

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.**

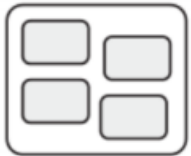
ZOOM-OUT,-IN

—What's Software Design?

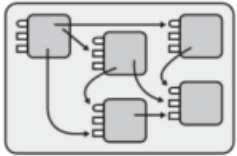
Design is needed at several different levels of detail in a system:



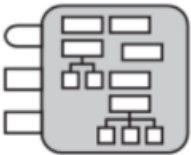
System



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**,

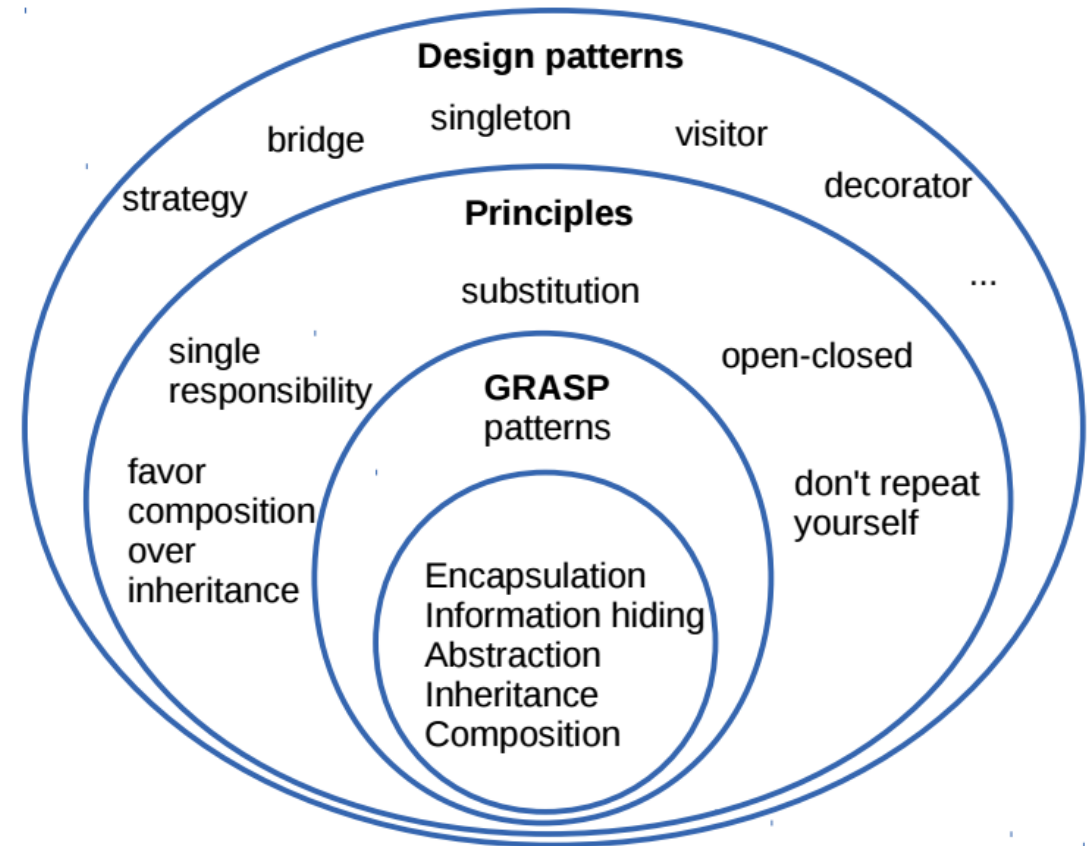
is terribly important and *anything but trivial*

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 than 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)

