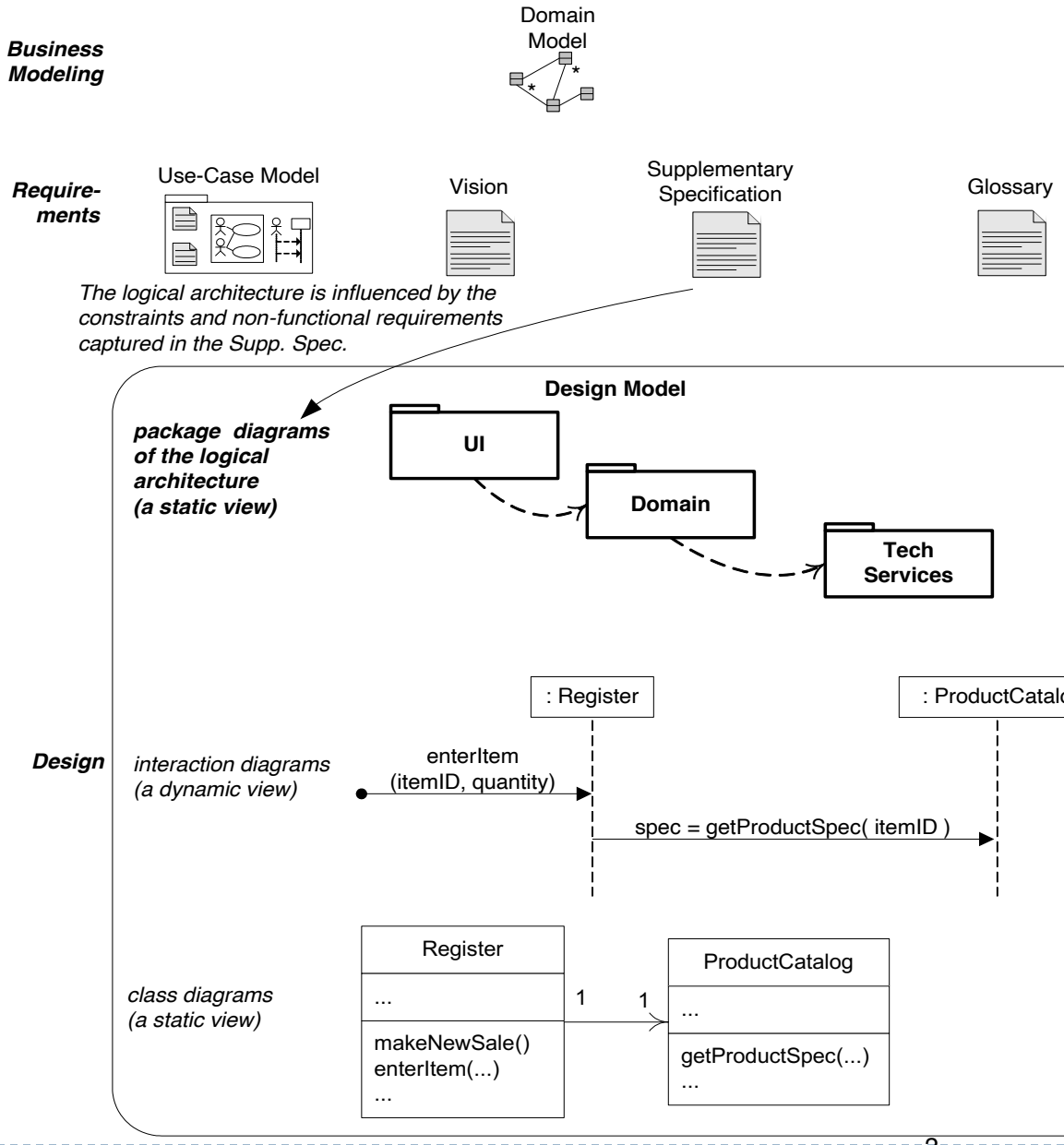
# Software Design

**Sample UP Artifact Relationships**



# Different Design Aspects

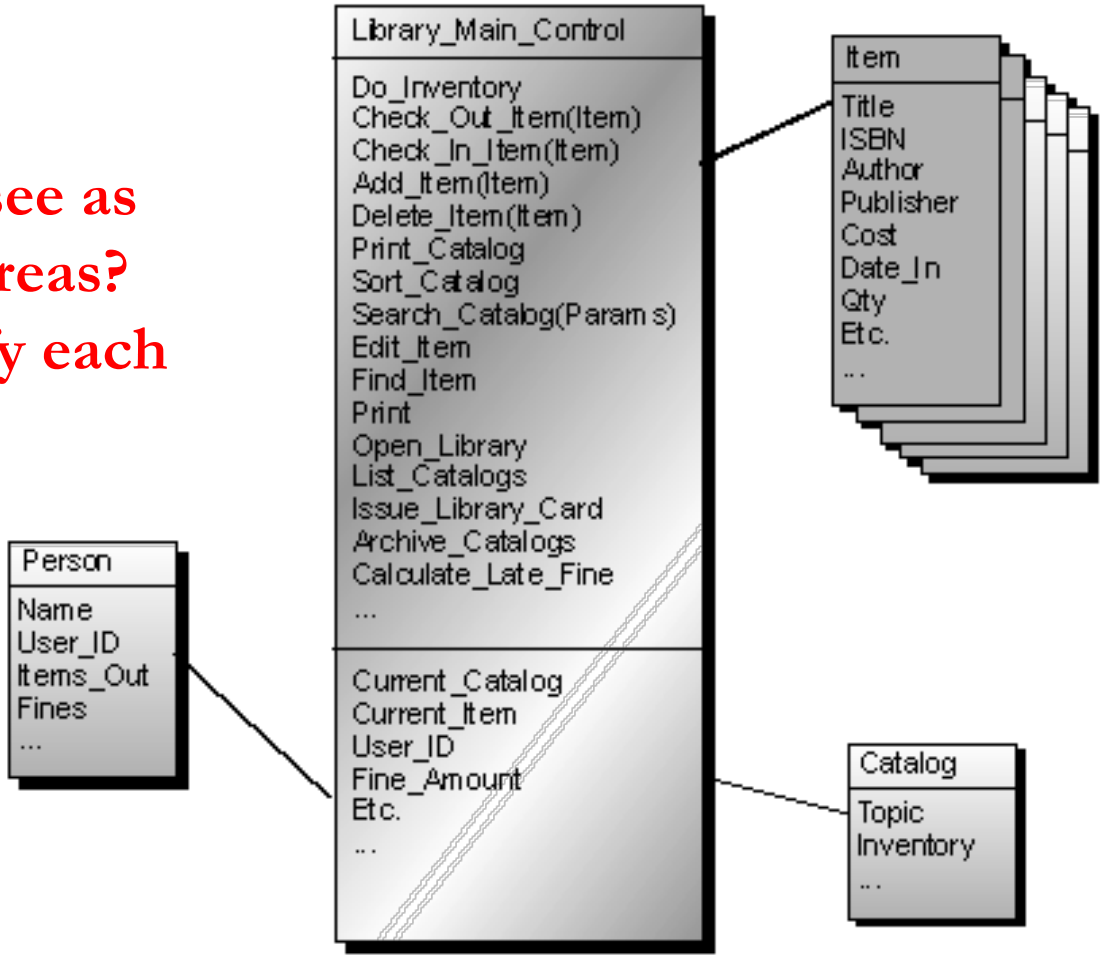
- *Architecture design*:
  - The division into subsystems and components,
    - How these will be connected.
    - How they will interact.
    - Their interfaces.
- *Class design*:
  - The various features of classes.
- *User interface design*
- *Algorithm design*:
  - The design of computational mechanisms.
- *Protocol design*:
  - The design of communications protocol.

# What is Software Design?

- A software design expresses a solution to a problem in programming language independent terms.
- This permits a design to be implemented in any programming language.

# Simple Library system - Existing design

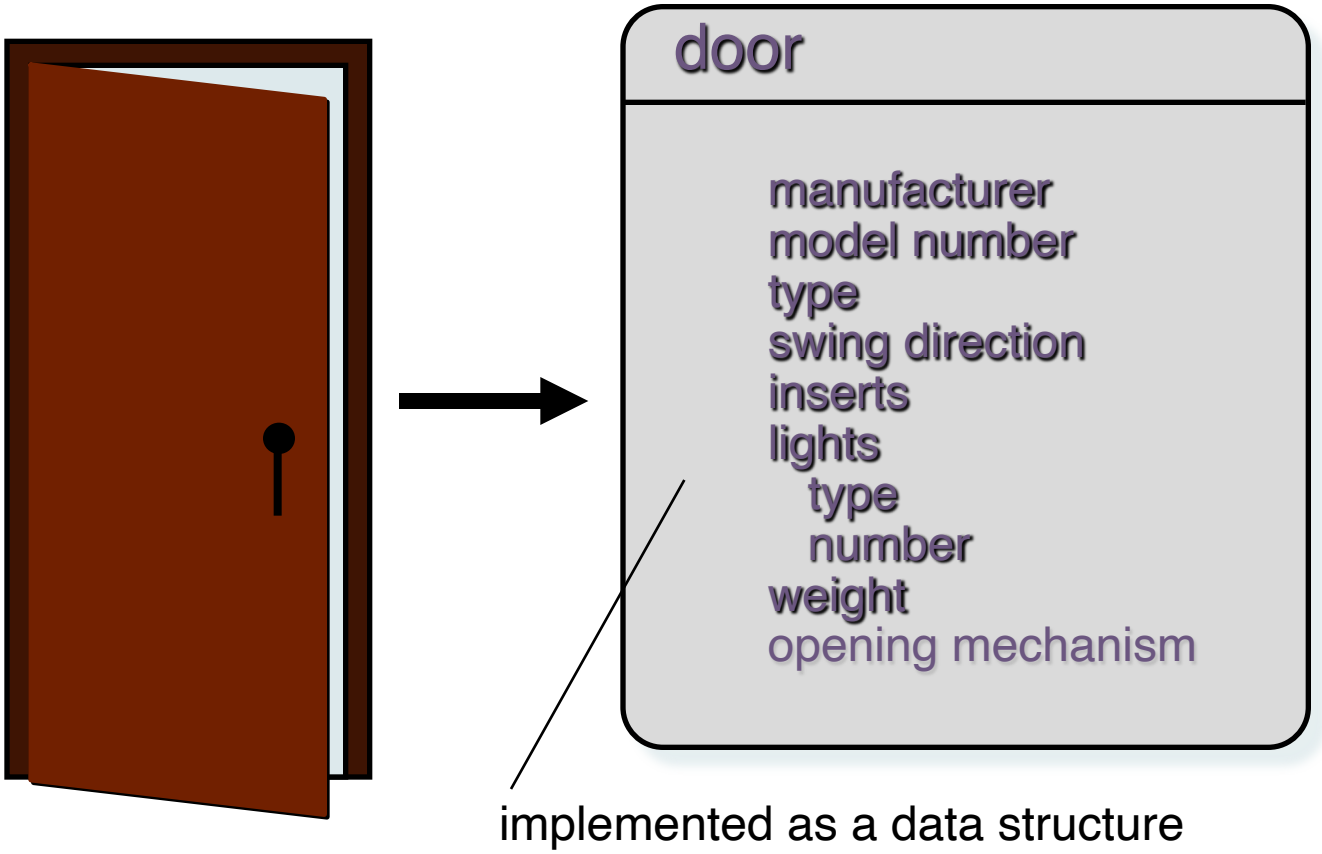
**What areas do you see as potential problem areas? Why did you identify each of those areas?**



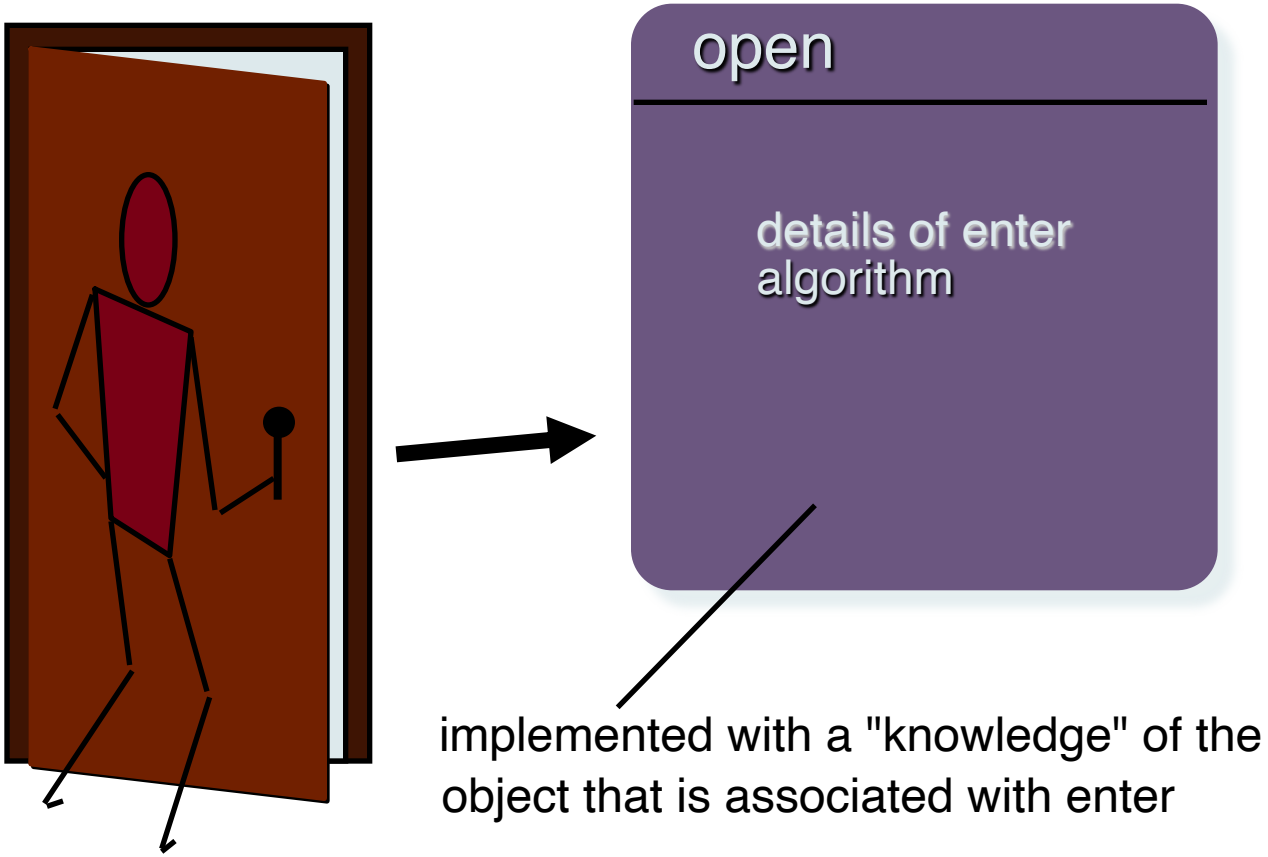
# Fundamental Concepts

- Abstraction—data, procedure, control
- Separation of concerns—any complex problem can be more easily handled if it is subdivided into pieces
- Modularity—compartmentalization of data and function
- Hiding—controlled interfaces
- Refinement—elaboration of detail for all abstractions
- Design Classes—provide design detail that will enable analysis classes to be implemented
- Functional independence—High Cohesion and Low coupling
- **Patterns**—"conveys the essence" of a proven design solution

# Data Abstraction



# Procedural Abstraction



# Separation of Concerns

- Any complex problem can be more easily handled if it is **subdivided into pieces** that can each be solved and/or optimized independently
- A *concern* is a feature or behavior that is specified as part of the requirements model for the software
- By separating concerns into smaller, and therefore more manageable pieces, a problem takes less effort and time to solve.

