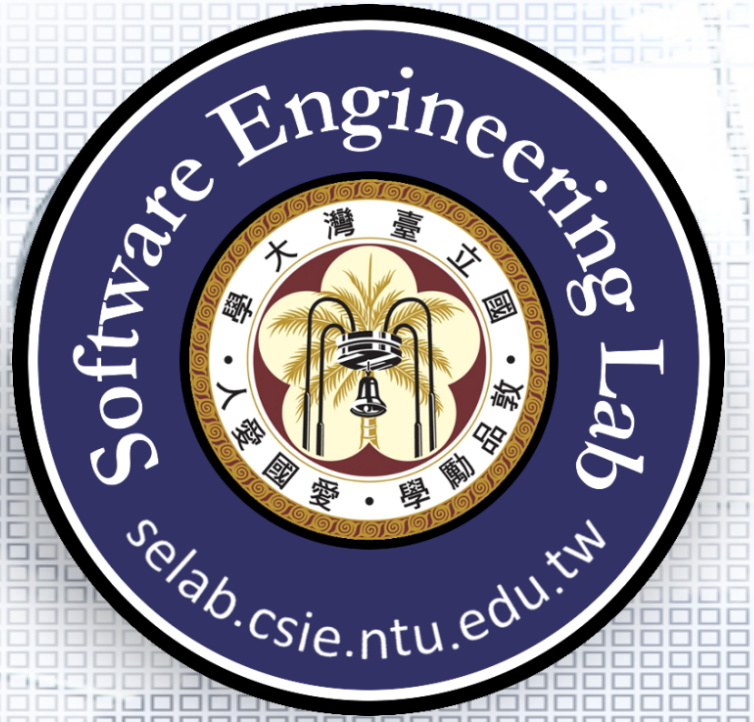


Design Pattern Concepts

Prof. Jonathan Lee (李允中)
CSIE Department
National Taiwan University

What's a Pattern?

- A pattern is a way to capture expertise.
- Refer to a **recurring best practice** documented as a solution to a problem in a given environment.
- A pattern language refers to patterns work together to solve problems in a particular area.
- Facilitate communication: when you know a pattern and use its name, a lot of information is communicated with just that single word or short phrase.

What is a Design Pattern?

- Design patterns capture solutions that have developed and evolved overtime.
- Software design patterns are derived from the concepts of patterns used in Architecture proposed by Christopher Alexander.
  - "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over."(AIS77)

Essential Elements of a Pattern

- Pattern name: describe a design problem, its solutions, and consequences in a word or two.
  - Strategy, Observer, Façade, and etc.
- Problem: describe when to apply the pattern.
- Solution: describe the elements that make up the design, their relationships, responsibilities, and collaborations.
- Consequence: describe the results and trade-offs of applying the pattern.
  - Include its impact on a system's flexibility, extensibility, or portability.

Design Patterns Category$_1$

- Design patterns are classified by two criteria:
  - Purpose: reflect what a pattern does, including creational, structural, and behavioral.
  - Scope: specifies whether the pattern applies primarily to classes or to objects, including class pattern, object pattern.
- Scope
  - Class patterns deal with relationships between classes and their subclasses, which are established through inheritance, that is, static and fixed at compile-time. Examples: Factory Method, Interpreter, Template Method, and Adapter (class).
  - Object patterns deal with object relationships, which can be changed at run-time. Adapter (object)

Design Patterns Category$_2$

- Creational: Involve object creation.
  - Factory Method, Abstract Factory, Builder, Prototype, and **Singleton**.
- Structural: compose classes or objects into larger structures.
  - Adapter, Bridge, Composite, Decorator, Façade, **Flyweight**, and Proxy.
- Behavioral: Concern with how classes and objects interact and distribute responsibility.
  - Interpreter, Template, Chain of Responsibility, Command, Iterator, Mediator, **Memento**, Observer, State, Strategy, and Visitor.

Design Patterns Category$_3$

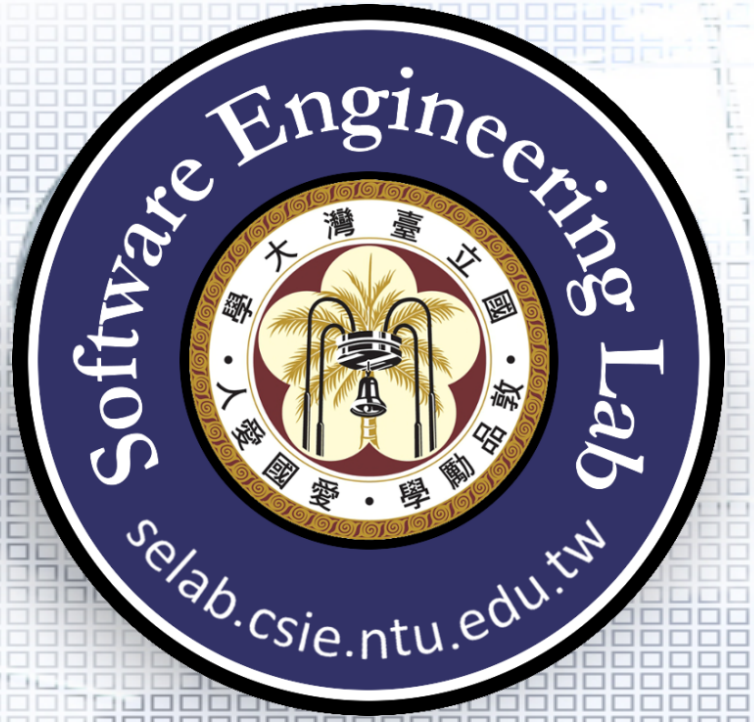
- Composite is often used with Iterator or Visitor.
  - Composite: compose objects into a **tree structure** to represent a part-whole hierarchy.
  - Visitor: Collect **states** from a Composite.
- Prototype is often an alternative to Abstract Factory.
  - Prototype: make new instances by copying existing ones.
- Composite and Decorator have a same iterative structure.
  - Have a same structure, but different meanings (termination).

Software Design

- Starting with problem statements (requirements)
- Modeling with class diagrams (concrete classes)
  - Noun: Class; Attribute
  - Verb: Operation; Relation
  - Option: Attribute with boolean type; Operation with multiple operations
- Refactoring with a design process involving the use of:
  - object-oriented concepts,
  - design principles, and
  - design patterns.