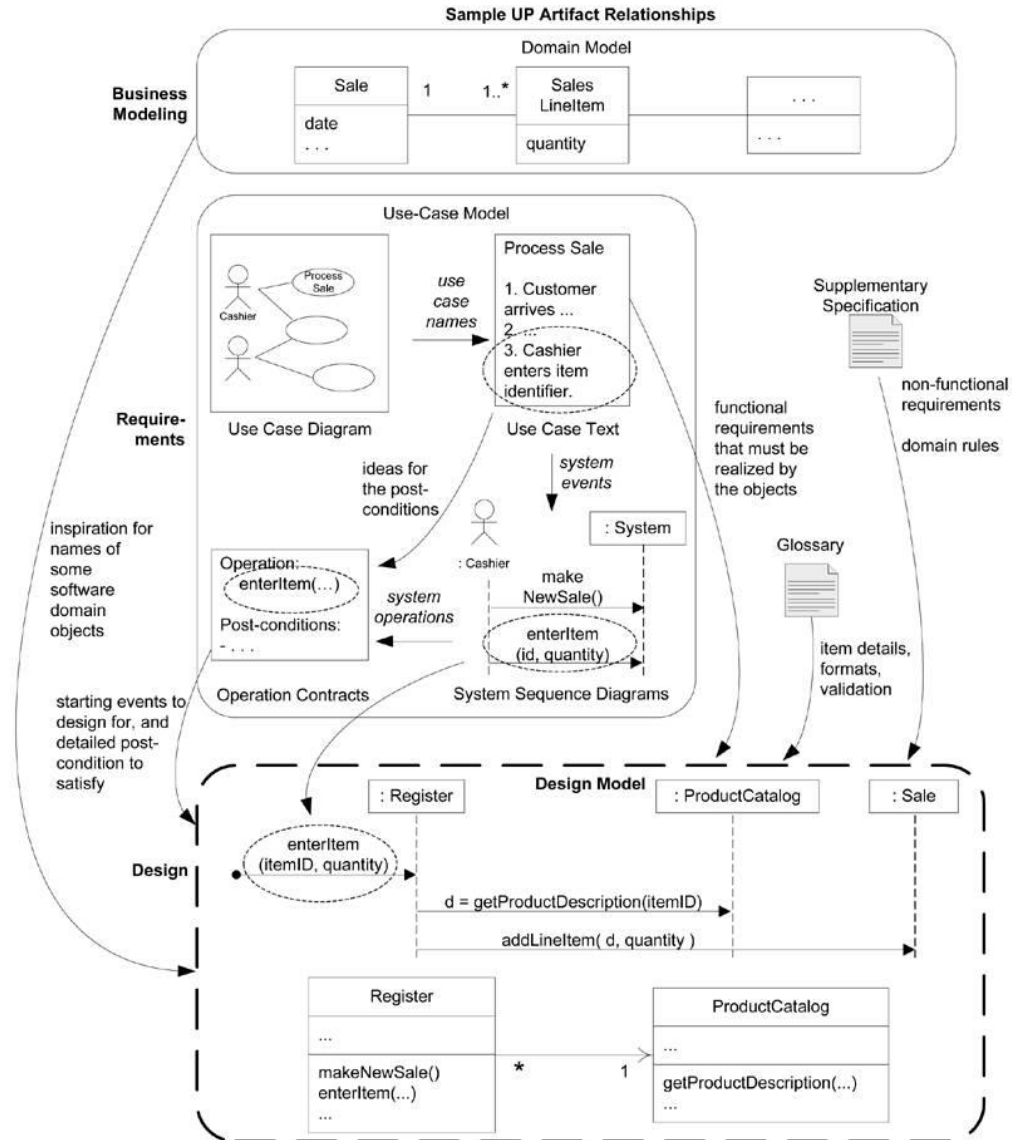
Operation Contracts

- 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 \neq 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

