Quality, Design principles and practices

DASS – Spring 2024

IIIT Hyderabad

What is Quality ?

- It’s a really vague question !

- Quality is always relative
  - Entity of interest
  - Viewpoint
  - Attributes that we want to assess
  - Levels of abstraction

imagesource@gadgetflow.com

How do we assess quality ?

- Is the following system modifiable?
  - Background color of the user interface is changed merely by modifying a resource file.
  - Dozens of components must be changed to accommodate a new data file format.

- A reasonable answer is
  - **yes** - with respect to changing background color
  - **no** - with respect to changing file format

Quality Attributes

The totality of *features and characteristics of a product, process or service that bear on its ability to satisfy stated or implied needs*

- The features and characteristics are called as quality attributes
  - Ex: Reliability, Robustness, Efficiency, Usability, Safety, Security, Fault-tolerance, Correctness, …..
- Useful in specifying requirements and evaluation
- Can be conflicting or supportive

Quality Attributes – for Credit risk analysis (predicting NPA?)

Types of Quality Attributes

Business Qualities	• Time to Market; Project lifetime; target market; cost and benefit; projected roll out; integration with legacy; etc.
System Qualities (includes Data and Model qualities)	• Performance, Availability, Modifiability, Testability, Usability Reliability, Security, Safety, Explainability, Fairness, Interpretability, Reproducibility, etc.
Architectural Qualities	• Conceptual integrity; Correctness; Completeness; Buildability

UML Vs Design Principles

- Knowing UML doesn't mean you know design
- UML is just a notation for representing your designs
- Designs may be represented/documented using other tools/languages/technologies…

" The critical design tool for software development is a mind well educated in **design principles**…"

OO principles such as GRASP (General Responsibility Assignment Software Patterns), Gang-of-Four Design Patterns, etc. help achieve better design.

Responsibility Driven Design (RDD)

RDD – Thinking about how to assign responsibility to collaborating objects

- A *responsibility* is a contract or obligation of a class in terms of its role
- Responsibilities can be of two types
  - Doing
    - Doing something itself, such as creating an object or performing a computation
    - Initiating action in other objects
    - Controlling and coordinating activities in other objects
  - Knowing
    - Knowing about private encapsulated data
    - Knowing about related objects
    - Knowing about things it can derive or calculate

For example:

" a *Sale* object is responsible for creating *SalesLineItems* objects "

" a *Sale* object is responsible for knowing its *total*"

Note that responsibility is not the same as a method !

→ Methods fulfill responsibilities alone or in collaboration with other methods/objects

Categories of responsibilities

- Setting and getting the values of attributes
- Creating and initializing new instances
- Loading to and saving from persistent storage
- Destroying instances
- Adding and deleting links of associations
- Copying, converting, transforming, transmitting or outputting
- Computing numerical results
- Navigating and searching
- Other specialized work

Assigning Responsibilities – Thumb rules

- All the responsibilities of a given class should be *clearly related*.
- If a class has too many responsibilities, consider *splitting* it into distinct classes
- If a class has no responsibilities attached to it, then it is probably *useless*
- When a responsibility cannot be attributed to any of the existing classes, then a *new class* should be created

- Use **"Patterns"** if applicable

Why Patterns?