Object-Oriented Concepts

- Inheritance (Generalization, Classification)
- Polymorphism
- Abstraction
- Encapsulation
- Delegation (Dependency)
- Association
- Composition (Aggregation)



Object-Oriented Design Principles

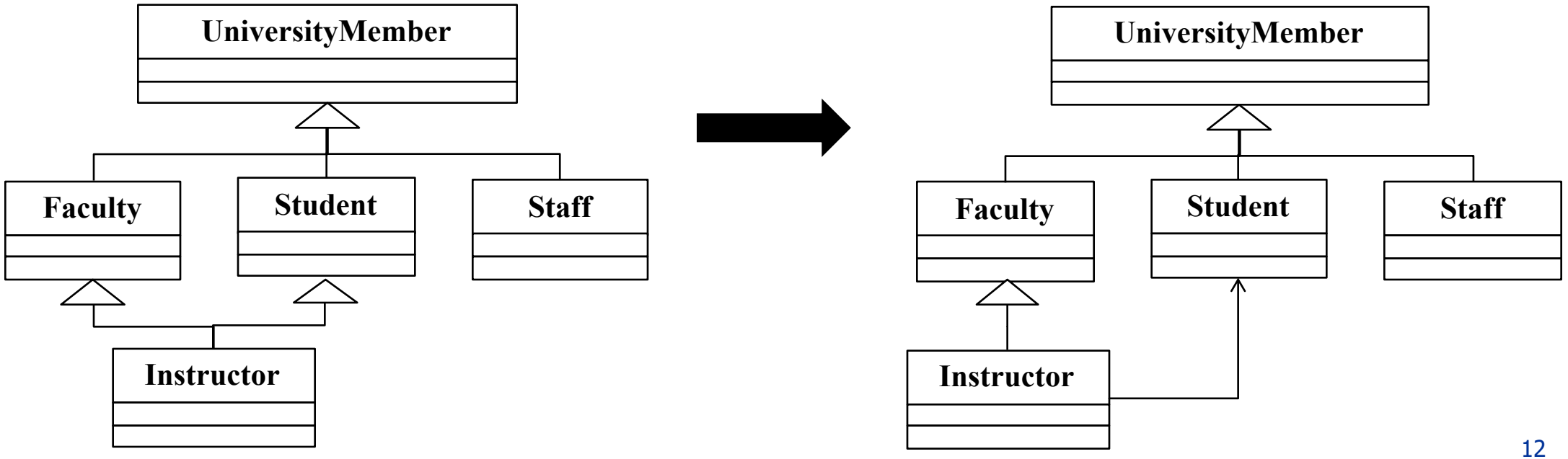
Prof. Jonathan Lee (李允中)
CSIE Department
National Taiwan University

Fundamental Design Principle

- Fundamental design principle: High cohesion, low coupling
- All the object-oriented design principles strive to address the above fundamental design principle.
- Each object-oriented design principle is limited to its own applicable cases, not an all-purpose-remedy.

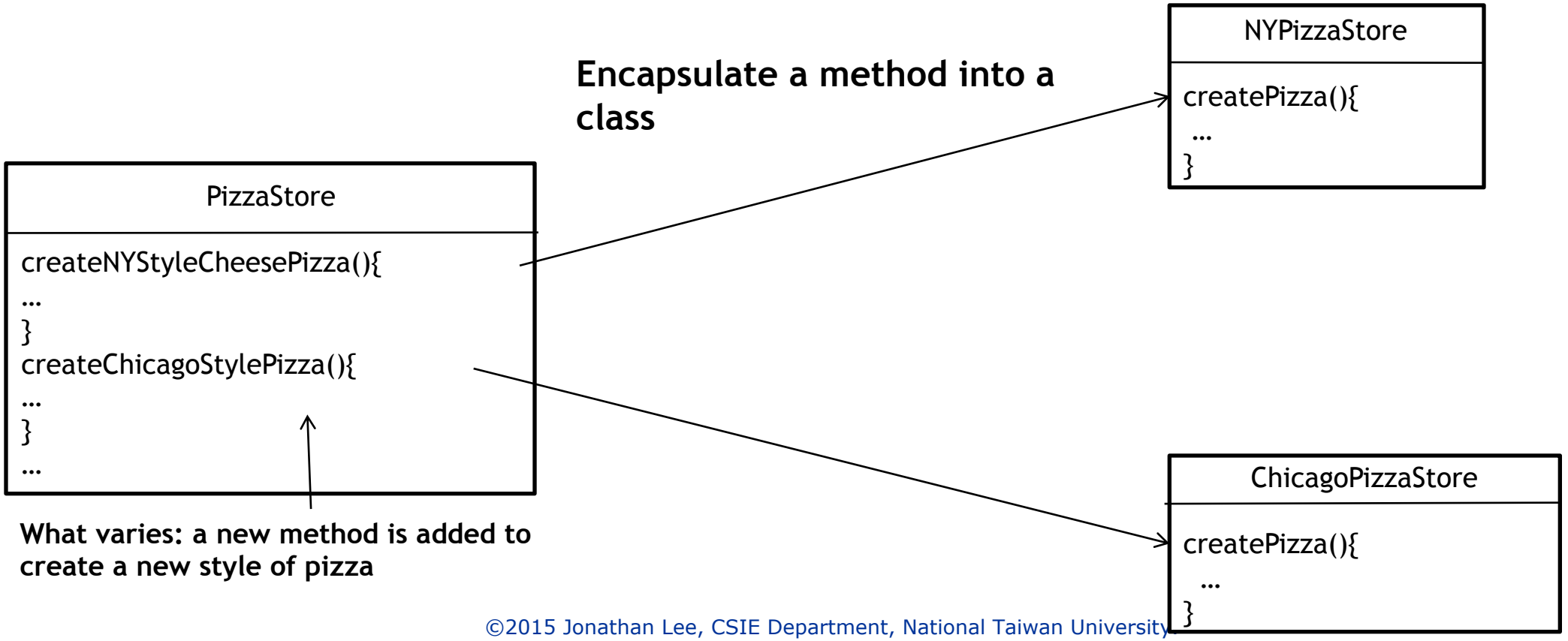
Inherit the most important features and delegate the rest

- As a software designer, you determine which features are more important than the others.
- Extend functionality by sub-classing and composing its instances with existing ones.