# Modularity

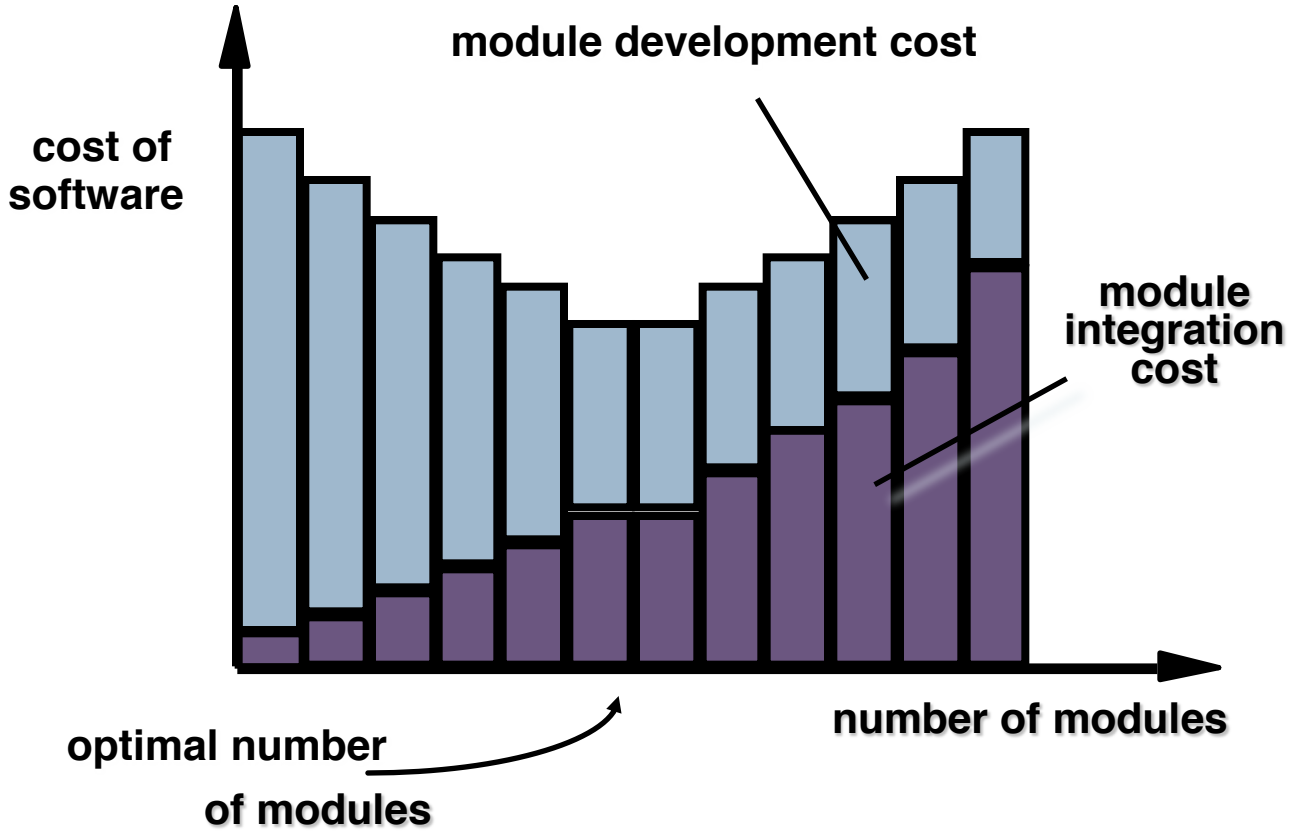
- "modularity is the single attribute of software that allows a program to be intellectually manageable" [Mye78].

- **Monolithic software (i.e., a large program composed of a single module) cannot be easily grasped by a software engineer.**

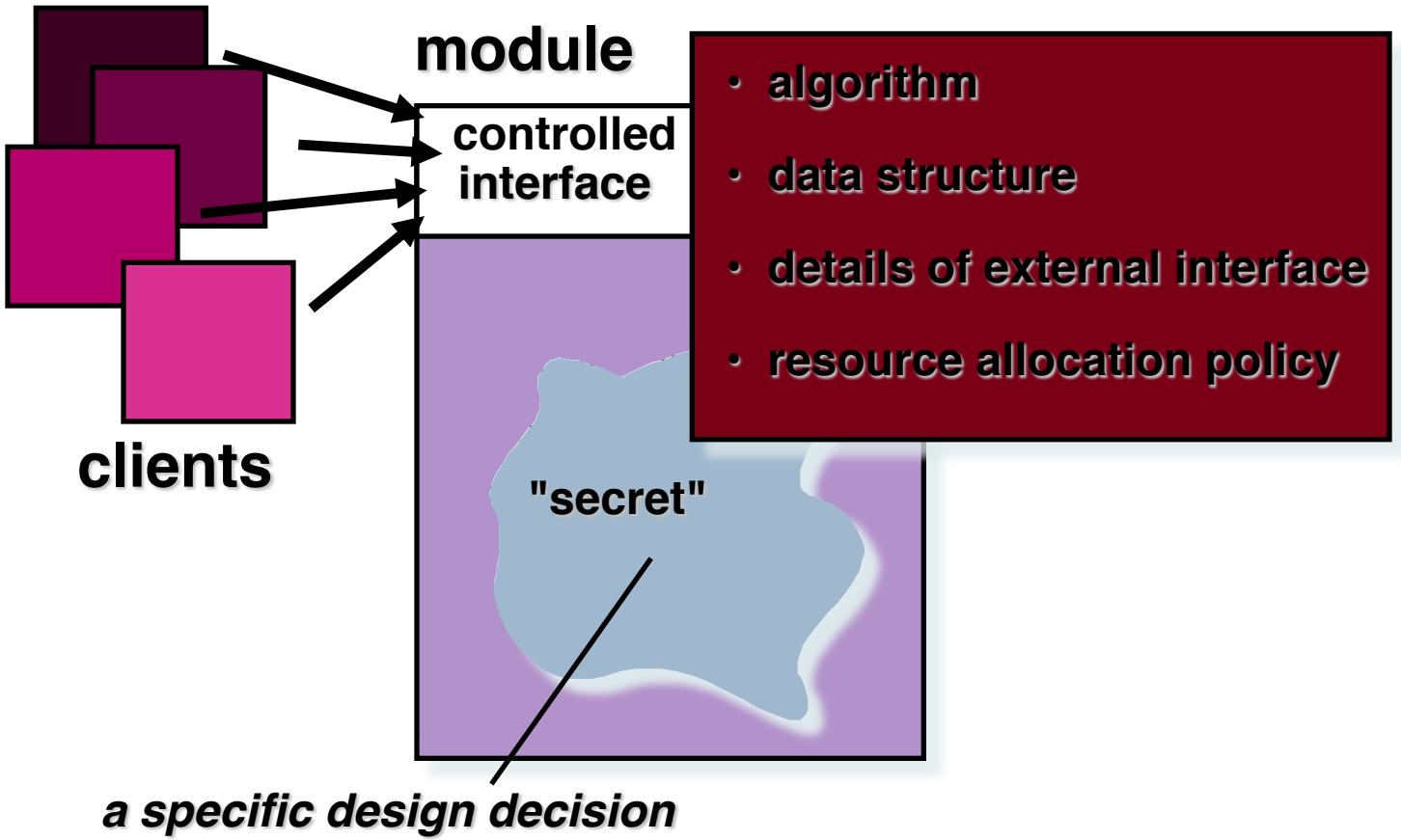
- In almost all instances, you should break the design into many modules, hoping to make understanding easier and as a consequence, reduce the cost required to build the software.

# Modularity: Trade-offs

***What is the "right" number of modules for a specific software design?***



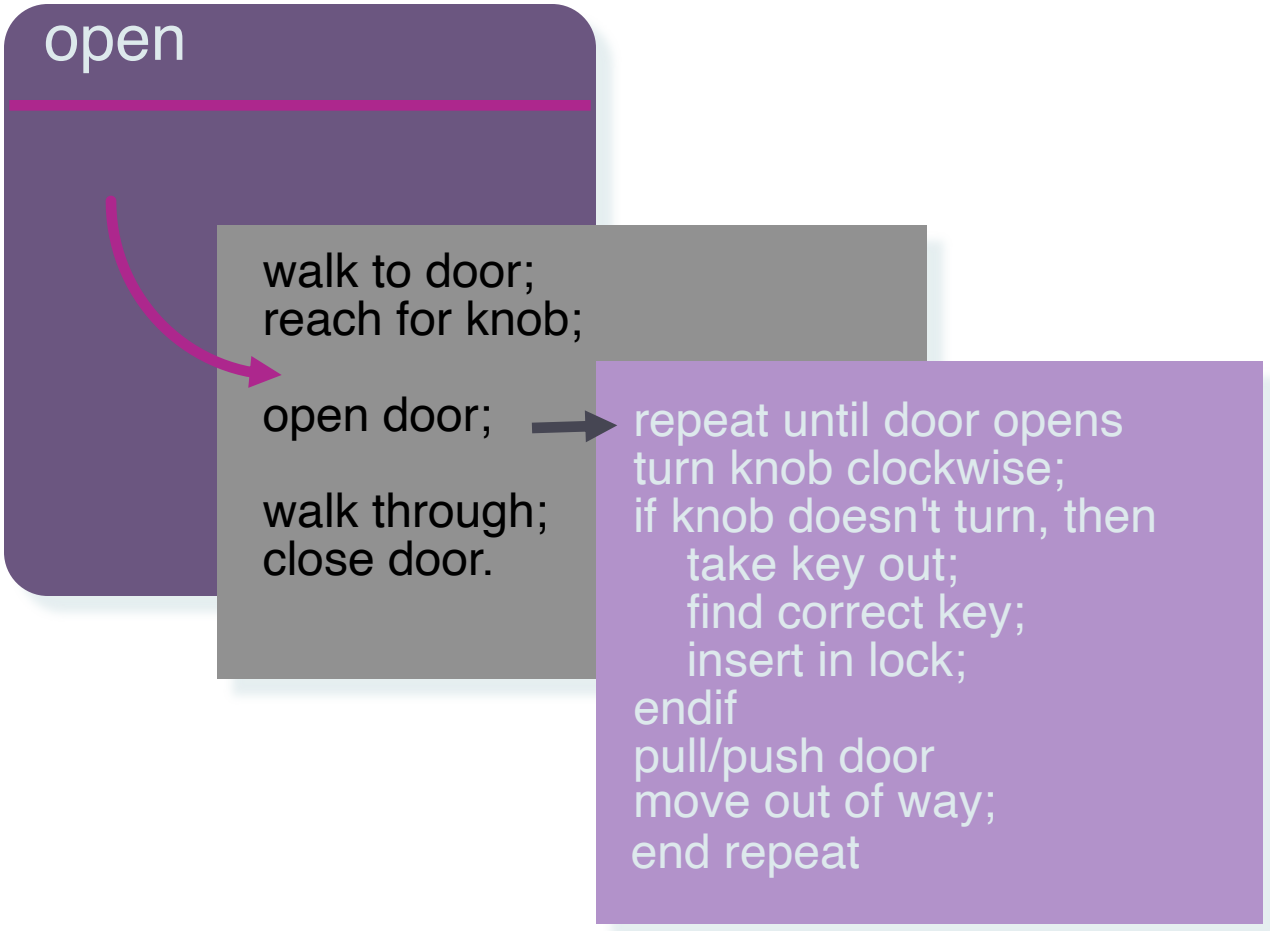
# Information Hiding



# Why Information Hiding?

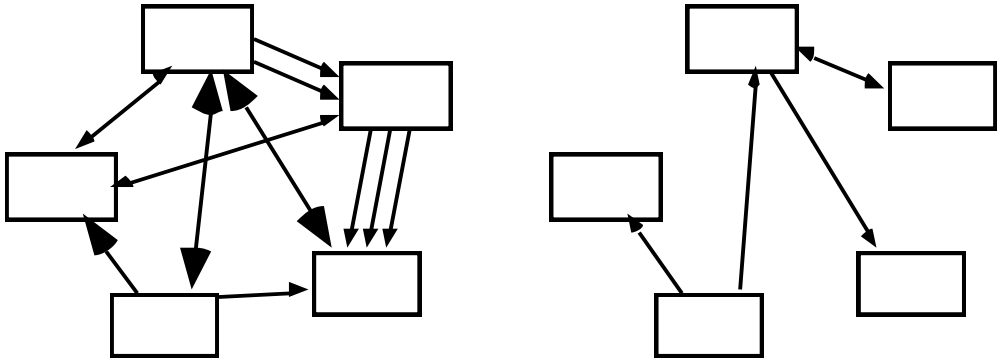
- reduces the likelihood of "side effects"
- limits the global impact of **local design** decisions
- emphasizes communication through **controlled interfaces**
- discourages the use of global data
- leads to **encapsulation**—an attribute of high quality design
- results in higher quality software

# Stepwise Refinement



# Coupling

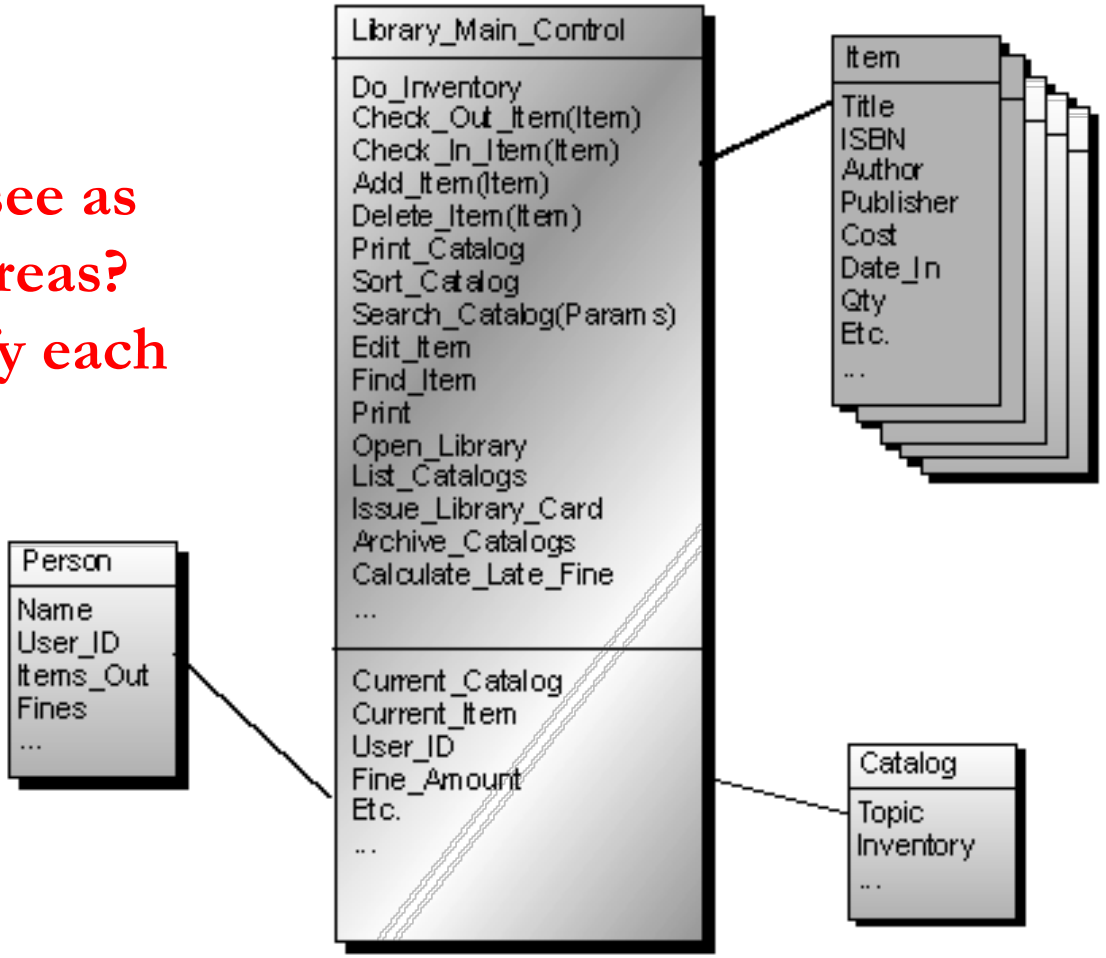
- *Coupling* occurs when there are *interdependencies* between one module and another



  - When interdependencies exist, changes in one place will require changes somewhere else.
  - A network of interdependencies makes it hard to see at a glance how some component works.