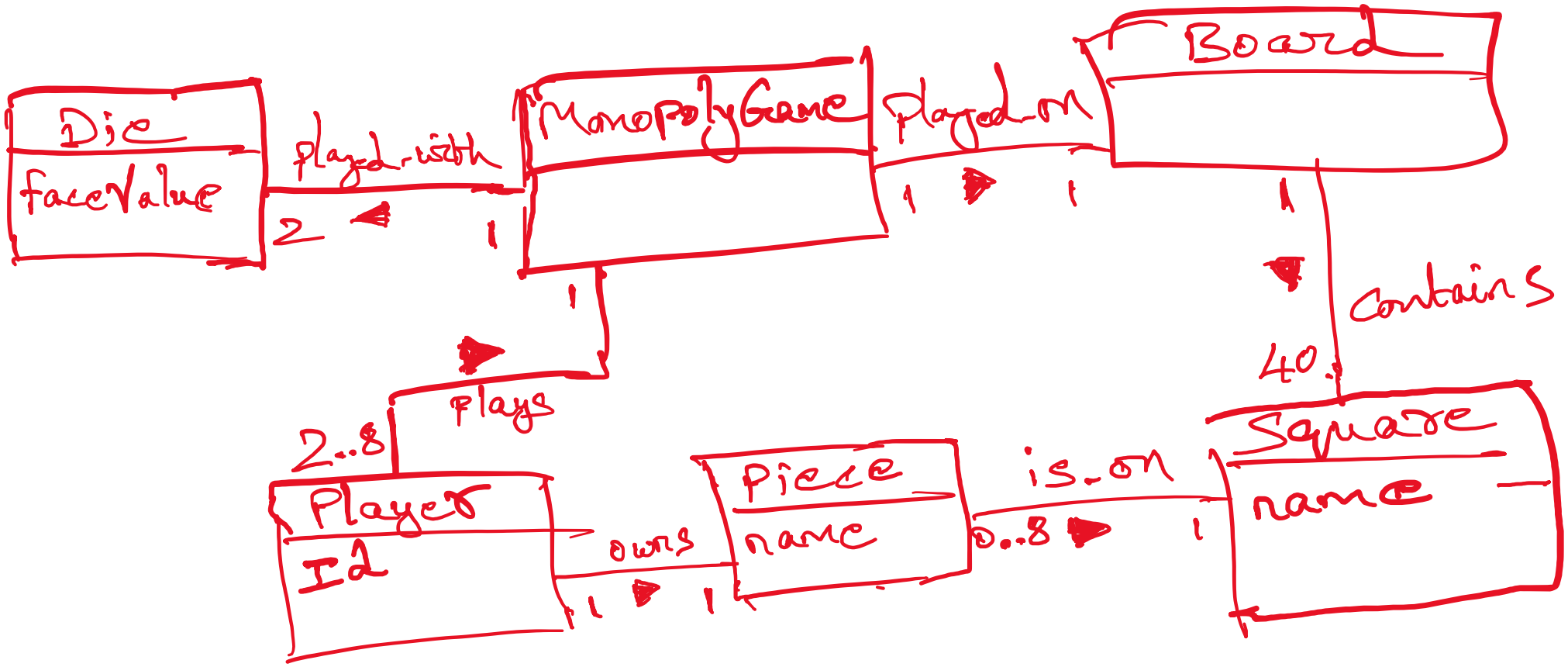
- Design for re-use is difficult
- Experienced designers:
  - Rarely start from first principles
  - Apply a working "handbook" of approaches
- Patterns make this ephemeral knowledge available to all
- Support evaluation of alternatives at higher level of abstraction

Discussion question:

"New Pattern" is an Oxymoron

OO Design Example

Problem: Design a simple Monopoly game



In a Monopoly game, who creates the Square Object ?

GRASP – Creator

➔**Doing** responsibility

➔What would you choose? And Why?

Name: CREATOR

Problem: Who creates an A?

Solution: Assign class B the responsibility to create an instance of class A if one of these is true:

(1) B contains A

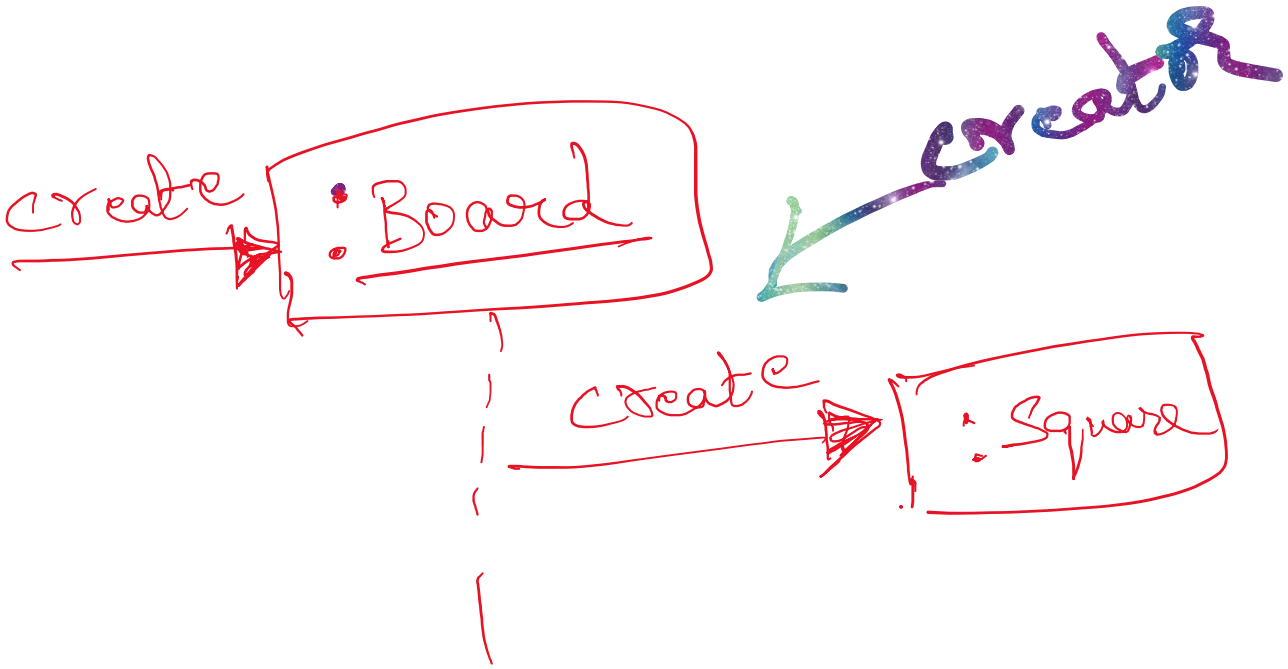
(2) B compositely aggregates A

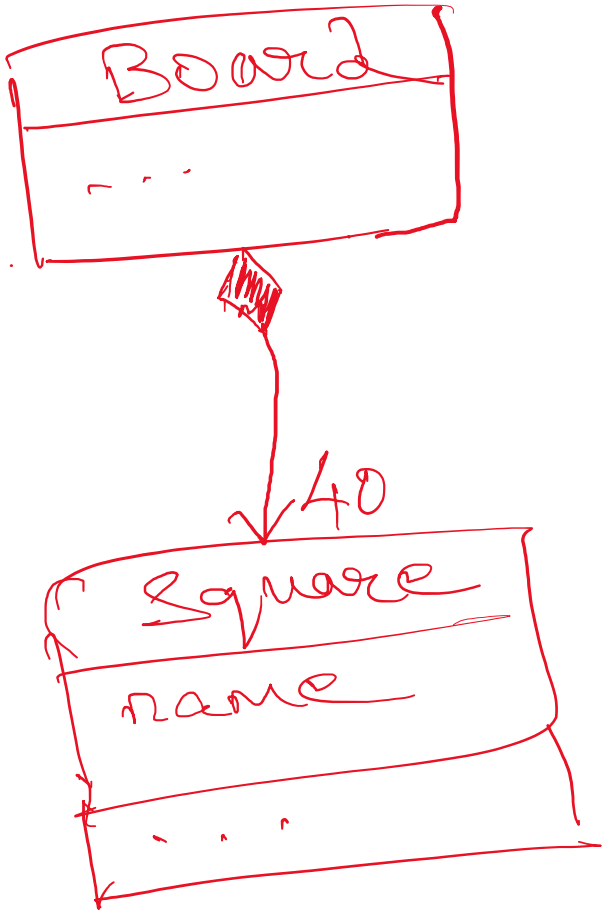
(3) B has the initializing data for A

(4) B records A

(5) B closely uses A

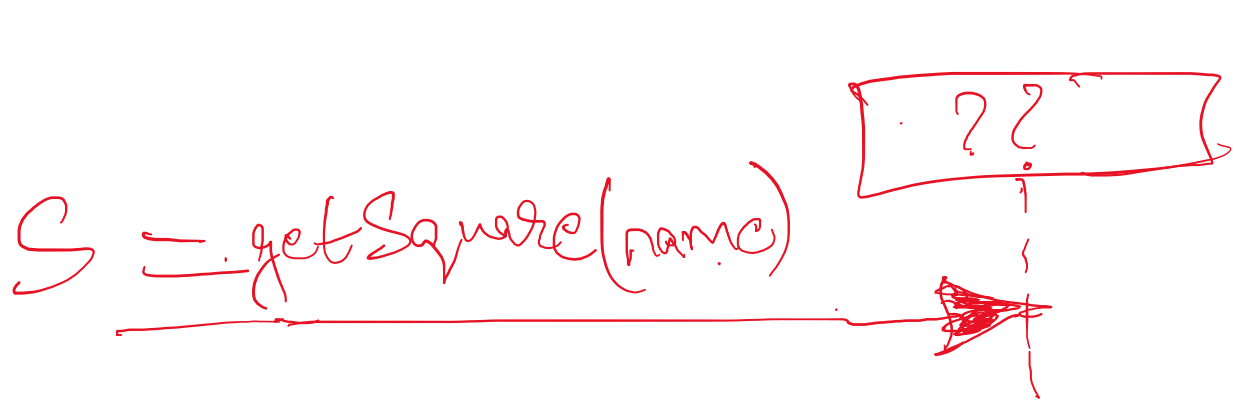
Applying creator pattern

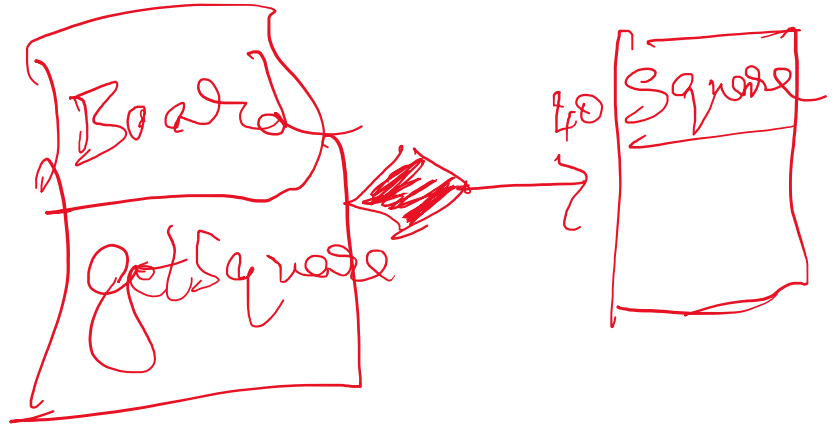




GRASP – Information Expert

Who knows about a Square object, given a key?





Name: Information Expert