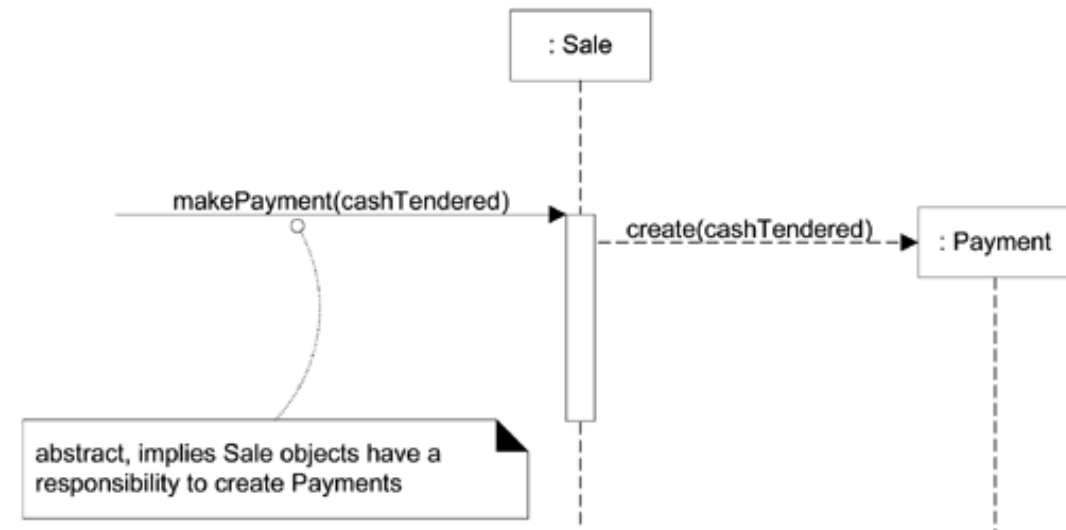
Drawing a UML interaction diagram decisions about responsibility assignments are made (**realized as methods**)

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



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
<i>Creator</i>	Who should be responsible for creating a new instance of some class?
<i>Information Expert</i>	What is a general principle of assigning responsibilities to objects?
<i>Low Coupling</i>	How to support low dependency, low change impact, and increased reuse?
<i>Controller</i>	What first object beyond the UI layer receives and coordinates ("controls") a system operation?
<i>High Cohesion</i>	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

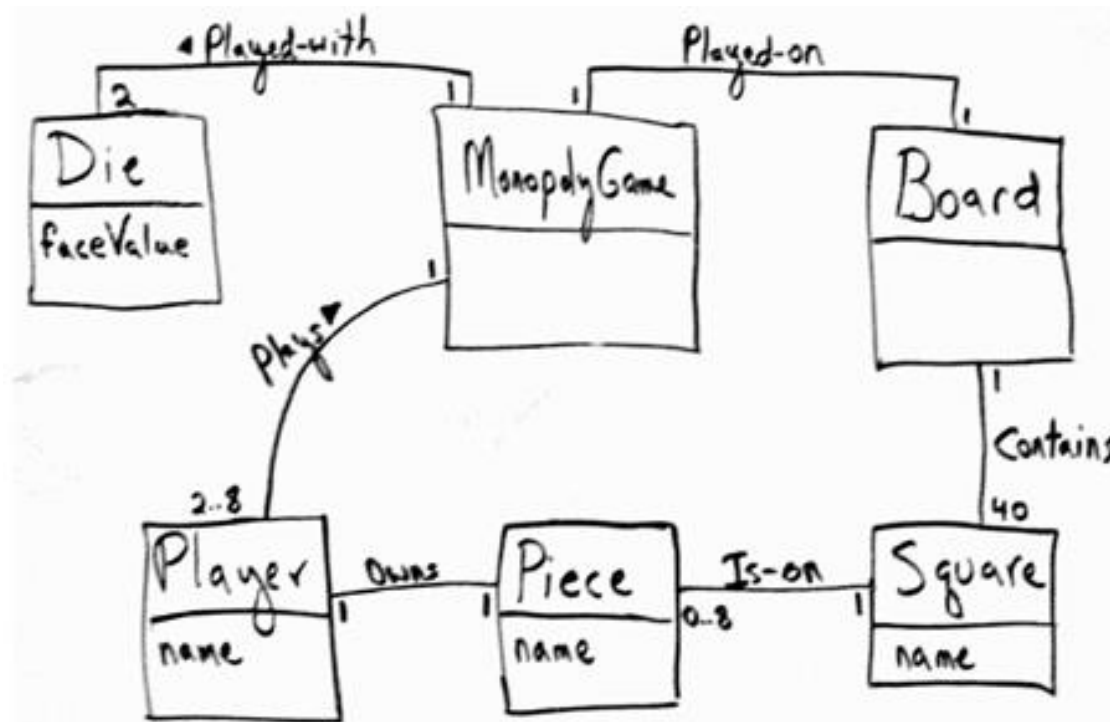
Since we are just **starting** the OO design, we haven't defined any software classes yet