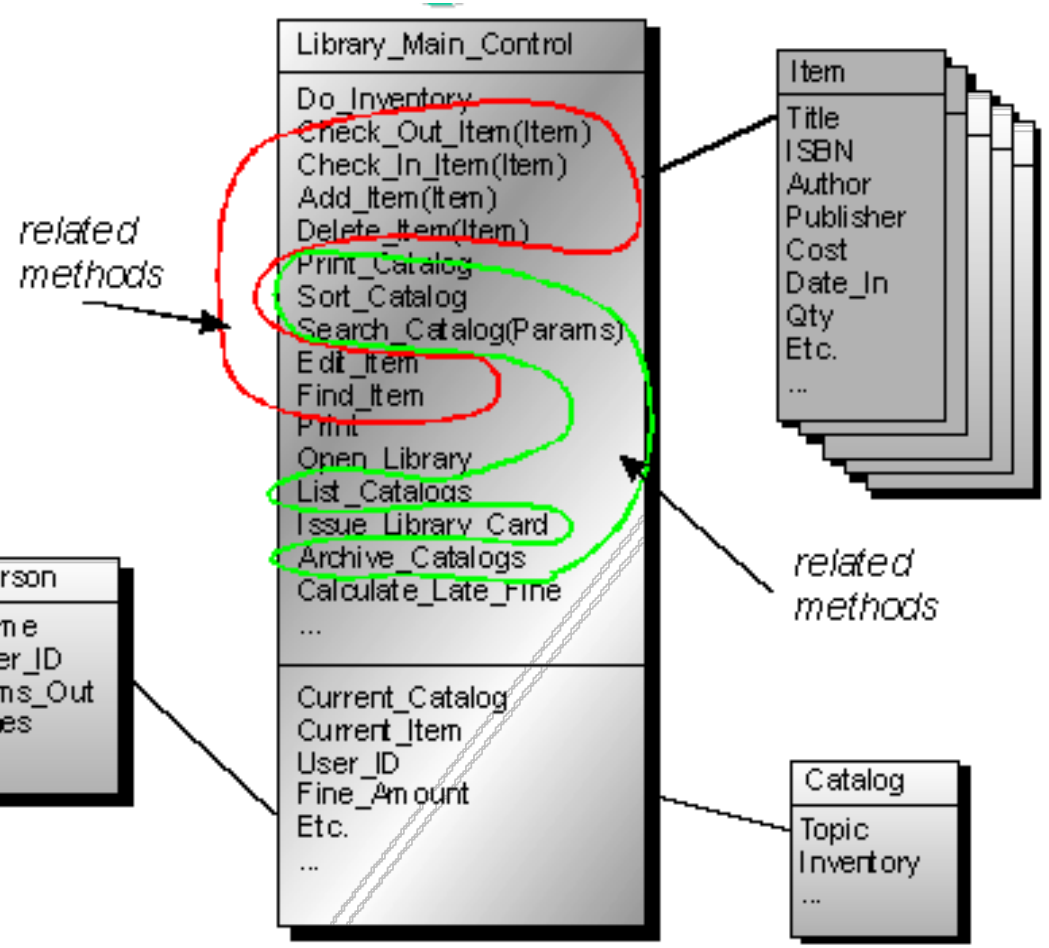
# Revisting Library system - Existing design

**What areas do you see as potential problem areas? Why did you identify each of those areas?**



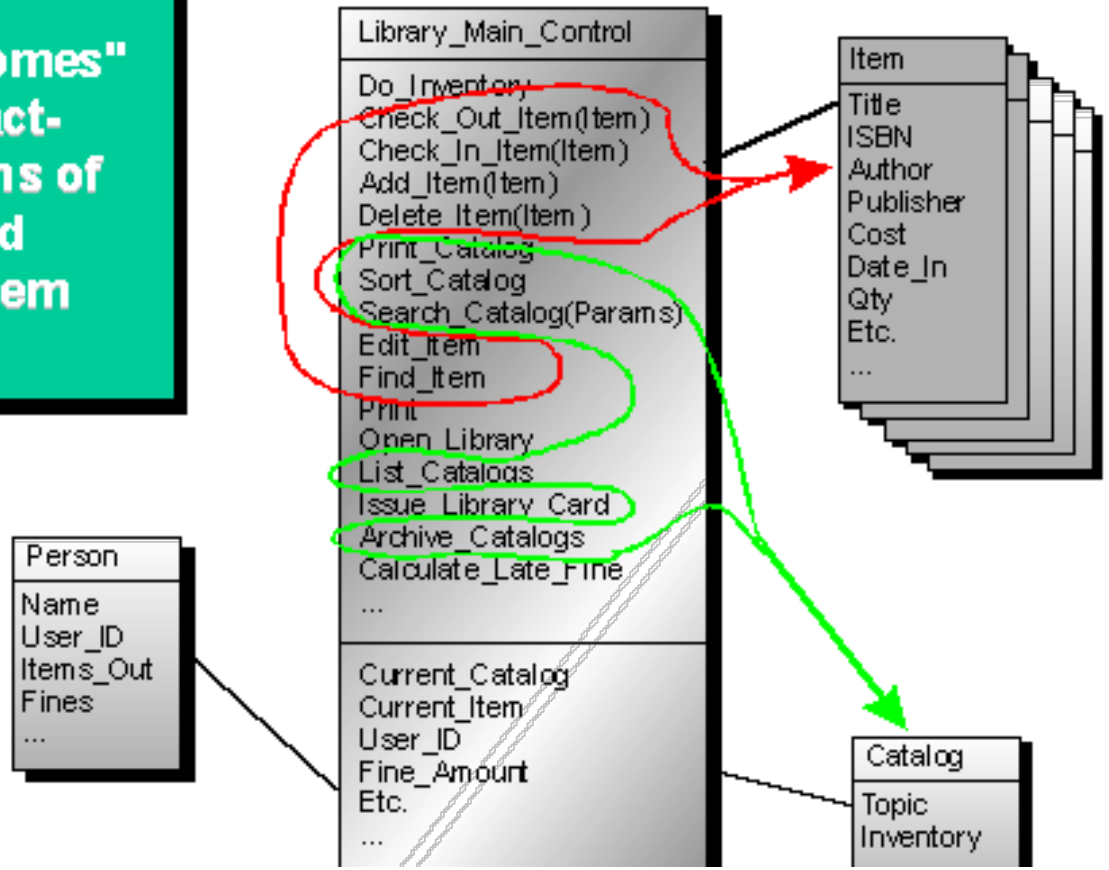
# Library system – Changing the design

**Step 1:**
**Identify or categorize related attributes and operations according to contracts.**

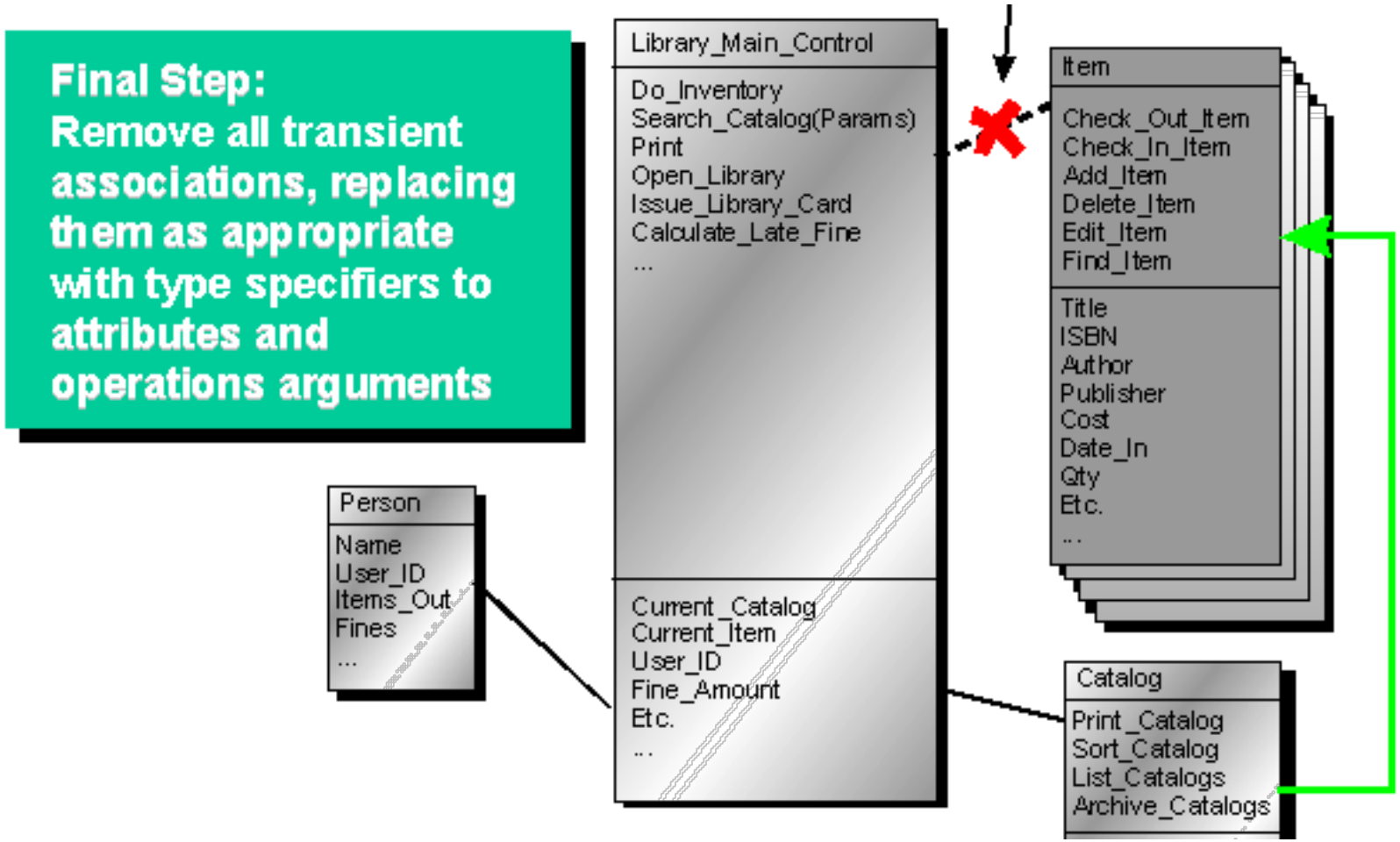


# Library system – Changing the design

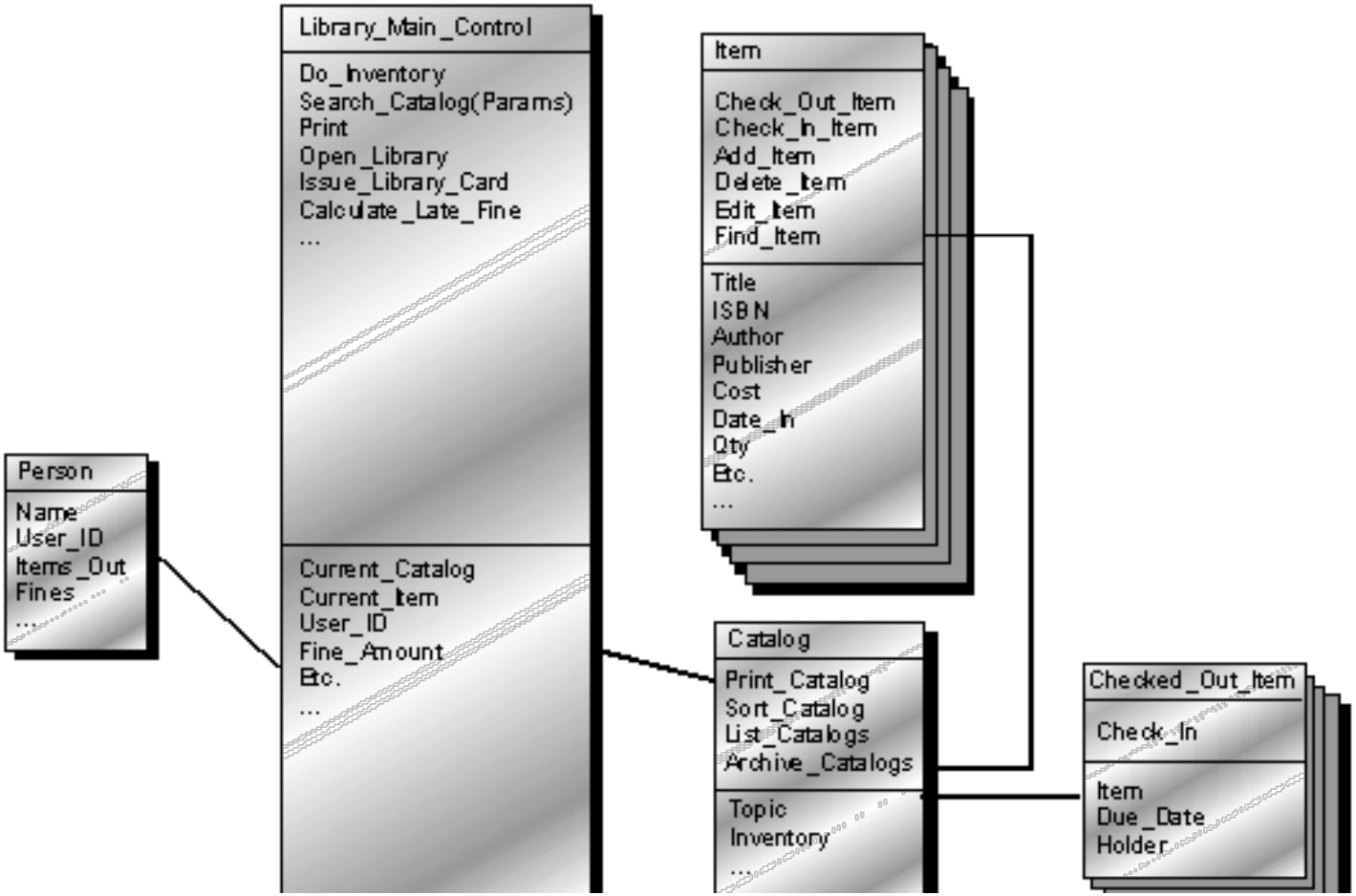
**Step 2:**
**Find "natural homes" for these contract-based collections of functionality and them migrate them there**



# Library system – Changing the design



# Library system – Changing the design



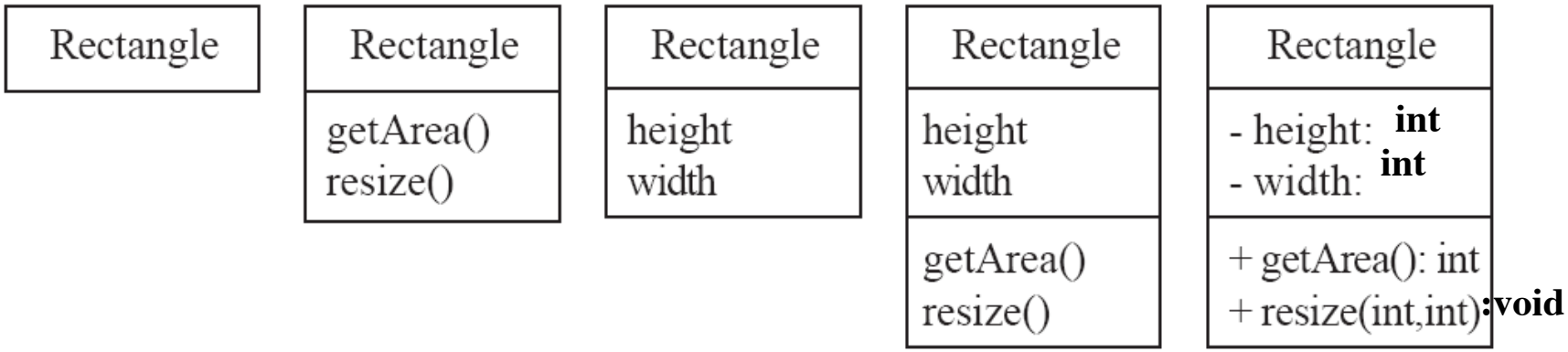
# Essentials of UML Class Diagrams

- *The main symbols shown on class diagrams are:*
  - *Classes*
    - represent the types of data themselves
  - *Associations*
    - represent linkages between instances of classes
  - *Attributes*
    - are simple data found in classes and their instances
  - *Operations*
    - represent the functions performed by the classes and their instances
  - *Generalizations*
    - group classes into inheritance hierarchies

