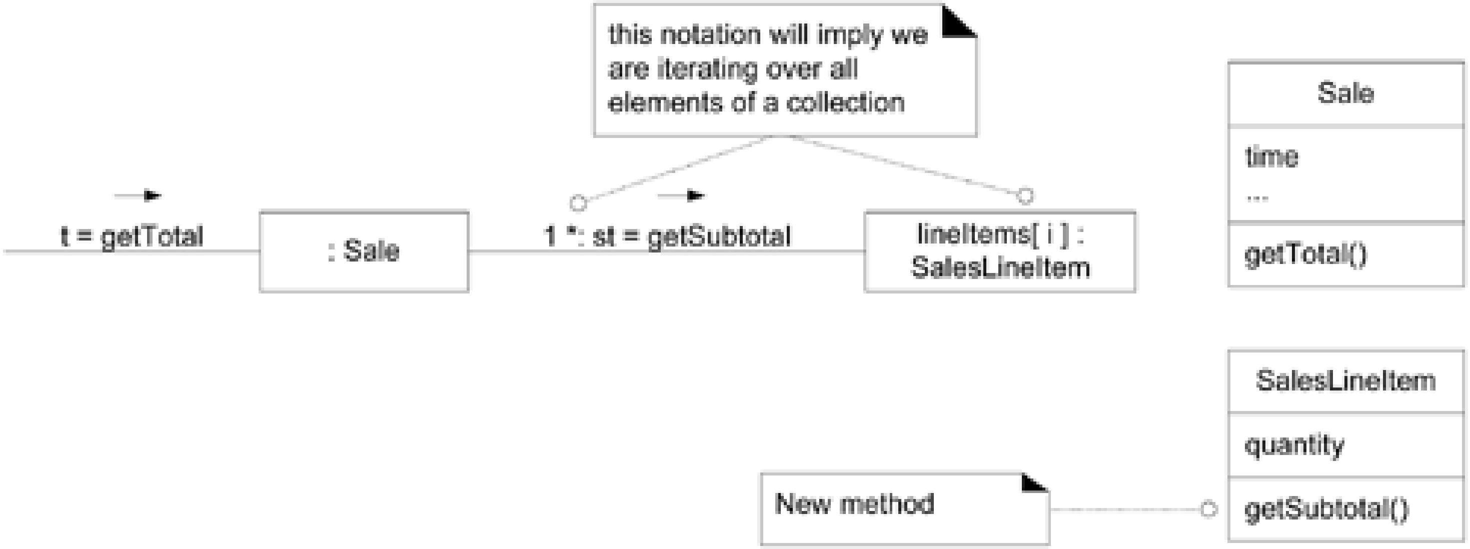
*What information is necessary to determine the **line-item subtotal**?*

SalesLineItem.quantity**,**
ProductDescription.price are necessary

SalesLineItem knows its quantity and its associated **ProductDescription**, and so **SalesLineItem** should determine the subtotal

SalesLineItem it is the information expert, ***getsubtotal***

Information Expert —POS


this notation will imply we are iterating over all elements of a collection
t = getTotal
: Sale
1 *: st = getSubtotal
lineItems[i] : SalesLineItem
Sale
time
...
getTotal()
SalesLineItem
quantity
getSubtotal()
New method