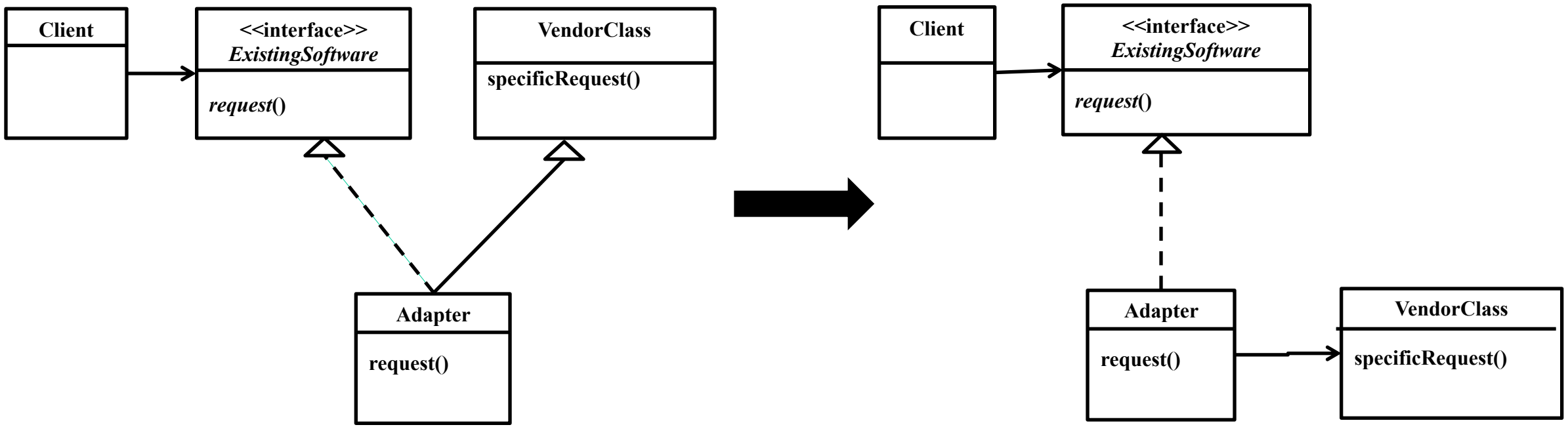
Encapsulate what varies

- Including method, attribute, part of method body, request (method invocation), iteration and etc.



Favor composition over inheritance

- Provide new functionality with more flexibility at run-time.



Composition vs Inheritance$_1$

- The two most common techniques for reusing functionality in object-oriented systems are class inheritance and object composition.
- Reuse by subclassing is often referred to as white-box reuse, that is, the internals of parent classes are often visible.
- Reuse by object composition is done by obtaining new functionality via composing objects to get more complex functionality, and is called black-box reuse, that is, no internal details of objects are visible.

Composition vs Inheritance$_2$

- Inheritance:
  - Advantage: Straightforward to use; easy to modify the implementation being reused through override.
  - Disadvantage: cannot change the implementation at run-time; any change in the parent's implementation will force the subclass to change.
- This implementation dependency limits flexibility, and ultimately reusability.
- A remedy to this is to inherit from abstract classes (only partial implementation).

Composition vs Inheritance$_3$

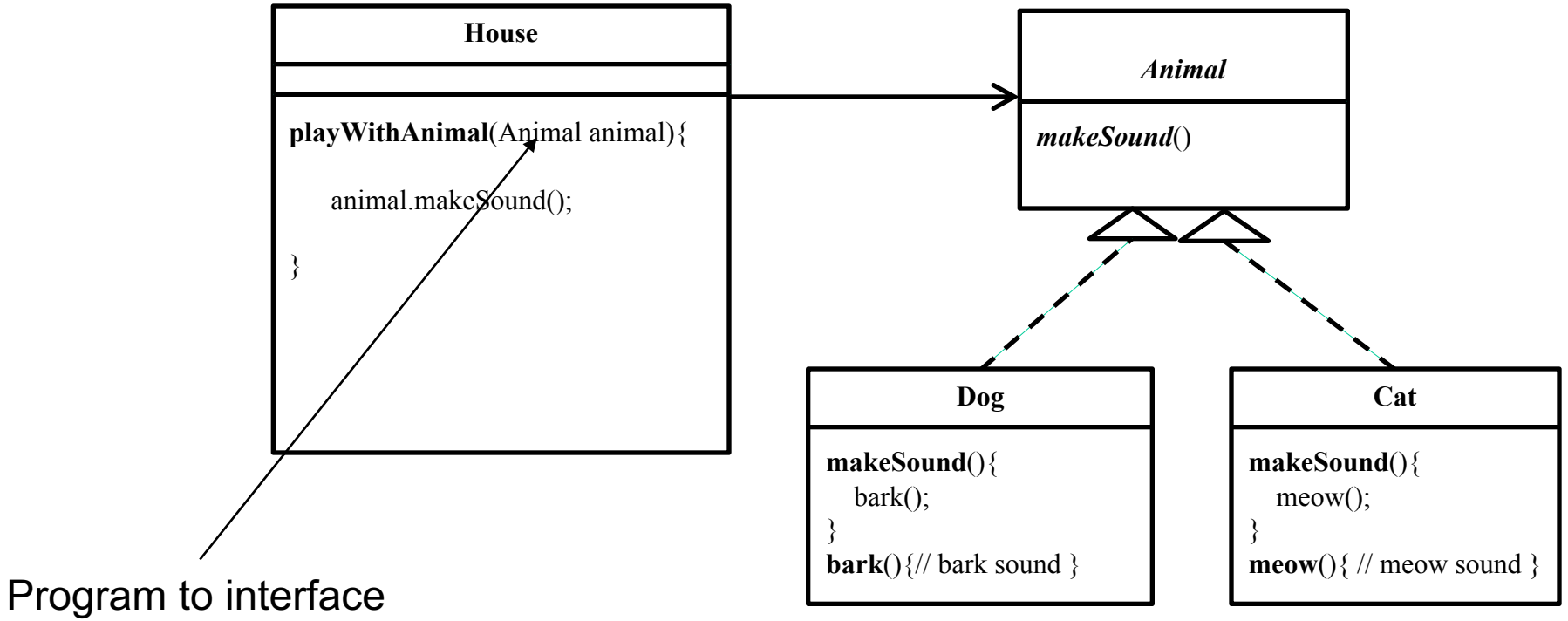
- Composition (Advantage)
  - Composition requires objects to respect each other's interfaces, and therefore, we don't break encapsulation.
  - Fewer implementation dependencies.
  - Classes and class hierarchies will remain small and will be less likely to grow into unmanageable monsters.
- Disadvantage: Dynamic, highly parameterized software is harder to understand than more static software.

