SWEN30006
Software Design and Modelling

GRASP:
More Objects with Responsibilities

Textbook: Larman Chapter 25

“Luck is the residue of design."

—Branch Rickey

Learning Objectives

On completion of this topic you should be able to:

- Apply four of the GRASP principles or patterns for Object-Oriented Design.
  - Polymorphism
  - Indirection
  - Pure Fabircation
  - Protected Variation

(Revisited) GRASP: General Responsibility Assignment Software Patterns/Principles

- **GRASP – Definition:** A set of patterns (or principles) of assigning responsibilities into an object.
- GRASP aids in applying design reasoning in a methodical, rational, and explainable way
  - Given a specific category of problem, GRASP guides the assignment of responsibilities to objects
- Useful to know and support Responsibility-Driven Design (RDD)
- **Pattern – Definition:**
  - A *named* and *well-known* problem/solution pair that can be applied in new contexts
  - A recurring successful application of expertise in a particular domain

(Last Week) GRASP: Learning Aid for RDD

- This subject will cover 9 GRASP patterns/principles
  - Creator
  - Information Expert
  - Low Coupling
  - High Cohesion
  - Controller*
  - Polymorphism
  - Indirection
  - Pure Fabrication
  - Protected Variations

*Controller will be covered in Workshop supplementary material

(This Week) Domain Model & GRASP

- More about *Domain* Model: Generalization-specialization hierarchy
- Another 4 GRASP patterns/principles
  - Polymorphism
  - Indirection
  - Pure Fabrication
  - Protected Variations

Domain Model:
Generalization-specialization hierarchy

Domain Model: Similar Concepts

- In the problem domain, there might be concepts that look very similar

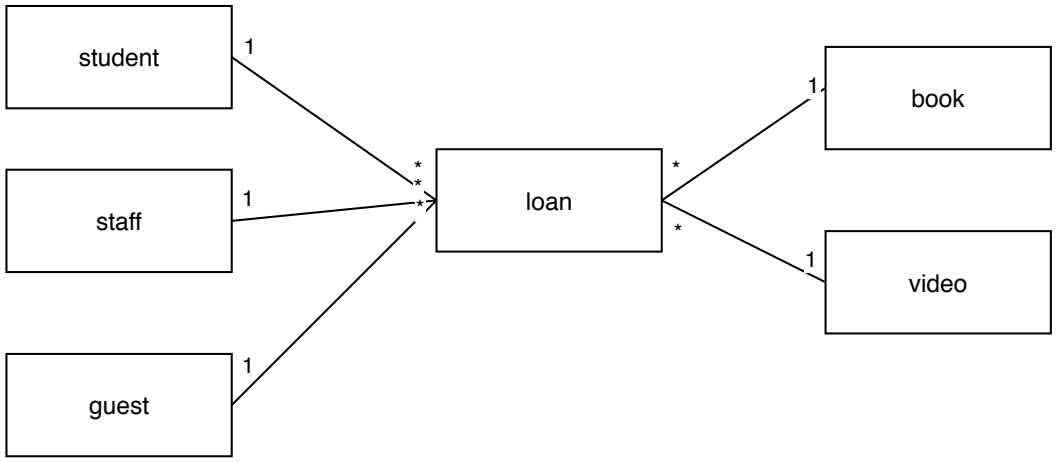
Week 2 Lecture's exercise: Identifying conceptual class

The library has **users** who can borrow **items**.

The library holds both **books** and **videos**: all **items** have a *title* and *year* of production (and a due date if they have been borrowed), while only books have *authors* and only *videos* have a duration

There are different kinds of **users**: **students**, **staff**, and **guests**. The only difference between these user types is that they have a different borrowing limit (max number of items they can borrow) which is based on their user type.

The library keeps track not only of which items are out on **loan** now, but the complete **history** of which users borrowed which items for which periods.

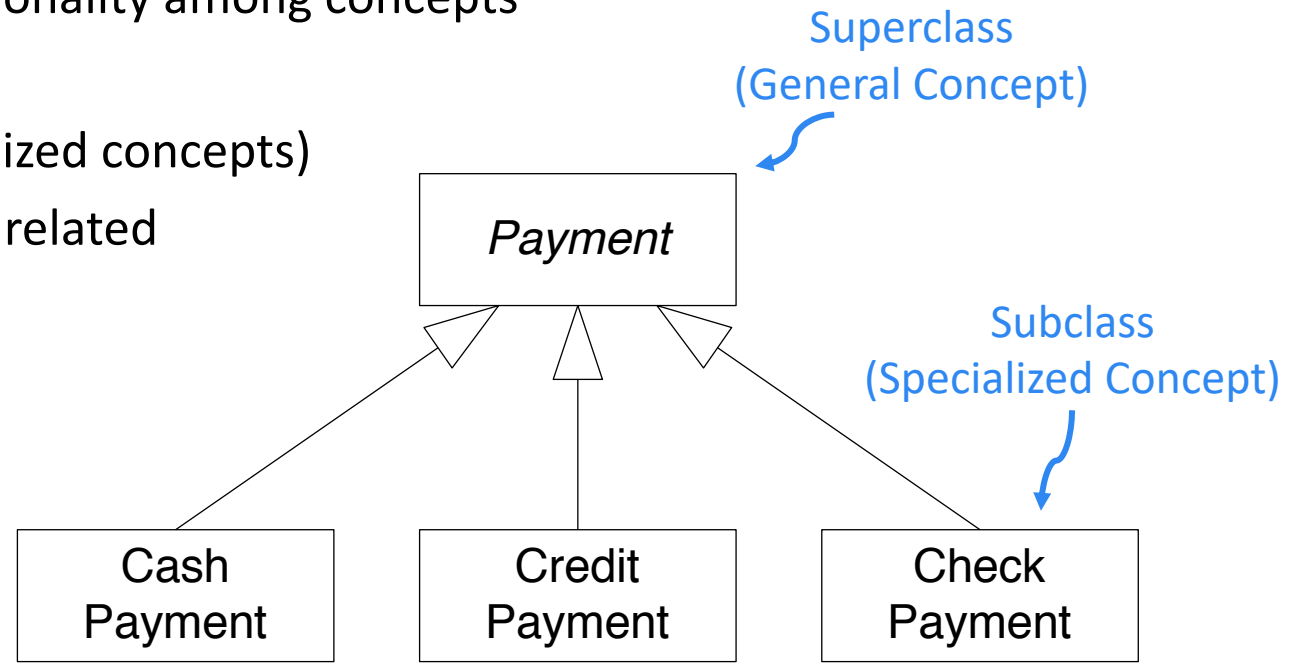


Partial domain model

- The conceptual classes are not well organized.
- The association between items, books, and video is missing. Same for users

Domain Model: Generalization-Specialization Class Hierarchy (or Class Hierarchy)

- When concepts are very similar, it is valuable to organise them
- ***Generalization:*** The activity of identifying commonality among concepts
  - Define *superclass* (general concept)
  - Define relationships with *subclasses* (specialized concepts)
  - Conceptual subclasses and superclasses are related in terms of set membership
- *Why?*
  - Economy of expression
  - Reduction in repeated information
  - Improved comprehension



A Domain Class Model

The class hierarchy of the domain model often inspires the design of software objects in the design model.

See more in Textbook Chapter 31

Domain Model: Generalization-Specialization Class Hierarchy (or Class Hierarchy)

Guideline for creating subclasses:

1. Subclass has additional attributes of interest
2. Subclass has additional associations of interest
3. Subclass is operated on, handled, reacted to, or manipulated differently to the other classes in noteworthy ways

Modelling Guidelines:

a) Declare superclasses abstract

b) Append the superclass name to the subclass

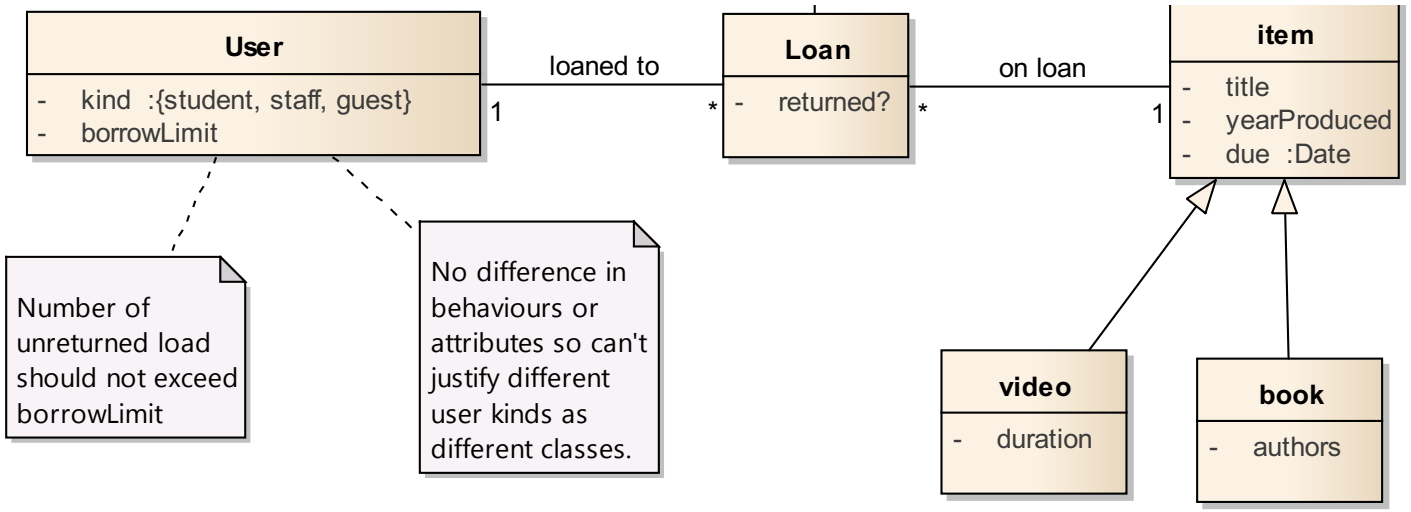
Class Hierarchy: Example

The library has **users** who can borrow **items**.

The library holds both **books** and **videos**: all **items** have a *title* and *year* of production (and a due date if they have been borrowed), while only books have *authors* and only *videos* have a duration

There are different kinds of **users**: **students**, **staff**, and **guests**. The only difference between these user types is that they have a different borrowing limit (max number of items they can borrow) which is based on their user type.

The library keeps track not only of which items are out on **loan** now, but the complete **history** of which users borrowed which items for which periods.



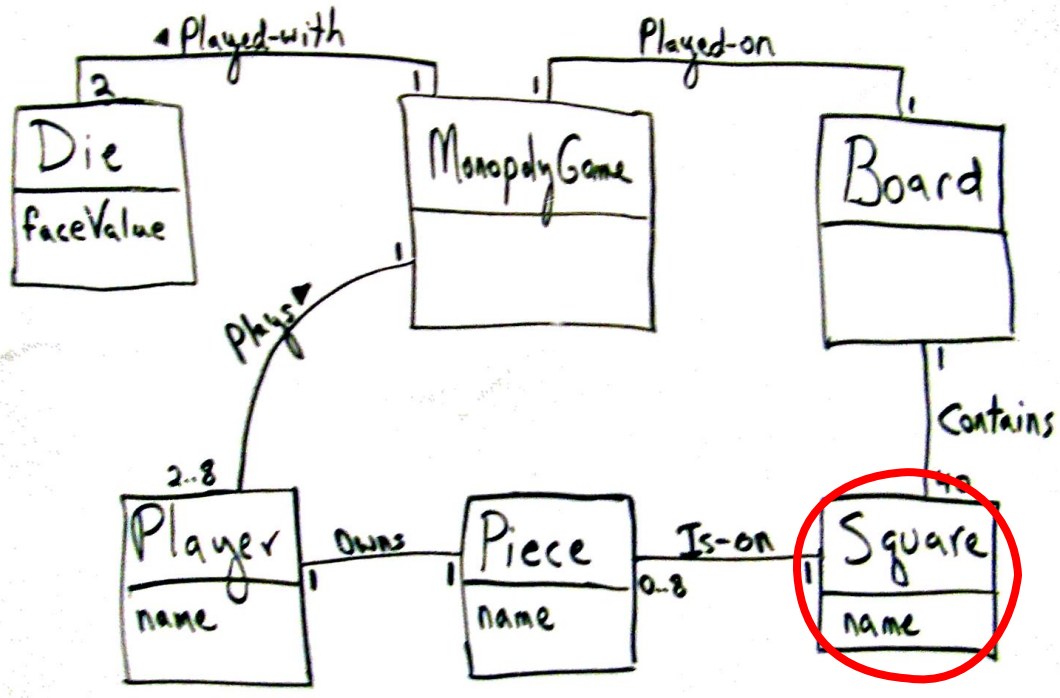
Partial domain model

GRASP: Polymorphism

Polymorphism

- **Problem (1):** How handle alternatives based on type?
  - Conditional variation: Adding new alternatives and if-then-else (or case-statement) are required many places
  - E.g., behaviour varies based on some conditions
- **Problem (2):** How to create pluggable software components?
  - E.g., Client-Server relationship: How to replace Server component without affecting Client?
- **Solution:** 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
  - Polymorphic operations: Operations with the same interface
  - ***Corollary*:** Do not test for the type of an object and use conditional logic to perform varying alternatives based on type.