Calculating the *Sale* total

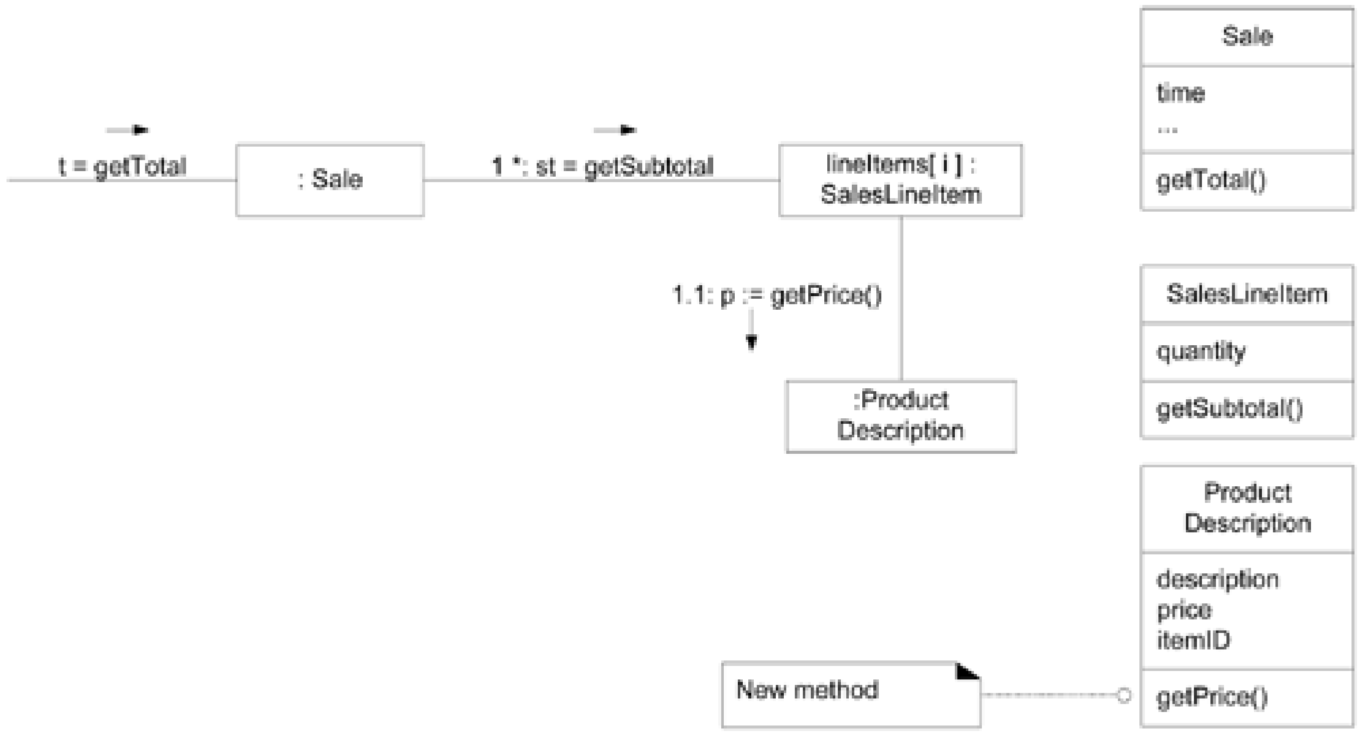
Information Expert —POS

What about the price of the product?

ProductDescription is an information expert regarding product price, it knows!

therefore, **SalesLineItem** sends **ProductDescription** a message asking for the product price

Information Expert —POS



Calculating the *Sale* total

Information Expert —POS

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

Design Class	Responsibility (knowing)
Sale	knows sale total
SalesLineItem	knows line-item subtotal
ProductDescription	knows product price

Information Expert - Comments

Information Expert is a basic guiding principle that frequently used in the assignment of responsibilities and object design

Information Expert expresses the common "intuition" that objects do things related to the information they have,

But too much "intuition“ make cause problems, will see!
SOLID?

Information Expert —Animation principle

The fulfillment of a responsibility often requires information that is spread across different classes of objects

Many "partial" information experts collaborate in the task

- Sales total problem ultimately required the collaboration of three classes of objects
- Whenever information is spread across different objects, they will need to interact via messages to share the work

Information Expert —Animation principle

Software object does those operations to the inanimate real-world thing it represents

In OO, 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

The “animation” principle in object design; it is like being in a cartoon where everything is alive.

3. Low Coupling

Problem	***How to support low dependency, low change impact, and increased reuse?***
Solution	Assign a responsibility so that (unnecessary) coupling remains low Evaluative, use this principle to evaluate alternatives

What is Coupling Anyway?

TypeX has an attribute that refers to a **TypeY** instance or **TypeY** itself

A **TypeX** calls on services of a **TypeY** object

TypeX has a method that references a **TypeY** instance or **TypeY** itself by any means; parameter, local variable, return object

TypeX is a direct or indirect subclass of **TypeY**

TypeY is an interface, and **TypeX** implements that interface

Low Coupling

Benefits:

- Un-coupled components are not affected by changes in each other
- Simpler to understand in isolation
- Easier reuse and test

Related Patterns or Principles:

- *Protected Variation*

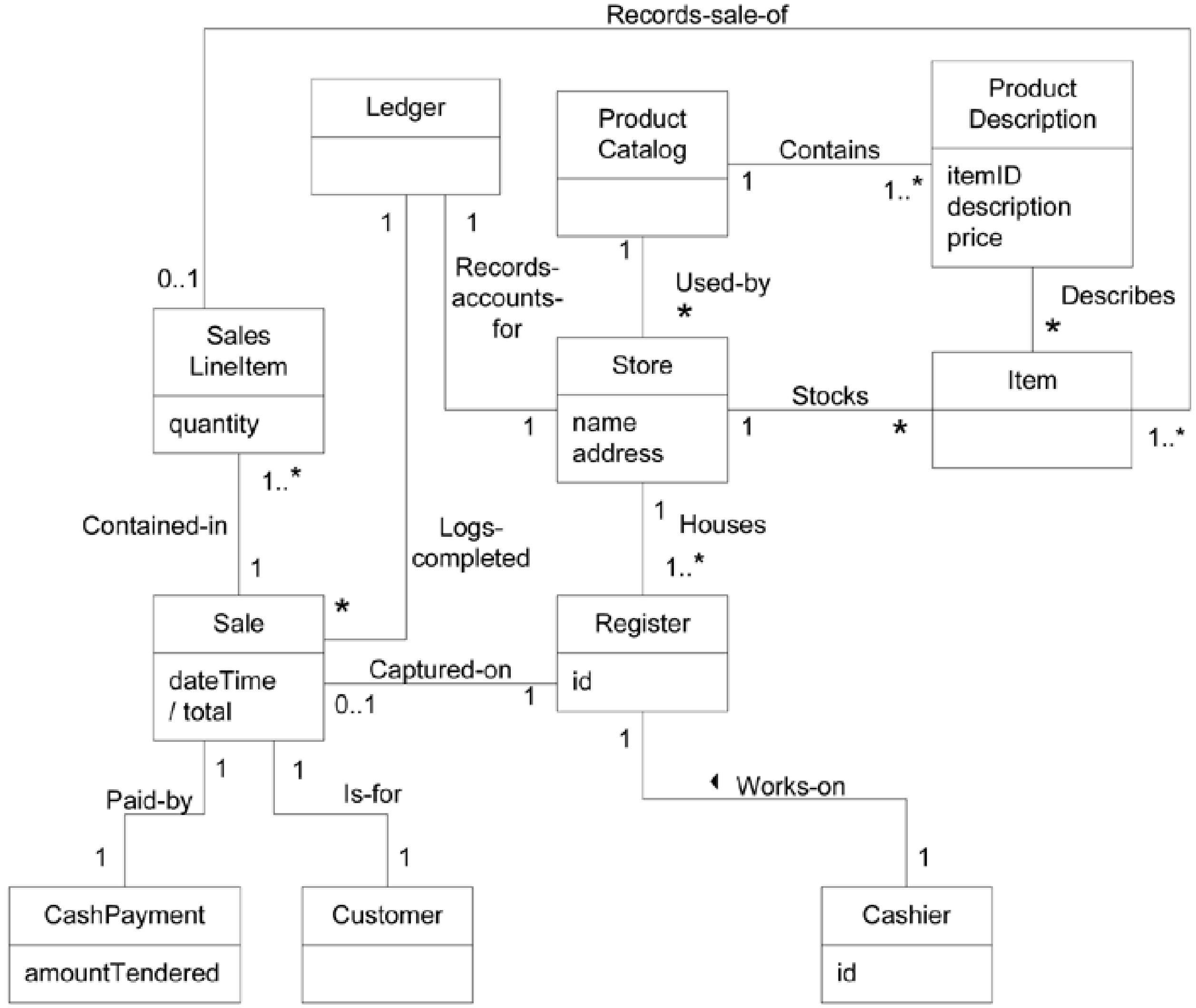
Low Coupling —POS

*What class should create a **Payment** instance and associate it with the **Sale**?*

Creator suggests **Register** since a **Register** "records" a **Payment** in the real-world domain (**LRG**)

This assignment of responsibilities couples the **Register** class to knowledge of the **Payment** class