Composition + Inheritance

- Ideally, you shouldn't have to create new components to achieve reuse. You should be able to get all the functionalities you need just by assembling existing components through object composition.
  - But, it is rarely the case, because the set of available components is never quite rich enough in practice.
  - Reuse by inheritance makes it easier to make new components that can be composed with old ones.
- **Extend functionality by sub-classing and composing its instances with existing ones.**

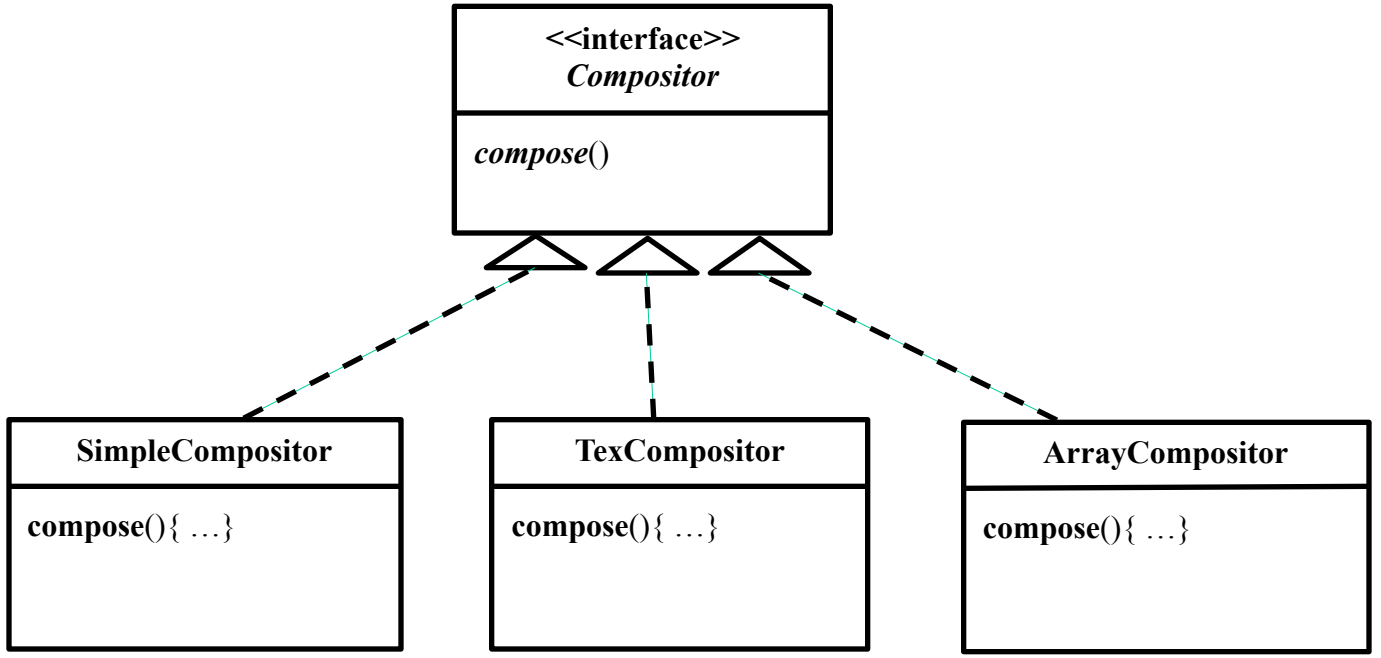
Program to interface, not implementation

- Exploit polymorphism/supertype to assign the concrete implementation object at run time.



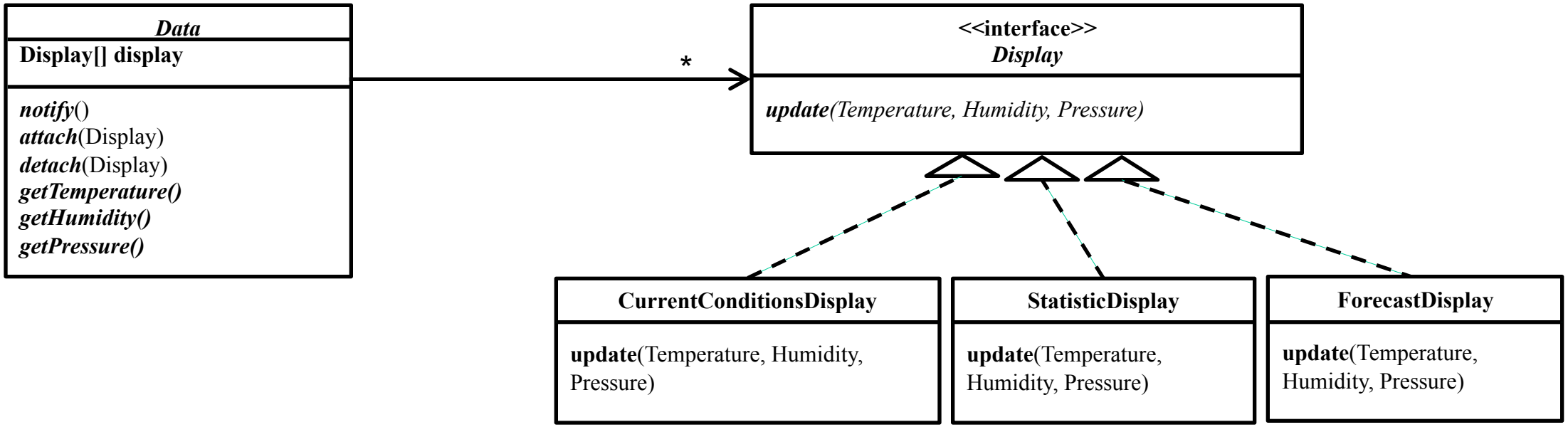
Classes should be open for extension but closed for modification

- Open-Close Principle: Allow classes to be easily extended to incorporate new behavior without modifying existing codes.