Example: Monopoly



Monopoly Game Simulation Domain Model

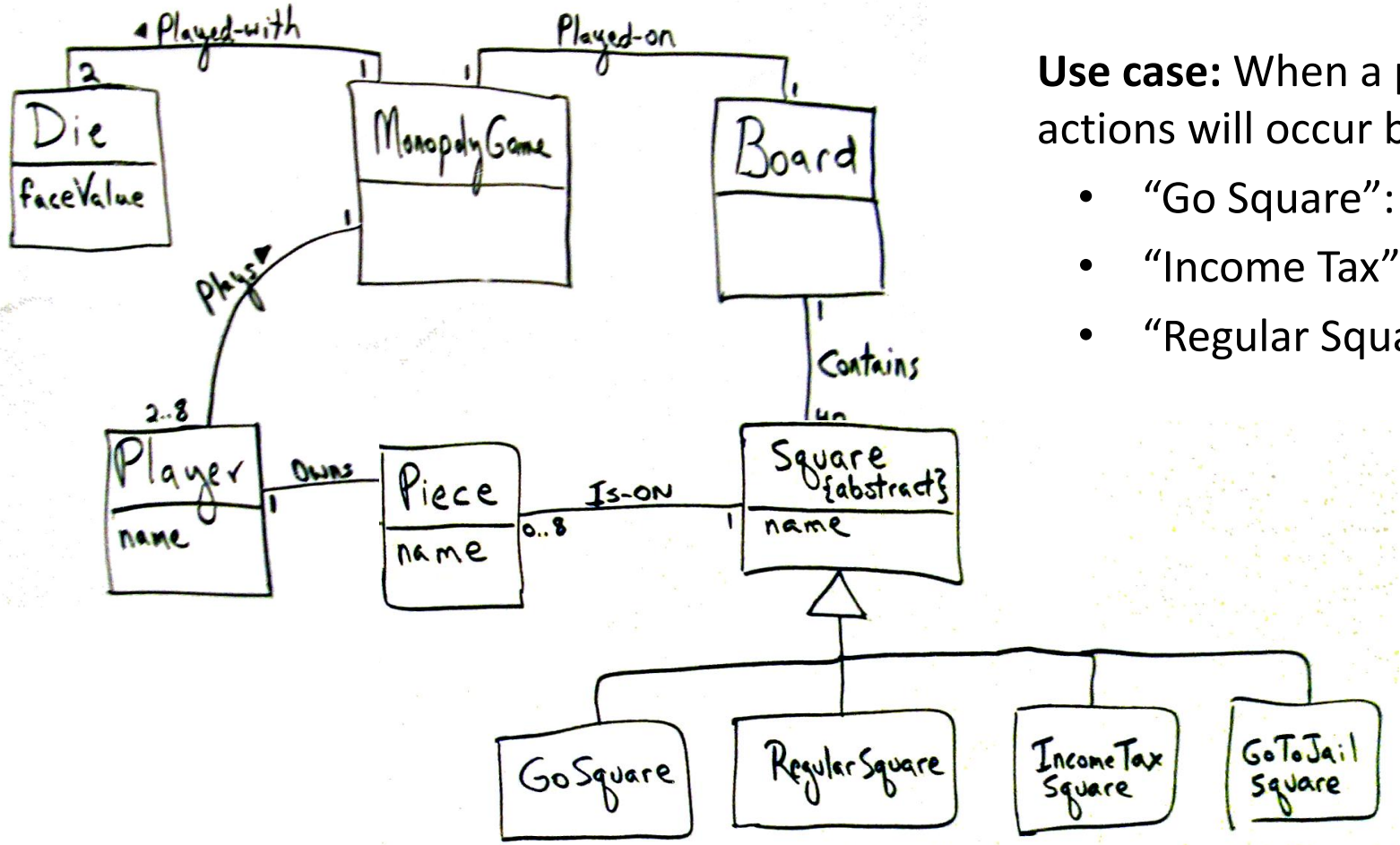
Use case: When a player lands on a square, different actions will occur based on the types of square, e.g.,

- "Go Square": A player receives $200
- "Income Tax": A player pays a tax of the income
- "Regular Square": A player does nothing

Realization: The different types of Square can be seen as concepts in the domain as well. It should be captured in the *domain* model

→ Since the concepts are similar, use a **generalization specialization class hierarchy (class hierarchy)** to organize these concepts

Example: Monopoly – Update *Domain* Model

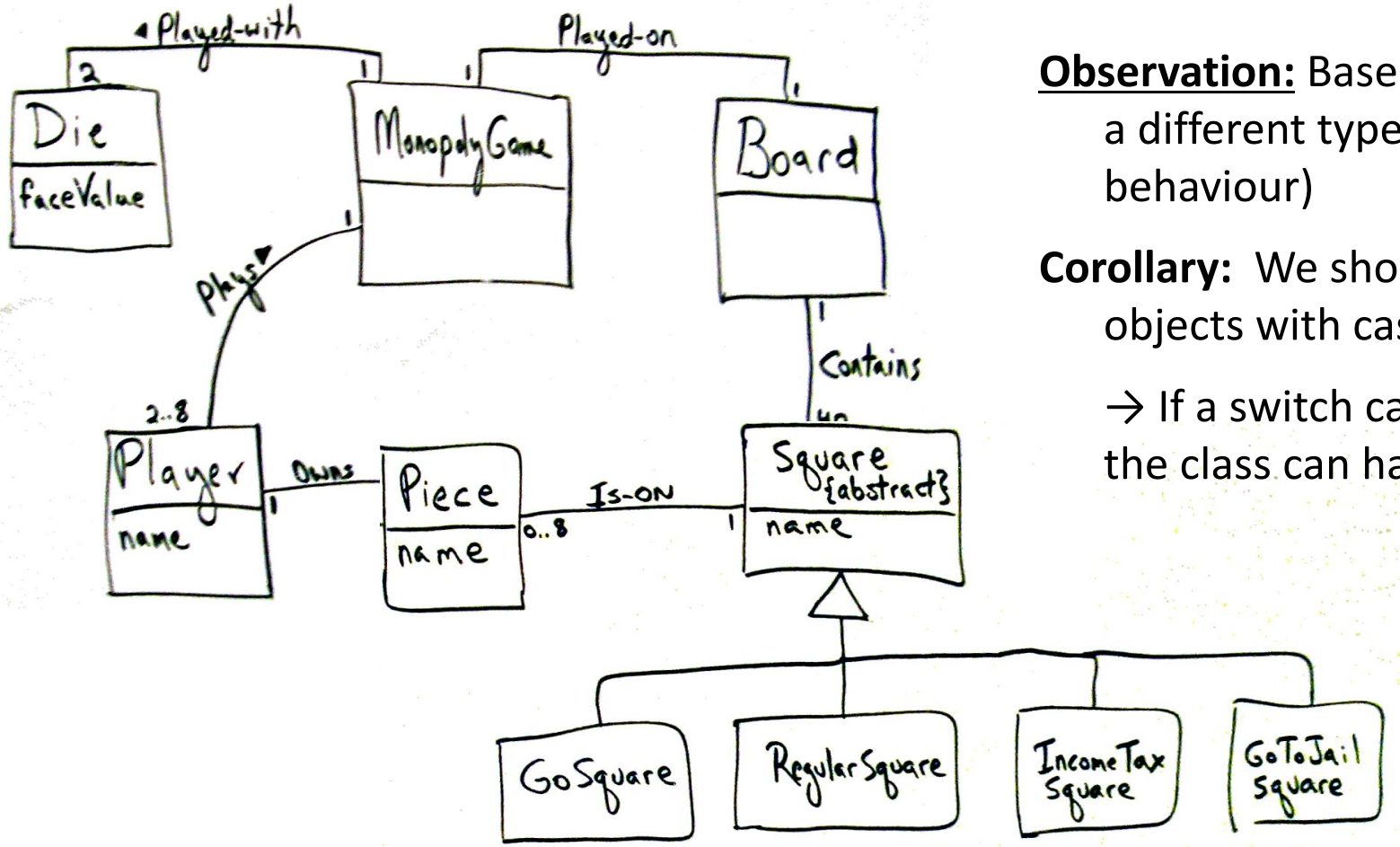


Monopoly Game Simulation Domain Model

Use case: When a player lands on a square, different actions will occur based on the types of square, e.g.,

- "Go Square": A player receives $200
- "Income Tax": A player pays a tax of the income
- "Regular Square": A player does nothing

Example: Monopoly – Create *Design* Model (1)



Monopoly Game Simulation Domain Model

Observation: Based on Use case and domain model, a different type of square has a different rule (i.e., behaviour)

Corollary: We should not design of our software objects with case logic (e.g., a switch statement)

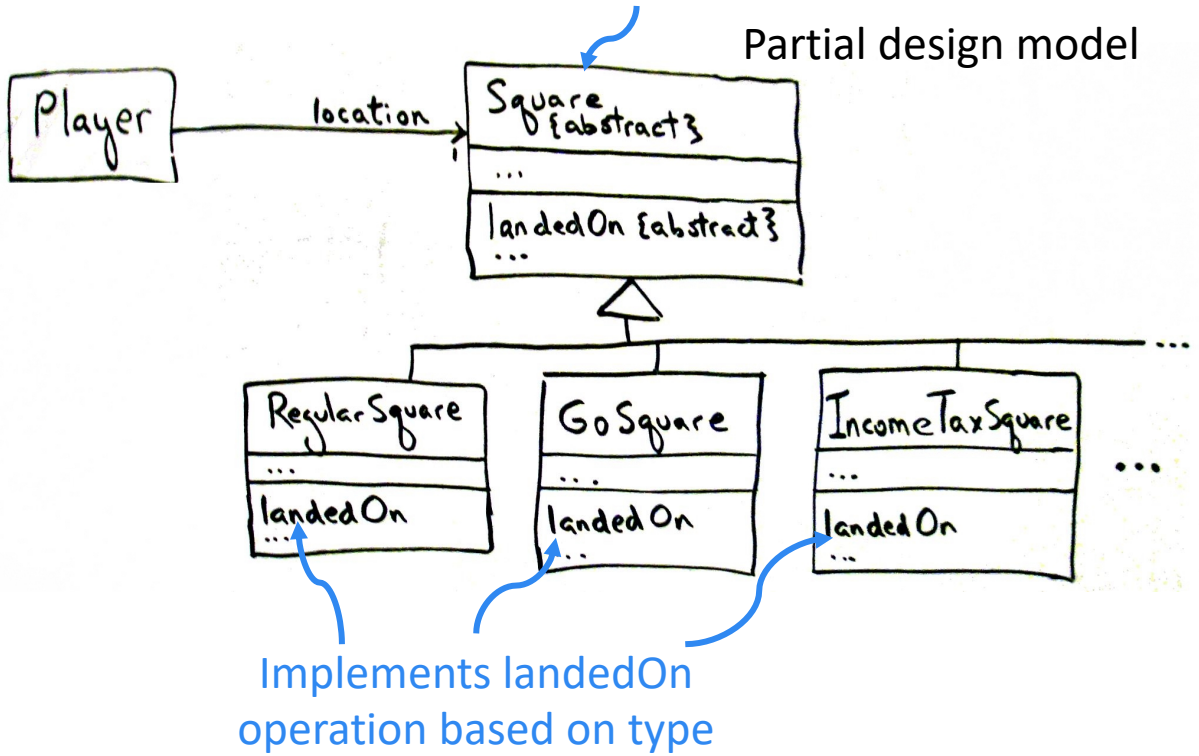
→ If a switch case (or if-else) is used in Square class, the class can have *low cohesion* and can be bloated!