# Classes

- A class is simply represented as a box with the name of the class inside
  - The diagram may also show the attributes and operations
  - The complete signature of an operation is:

    operationName(parameterName: parameterType …): returnType

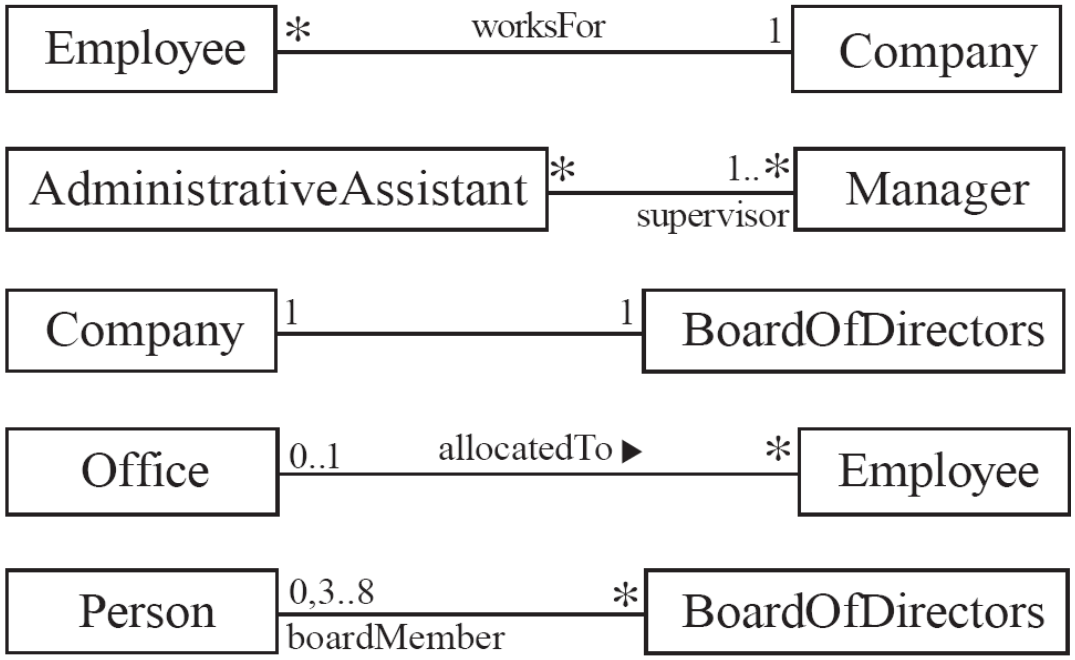


# Associations and Multiplicity

**An *association* is used to show how two classes are related to each other**

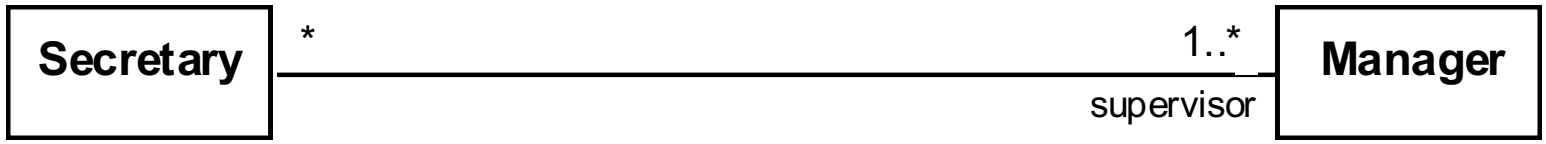
Symbols indicating *multiplicity* are shown at each end of the association

Each association can be labelled, to make explicit the nature of the association



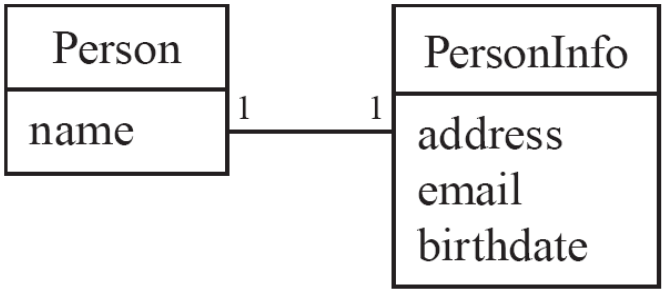
# Analyzing and validating associations

- **Many-to-many**
  - A secretary can work for many managers
  - A manager can have many secretaries
  - Secretaries can work in pools
  - Managers can have a group of secretaries
  - Some managers might have zero secretaries.
  - Is it possible for a secretary to have, perhaps temporarily, zero managers?

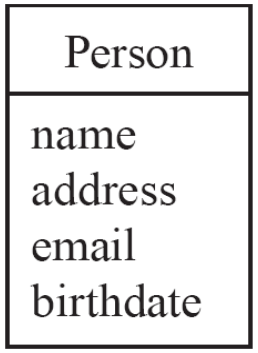



# Analyzing and validating associations

- Avoid unnecessary one-to-one associations
- Avoid this




do this




# Directionality in associations

- Associations are by default are undefined, though many tools treat these as *bi-directional*.
- It is possible to limit the direction of an association by adding an arrow at one end


# Generalization

- Specializing a superclass into two or more subclasses
  - The *discriminator* is a label that describes the criteria used in the specialization


# Associations versus generalizations in object diagrams

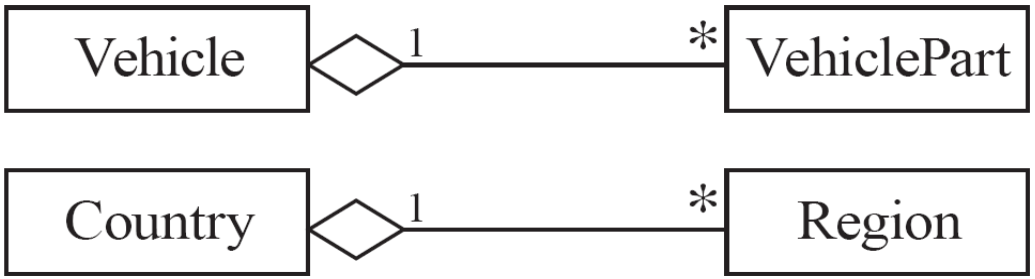
- Associations describe the relationships that will exist between *instances* at **run time**.
  - When you show an instance diagram generated from a class diagram, there will be an instance of *both* classes joined by an association

- Generalizations describe relationships between *classes* in class diagrams.
  - They do not appear in instance diagrams at all.
  - An instance of any class should also be considered to be an **instance** of each of that class's **superclasses**

# More Advanced Features: Aggregation

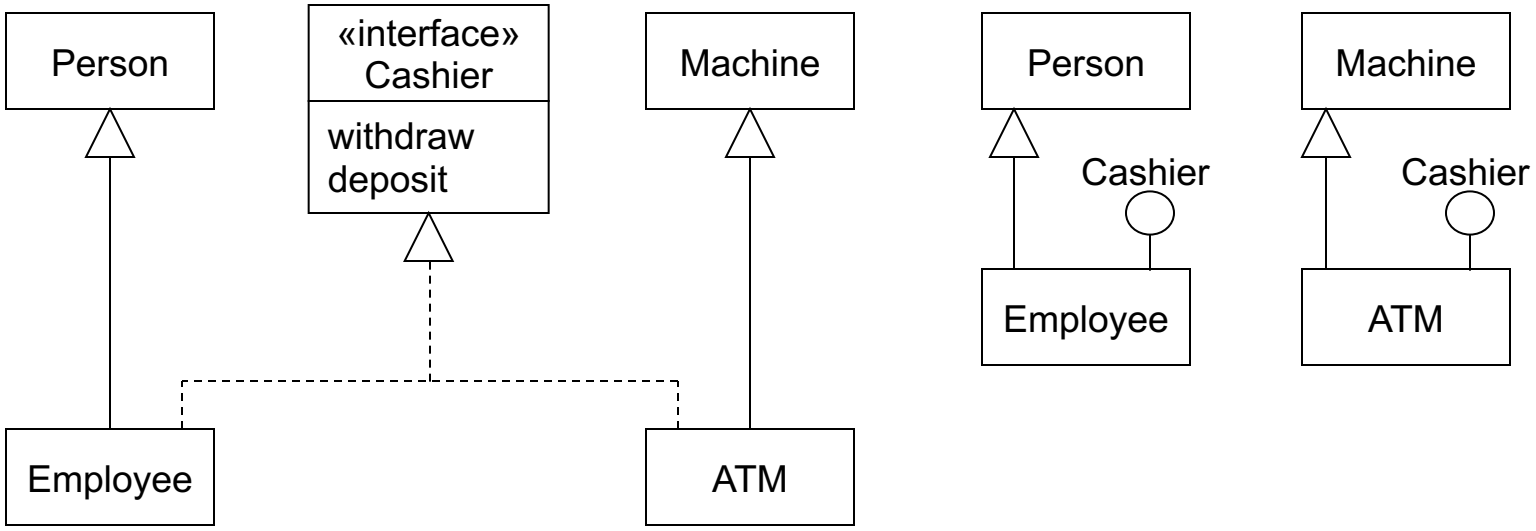
- Aggregations are special associations that represent 'part-whole' relationships.
  - The 'whole' side is often called the *assembly* or the *aggregate*
  - This symbol is a shorthand notation association named isPartOf

- As a general rule, you can mark an association as an aggregation if the following are true:
  - You can state that
    - the parts 'are part of' the aggregate
    - or the aggregate 'is composed of' the parts
  - When something **owns** or controls the aggregate, then they also own or control the parts




# Interfaces

- An interface describes a *portion of the visible behaviour* of a set of objects.
  - An *interface* is similar to a class, except it lacks instance variables and implemented methods

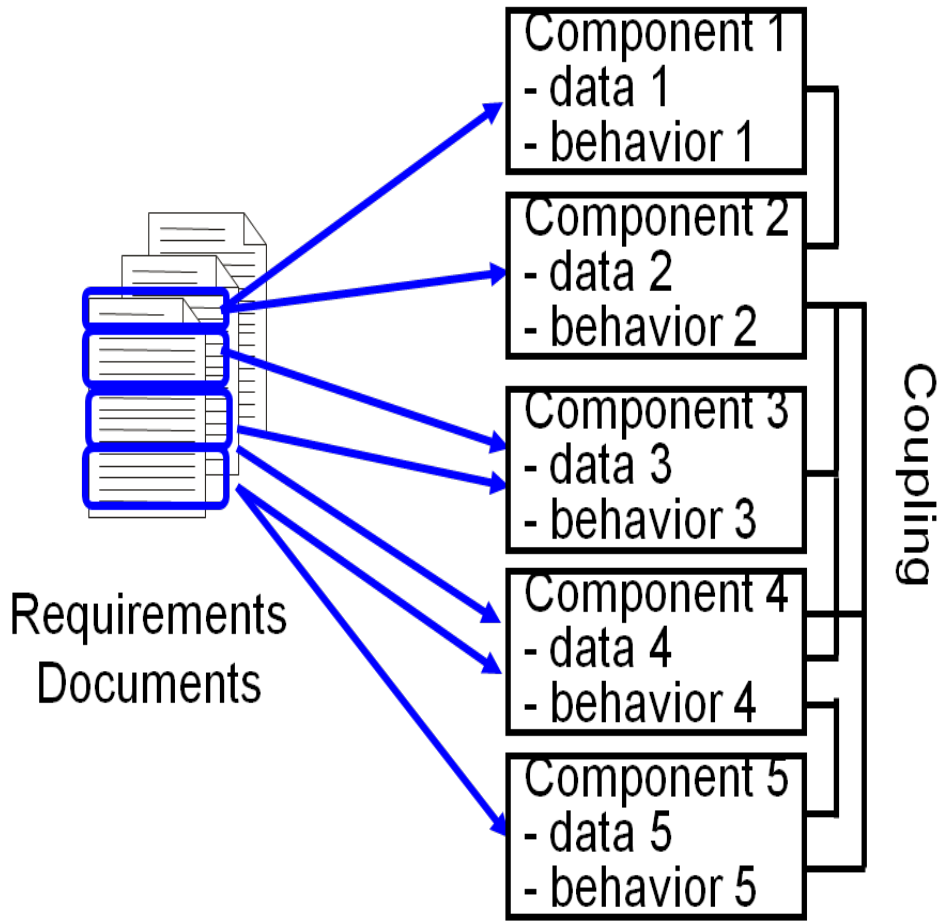



# Suggested sequence of activities

- Identify a first set of candidate **classes**
- Add **associations** and **attributes**
- Find **generalizations**
- List the main **responsibilities** of each class
- Decide on specific **operations**
- **Iterate** over the entire process until the model is satisfactory
  - Add or delete classes, associations, attributes, generalizations, responsibilities or operations
  - Identify interfaces
  - Apply design patterns

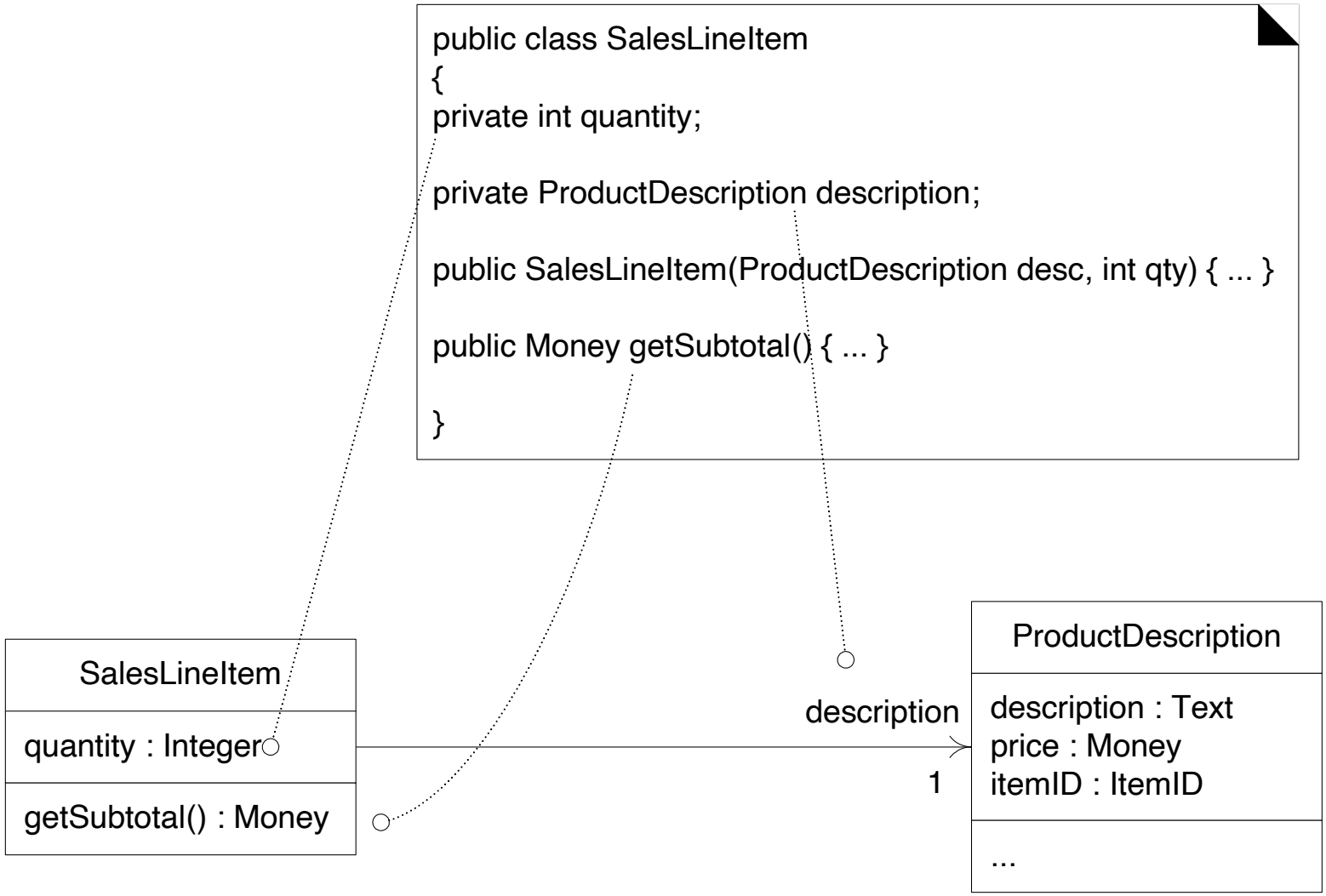
***Don't be too disorganized. Don't be too rigid either!***

# Mapping Requirements to Design Components




- Design must satisfy requirements
  - Everything (data and behavior) in the requirements must be mapped to the design components
  - Decide what functionality goes into which component
- As you do the mapping, assess functional cohesion and coupling
  - Strive for **loose** coupling and **high** cohesion