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

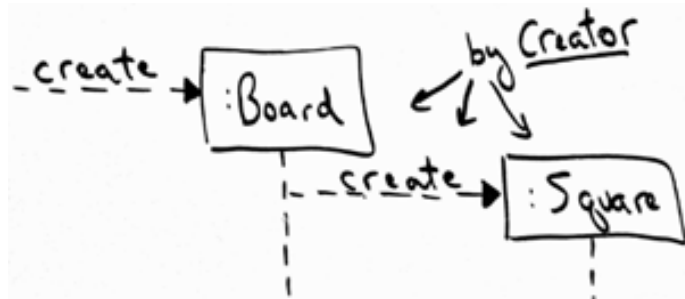
*What object should create the **Square** objects?*

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**

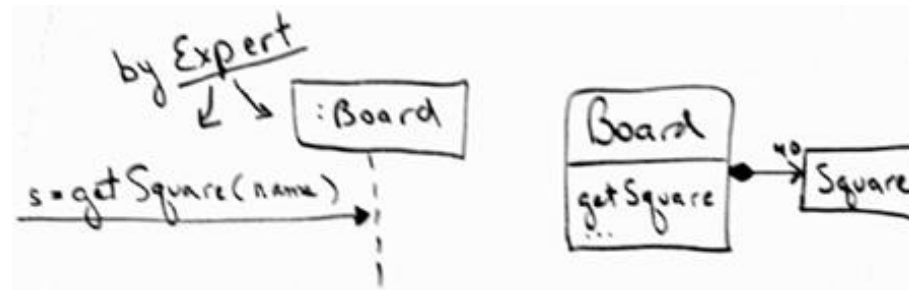
As has been previously decided, a software **Board** will **aggregate** all the **Square** objects

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)?