Example: Monopoly – Create *Design* Model (2)

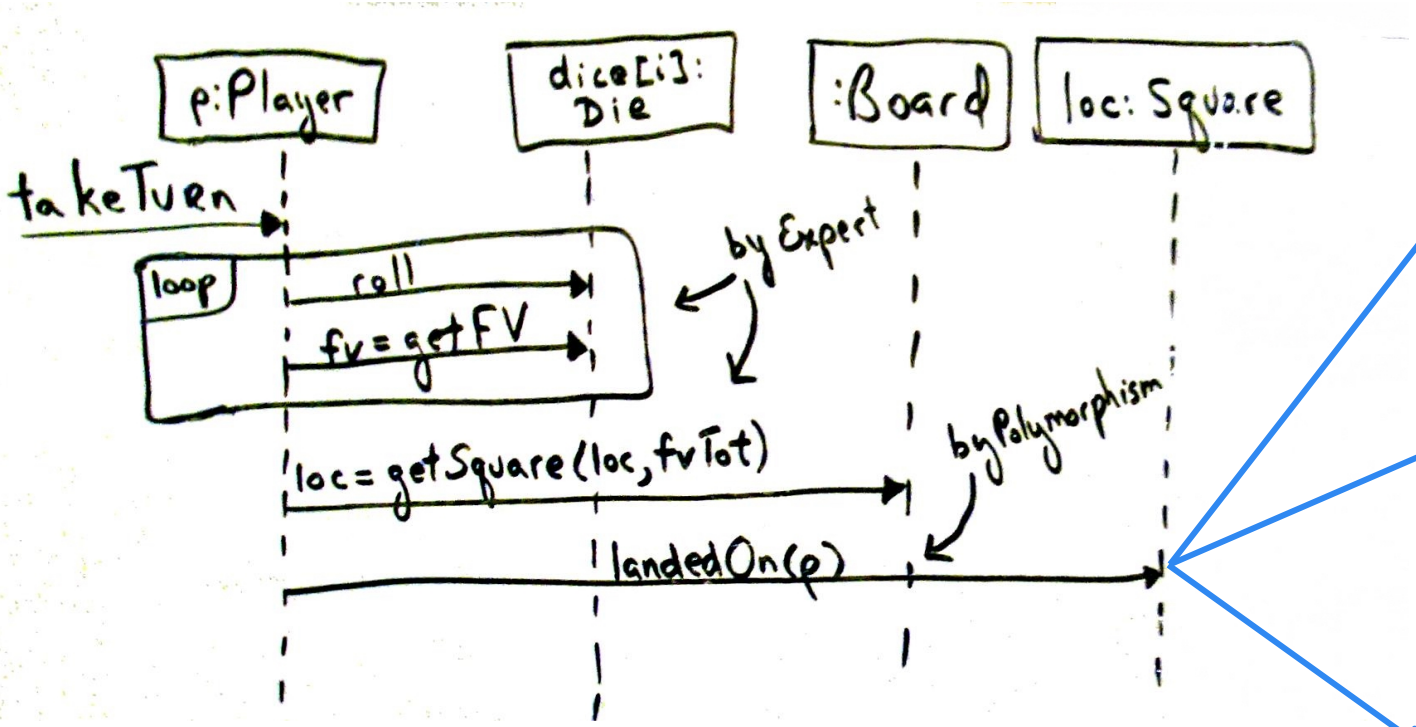
Analysis: There are alternatives based on type of Squares
→ Polymorphism should be used.

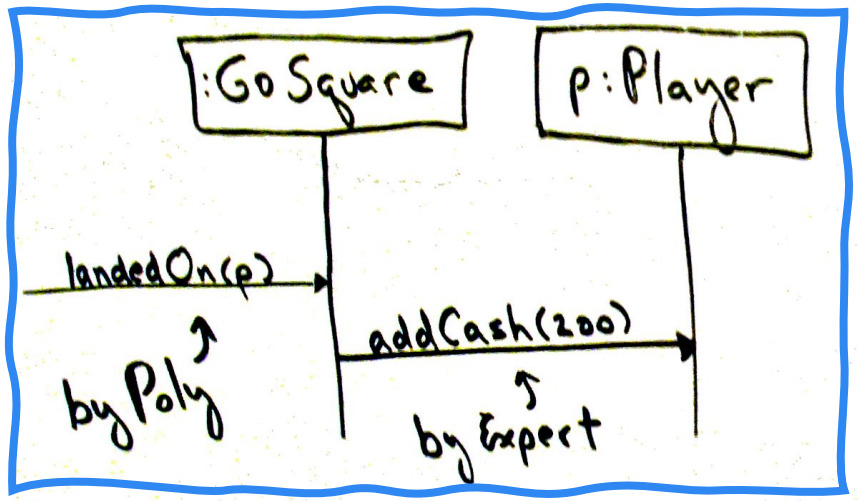
Approach: Based on Polymorphism, we should create a polymorphic operation for each type for which the behavior varies.

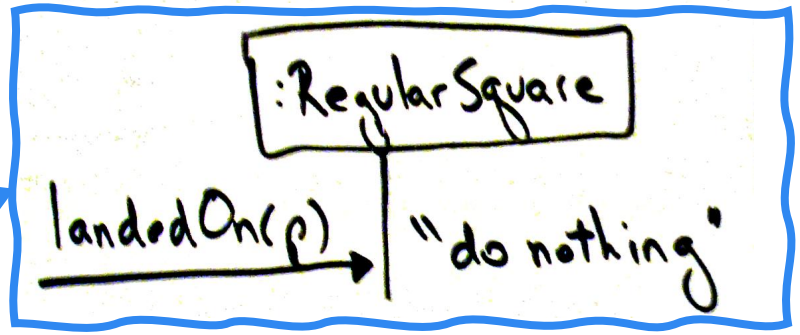
- What are types that Square varies?
  → ”Go Square”, “RegularSquare”, “IncomeTaxSquare”, so on
- What is the operation that varies?
  → The behaviour of Square will vary when a player *lands on* a square
  → Then, the responsibility of “landedOn” should be a *polymorphic operation*

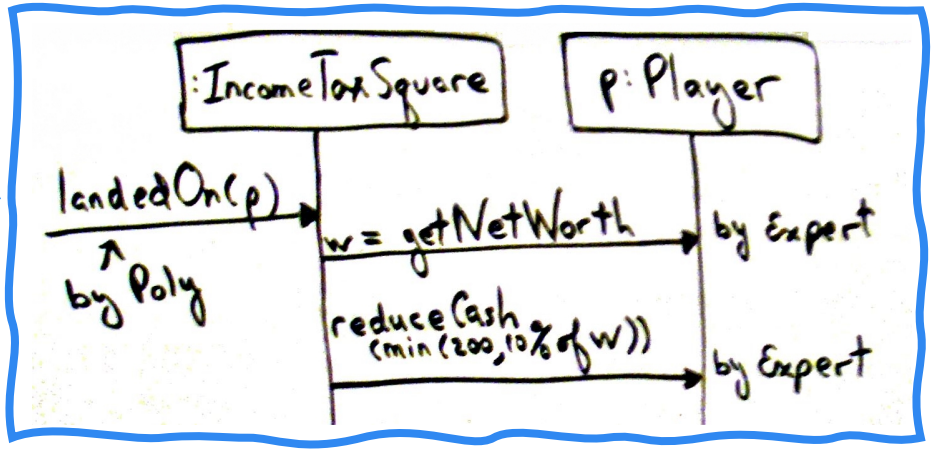


Example: Monopoly









The behaviour of square is based on the type of Square that Board returns to Player

Polymorphism: Discussion

- Polymorphism is a *fundamental* pattern in designing how a system is organised to *handle similar variations*
  - Based on Polymorphism, a design is easily extended to handle new variations
- **Contradictions**:
  - Designing systems with interfaces and polymorphism for speculative “future-proofing” against an unknown variation could be useful.
  - However, the unnecessary effort might occur if polymorphism is applied at the points where variations, in fact, are improbable and will never actually arise.

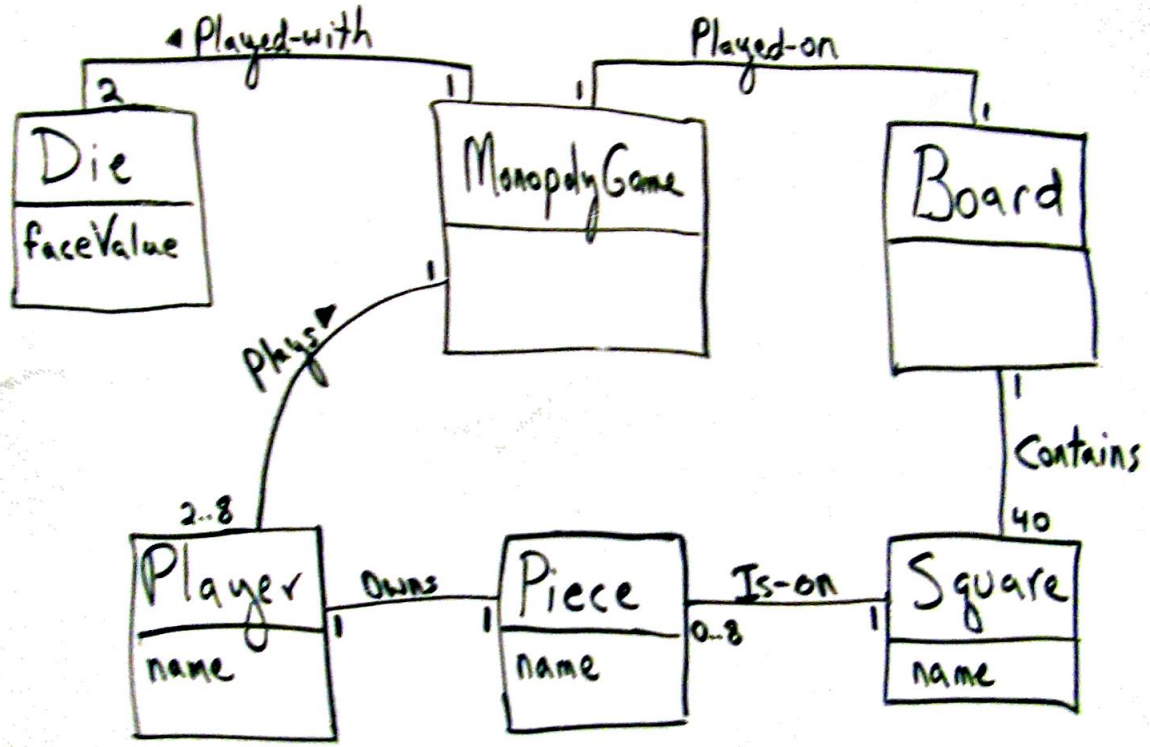
GRASP:

Pure Fabrication

Pure Fabrication

- **Problem:** Which object should have responsibility when you do not want to violate *High Cohesion* and *Low Coupling*, but solutions offered by other patterns (e.g. Expert) are not appropriate?
  - Domain model often inspires the design of software objects (achieving low representational gap)
  - In many situations, assigning responsibilities only based on the domain model often leads to the problem of poor cohesion or coupling.
- **Solution:** Assign a highly cohesive set of responsibilities to an *artificial* or convenience class that does not represent a problem domain concept
  - Made up a class to support high cohesion, low coupling, and reuse

Example: Monopoly

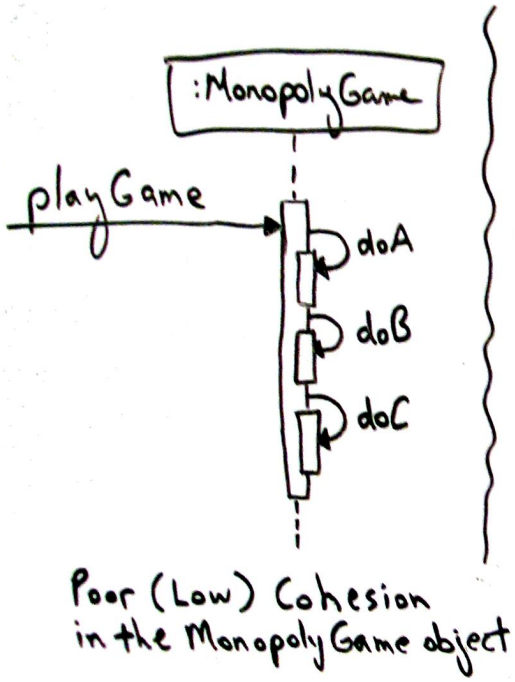


Monopoly Game Simulation Domain Model

Who should be responsible for rolling the dice?