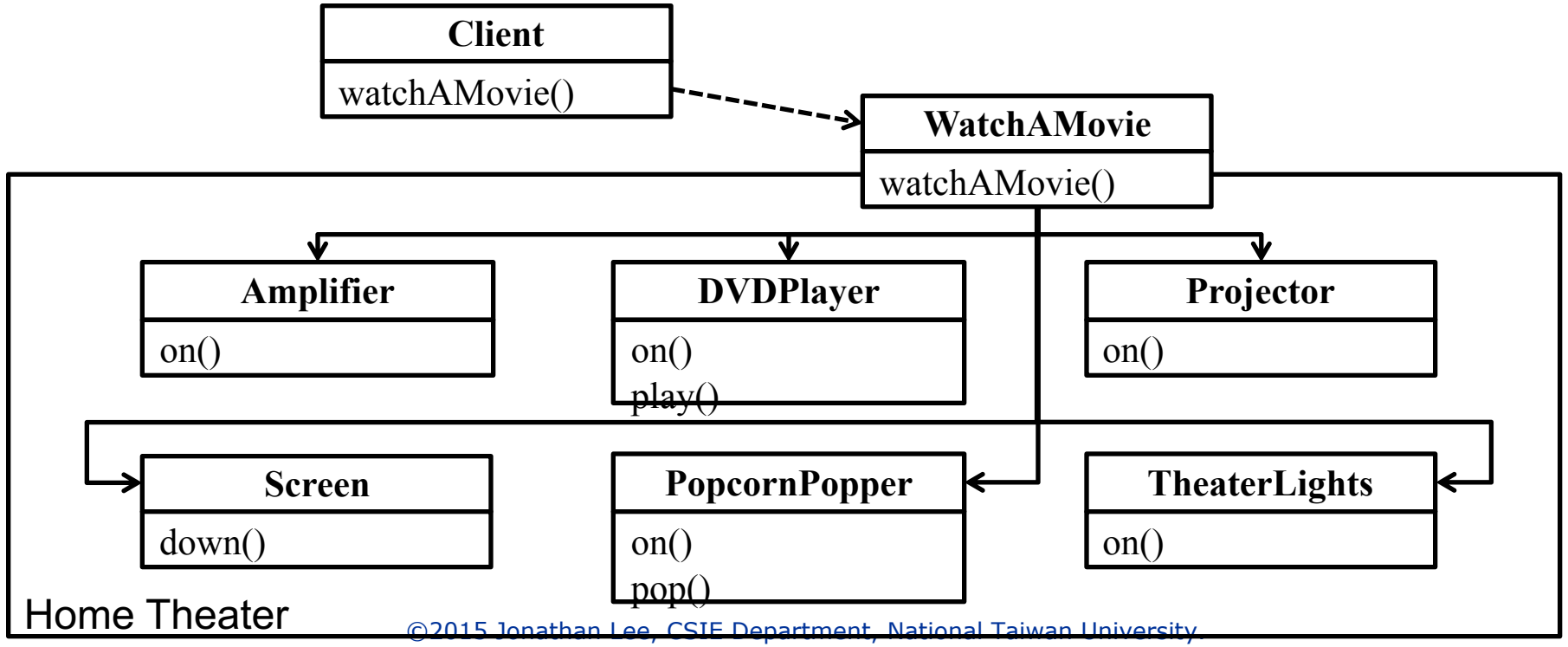
Strive for loosely coupled designs between objects that interact

- Loosely Coupled Principle: Objects can interact but have very little knowledge of each other to minimize the interdependency between objects.



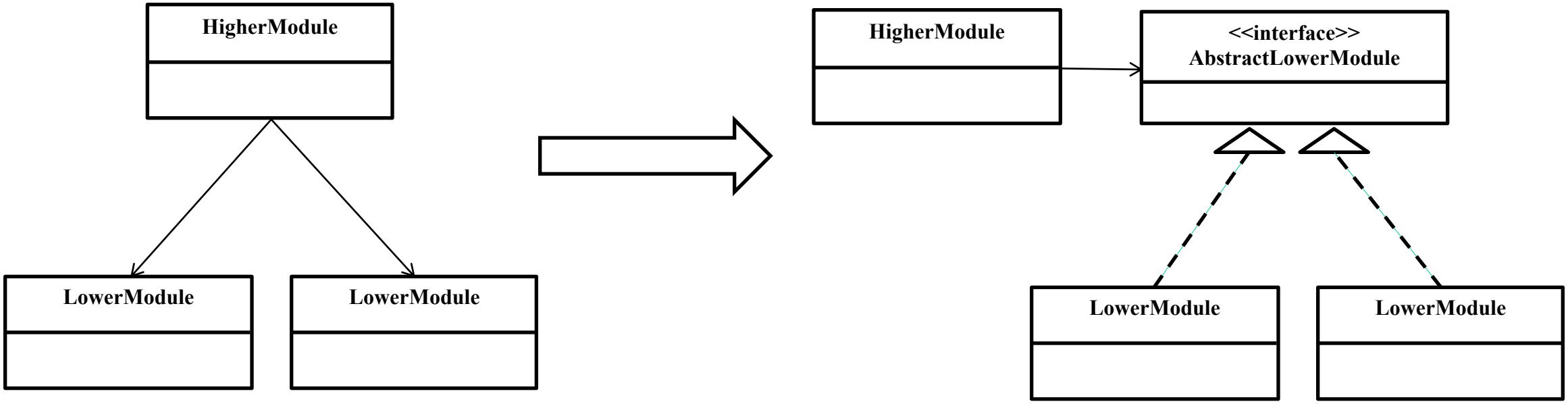
Only talk to your friends

- Least Knowledge Principle: The least we know about the subsystem components, the better by providing a unified interface to a set of interfaces in a subsystem.



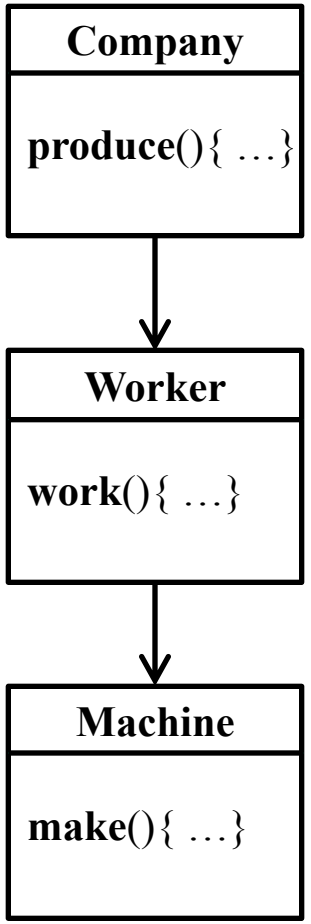
Depend on abstractions. Do not depend on concrete classes

- Dependency Inversion Principle: Enforce both the high level components and low level components to depend on abstractions.
  - A high level component is a class with behavior defined in terms of other low level components.