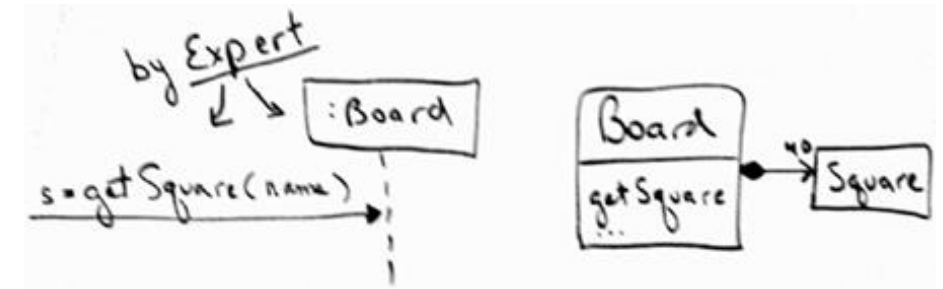
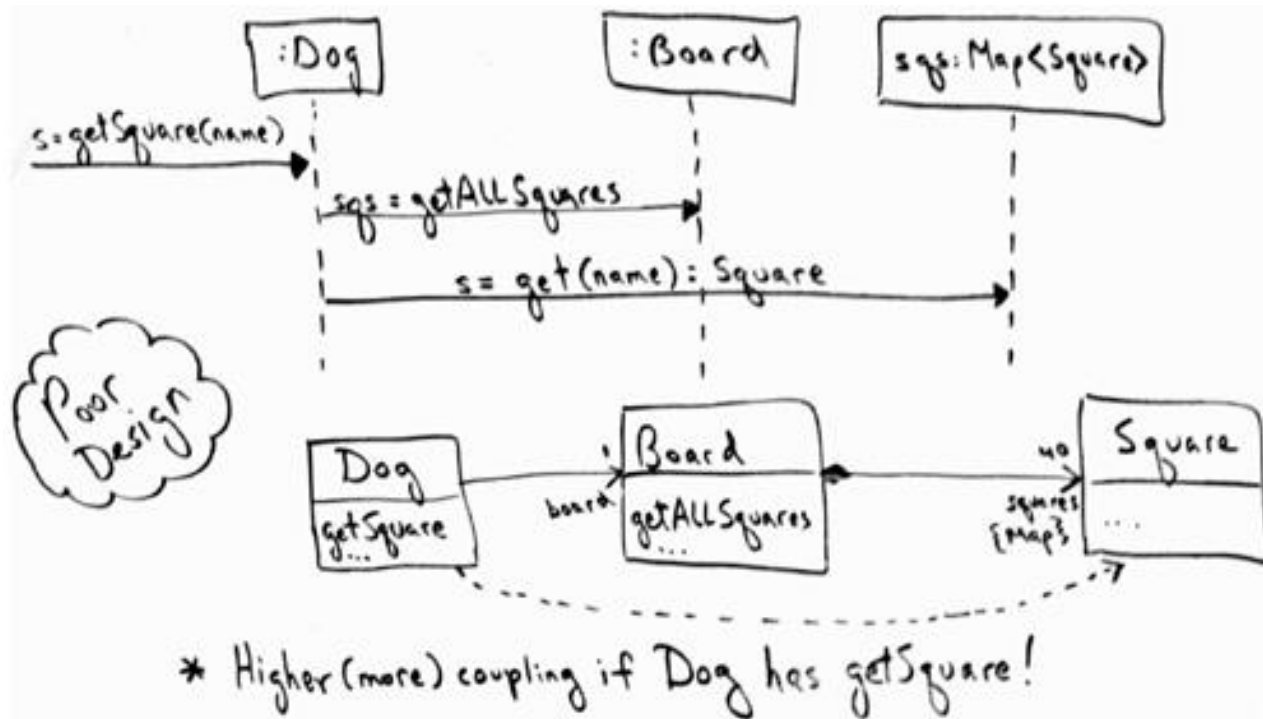
Consider the impact in terms of low coupling

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

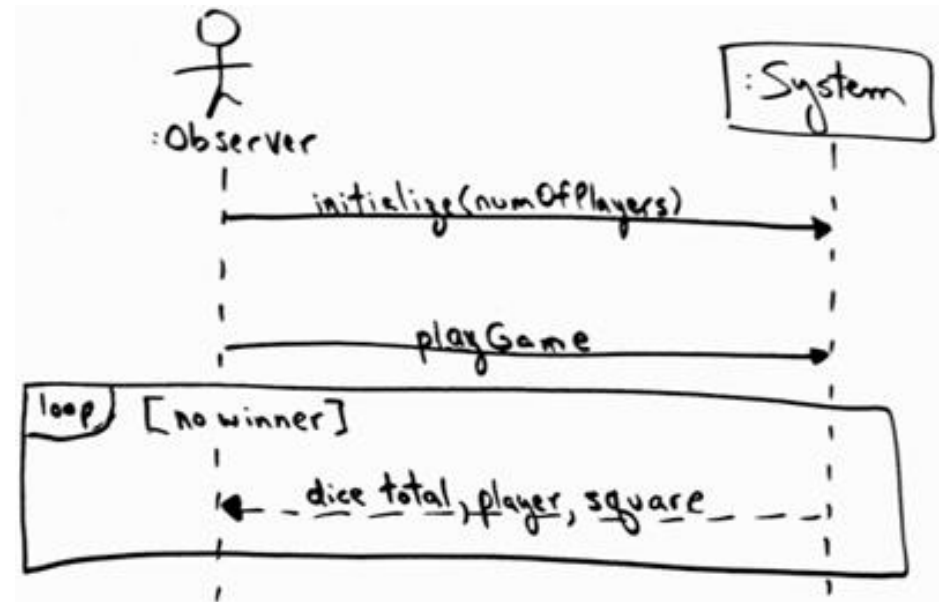
Once the UI objects **pick up the mouse event** they need to **delegate** (forward the task to another object) the request to **domain** objects in the domain layer