POS Partial Domain Model

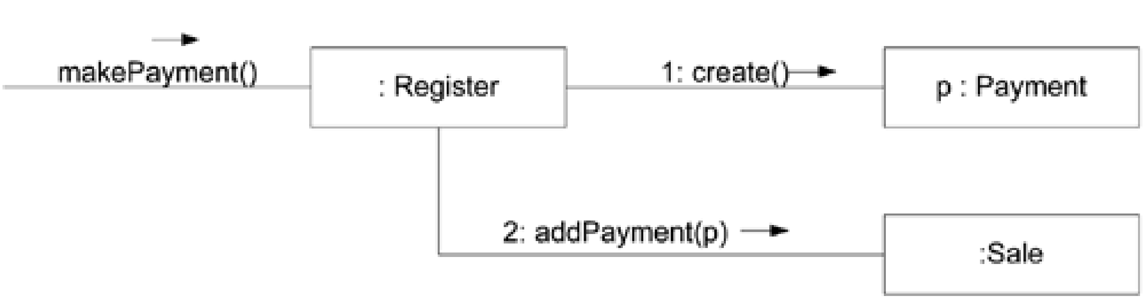


Low Coupling —POS

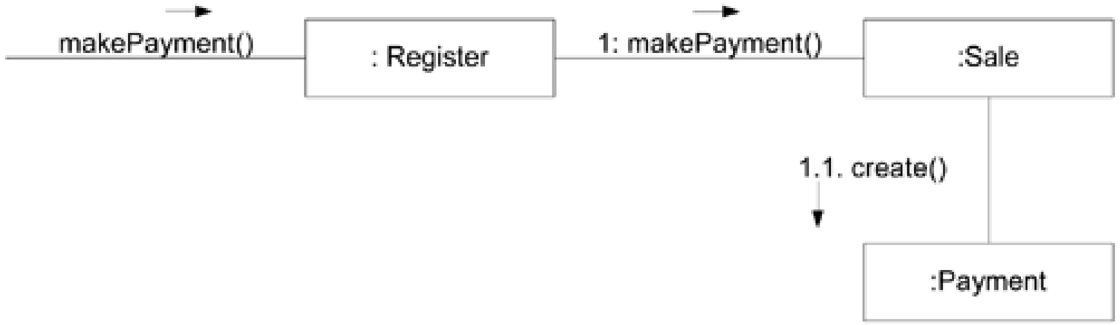
Low coupling suggests **Sale** should create **Payment**.

Which is best?

The second design gives lower coupling and, other things being equal, is probably preferred



Register creates Payment



Sale creates Payment

Low Coupling —Comments

Low Coupling is a principle to keep in mind during all design decisions

Low Coupling is an underlying goal to continually consider while evaluating design decisions

In general, classes that are inherently generic in nature and with a high probability for reuse should have especially low coupling

Low Coupling —Comments

Low Coupling taken to **excess** yields a poor design

A design with a few in-cohesive, **bloated**, and complex active objects that do all the work and with many passive low-coupled objects that act as simple data repositories

Low Coupling is an **evaluative** guideline

4. Controller

Problem	***What first object beyond the UI layer receives and coordinates ("controls") a system operation?***
Solution	Assign the responsibility to a class representing one of the following choices: 1. Represents the overall "system", a "root object", a “device” the software is running within, or a major “subsystem” (these are all variations of the **Facade Controller**) 2. Represents a use case scenario within which the system event occurs, (a use-case session controller)

Controller

This is a delegation pattern

UI, user interface, layer (boundary objects) shouldn't contain application logic

UI objects must delegate work requests to another layer

The controller pattern summarizes the common choices for the object to receive the work request

Controller

A common defect of controllers results from over-assignment of responsibility “**Bloated**" controllers

Solution: more controllers are needed or the controller is not delegating enough

Controller

Benefits:

- Increased potential for reuse and pluggable interfaces
- Opportunity to reason about the state of the use case

Related Patterns or Principles:

- *Command Pattern*
- *Facade Pattern*
- *Layers*
- *Pure Fabrication*

Controller —POS

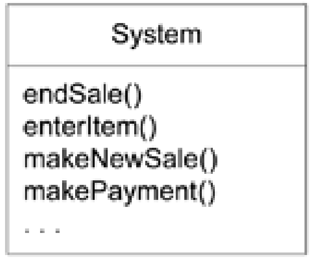
The POS application contains several system operations

During analysis (e.g. **SSD**), system operations may be assigned to the class System, to indicate they are **System** operations

There will not be software class named System though

During design, a controller class is assigned the responsibility for system operations