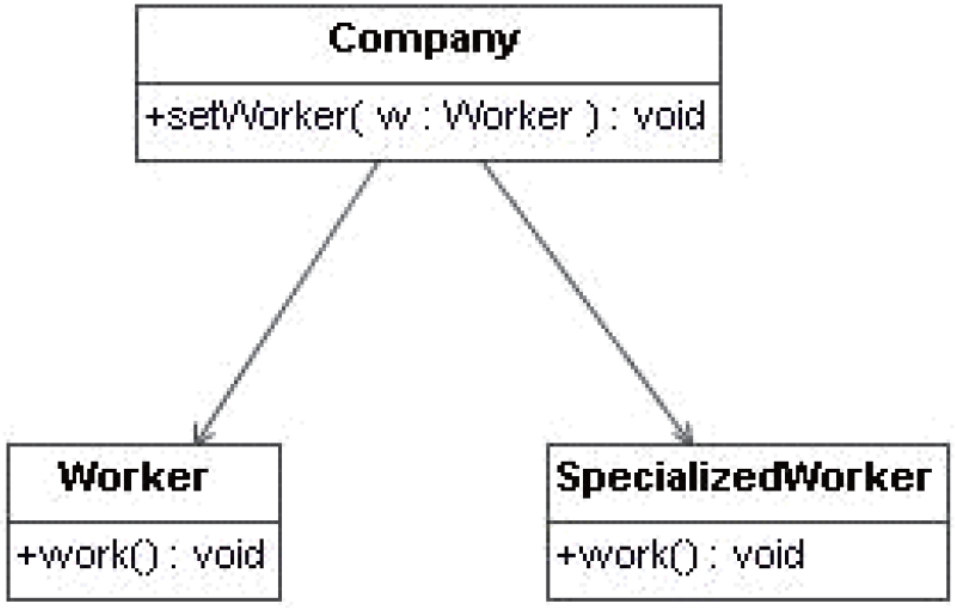
Dependency Inversion Principle (DIP)

- Start with high level abstractions and elaborate with details in a stepwise manner to make the design align with the real world scenarios.
- Dependency Inversion Principle: Enforce both the high level components and low level components to depend on abstractions.



Violate DIP: Example Company

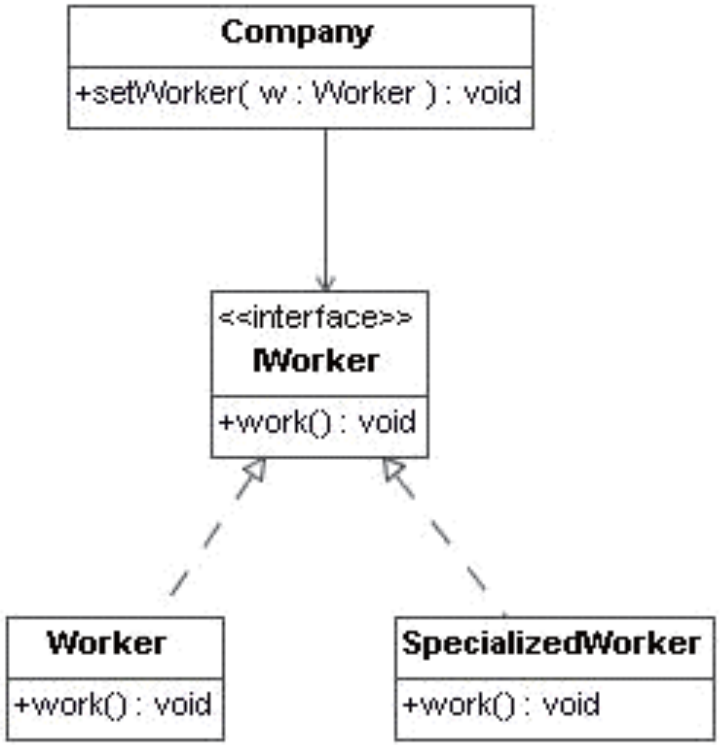
- Company NTU has established a new business which requires a new kind of workers, called SpecializedWorker. In order to reflect this change, the Company Class (high level component) has been revised to include SpecializedWorker class. This is a clear violation of Dependency Inversion Principle.




```
class Worker {
        public void work () {…..}
}
class SpecializedWorker {
        public void work () { …..}
}
class Company {
        Worker worker;
       public void setWorker (Worker w) {worker = w;}
        public void produce() {worker.work ();}
}
```

Enforce DIP: Insert Isolation Layer

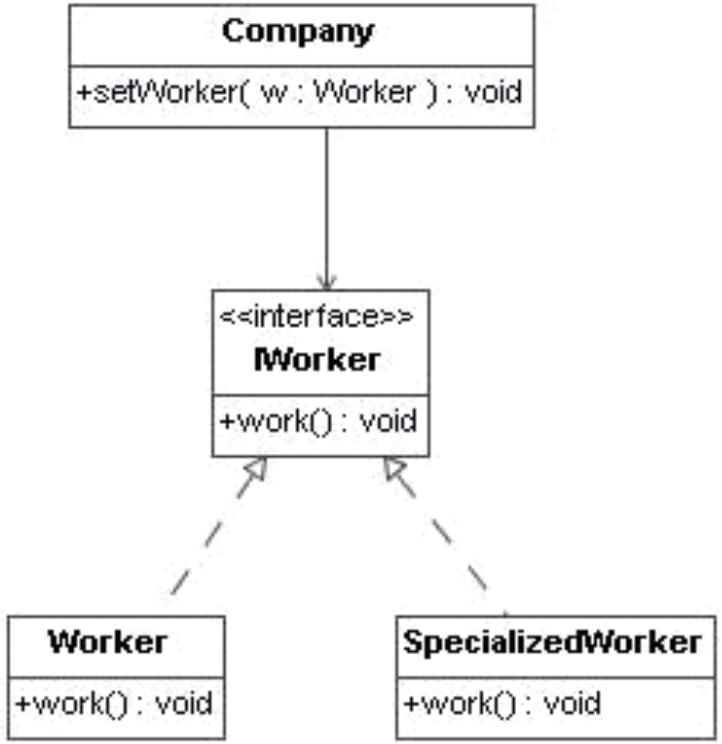
- Insert an isolation layer (Abstract layer) between the high level component and low level components to make both types of components dependent on this layer.