Problem: What is the basic principle to assign responsibilities to objects

Solution: Assign a responsibility to the class that has the information needed to fulfill it

GRASP – Low Coupling

Coupling is a measure of how strongly one element is connected to, has knowledge of, or depends on other elements.

Ex. Subclass is strongly coupled with Superclass

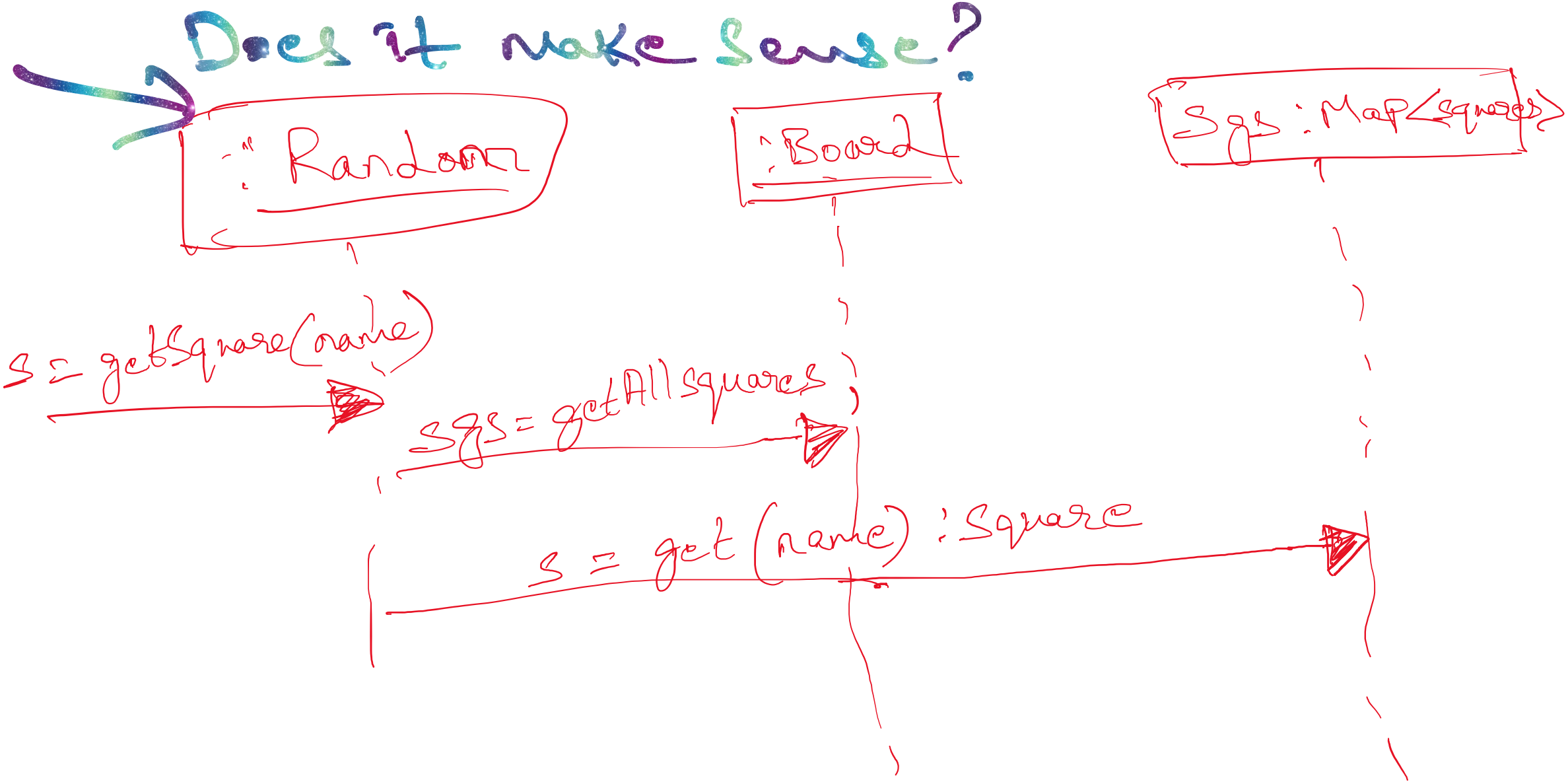
Object A that invokes methods of Object B has coupling to B's services