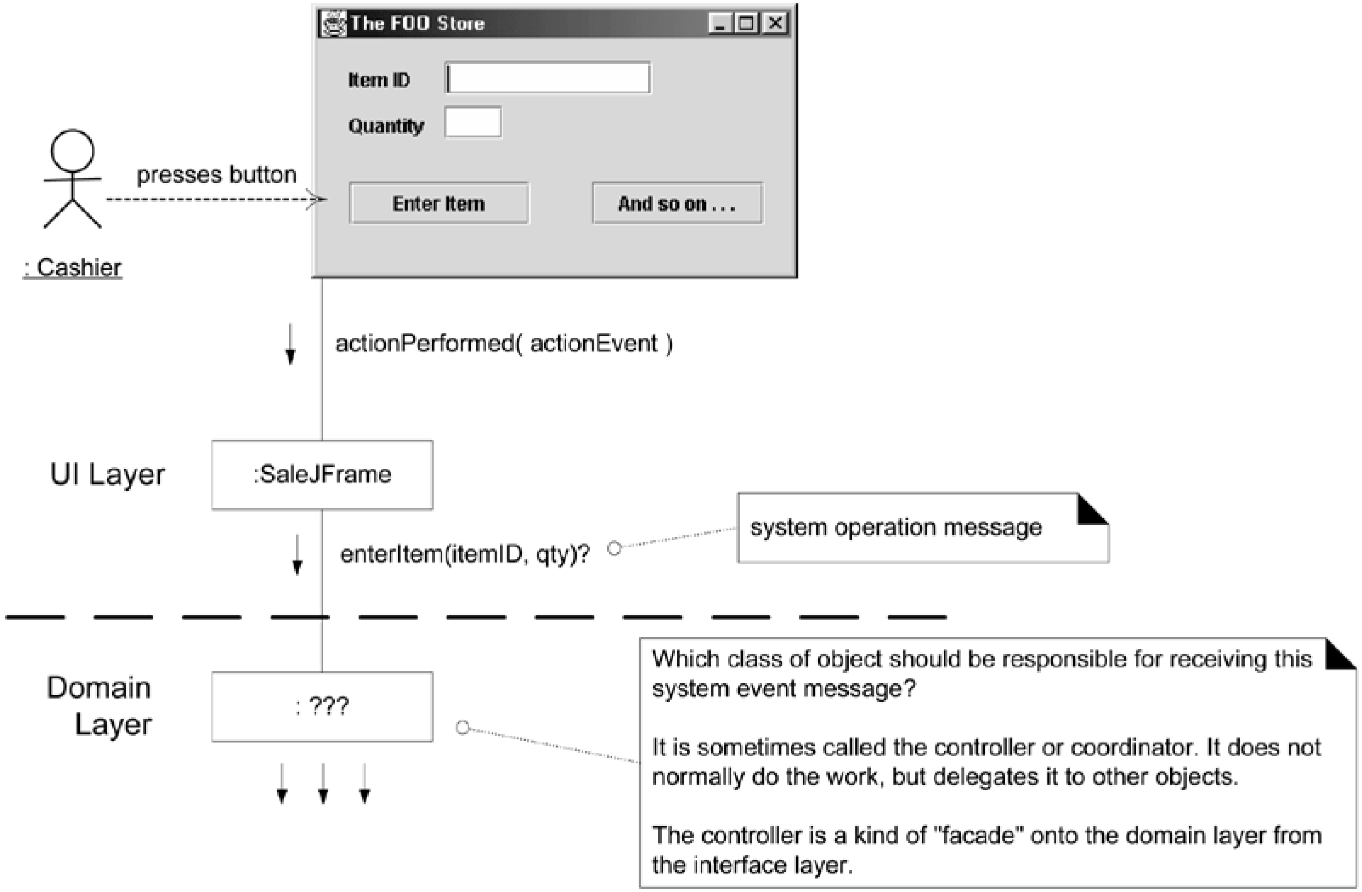
Some system operations of the NextGen POS application



System

endSale()
enterItem()
makeNewSale()
makePayment()
. . .

What object should be the Controller for enterItem?