- Based on the domain model & Expert, MonopolyGame class should be assigned for this responsibility

<u>Discussion:</u> It is acceptable, but lead to low cohesion



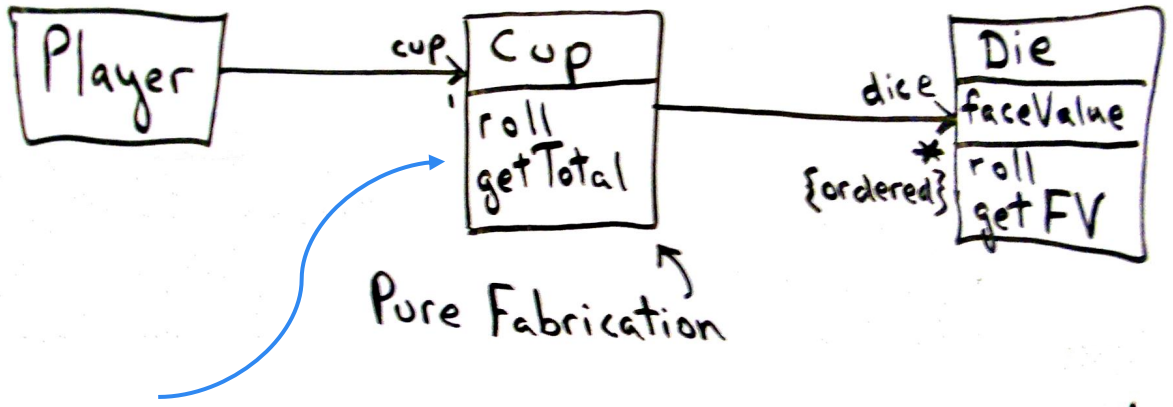
Based on the domain model, the design can violate high cohesion

Let's use **Pure Fabrication**!

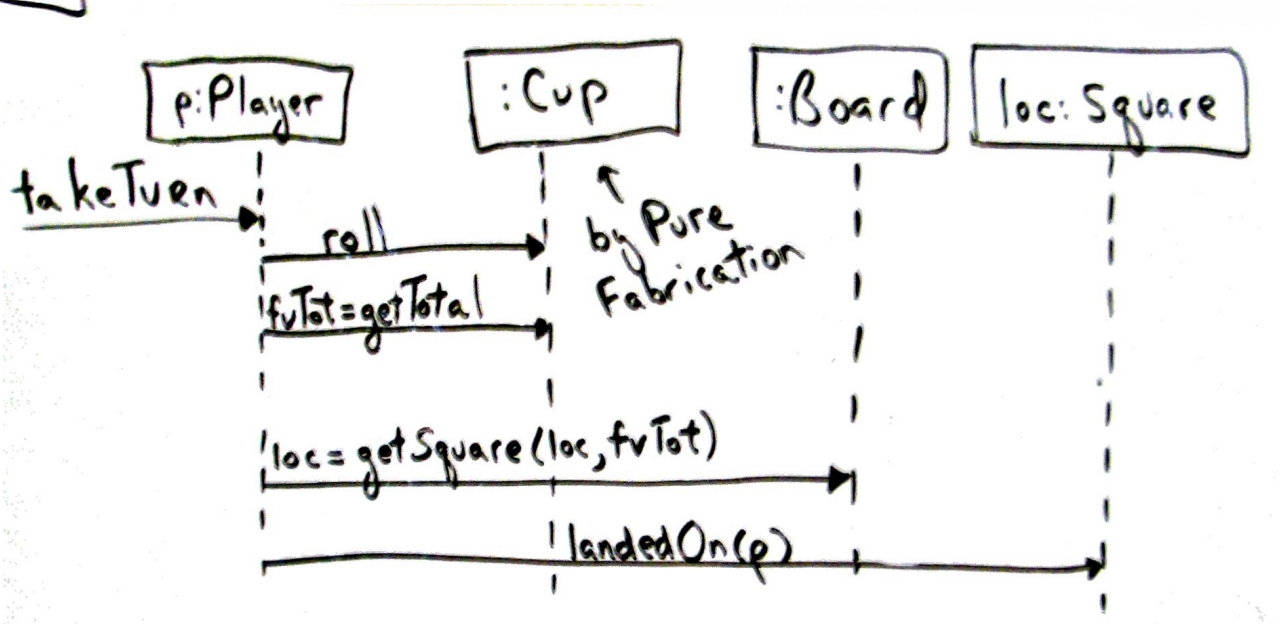
Example: Monopoly

Who should be responsible for rolling the dice?

Solution: By Pure Fabrication, let's create an artificial class which is responsible for rolling dice



In the domain, Cup was not used in Monopoly Game



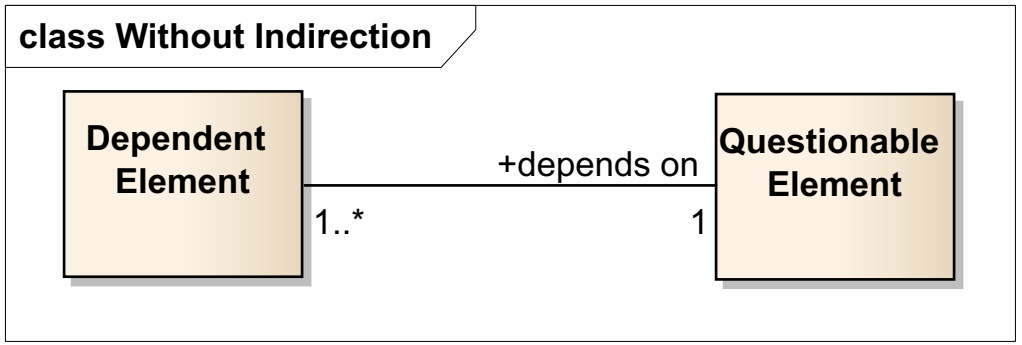
Pure Fabrication: Contradictions

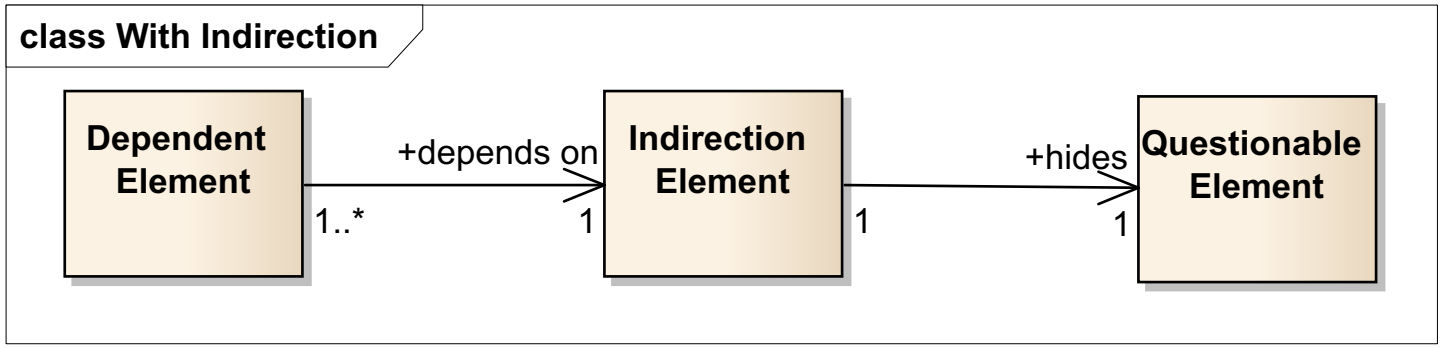
- Pure Fabrication should *not* be used as excuse to add new objects
- If overused Pure Fabrication, the design can end up with one class has one responsibility
  - Too many behavior objects that have responsibilities *not* co-located with the information required for their fulfillment
  - can adversely affect coupling (i.e., high dependencies)

GRASP: Indirection

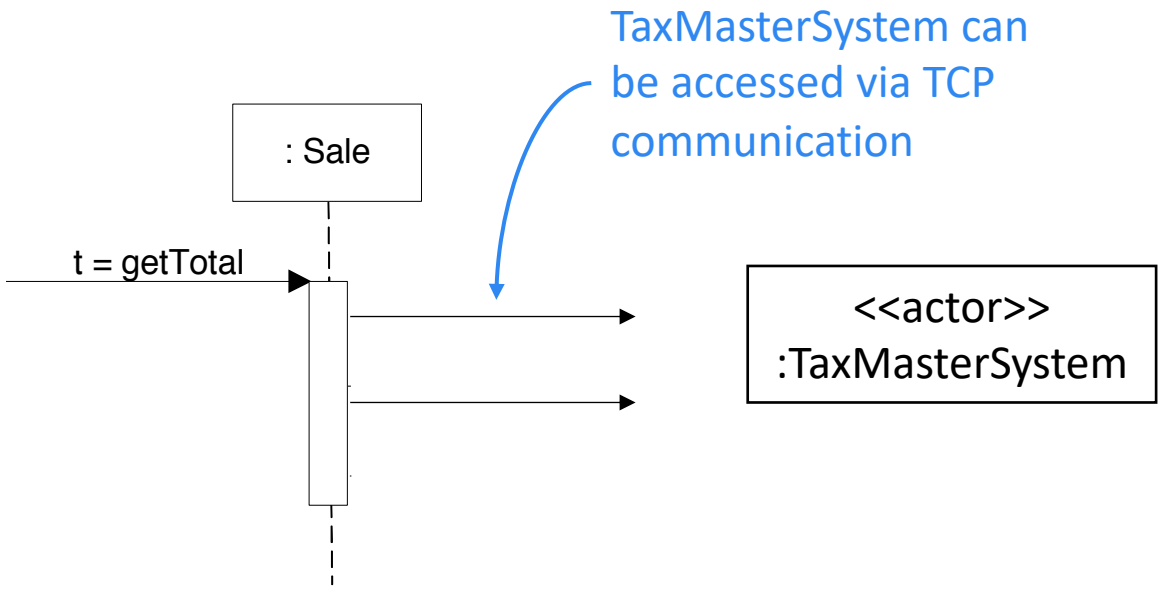
Indirection

- **Problem:** Where to assign a responsibility, to *avoid direct coupling* between two (or more) things?
  - How to de-couple objects so that low coupling is supported and reuse potential remains higher?
- **Solution:** Assign the responsibility to an *intermediate* object to mediate between other components or services so that they are not directly coupled.
  - The intermediary creates an *indirection* between the other components.





Example: POS



POS will use Tax Master System (an external service) to calculate tax. Who should be responsible for calculating tax?

- By Expert, Sale is responsible for calculating total. So, Sale should also calculate tax
- Should Sale be directly use TaxMasterSystem via TCP communication?