# Mapping UML class diagram to code



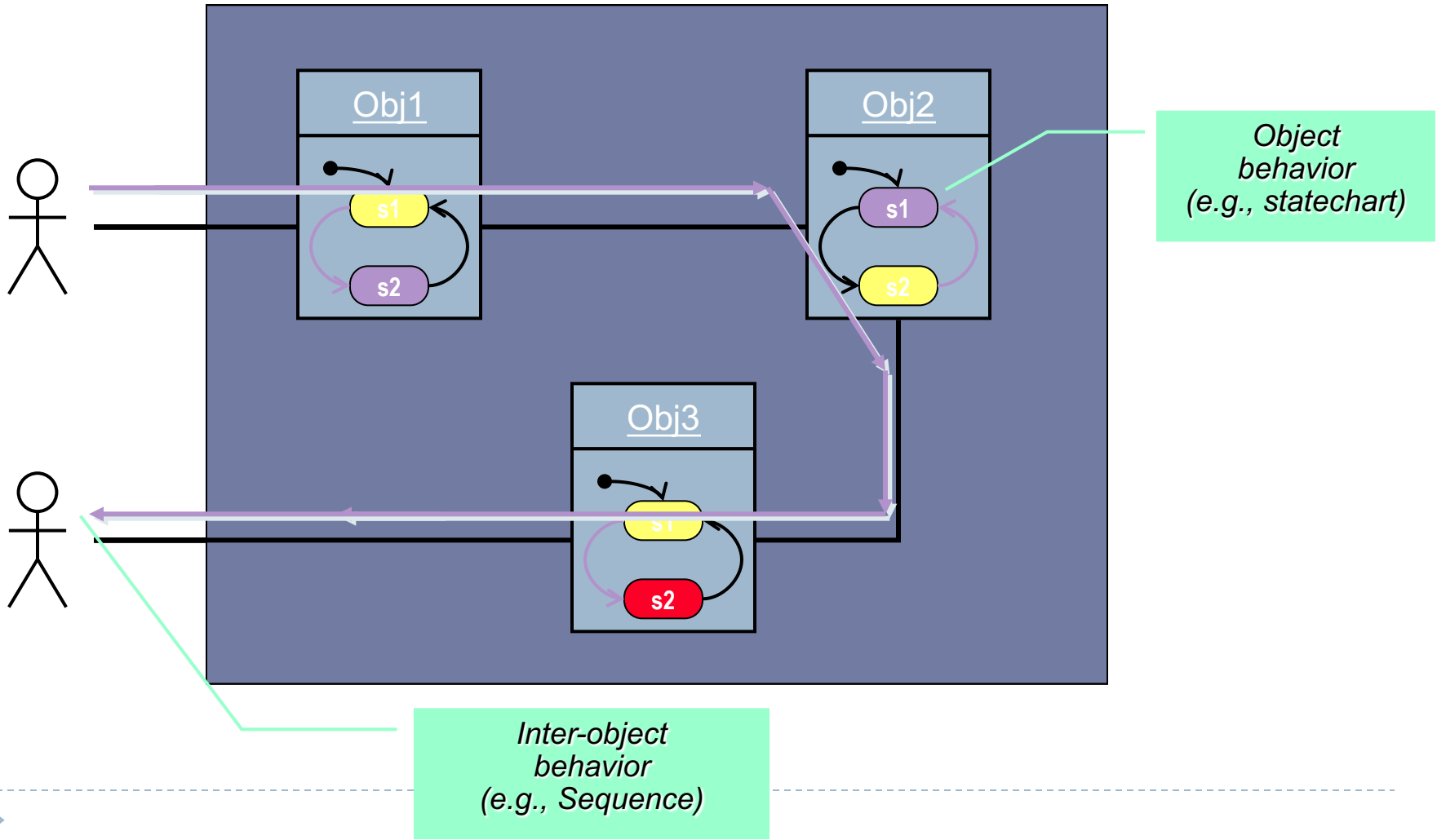
# UML Behavioral Models

# Specifying behavior using the UML

- Class models describe objects and their relationships
  - Behavior can be specified in terms of operation pre and postconditions, but behavior is not the primary focus of a class model
- Behavioral models in the UML
  - State diagrams: describe control aspects of a system – provides descriptions sequences of operations without regard for what the operations do.
  - Interaction models (Sequence diagrams): describe interactions among objects

# How Things Happen in UML

- In UML, all behavior results from the actions of (active) objects



# Interaction Diagrams

- Interaction diagrams are used to model the dynamic aspects of a software system
  - They help you to visualize how the system runs.
  - An interaction diagram is often built from a use case and a class diagram.
    - The objective is to show how a set of objects accomplish the required interactions with an actor.

# Interactions and messages

- Interaction diagrams show how a set of actors and objects communicate with each other to perform:
  - The steps of a use case, or
  - The steps of some other piece of functionality.

- The set of steps, taken together, is called an ***interaction***.

- Interaction diagrams can show several different types of communication.
  - E.g. method calls, messages send over the network
  - These are all referred to as ***messages***.

# Elements found in interaction diagrams

- Instances of classes
  - Shown as boxes with the class and object identifier underlined
- Actors
  - Use the stick-person symbol as in use case diagrams
- Messages
  - Shown as arrows from actor to object, or from object to object

# Sequence Diagrams – Modeling Interaction

1. Student selects Register for Courses option

2. System retrieves a list of the available courses

3. Student specifies the desired course

4. System shows a list of the available sections

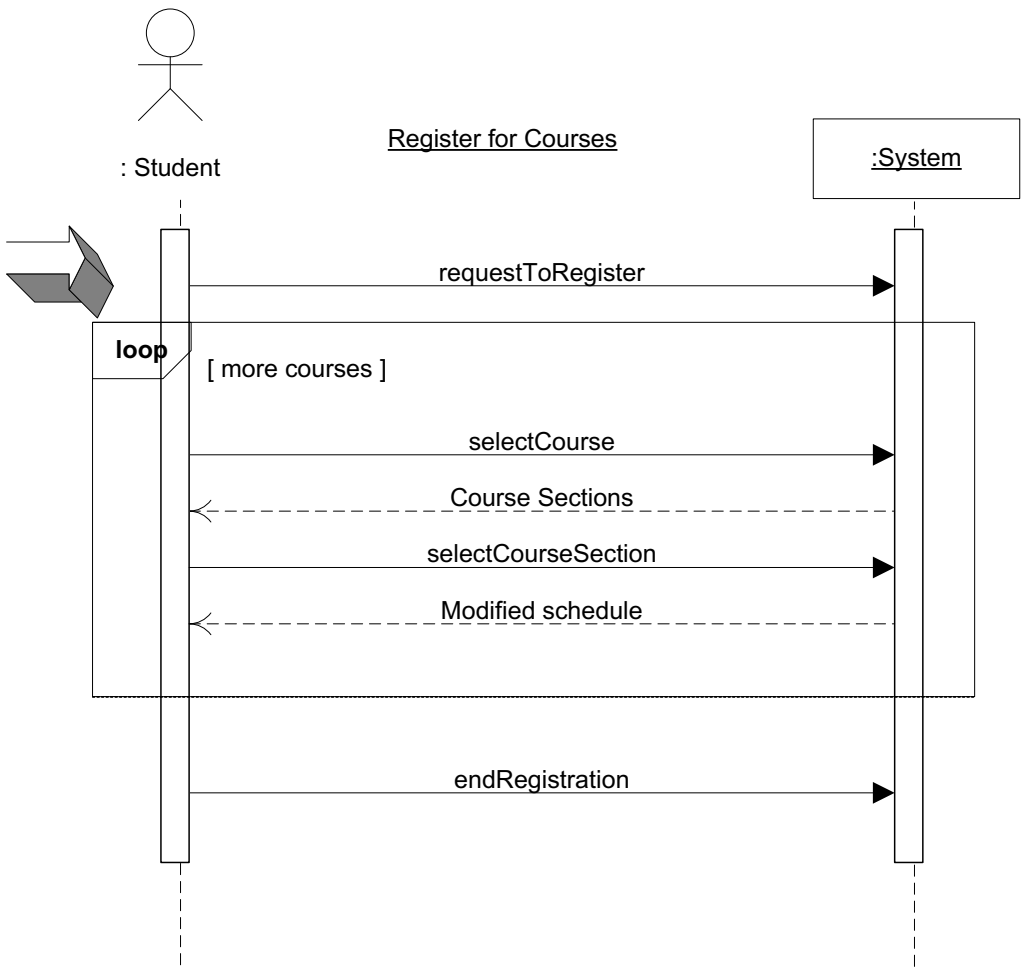
5. Student selects the course section

6. System verifies if the student has passed prerequisites

7. System add course section to student's Schedule

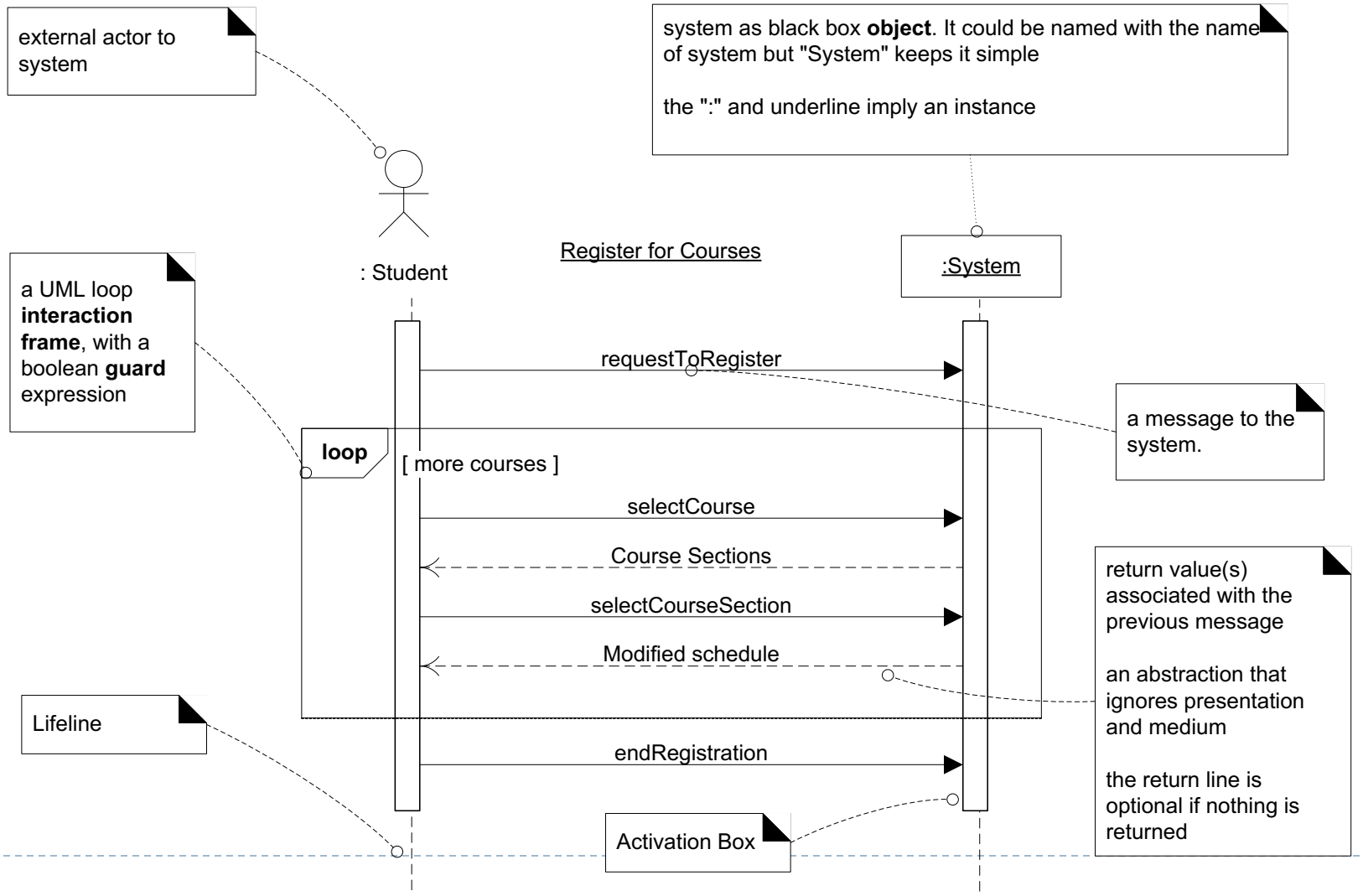
8. System displays modified student's Schedule

9. Steps 3-8 repeated until student finished




Based on System Sequence Diagrams
Larman, C. Applying UML and Patterns, 3rd Edition (Pearson Education, 2005)

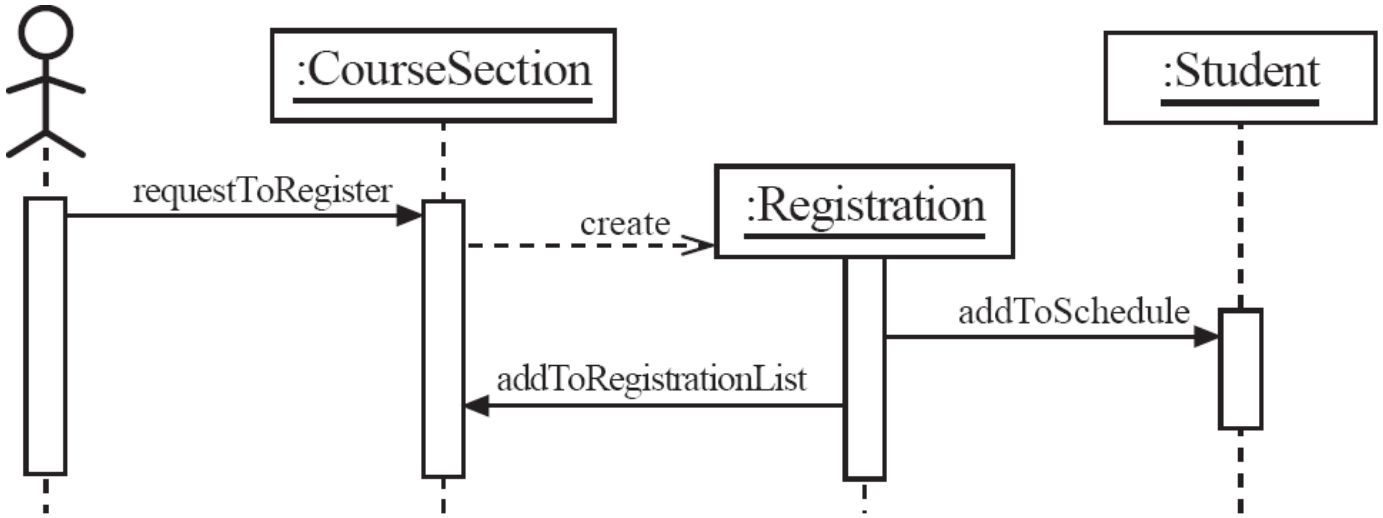
# Sequence Diagrams – Elements

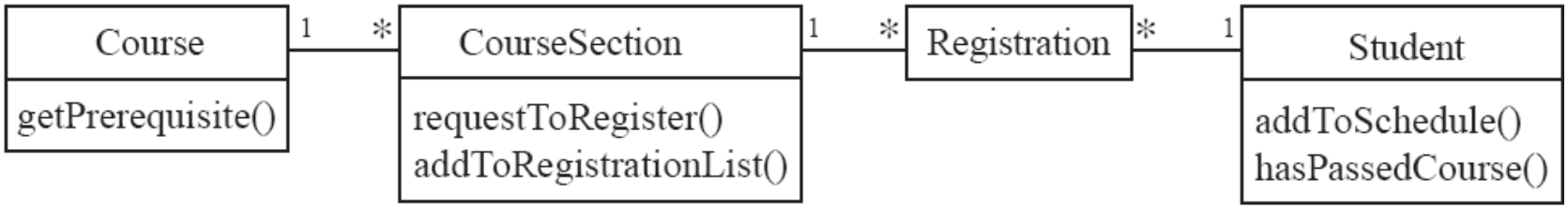


# Sequence diagrams

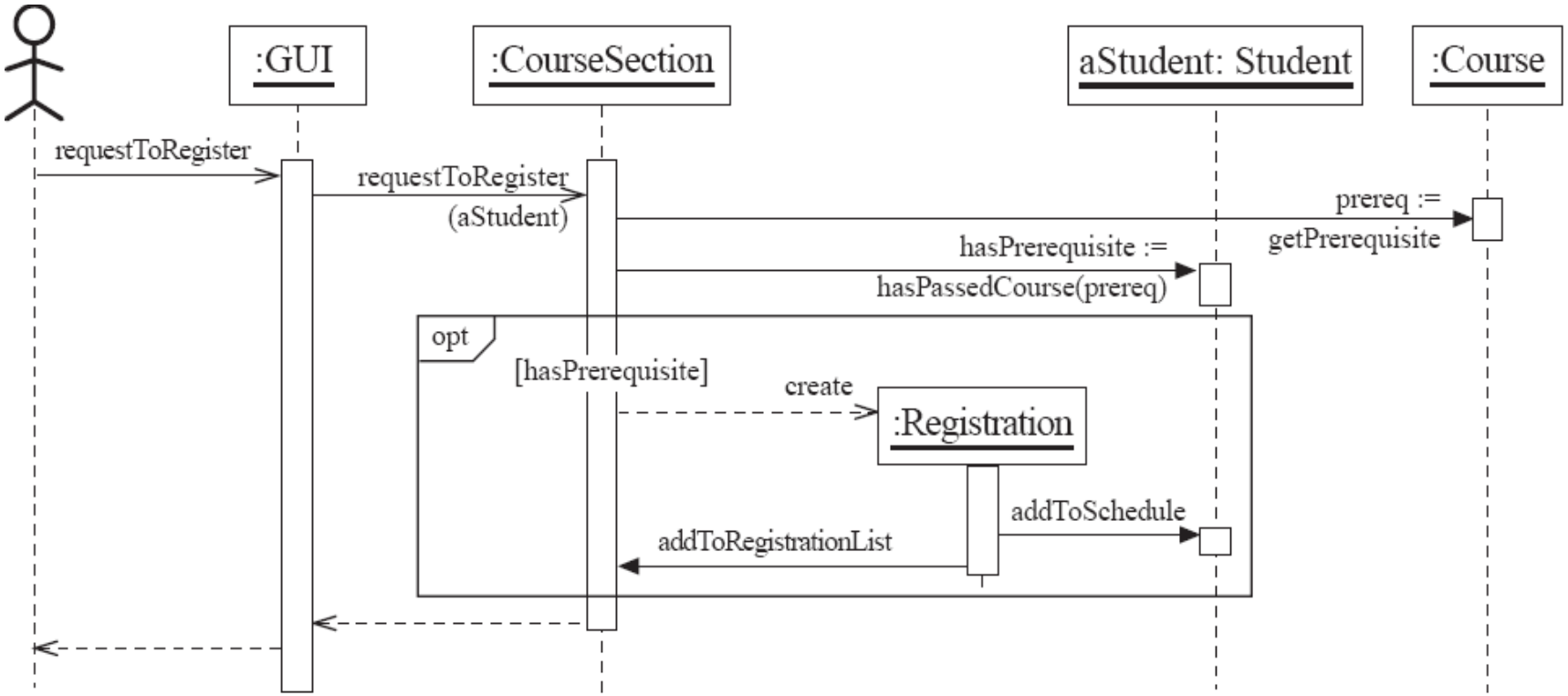
- A sequence diagram shows the sequence of messages exchanged by the set of objects performing a certain task
  - The objects are arranged horizontally across the diagram.
  - An actor that initiates the interaction is often shown on the left.
  - The vertical dimension represents time.
  - A vertical line, called a *lifeline*, is attached to each object or actor.
  - The lifeline becomes a broad box, called an *activation box* during the *live activation* period.
  - A message is represented as an arrow between activation boxes of the sender and receiver.
    - A message is labelled and can have an argument list and a return value.