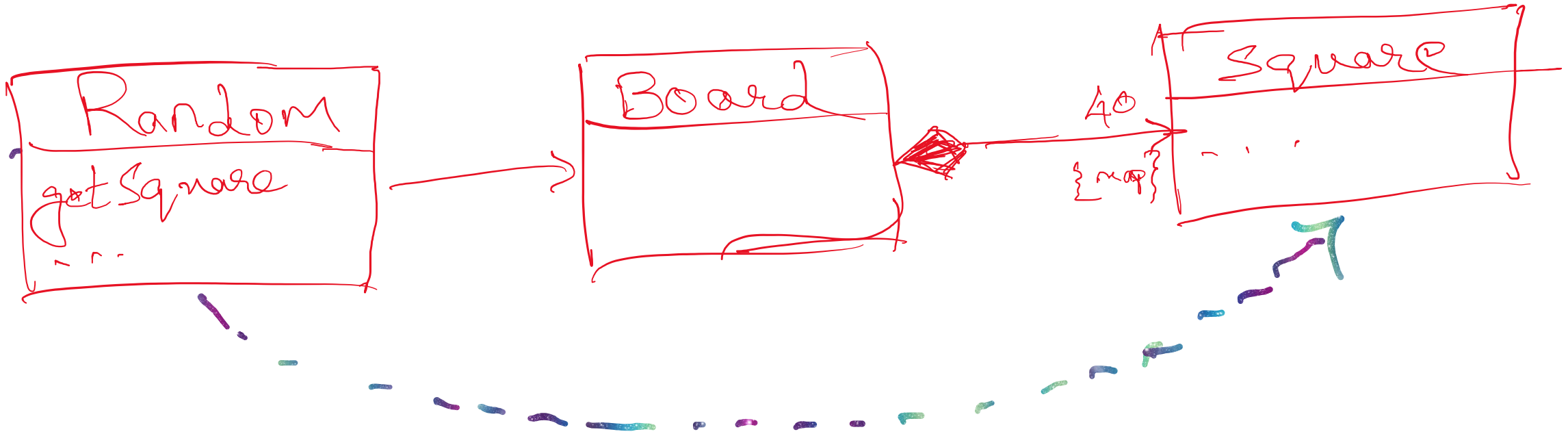
Monopoly Example:

Why Board over some other random object?

→ Information expert guides us to assign the responsibility to know a particular Square, given a unique name.

Does it make sense?





Poor design
(HIGH COUPLING?)

GRASP – Low coupling

Name: Low coupling

Problem: How to reduce the impact of change?

Solution: Assign responsibilities so that (unnecessary) coupling remains low. Use this principle to evaluate alternatives.