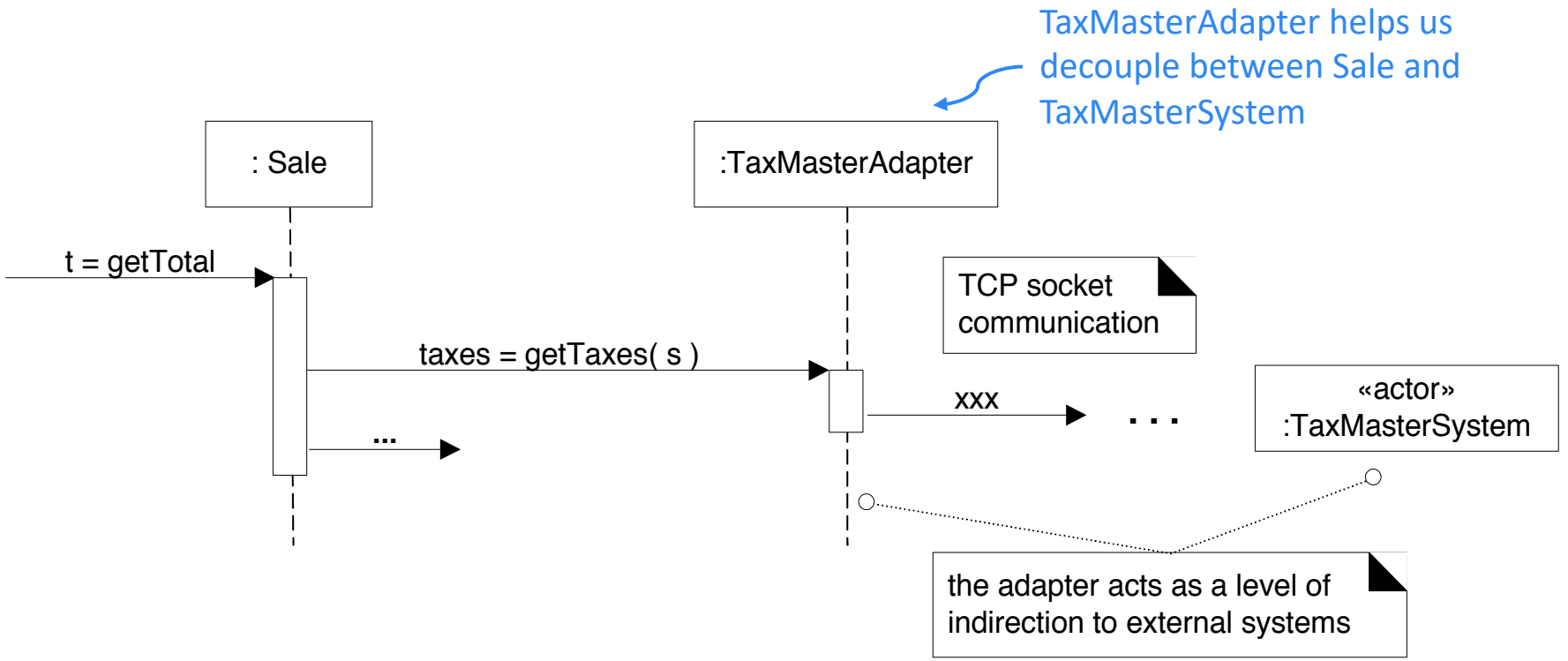
Discussion: If this responsibility is assigned to Sale,

→ a lot of TCP communication to TaxMasterSystem will be implemented in Sale

→ Sale class is highly coupled with TaxMasterSystem

Example: POS

Solution: By Indirection, the responsibility of communicating with TaxMasterSystem should be assigned to an intermediate class, i.e., TaxMasterAdapter



Indirection: Discussion

- Many Pure Fabrications are generated because of Indirection.
  - Indirection: Assigning any class as an intermediary
  - Pure Fabrications: Assign an artificial class (that does not represent the domain) as an intermediary
- The motivation for Indirection is usually Low Coupling (i.e., decouple objects for future reuse)
- Old adage:

  “Most problems in computer science can be solved by another level of indirection”
- Counter adage:

  “Most problems in performance can be solved by removing another layer of indirection!”

GRASP:
Protected Variations

Protected Variation

- **Problem:** How to design objects, subsystems, and systems so that the variations or instability in these elements does not have an undesirable impact on other elements?
  - Points of change include:

    Variation point: variations in existing system or requirements

    Evolution point: speculative variations that may arise in the future
- **Solution:**
  - Identify points of known or predicted variation or instability;
  - Assign responsibilities to create a stable interface* around them

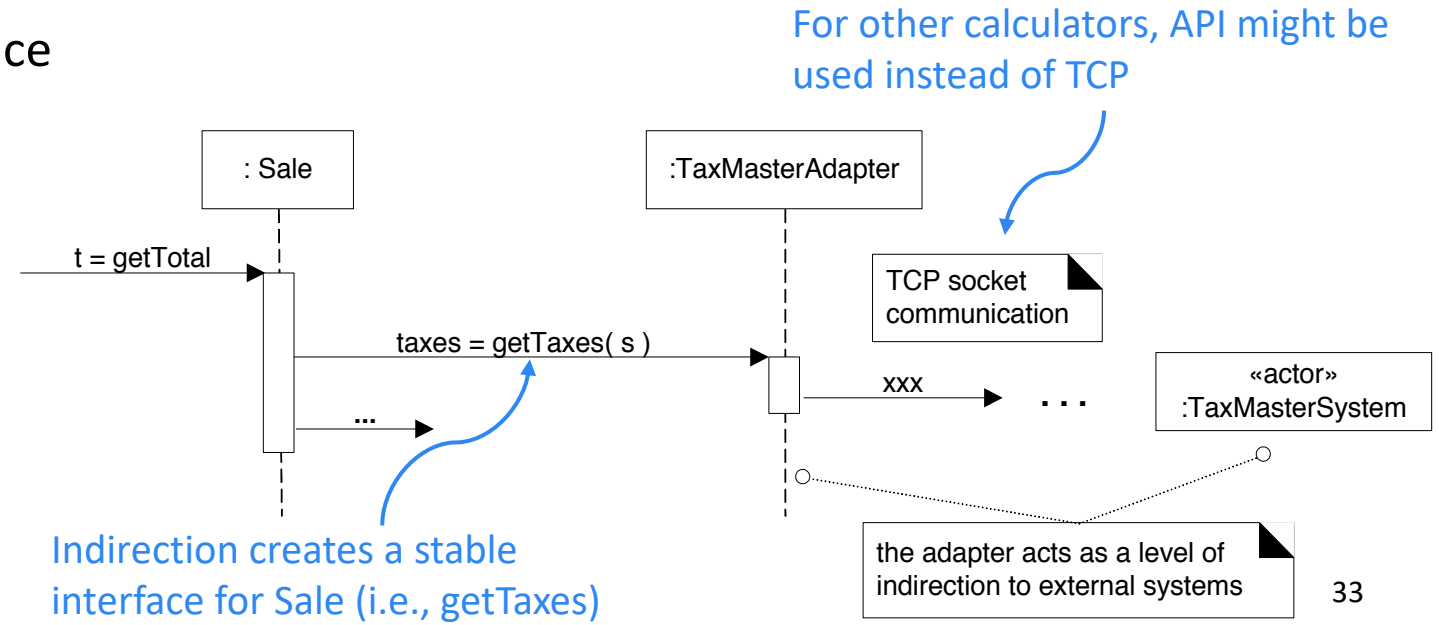
**Interface:* a means of access (not only a programming language interface or Java interface)

Example: POS

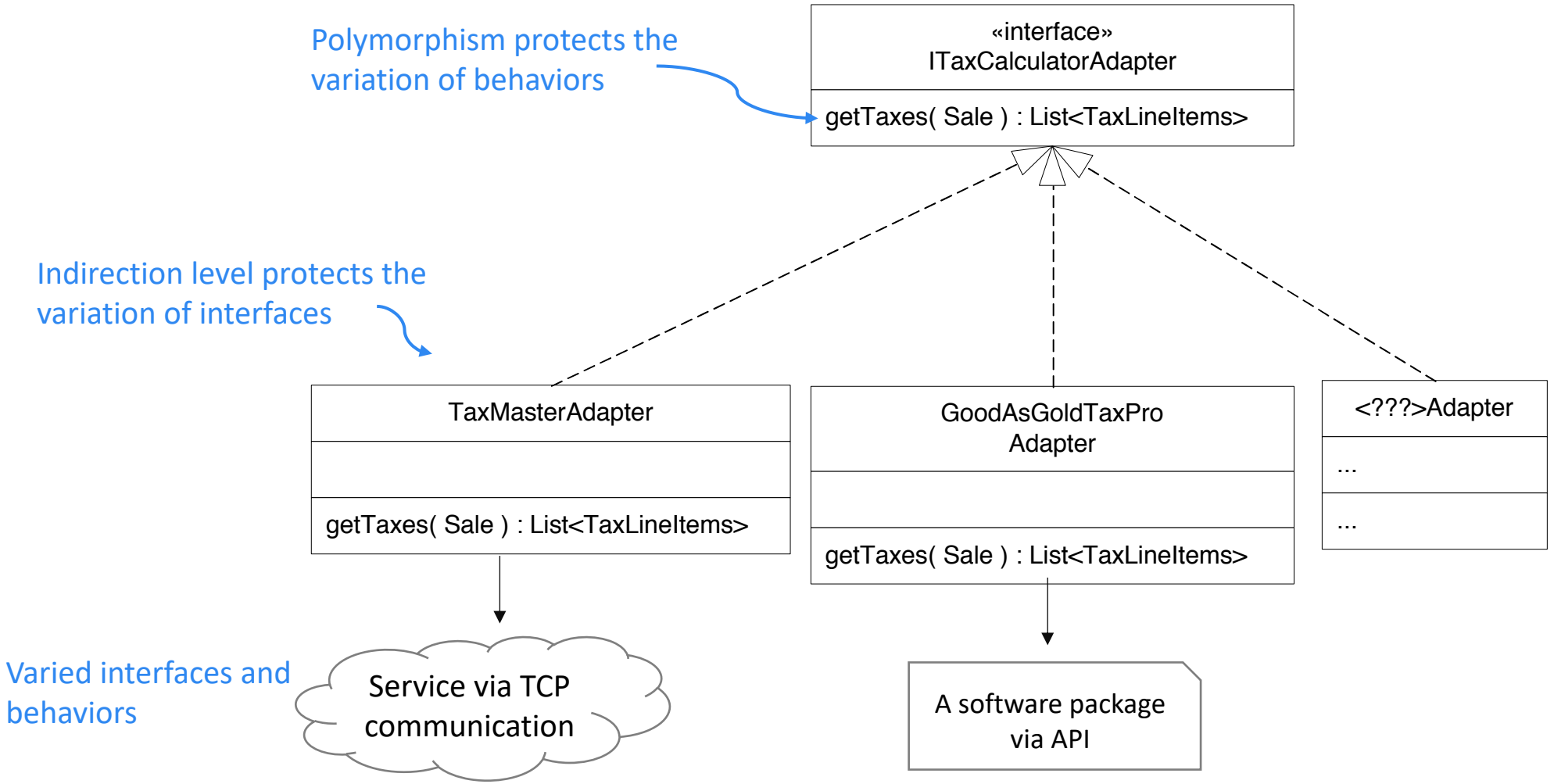
POS currently uses 'TaxMasterSystem' to calculate tax. In the future, other tax calculators might be used.

Analysis: To protected variation:

1. What is the point that will be known or predicted variation or instability?
   → The point of instability or variation is the different interfaces or APIs of external tax calculators
2. Assign responsibilities to create a stable interface* around them
   → Use Indirection to create a stable interface
   → yet, the behaviour of getTaxes() *varies* based on the type of tax calculators
   → By Polymorphism, let's make getTax() as a polymorphic operation!