# Sequence diagrams – an example




# Sequence diagrams – same example, more details



# State diagrams

# State diagrams

- A *state diagram* specifies the life histories of objects in terms of the sequences of operations that can occur in response to external stimuli.
  - For example, a state diagram can describe how an object responds to a request to invoke one of its methods.

- A state diagram describes behavior in terms of sequences of *states* that an object can go through in response to *events*.

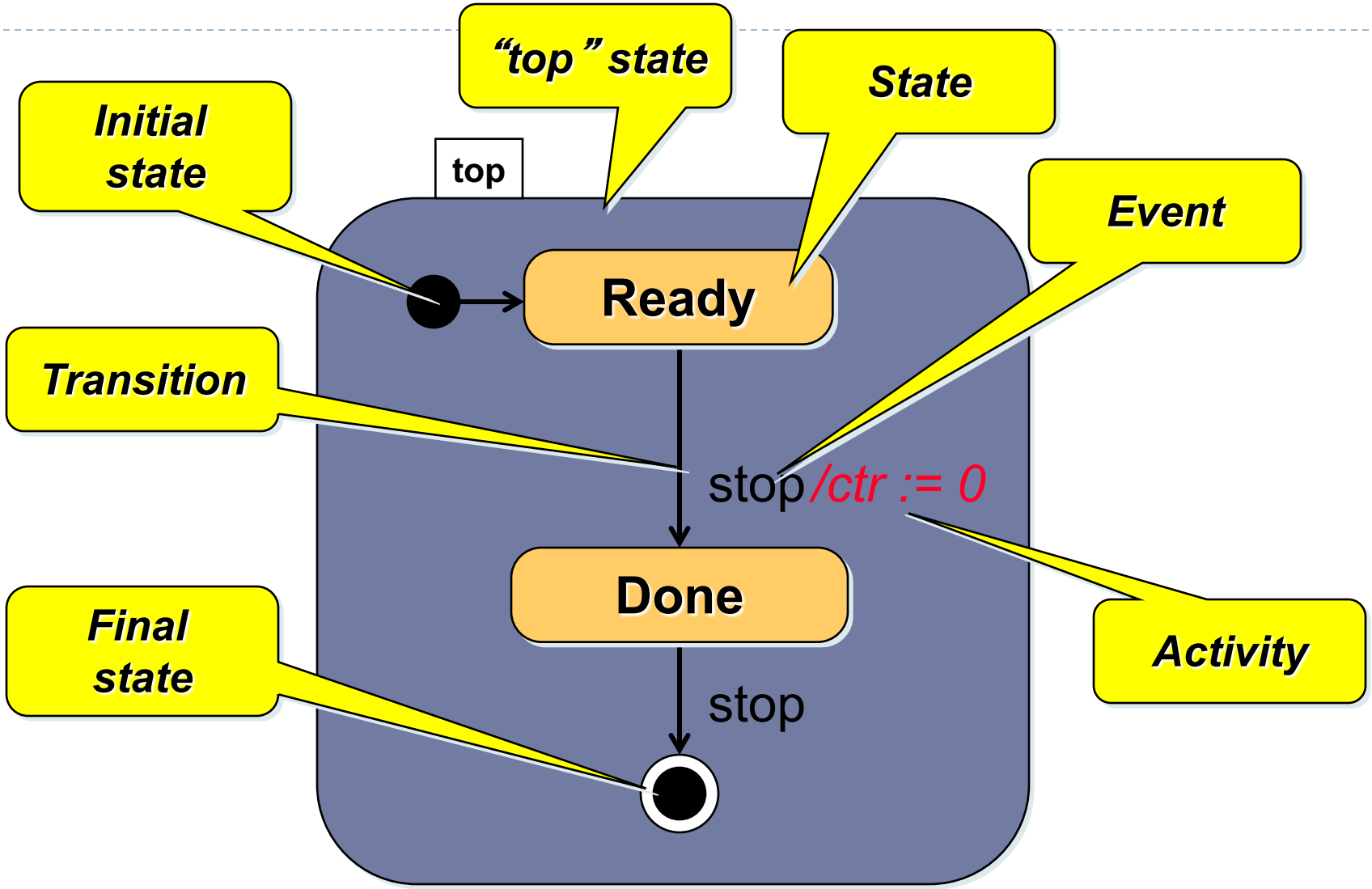
# Key Concepts

- An *event* is a significant or noteworthy occurrence at a point in time.
  - Examples of events: sending a request to invoke a method, termination of an activity.
  - An event occurs instantaneously in the time scale of an application.
- A *state* is a condition of an object during its lifetime.
  - For example, a student is in the registered state after completing course registration.
  - A state is an abstraction of an object's attribute values and links
    - For example, a bank account is in the Overdraft state when the value of its balance attribute is less than 0.

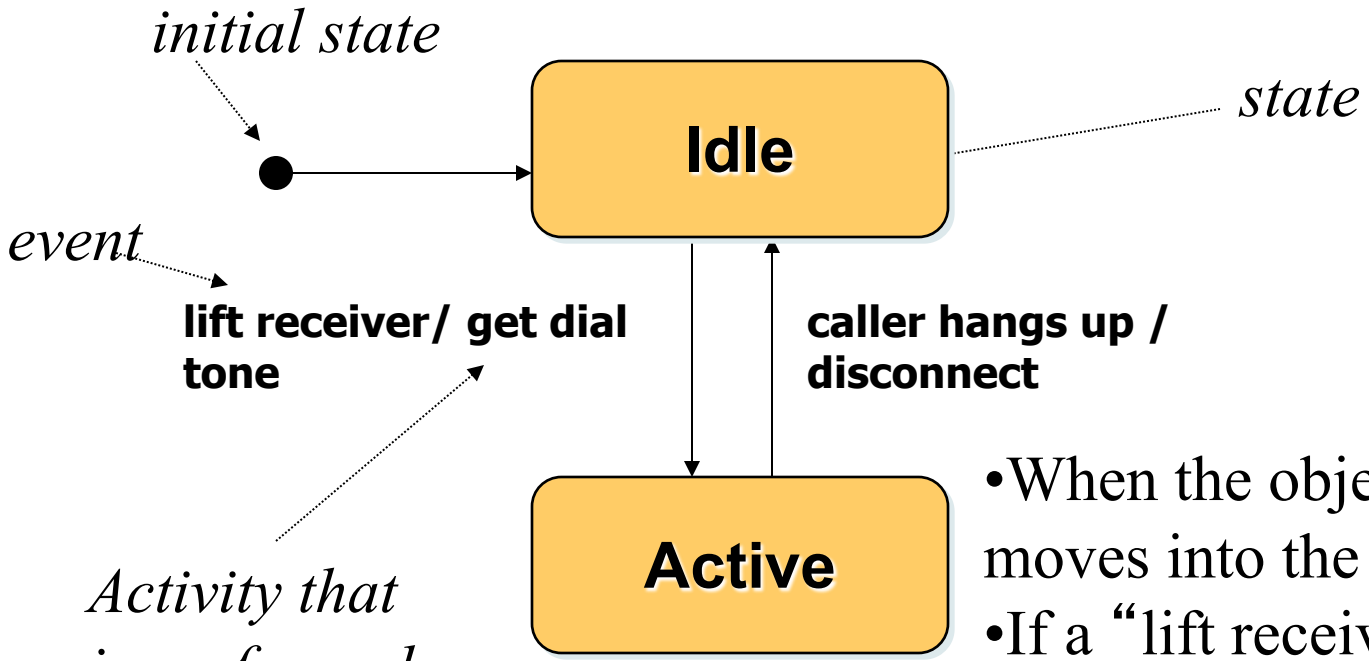
# Key Concepts - 2

- A *transition* occurs when an event causes an object to change from its current (source) state to a target state.
  - For example, if a student is in the registered state and then drops out of the program then the student is in the "not registered" state.
  - The source and target states can be the same.
  - A transition is said to fire when the change from source to target state occurs.
- A *guard condition* on a transition is a boolean expression that must be true for a transition to fire
- An *activity* is a behavior that is executed in response to an event.

# Basic UML State Diagram



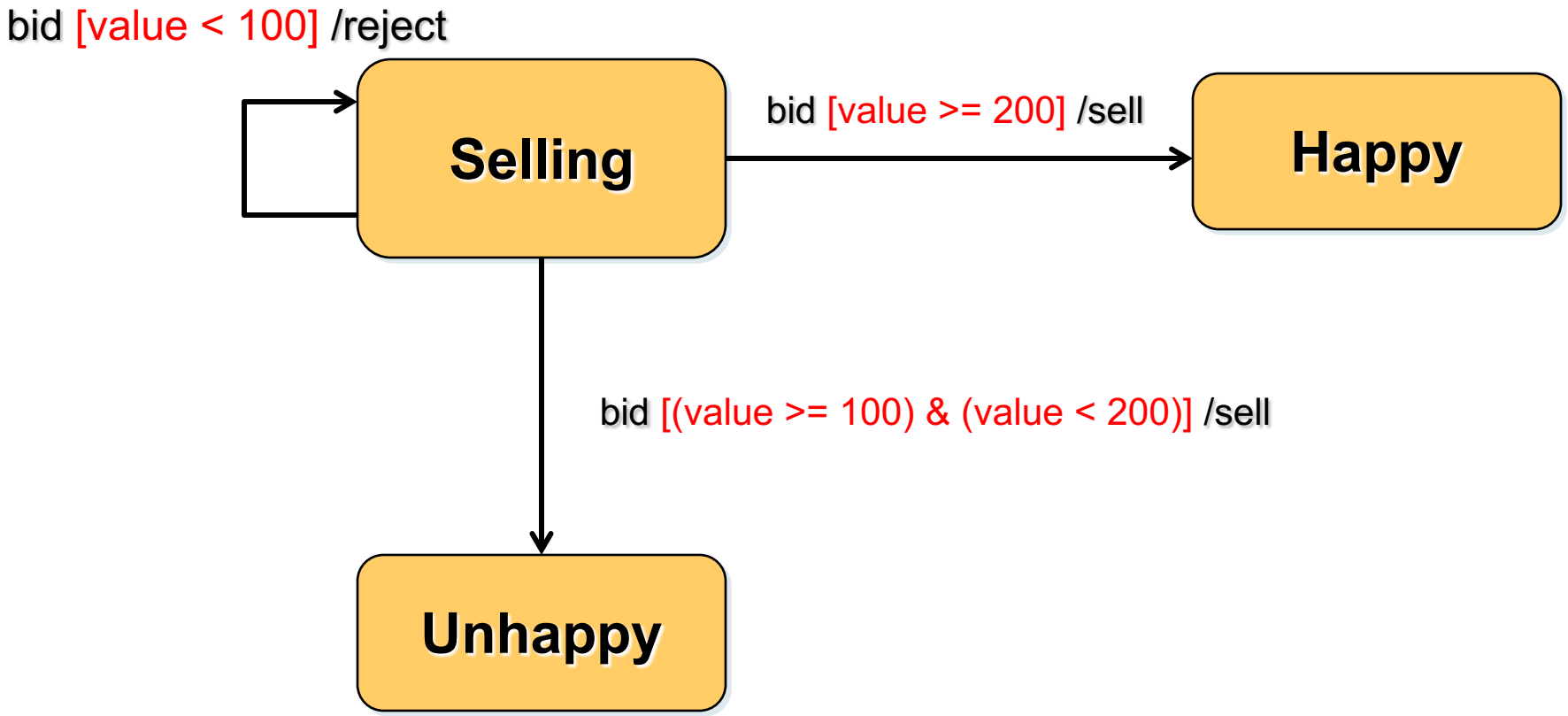
# Simple Example: Telephone Object



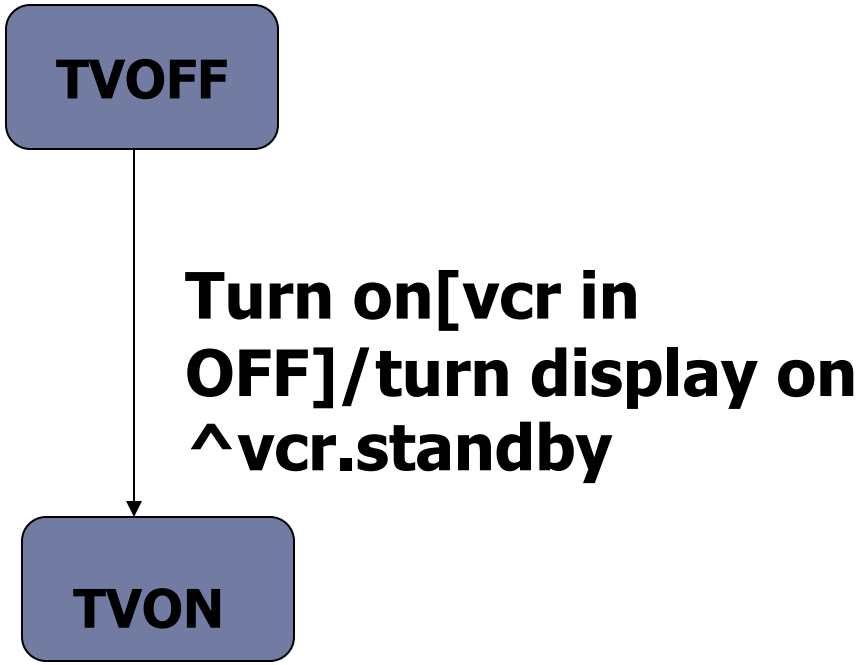

- When the object is created, it moves into the Idle state.
- If a "lift receiver" event is received the object moves from the Idle to the Active state. During the transition the "get dial tone" activity is executed.
- If a "caller hangs up" event occurs when in Active state, the object moves to the Idle state

# Guards

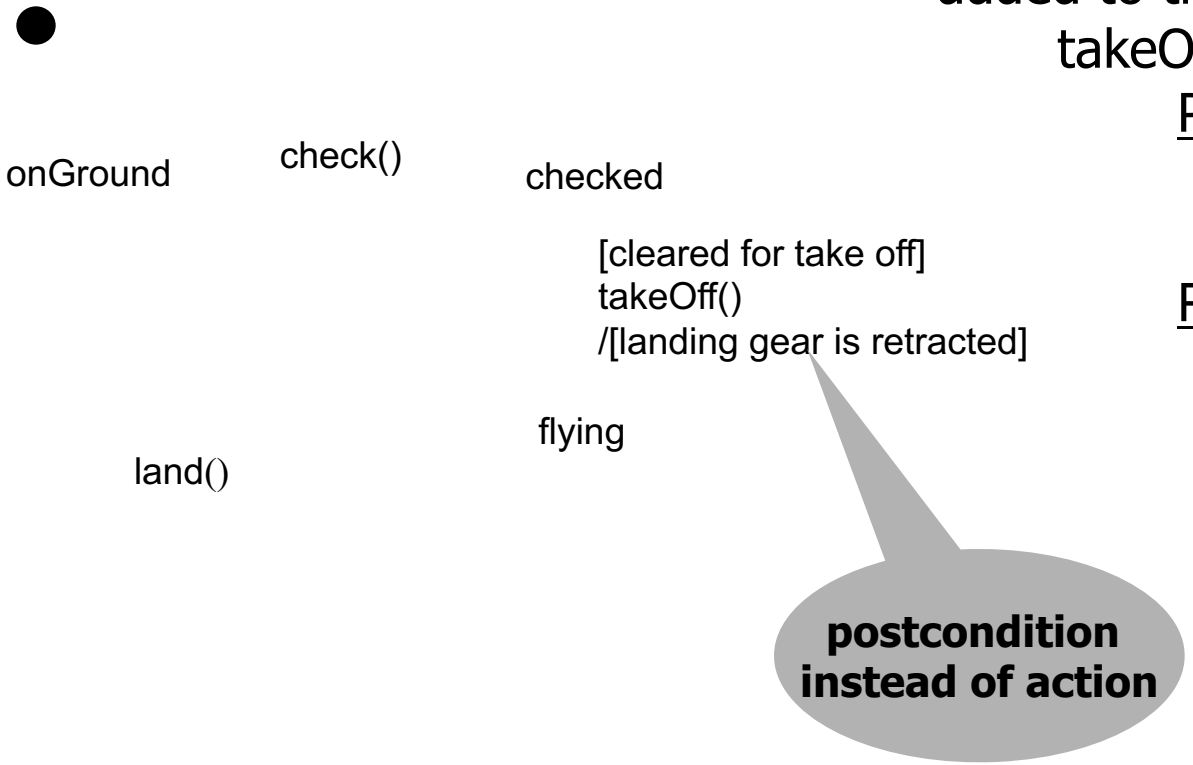
- Conditional execution of transitions
  - guards (Boolean predicates) must be side-effect free