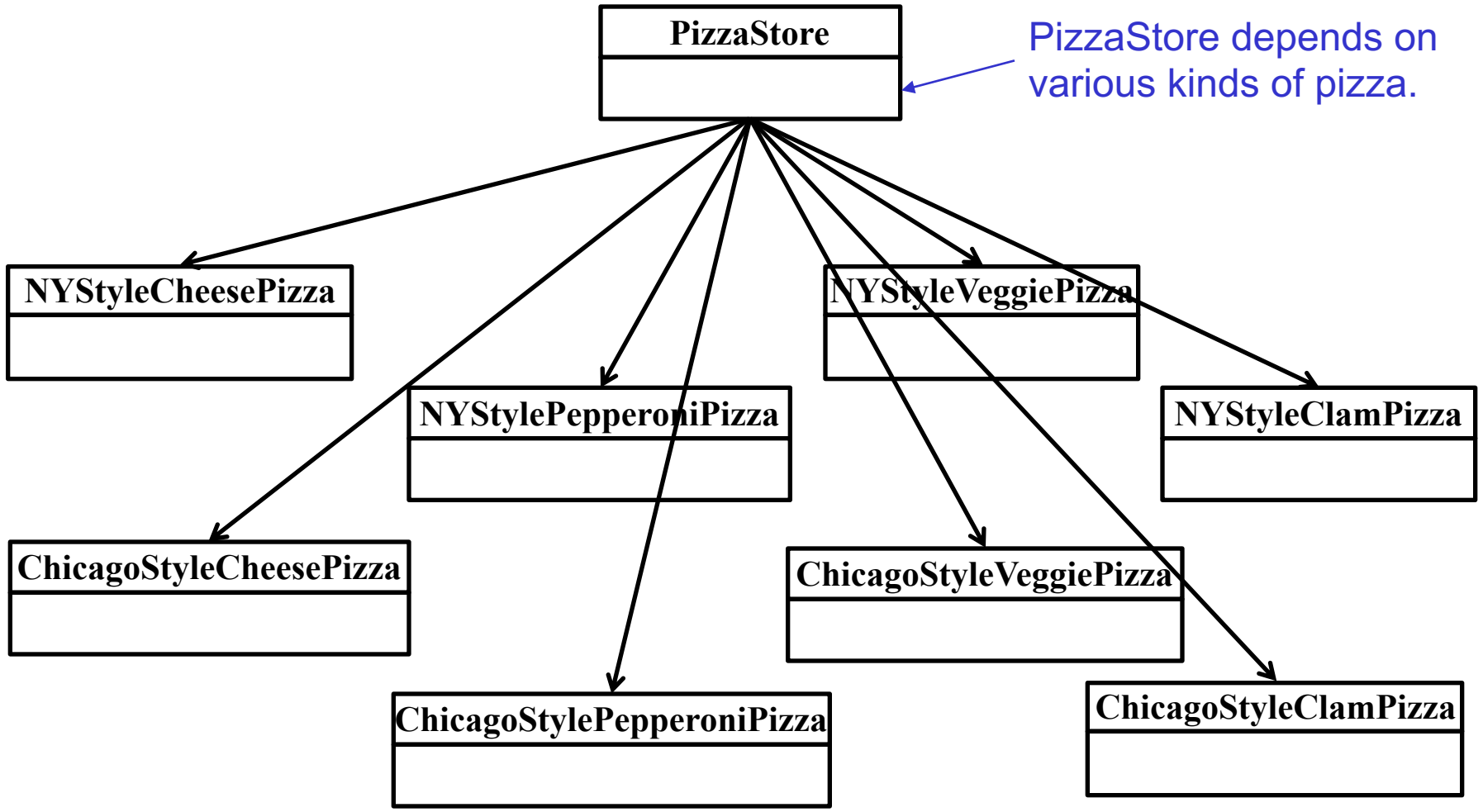

```
class Company {
        IWorker worker;
publich void setWorker (IWorker w) {
        worker = w;
}
public void produce () {
        worker.work (); }
}
```

Benefit of DIP

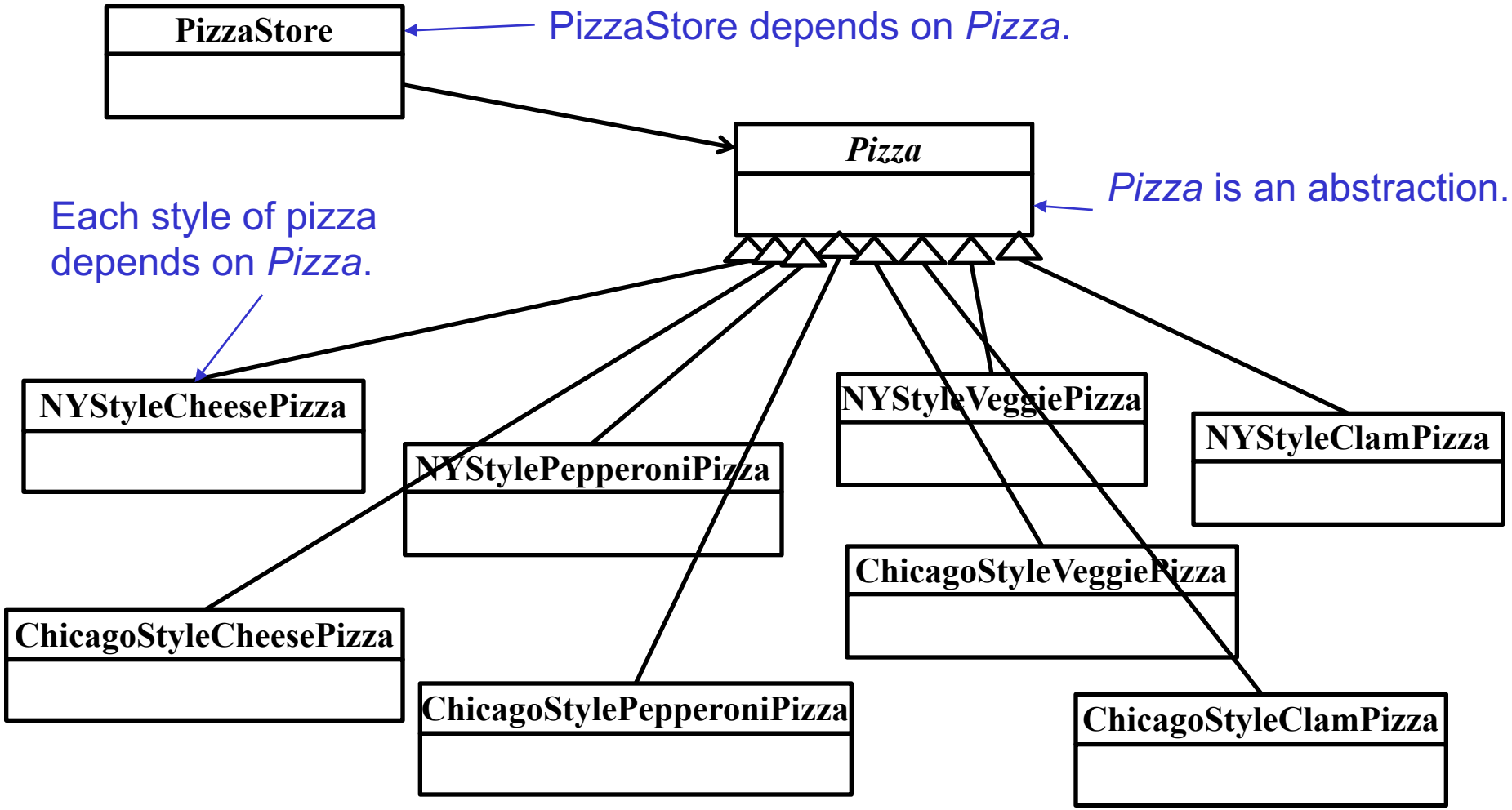
- High level components carry important business logic, while low level components carry algorithms and less important business logic. Dependency Inversion Principle provides an isolation layer to protect high level components from been changed when the low level components are changed.