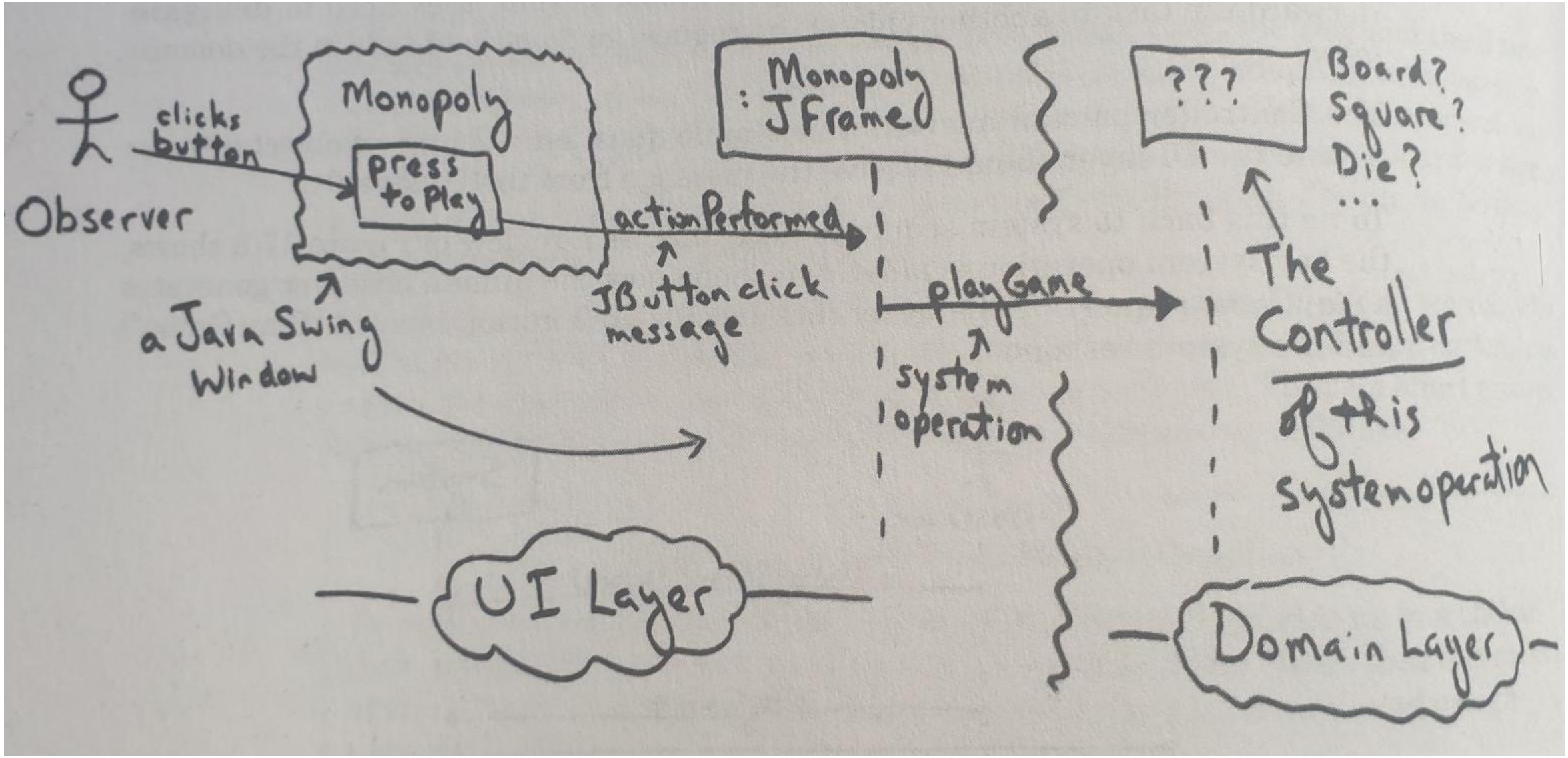
GRASP - Controller

Controller deals with the basic question of how to connect UI layer to the application logic layer.