# TV - Example



# Protocol State Machines




Equivalent to pre and post conditions added to the related operations:

takeOff()

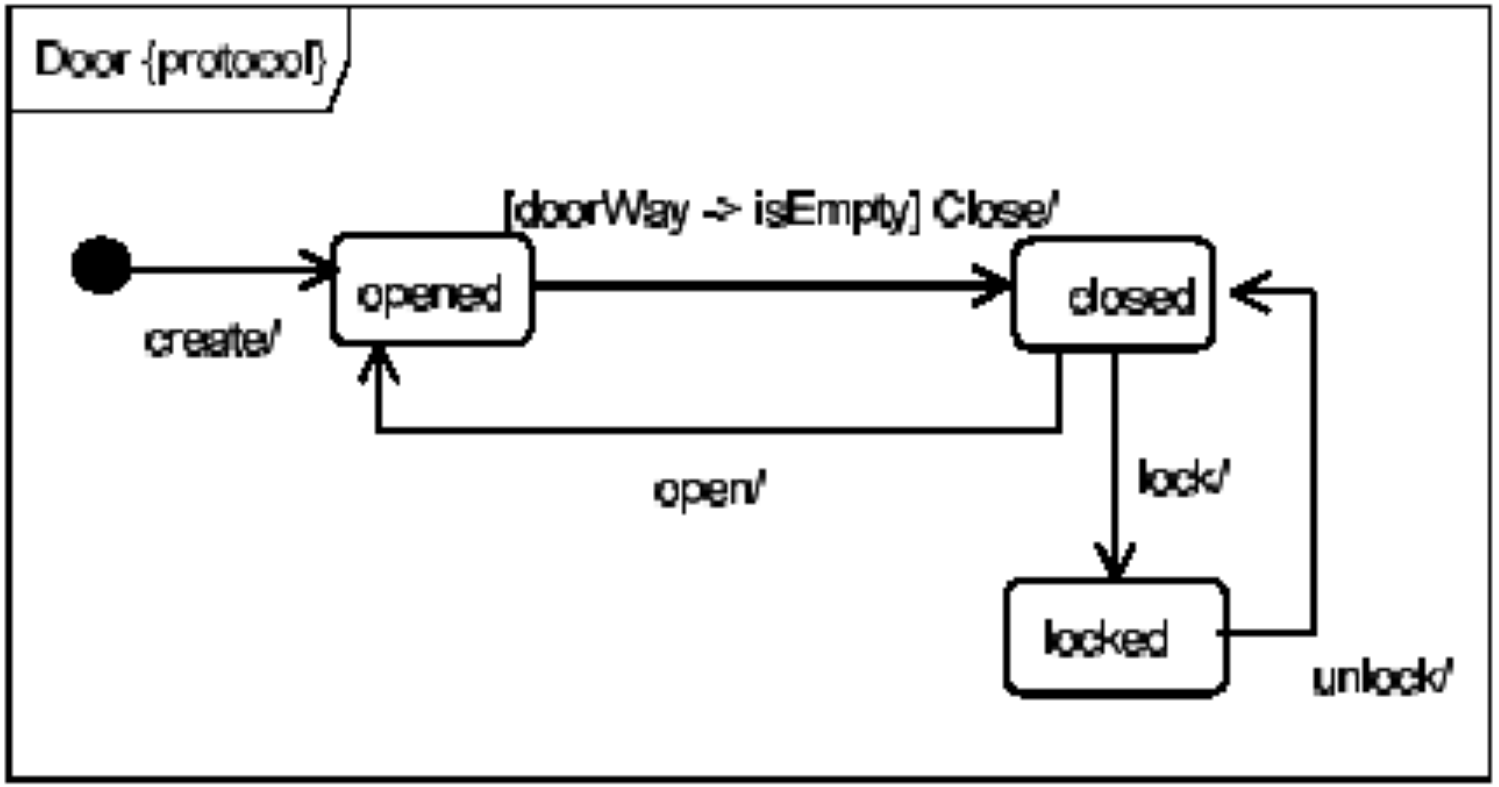
Pre

-in state "checked"
-cleared for take off

Post

-landing gear is retracted
-in state "flying"

# Another Example of a Protocol State Machine



# 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?

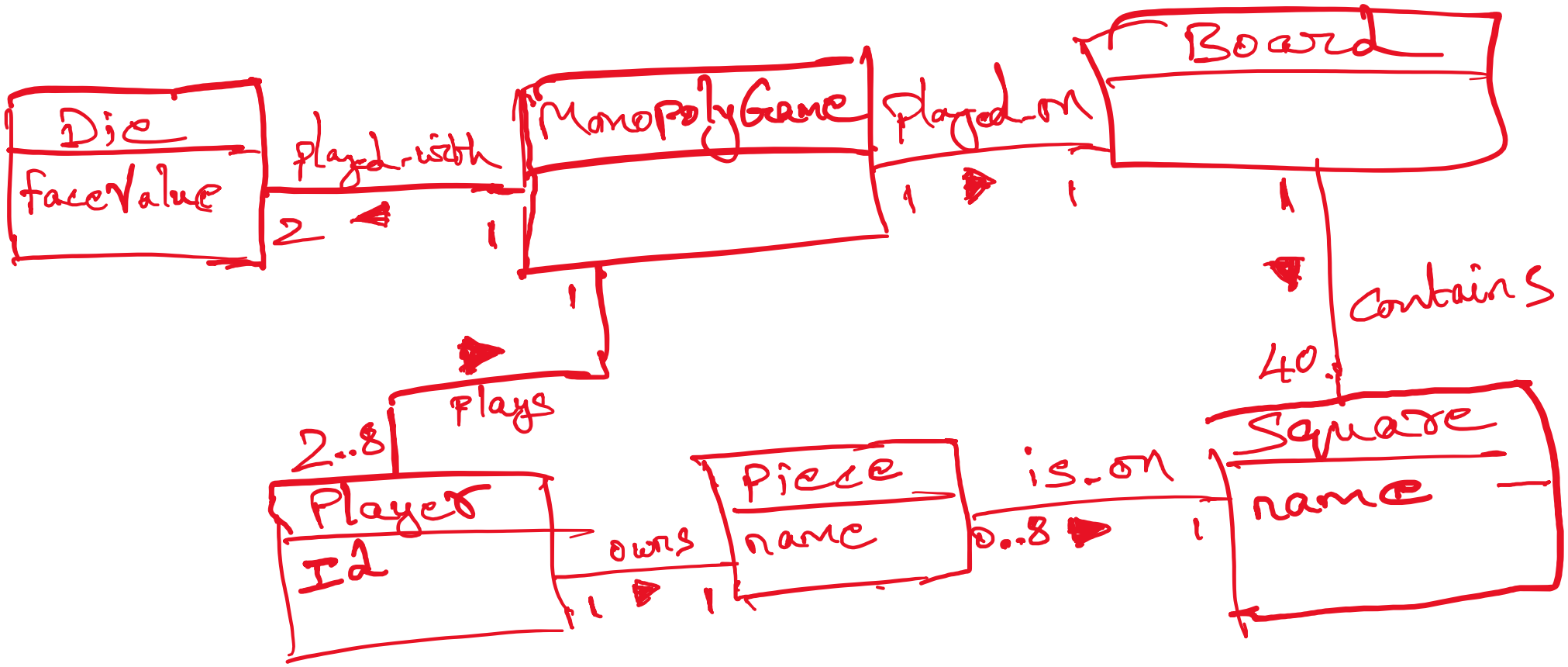
- Design for re-use is difficult
- Experienced designers:
  - Rarely start from first principles
  - Apply a working "handbook" of approaches
- Patterns make this ephemeral knowledge available to all
- Support evaluation of alternatives at higher level of abstraction