Example: POS


Polymorphism protects the variation of behaviors
«interface»
ITaxCalculatorAdapter
getTaxes(Sale) : List<TaxLineItems>
Indirection level protects the variation of interfaces
TaxMasterAdapter
getTaxes(Sale) : List<TaxLineItems>
GoodAsGoldTaxPro Adapter
getTaxes(Sale) : List<TaxLineItems>
<???>Adapter
...
...
Varied interfaces and behaviors
Service via TCP communication
A software package via API

Protected Variation: Discussion

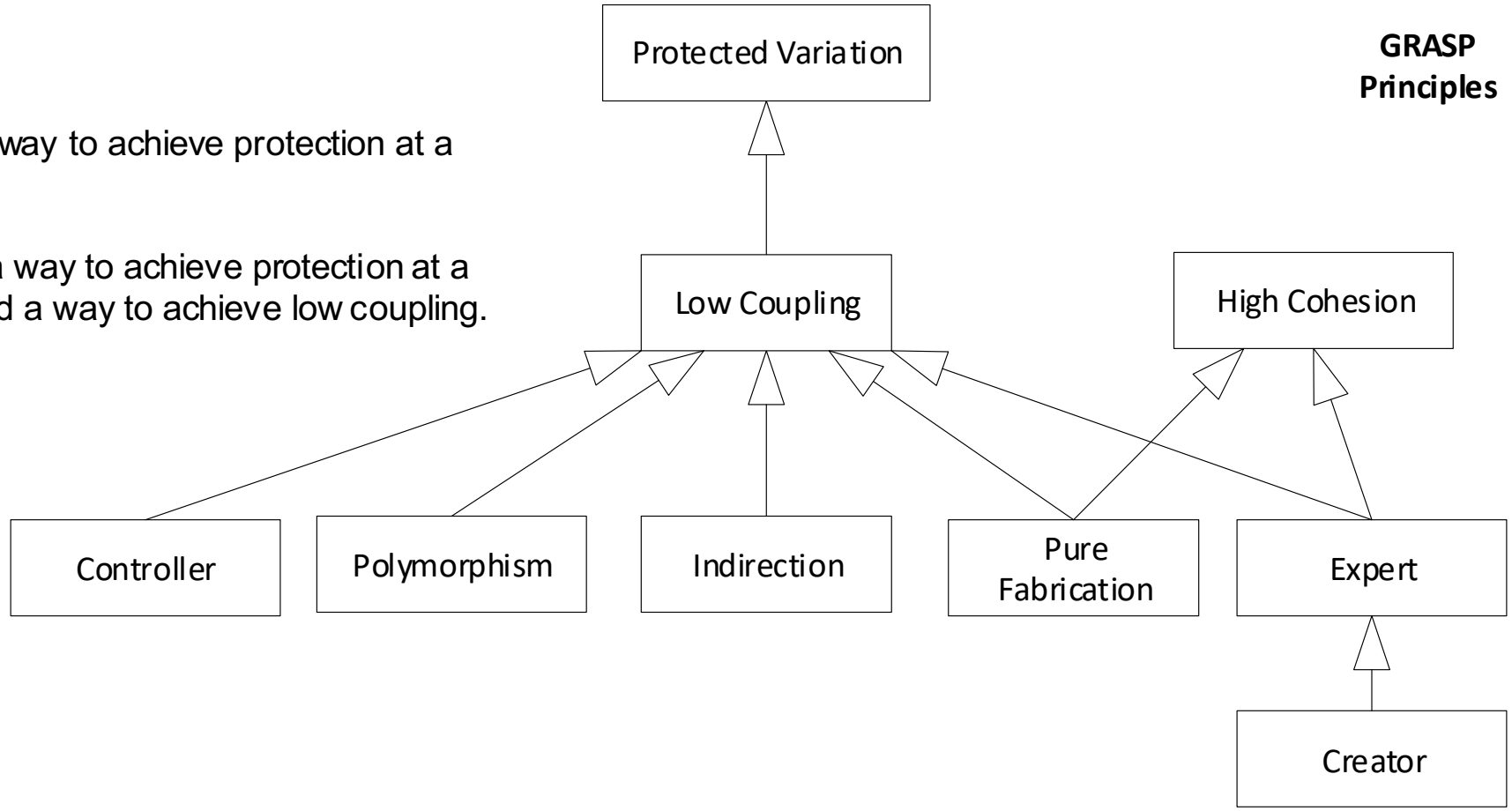
- Protected Variation (PV) is *very* important, fundamental principle of software design!
- PV is a root principle motivating most of the mechanisms and patterns in programming and design.
  - For example: Core Protected Variations Mechanisms, Data-Driven Design, Service Lookup, Interpreter-Driven Design, (See more in textbook)
- PV is equivalent to **Open-Closed Principle** (OCP), described by Bertrand Meyer
  - OCP: Modules should be both open (for extension; adaptable) and closed (to modification that affects clients).
  - A class can be closed w.r.t. attribute access (access methods only – no changes), but open to modification of underlying attributes (+ addition of new methods).
- **Contradictions**: Similar to Polymorphism
  - Cost of speculative "future-proofing" at evolution points can outweigh the benefits
  - It may be cheaper/easier to rework a simple "brittle" design

Relationship between GRASP Principles

For example:

Low coupling is a way to achieve protection at a variation point.

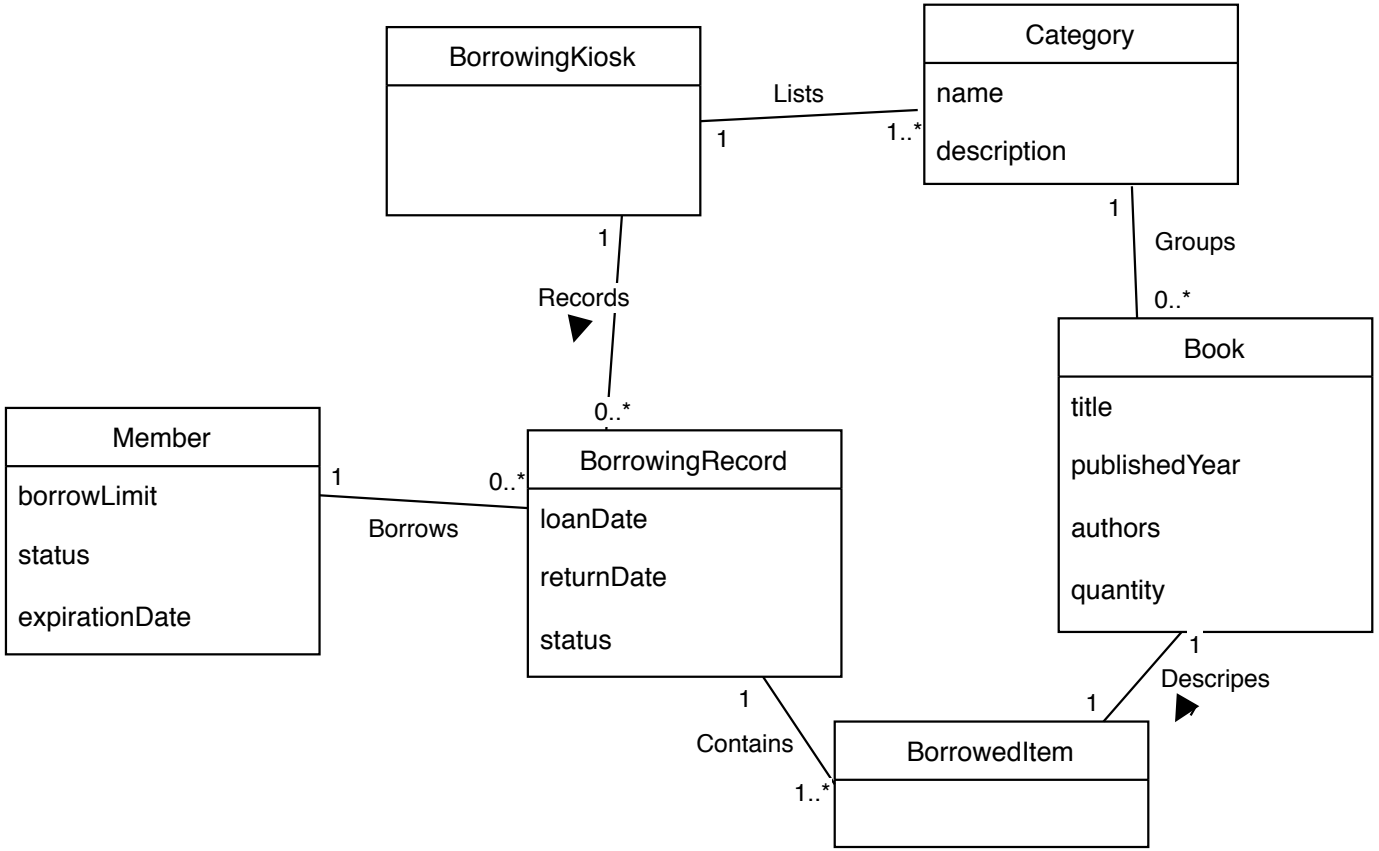
Polymorphism is a way to achieve protection at a variation point, and a way to achieve low coupling.



Summary & Remarks

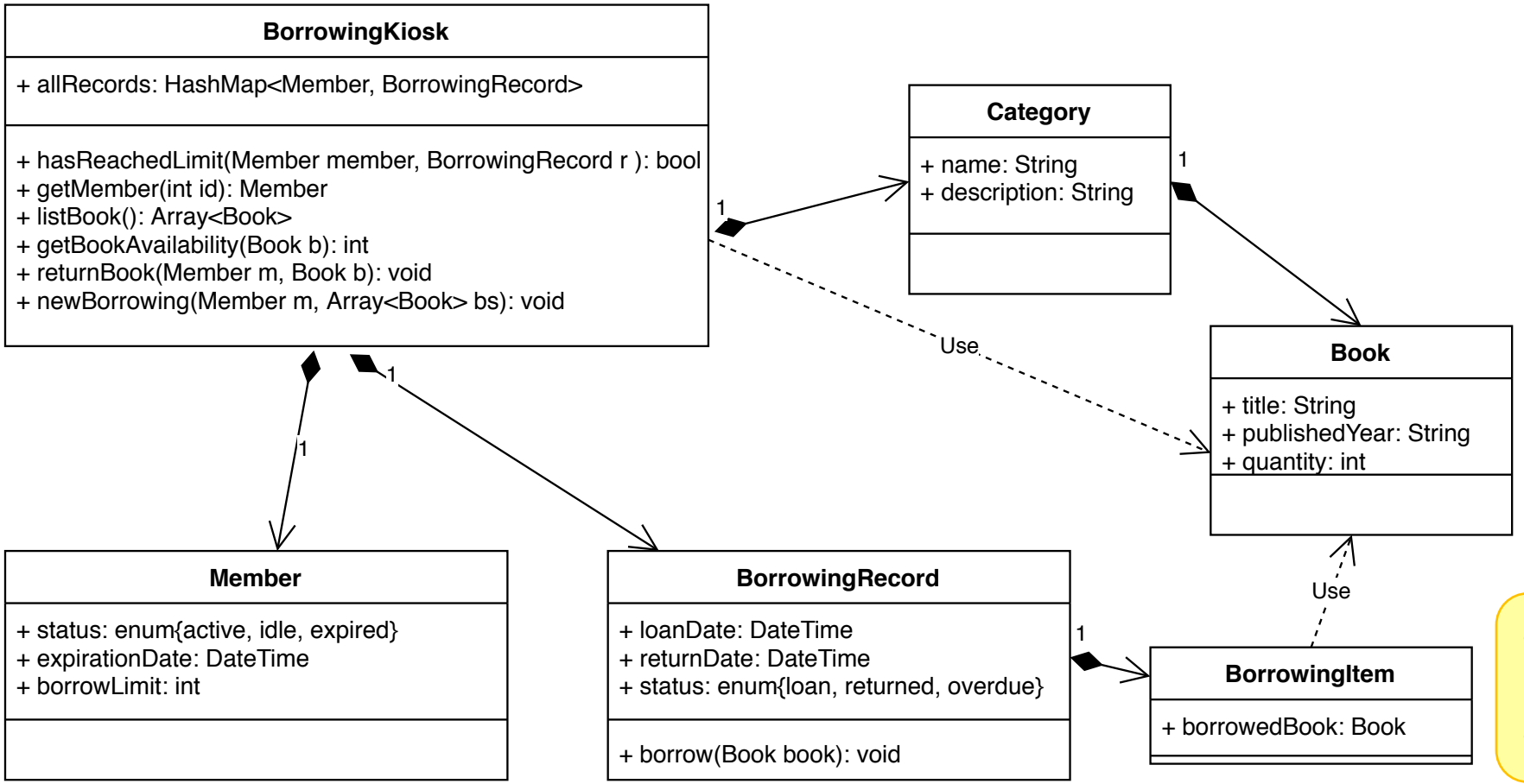
- Protected Variation, Low Coupling, and High Cohesion are a fundamental principle that we aim to achieve when designing software objects
- Use the remaining 6 GRASP patterns to achieve the principles