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

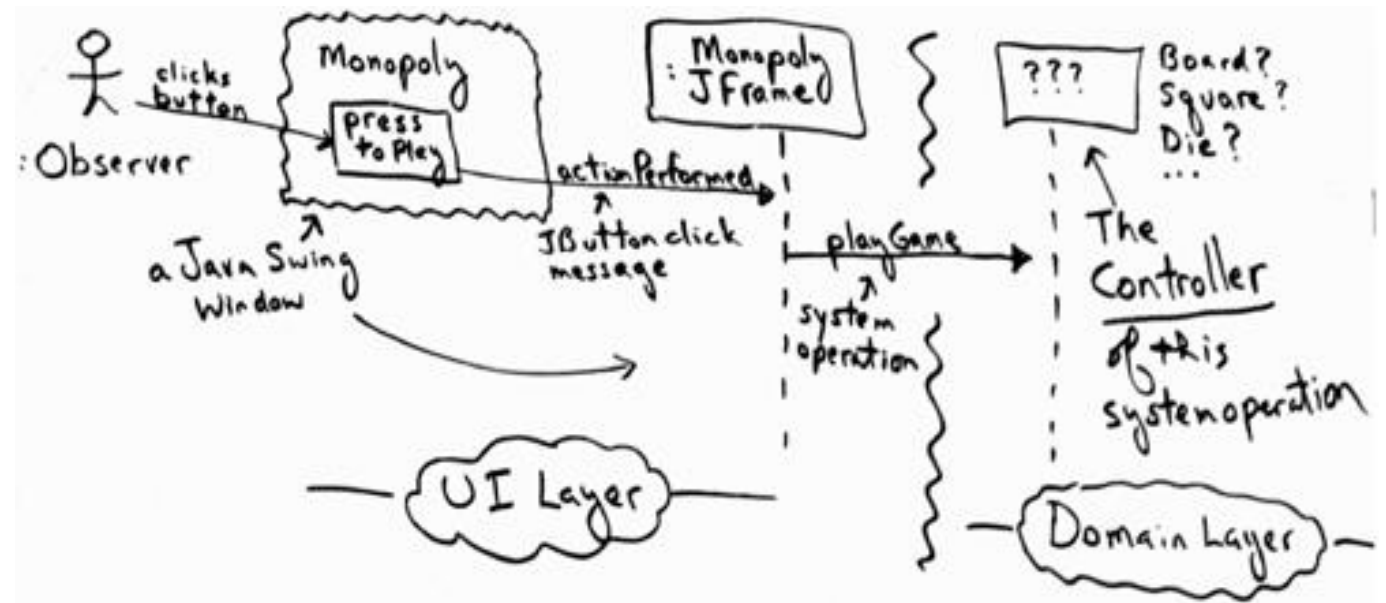


4. Controller Pattern

*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?