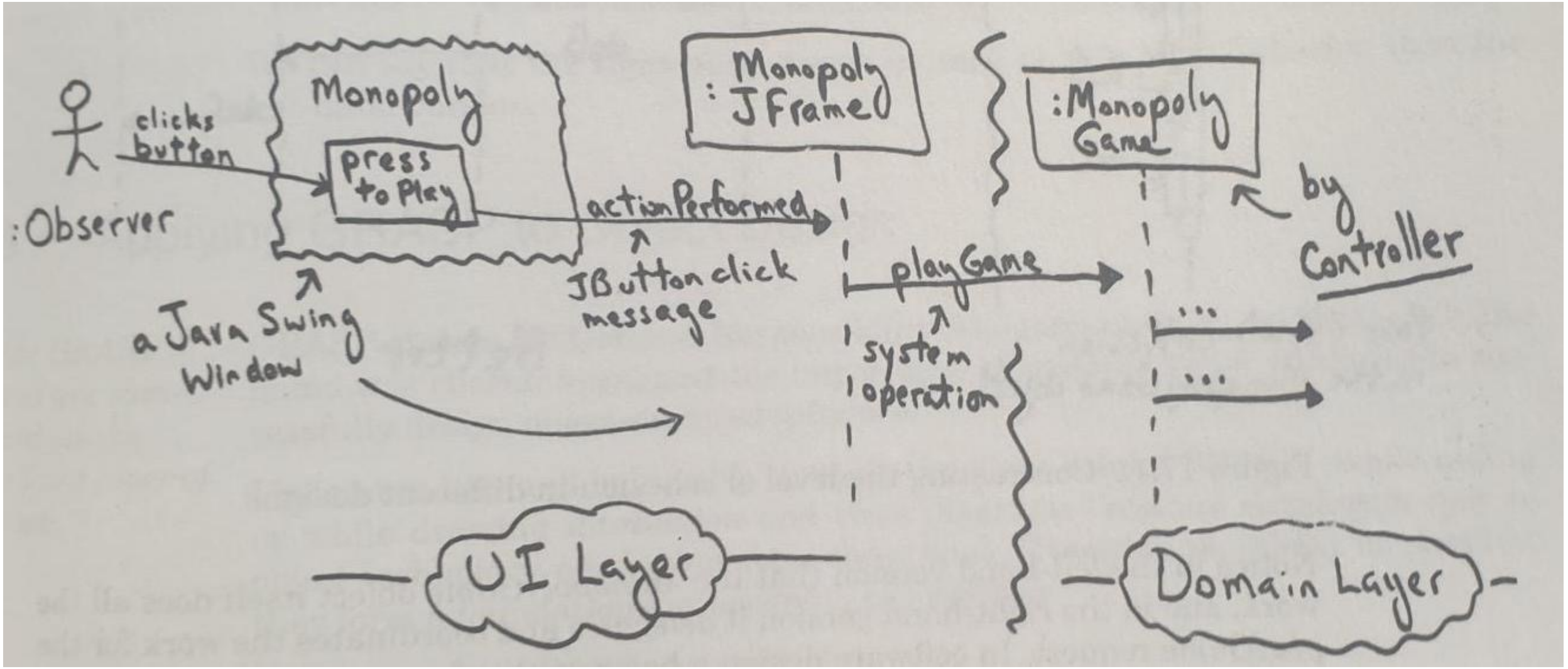
Model-View-Controller (MVC) Pattern

- The *Model* component: encapsulates core functionality; independent of input/output representations and behavior.
- The *View* components: displays data from the model component; there can be multiple views for a single model component.
- The *Controller* components: each view is associated with a controller that handles inputs; the user interacts with the system via the controller components.