Exercise: Library borrowing system



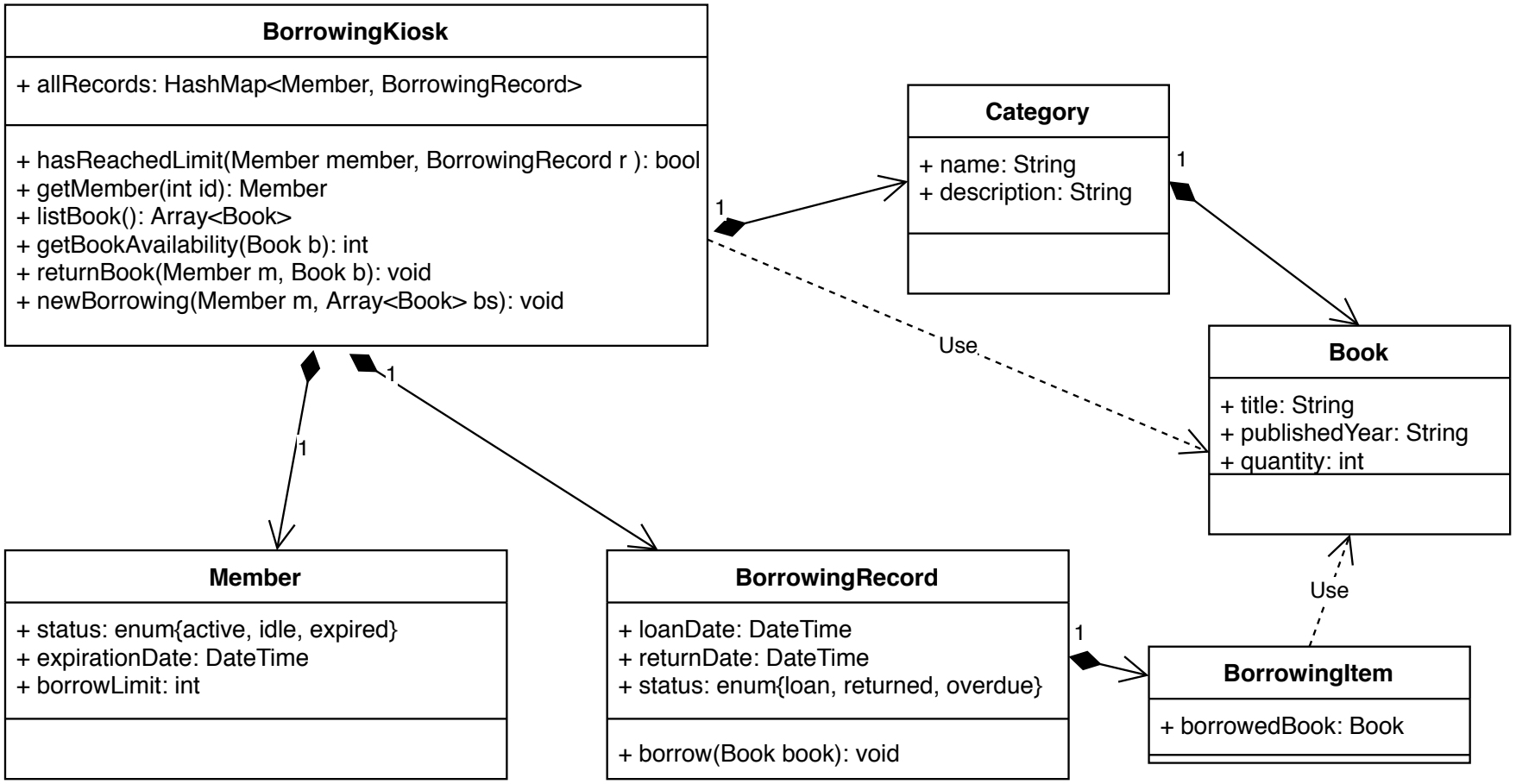
A domain model of Library Borrowing System

Exercise: Analyze the design model based on GRASP



- Which GRASP principles that are likely being violated?
- And how can we improve?

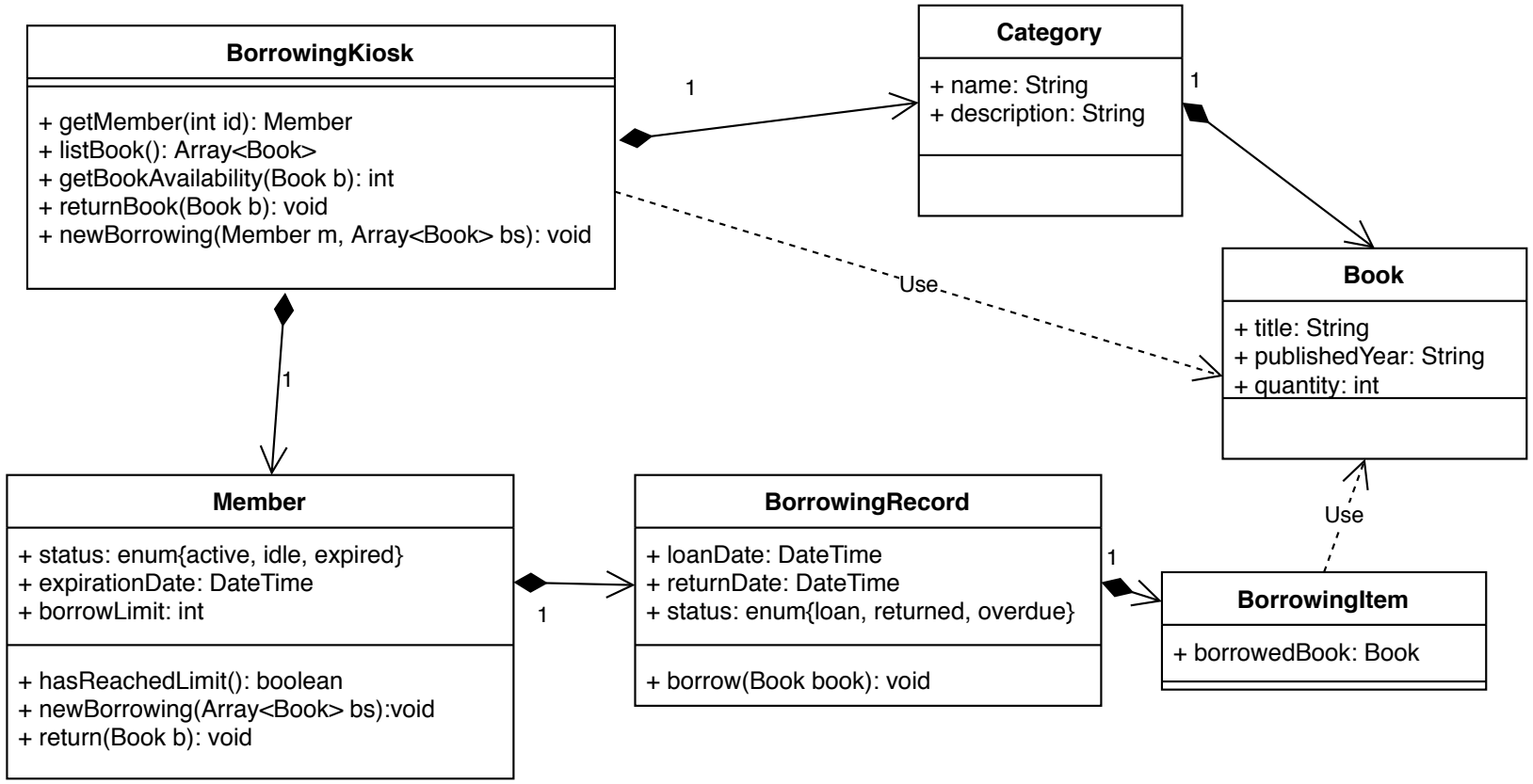
Exercise: Analyze the design model based on GRASP



- Which GRASP principles that are likely being violated?
- And how can we improve?

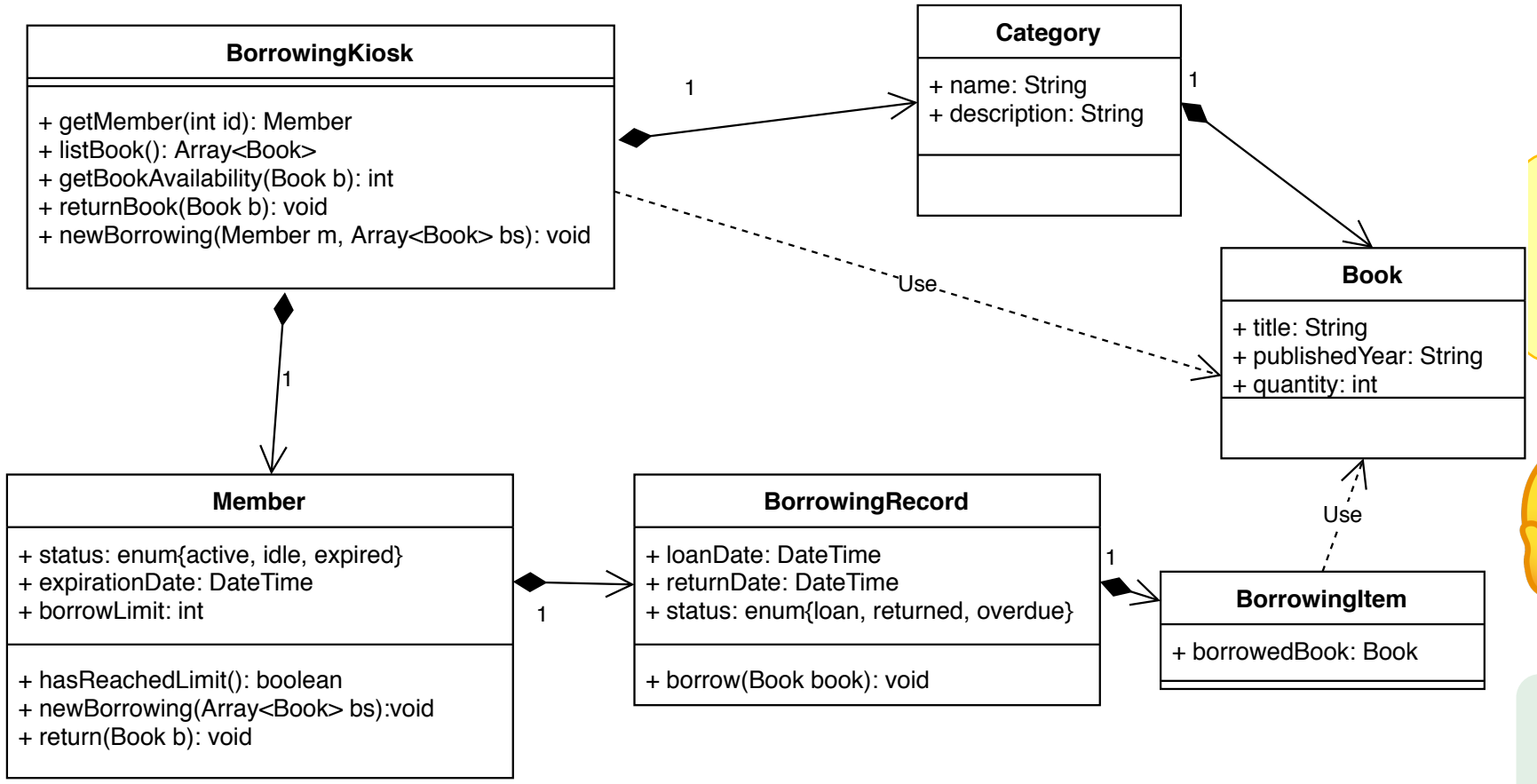
BorrowingKiosk is highly coupled with many classes. Also it has a low cohesion.