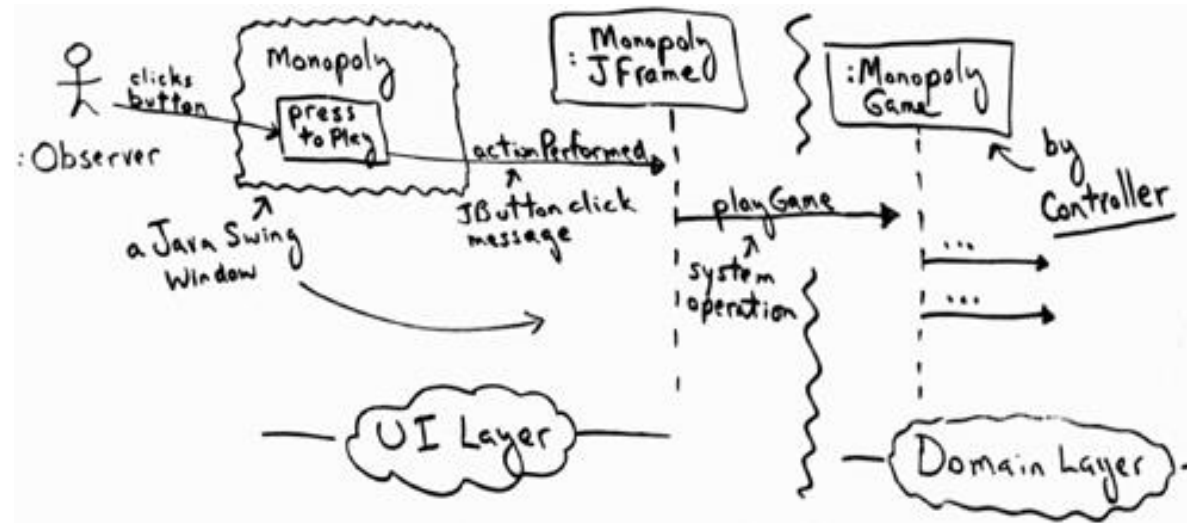
Controller Pattern

*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

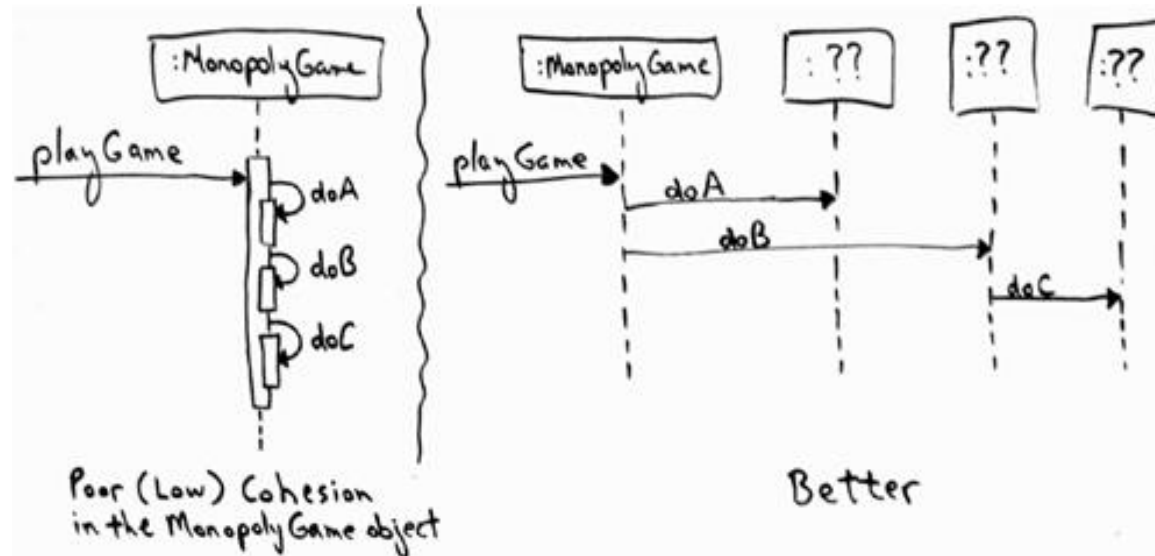


Connecting the UI layer to the domain layer of software objects

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 **General Responsibility Assignment Software Patterns**

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**