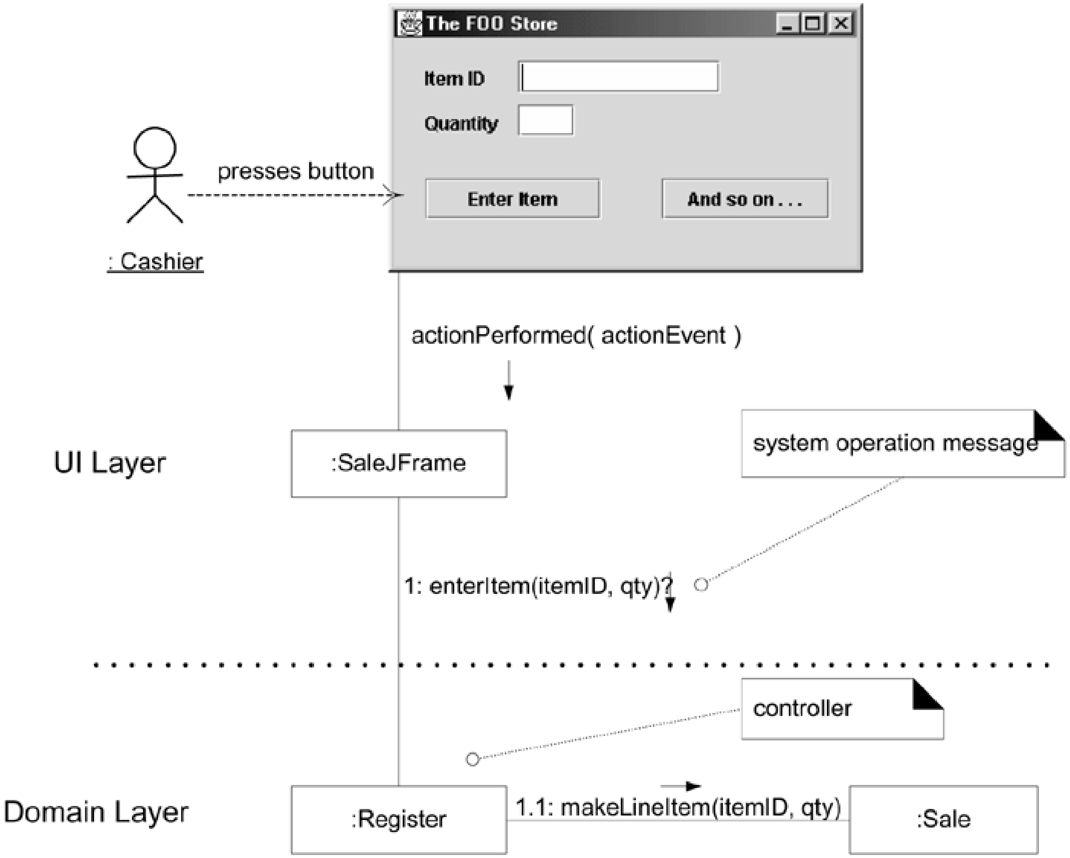
system operation message

Which class of object should be responsible for receiving this system event message?

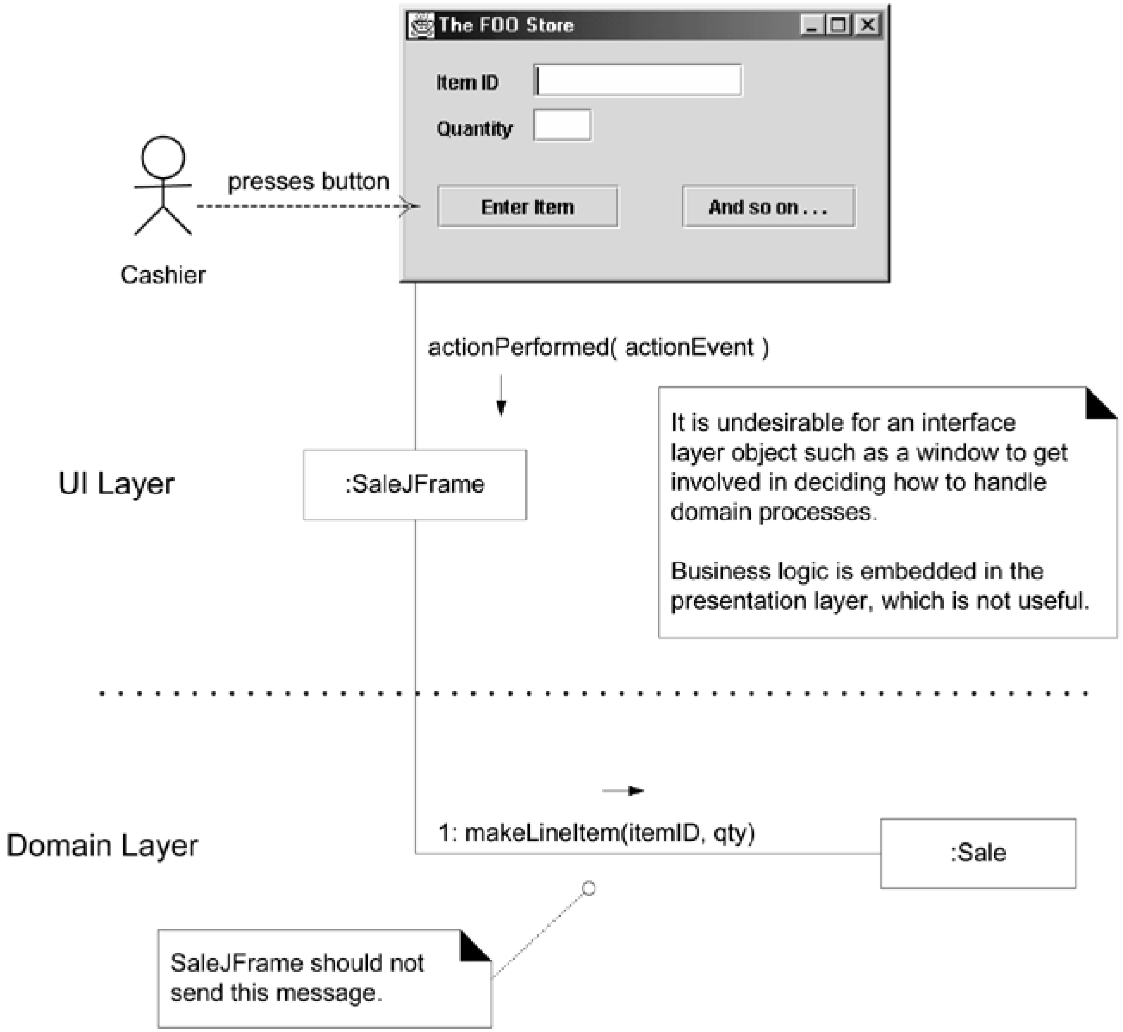
It is sometimes called the controller or coordinator. It does not normally do the work, but delegates it to other objects.