Exercise: Revision 1



To improve, we can lose coupling by assigning Member as an intermediate class between BorrowingKiosk and BorrowingRecord

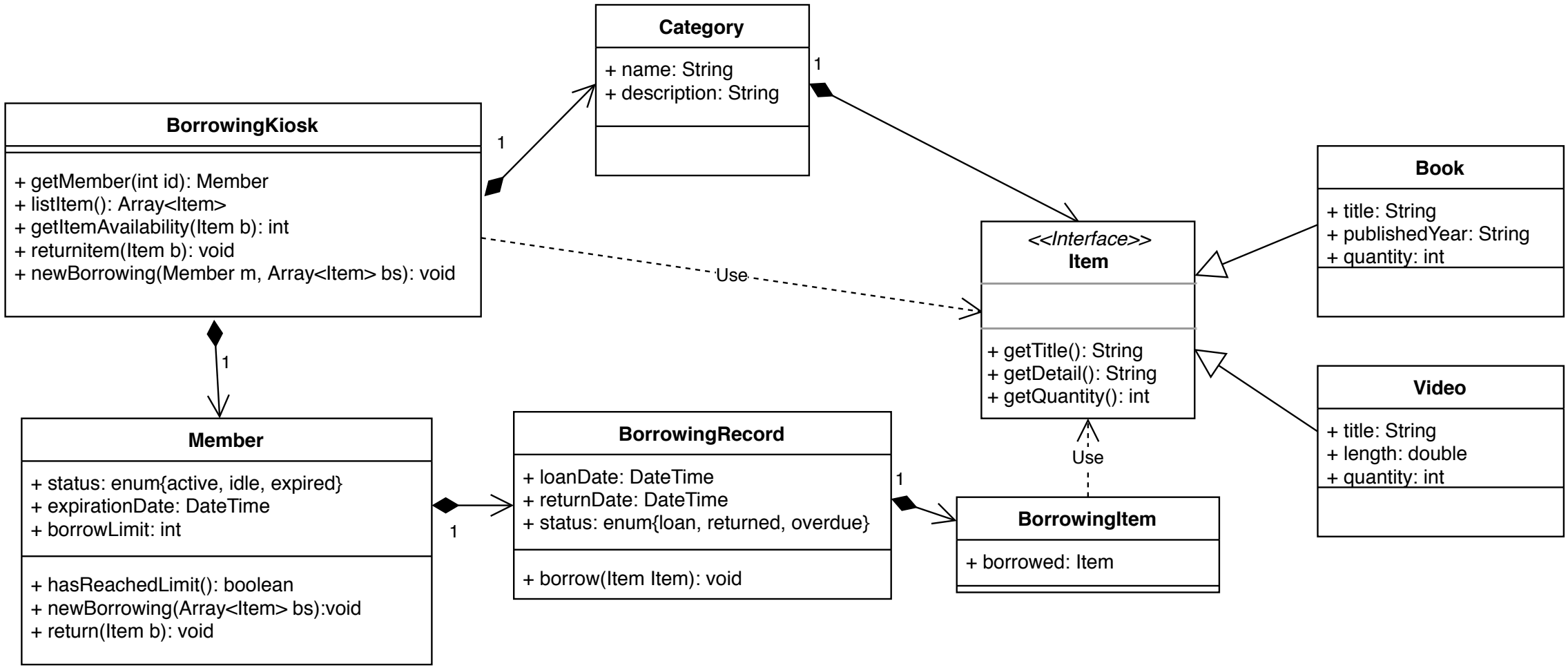
Exercise: Analyze the design model based on GRASP



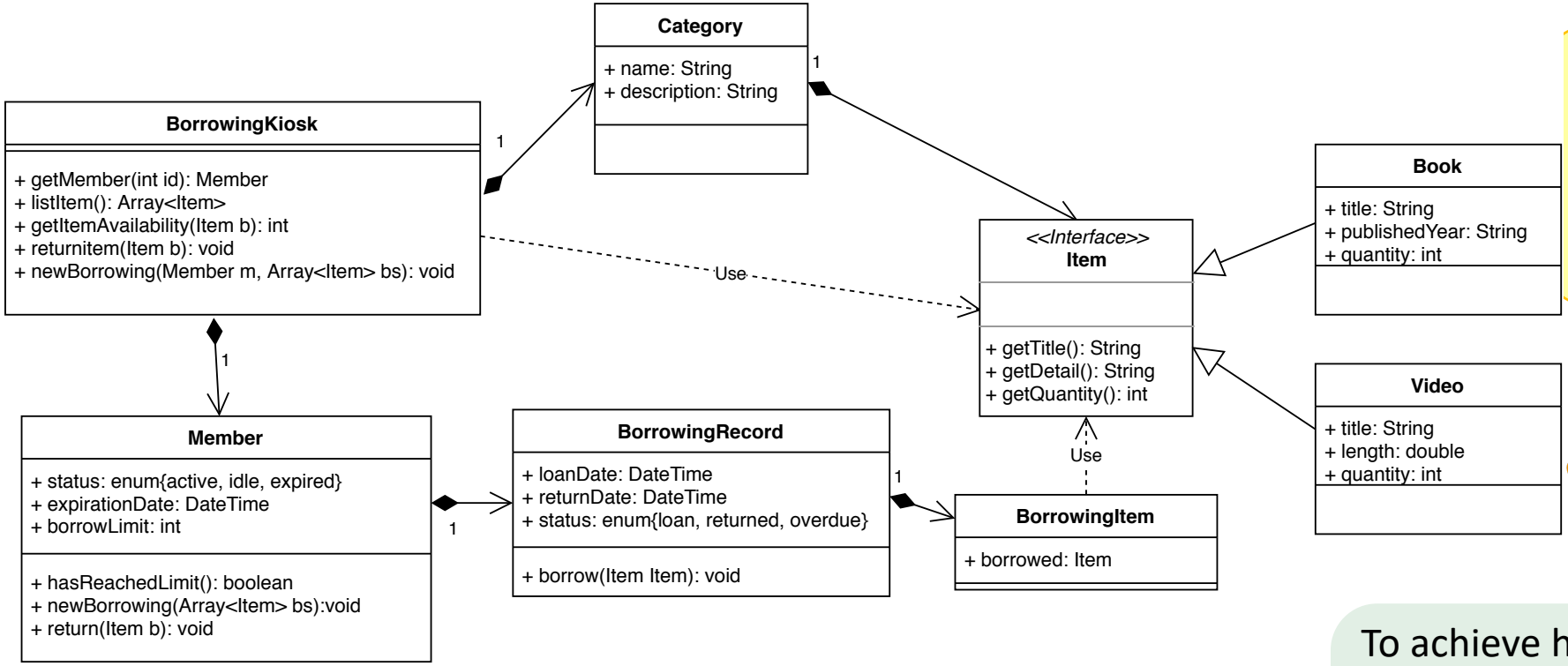
- What if in the future, a member can borrow a video via Kiosk as well. But a video has a duration attribute

Let's use polymorphism.

Exercise: Revision 2



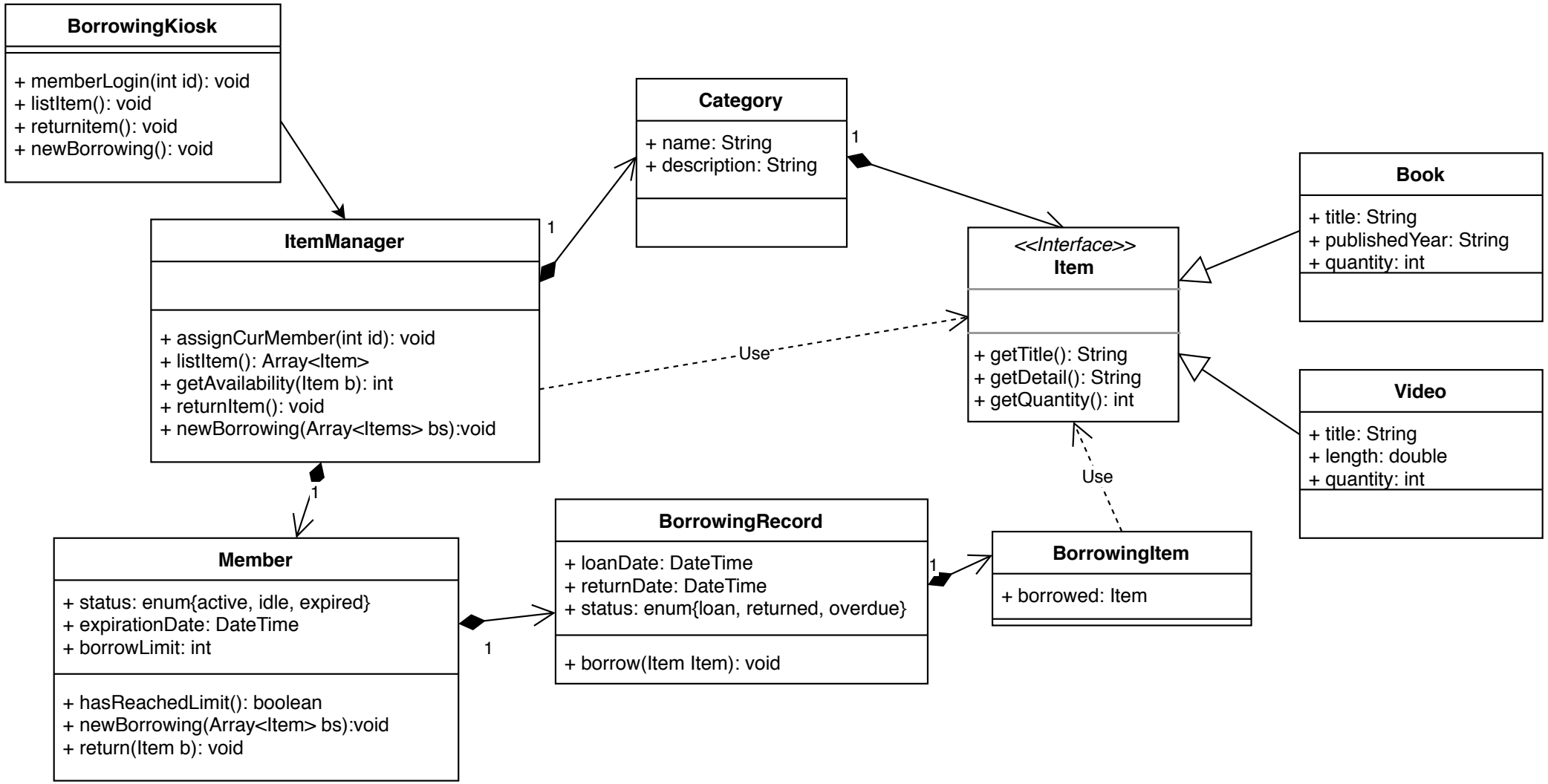
Exercise: Analyze the design model based on GRASP



- If we assign BorrowingKiosk be responsible for getting user inputs from UI (i.e., a controller), how can we improve the design?

To achieve high cohesion, i.e., let's BorrowingKiosk focuses on handling user inputs, we should create a new class to handle the items and borrowing (Let's called ItemManger).

Exercise: Revision 3



Lecture Identification

Lecturer: Patanamon Thongtanunam

Semester 2, 2020

These slides include materials from:

Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development, Third Edition, by Craig Larman, Pearson Education Inc., 2005.