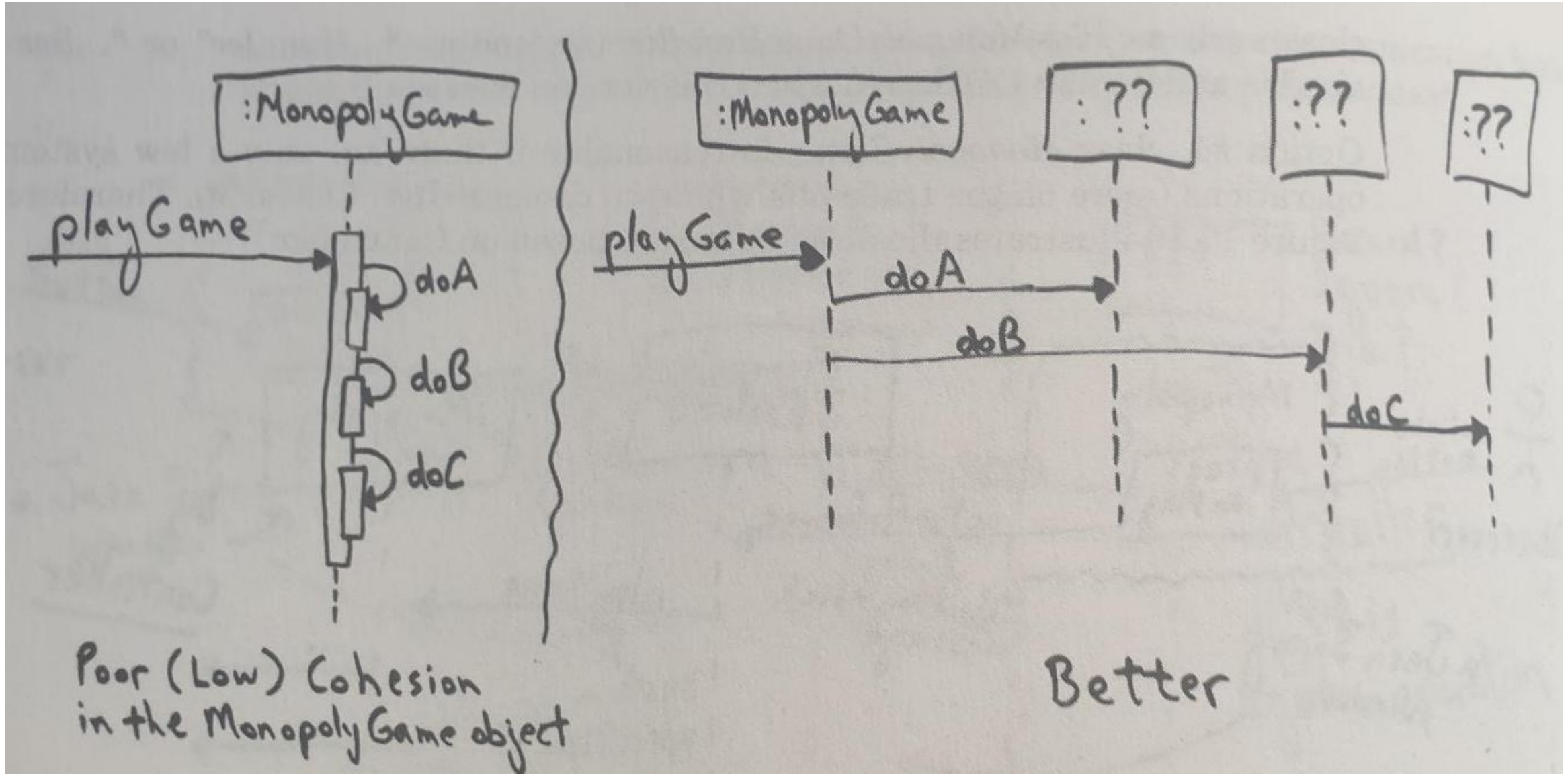
Monopoly example



Monopoly example – with Controller



GRASP – High Cohesion



GRASP – other patterns/principles

Polymorphism: Who is responsible when the behavior varies by type?

When related alternatives or behaviors vary by type (class), assign responsibility for the behavior – using polymorphic operations – to the types for which the behavior varies

Pure fabrication: Who is responsible when you are desperate, and do not want to violate high cohesion and low coupling?

Assign a highly cohesive set of responsibilities to an artificial or convenience "behavior" class that does not represent the problem domain concept – something made up, in order to support high cohesion, low coupling, and reuse.

Indirection: How to assign responsibilities to avoid direct coupling?

Assign the responsibility to an intermediate object to mediate between other components or services, so that they are not directly coupled.

Protected Variations: How to assign responsibilities to objects, subsystems, and systems so that the variations or instability in these elements do not have an undesirable impact on other elements?

Identify points of predicted variations or instability; assign responsibilities to create a stable "interface" around them

Pattern Types

- **Requirements Patterns**: Characterize families of requirements for a family of applications
  - The checkin-checkout pattern can be used to obtain requirements for library systems, car rental systems, video systems, etc.

- **Architectural Patterns**: Characterize families of architectures
  - The Broker pattern can be used to create distributed systems in which location of resources and services is transparent (e.g., the WWW)
  - Other examples: MVC, Pipe-and-Filter, Multi-Tiered

- **Design Patterns**: Characterize families of low-level design solutions
  - Examples are the popular Gang of Four (GoF) patterns

- **Programming idioms**: Characterize programming language specific solutions