The controller is a kind of "facade" onto the domain layer from the interface layer.

Desirable coupling of UI layer to Domain layer: Device Rep.



Less desirable coupling of interface layer to domain layer



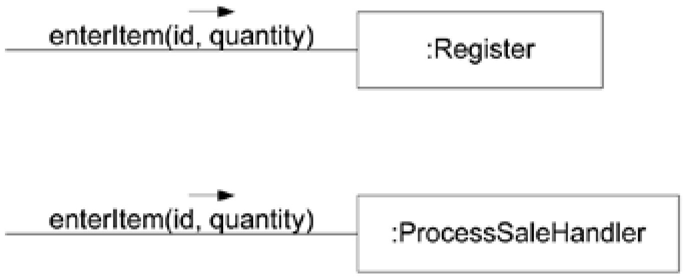
It is undesirable for an interface layer object such as a window to get involved in deciding how to handle domain processes.

Business logic is embedded in the presentation layer, which is not useful.

SaleJFrame should not send this message.

What object should be the Controller?

Controller choices



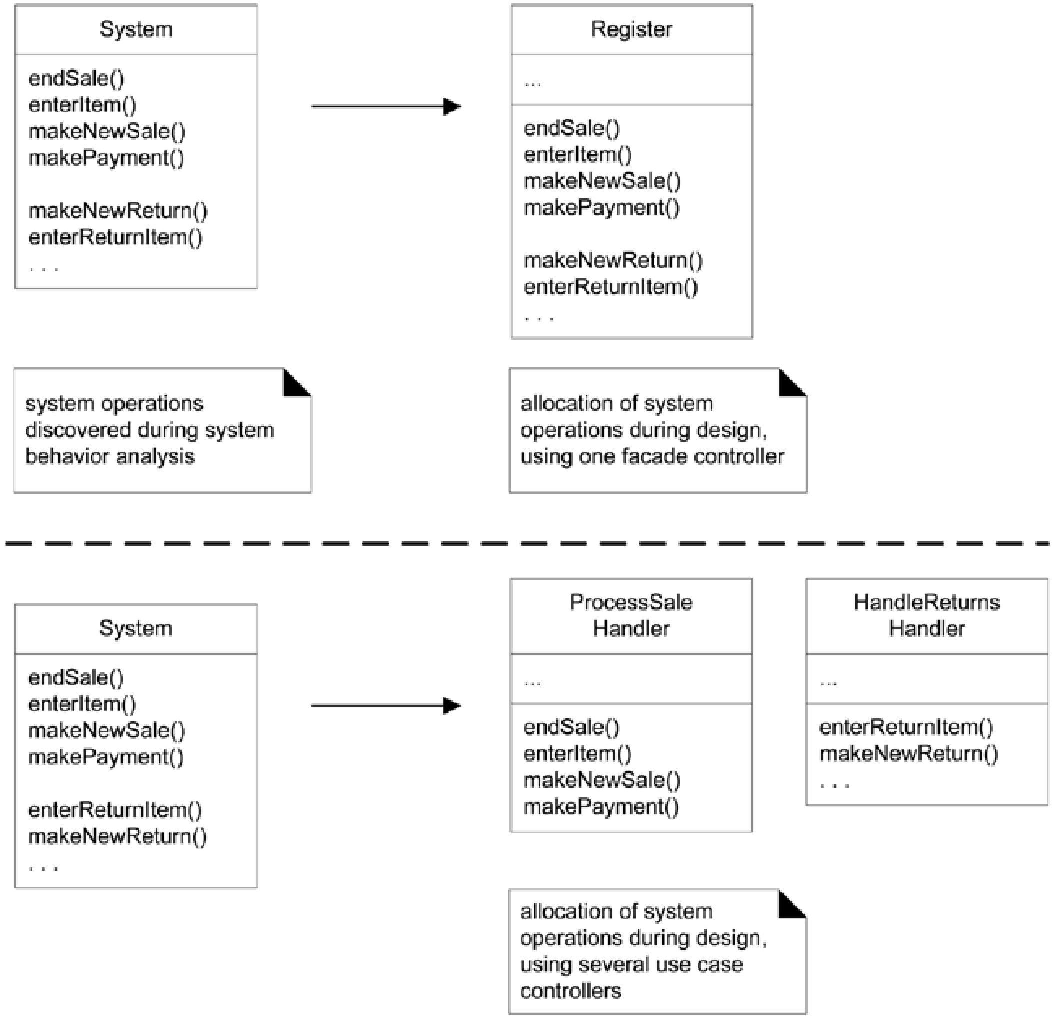
enterItem(id, quantity)

:Register

enterItem(id, quantity)

:ProcessSaleHandler

Allocation of system operations



System

endSale()
enterItem()
makeNewSale()
makePayment()

makeNewReturn()
enterReturnItem()
. . .