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

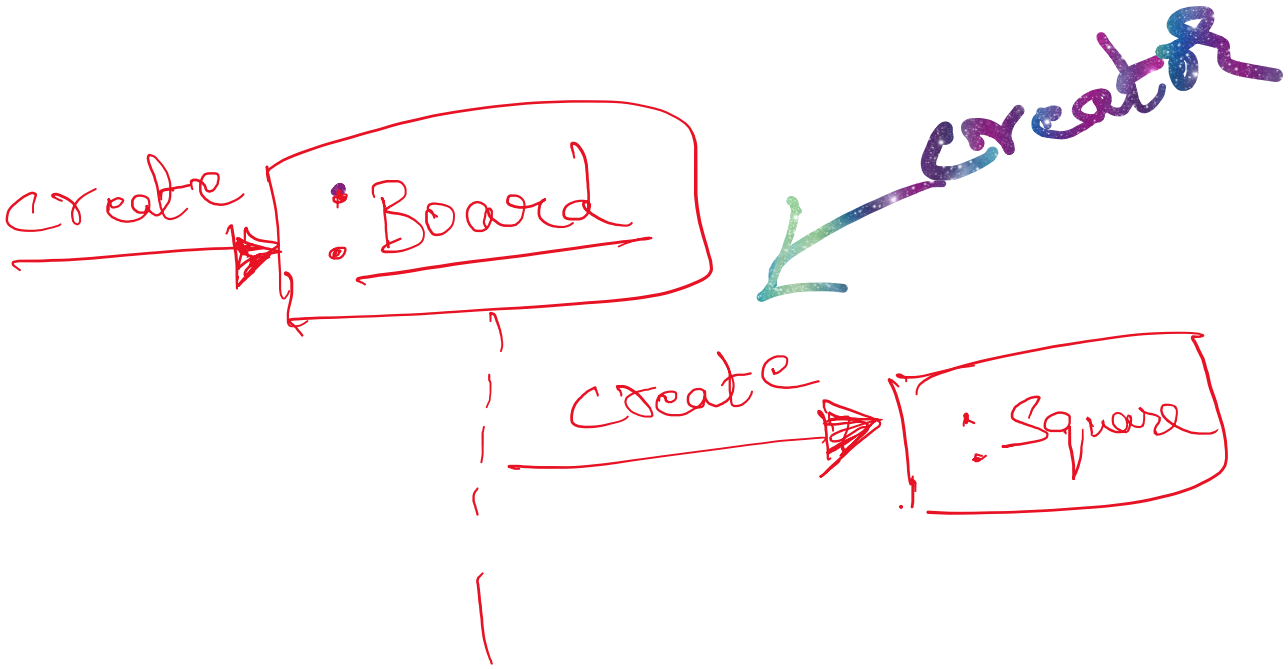
(2) B compositely aggregates A

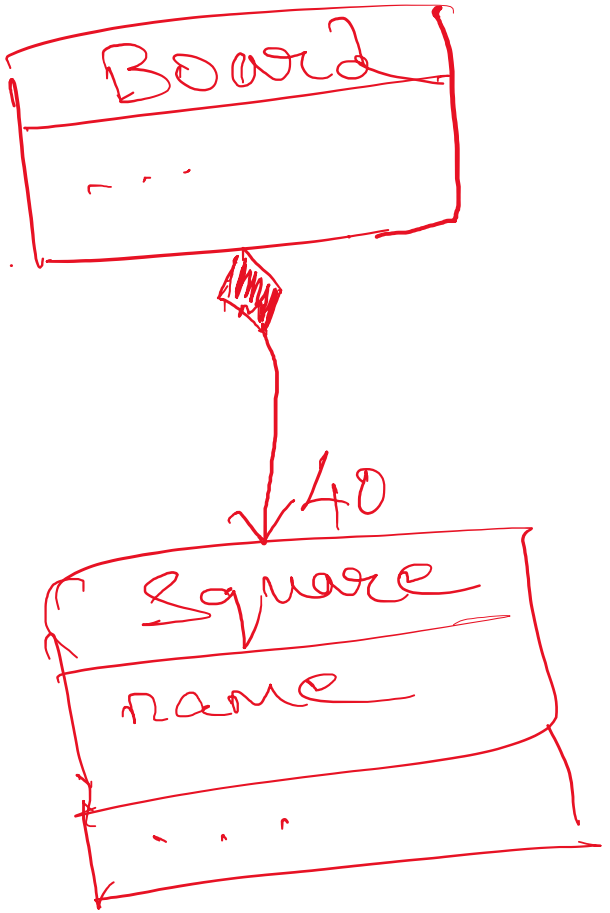
(3) B has the initializing data for A

(4) B records A

(5) B closely uses A

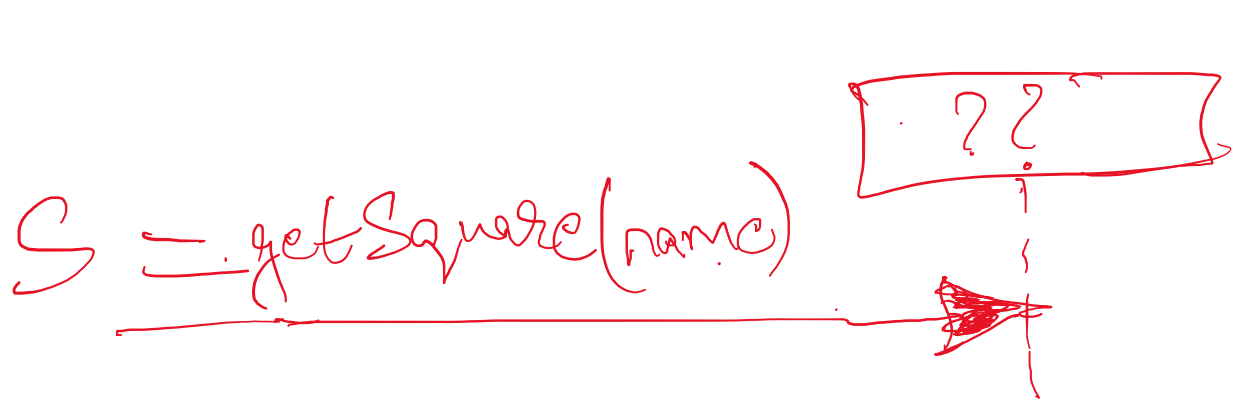
# Applying creator pattern

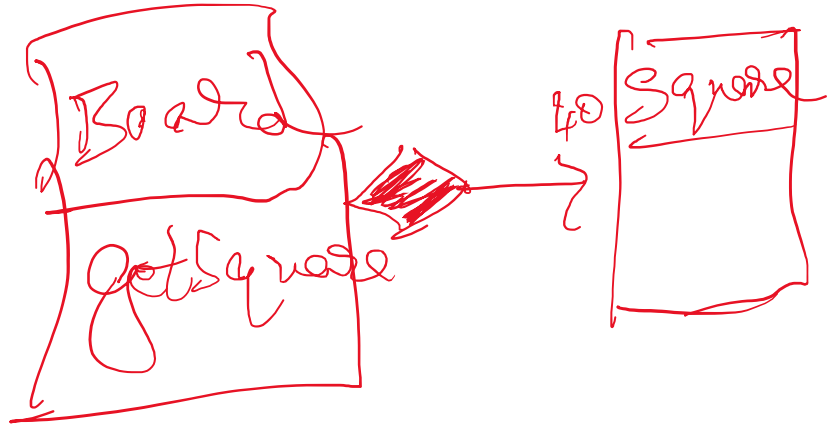


# GRASP – Information Expert

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

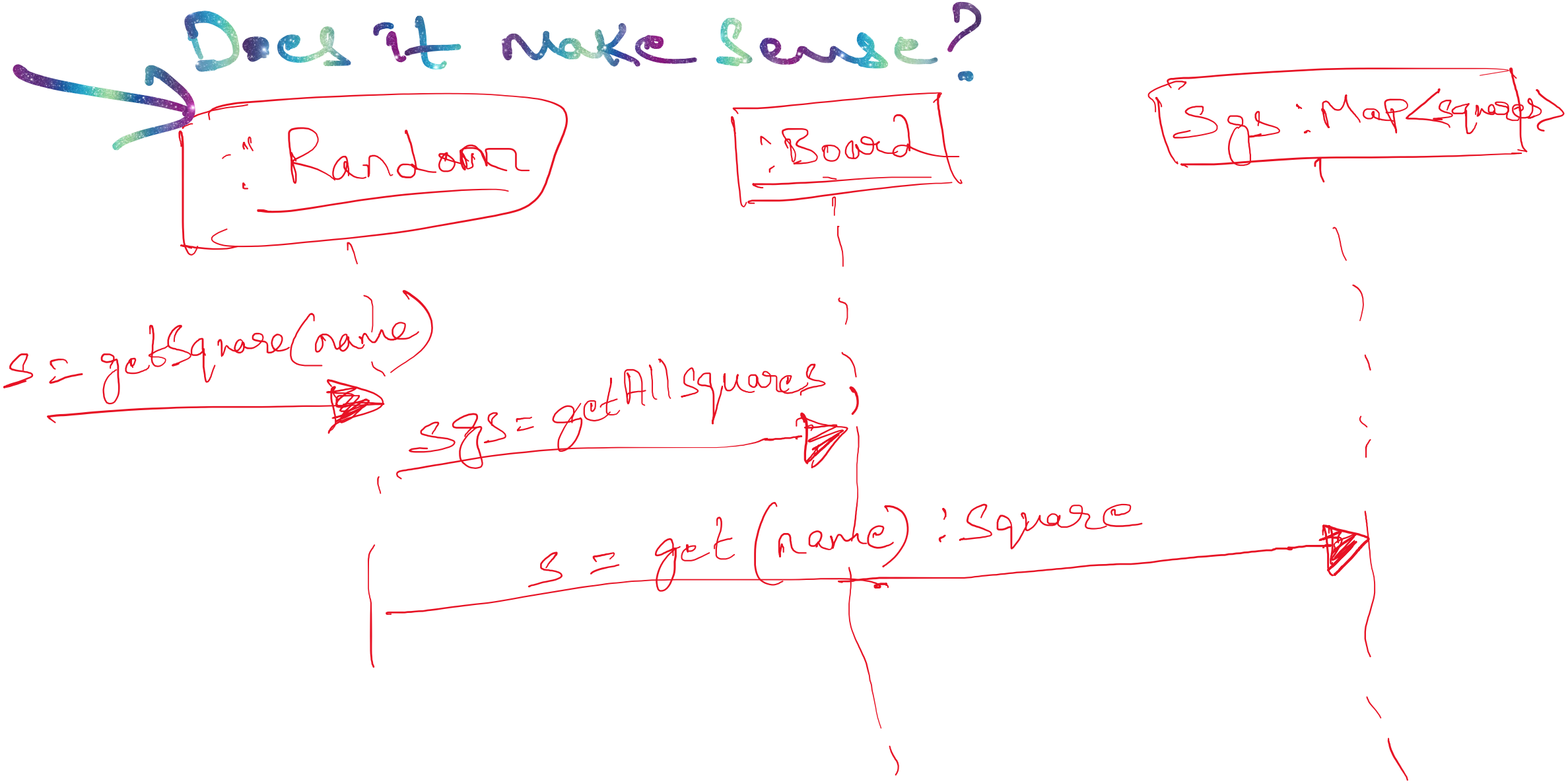
Object A that invokes methods of Object B has coupling to B's services

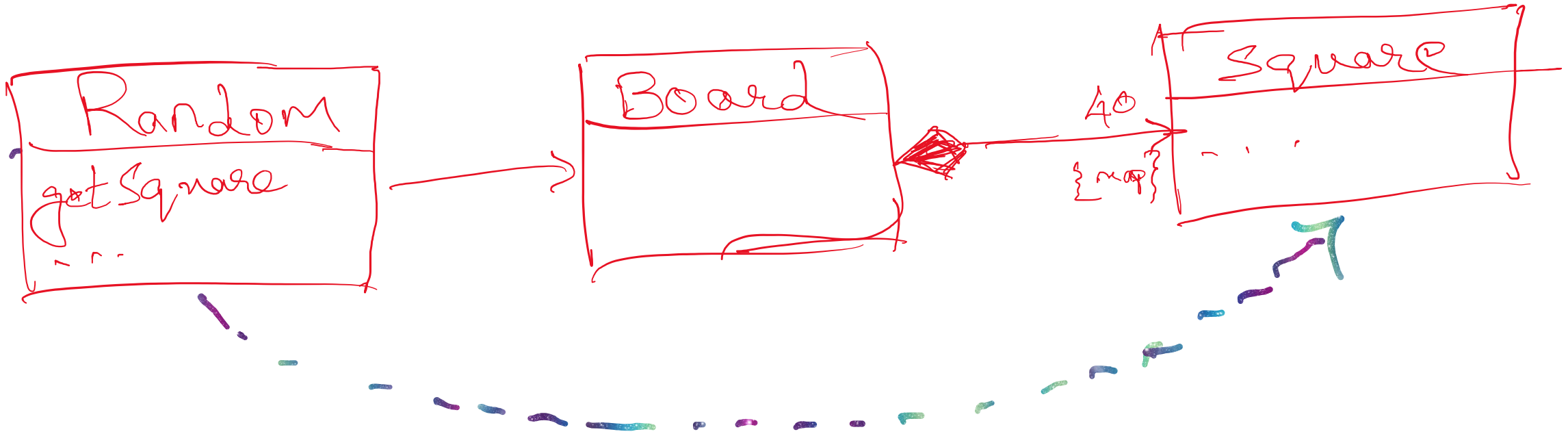
Monopoly Example:

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.

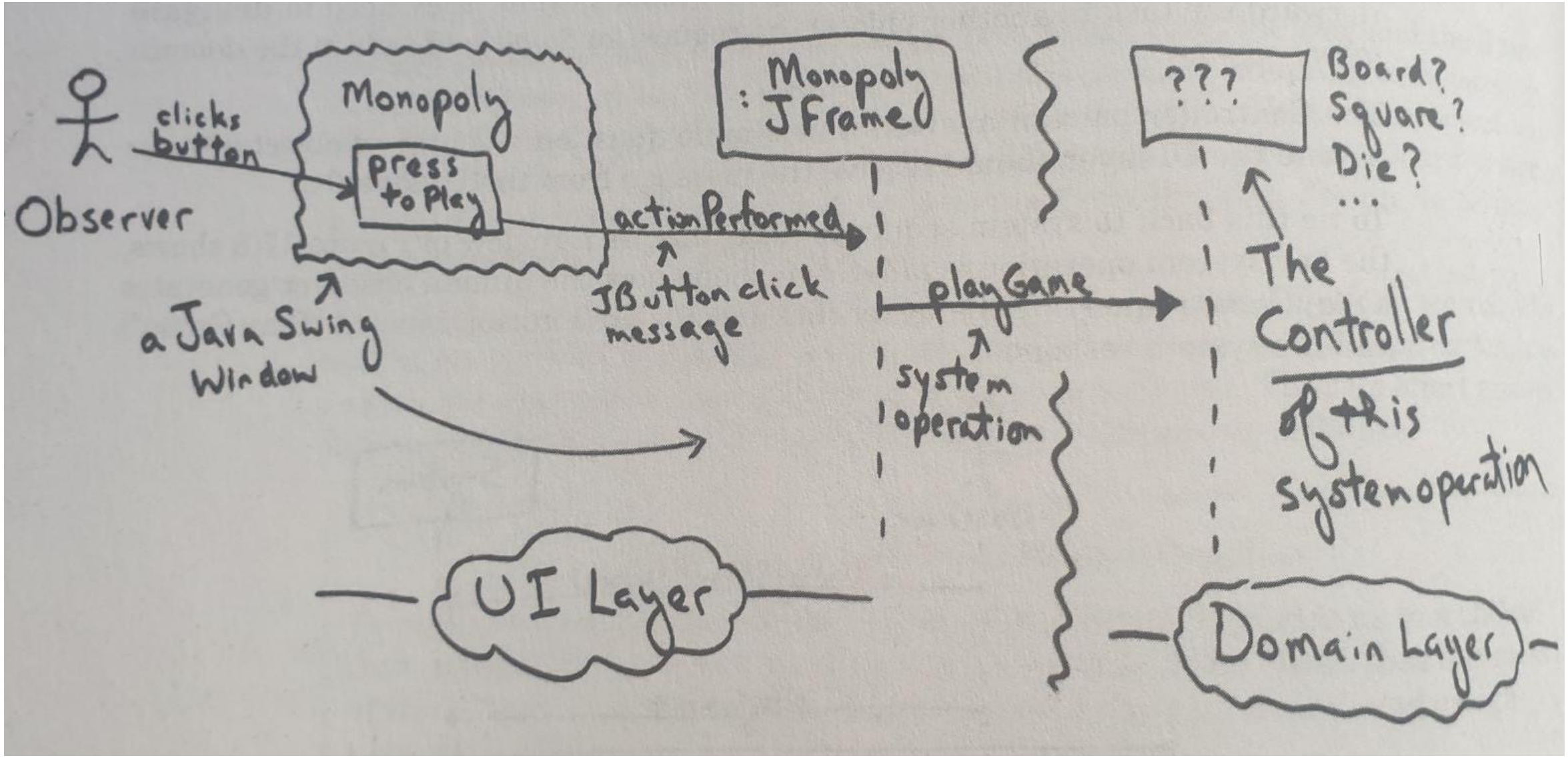
# GRASP - Controller

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

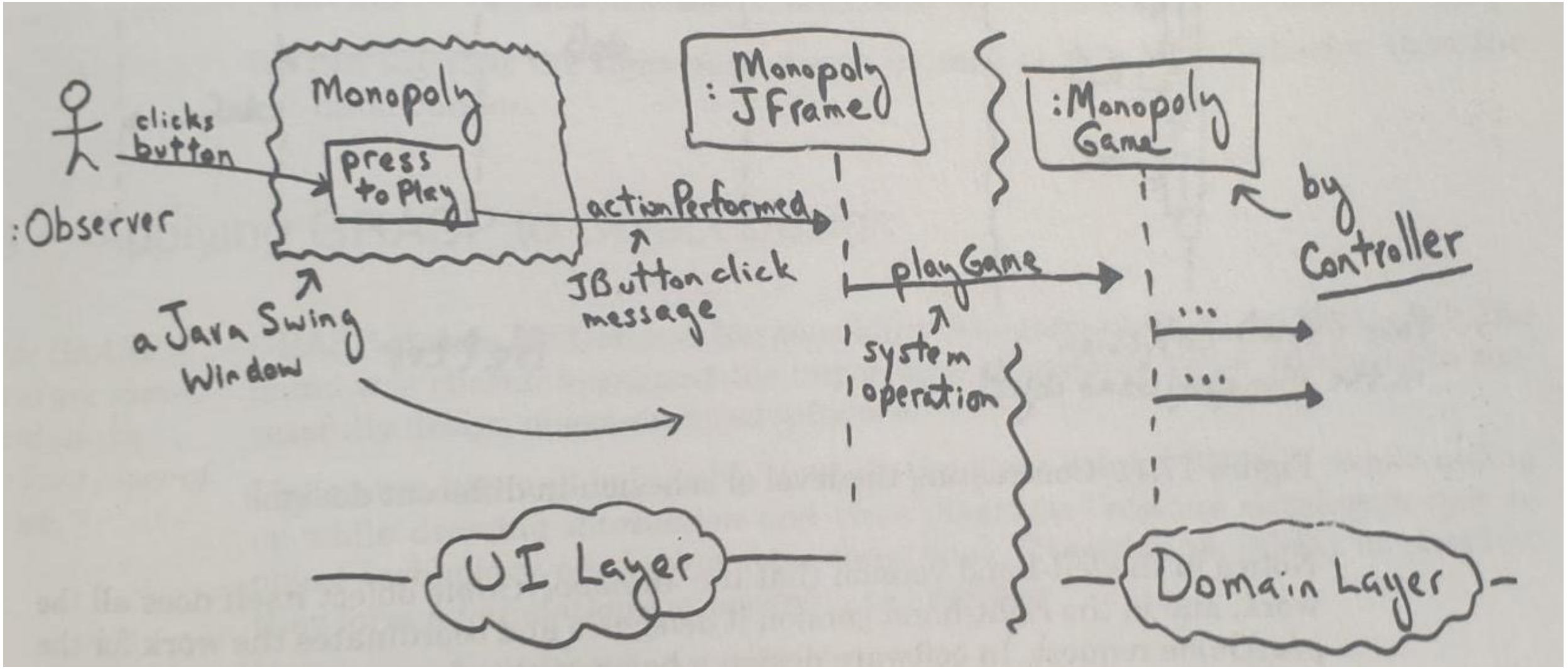
## Model-View-Controller (MVC) Pattern

- 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.

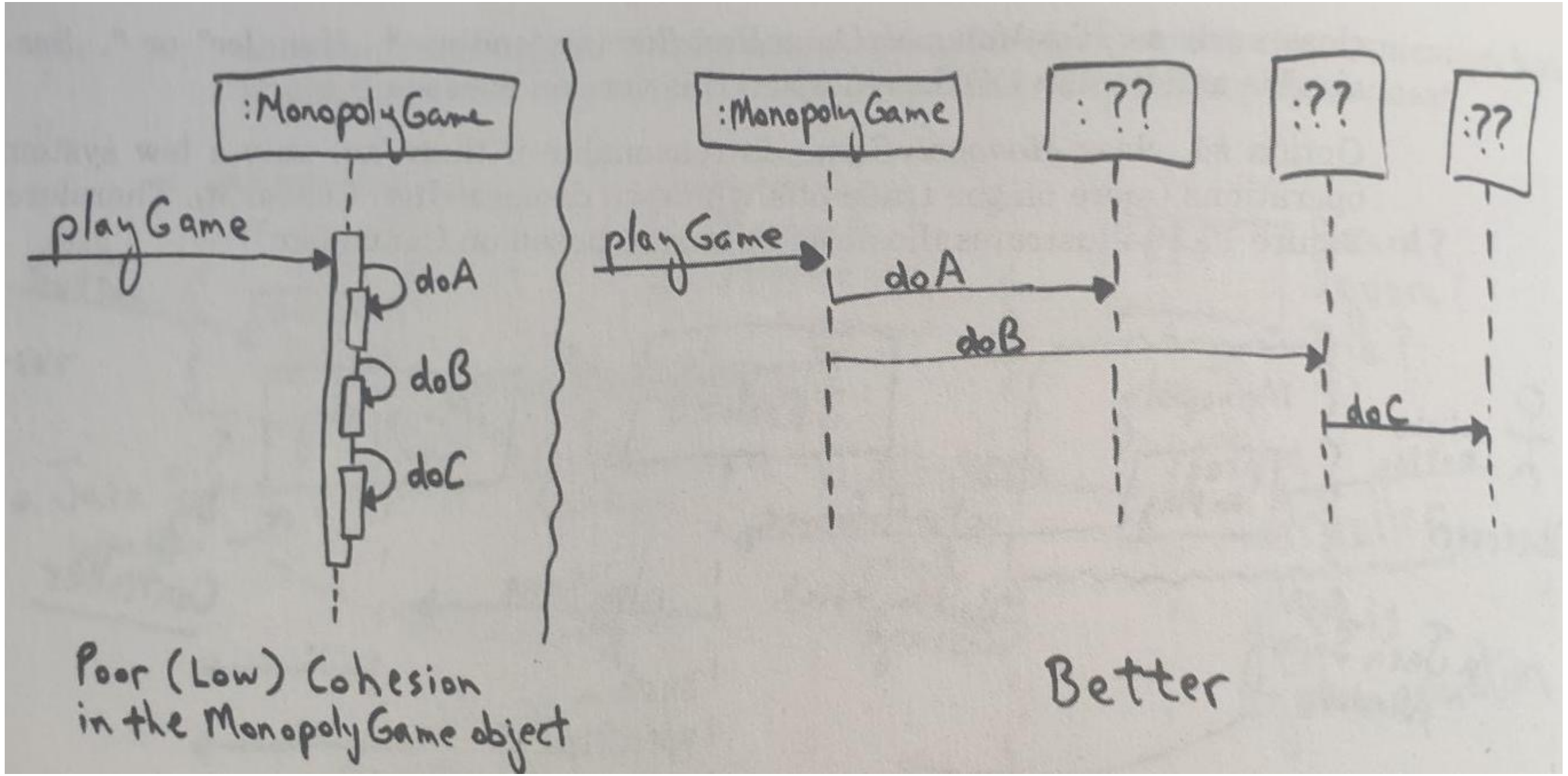
# Monopoly example



# 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