Register

...

endSale()
enterItem()
makeNewSale()
makePayment()

makeNewReturn()
enterReturnItem()
. . .

system operations discovered during system behavior analysis

allocation of system operations during design, using one facade controller

System

endSale()
enterItem()
makeNewSale()
makePayment()

enterReturnItem()
makeNewReturn()
. . .

ProcessSale Handler

...

endSale()
enterItem()
makeNewSale()
makePayment()

HandleReturns Handler

...

enterReturnItem()
makeNewReturn()
. . .

allocation of system operations during design, using several use case controllers

5. 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 Evaluative, use this principle to evaluate alternatives

High Cohesion

Coupling and Cohesion impact each other

Consider a class **A** that does two logically different things

A is coupled to the resources necessary to accomplish the two different things

If **A** were split into two classes each performing one of the logically different tasks

Each would only be coupled to the classes necessary to accomplish its task

High Cohesion –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. **S**OLID?
- It collaborates with other objects to share the effort if the task is large

High Cohesion —Analogy

A person takes on too many unrelated responsibilities—especially ones that should properly be delegated to others—then the person is **not effective**

This is observed in some managers who have not learned how to delegate.

These people suffer from low cohesion; they are ready to become "unglued." needs to **refocus!**

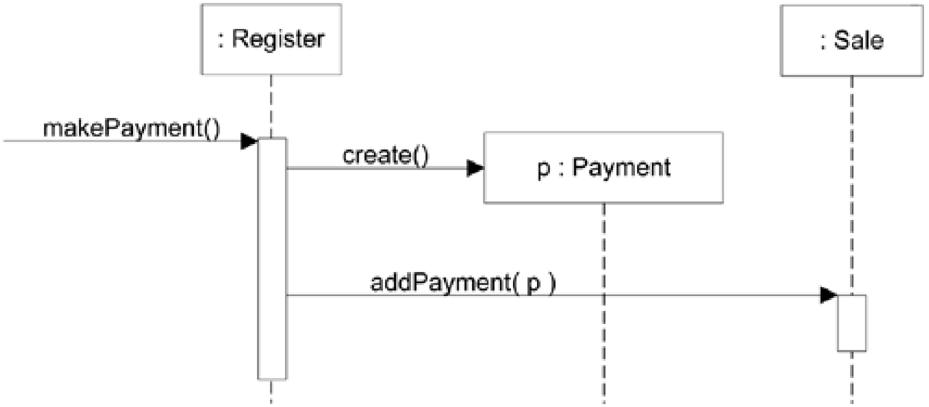
High Cohesion

Benefits:

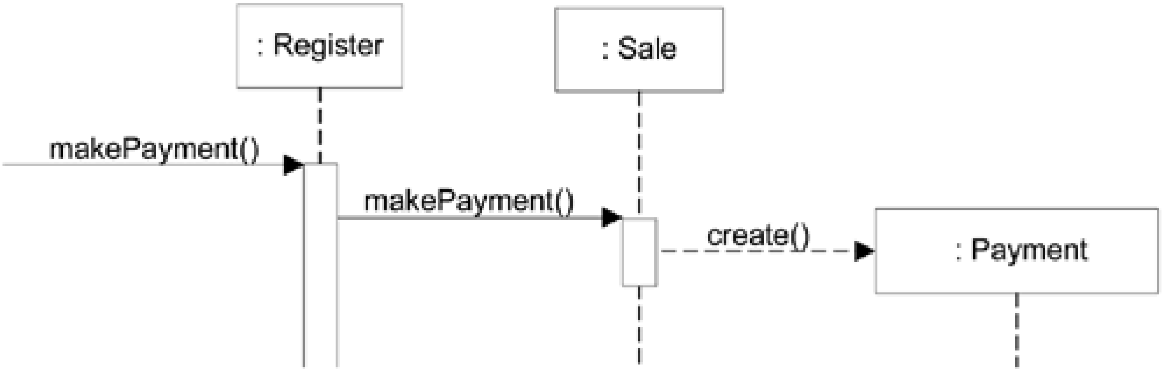
- Clarity and ease of comprehension of the design is increased
- Maintenance and enhancements are simplified
- Low coupling is often supported
- Reuse is increased

High Cohesion –POS

Register Creates Payment



Sale Creates Payment



Modular Design

Modularity is

the property of a system that has been **decomposed** into a set of **Cohesive and Loosely Coupled Modules** [Booch94].

At the basic object level, we achieve modularity by designing each **method** with a clear, single purpose and by grouping a related set of concerns into a class