Violate DIP: Example PizzaStore

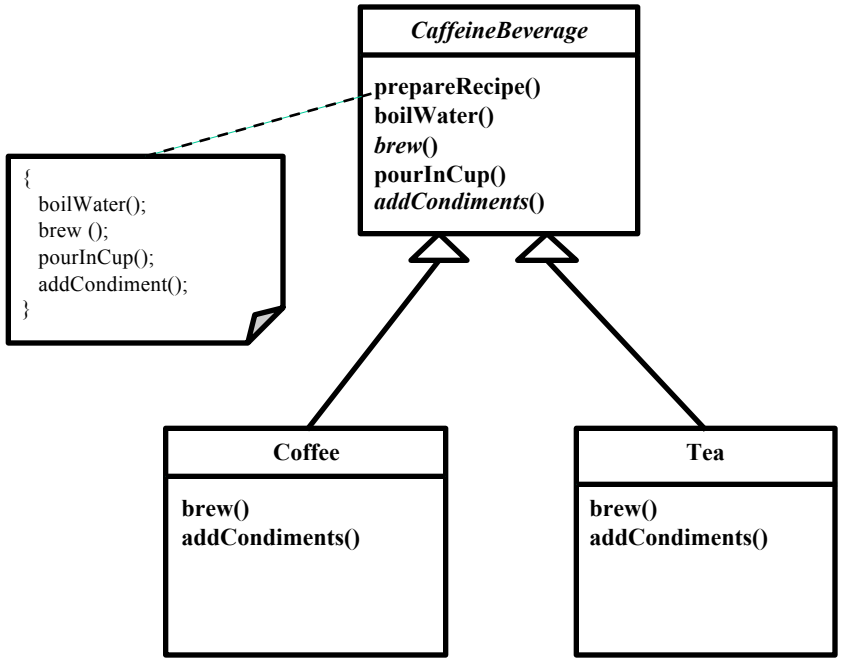


Enforce DIP: Insert Isolation Layer



Don't call us, we'll call you. (The Hollywood Principle)

- High level components call low level components, but not vice verse.



The high-level components control when and how

A low-level component never calls a high-level component directly

OO Design Principles (I)

- Inherit the most important features and delegate the rest.
  - As a software designer, you determine which features are more important than the others.
  - Extend functionality by sub-classing and composing its instances with existing ones.
- Encapsulate what varies.
  - Including method, attribute, part of method body, request (method invocation), iteration, steps in an algorithm, and etc.
- Favor composition over inheritance.
  - Provide new functionality with more flexibility at run-time.

OO Design Principles (II)

- Program to interface, not implementation.
  - Exploit polymorphism to assign the concrete implementation object at run time.
- Depend on abstractions. Do not depend on concrete classes. (Dependency Inversion Principle)
  - Enforce both the high level components and low level components to depend on abstractions.
  - A high level component is a class with behavior defined in terms of other low level components.

OO Design Principles (III)

- Classes should be open for extension but closed for modification. (Open-Close Principle)
  - Allow classes to be easily extended to incorporate new behavior without modifying existing codes.
- Strive for loosely coupled designs between objects that interact. (Loosely Coupled Principle)
  - Objects can interact but have very little knowledge of each other.
  - Minimize the interdependency between objects.

OO Design Principles (IV)

- Only talk to your friends. (Least Knowledge Principle)
  - The least we know about the subsystem components, the better.
  - Provide a unified interface to a set of interfaces in a subsystem.
- Don't call us, we'll call you. (The Hollywood Principle)
  - High level components call low level components, but not vice verse.
  - Abstract **common behaviors** into an abstract class with respect to steps of an algorithm.

OO Design Principles (V)

- A class should have only one reason to change. (Single Responsibility Principle)
  - Avoid the probability of too many changes in a class to run the risk of changing the class in the future.
  - Assign each responsibility to one class, and just one class.
  - This is probably the hardest principle to abide by.

- …..

Designing for Change: Guidelines$_1$

- The key to maximize reuse lies in **anticipating new requirements** and **changes to existing requirements**, and in designing your systems so that they can **evolve** accordingly.
- A design that doesn't **take change into account** risks major redesign in the future.
- Design patterns help avoid redesign by ensuring that **a system can change in specific ways.**

Designing for Change: Guidelines$_2$

- Create object indirectly.
- Avoid hard-coded requests.
- Limit platform dependencies.
- Hide object representation from clients.
- Algorithms likely to change should be isolated.
- Avoid tight coupling.
- **Extend functionality by subclassing and composing its instances with existing ones.**

Design Pattern vs Framework

- A typical framework contains several design patterns, but the reverse is never true.
- Framework always have a particular application domain. Whilst design patterns can be used in nearly any kind of application.
- Compound pattern: combine two or more patterns into a solution that solves a recurring or general problem.
  - For example: Model-View-Control
  - Model: Observer, View: Composite, Control: Strategy.

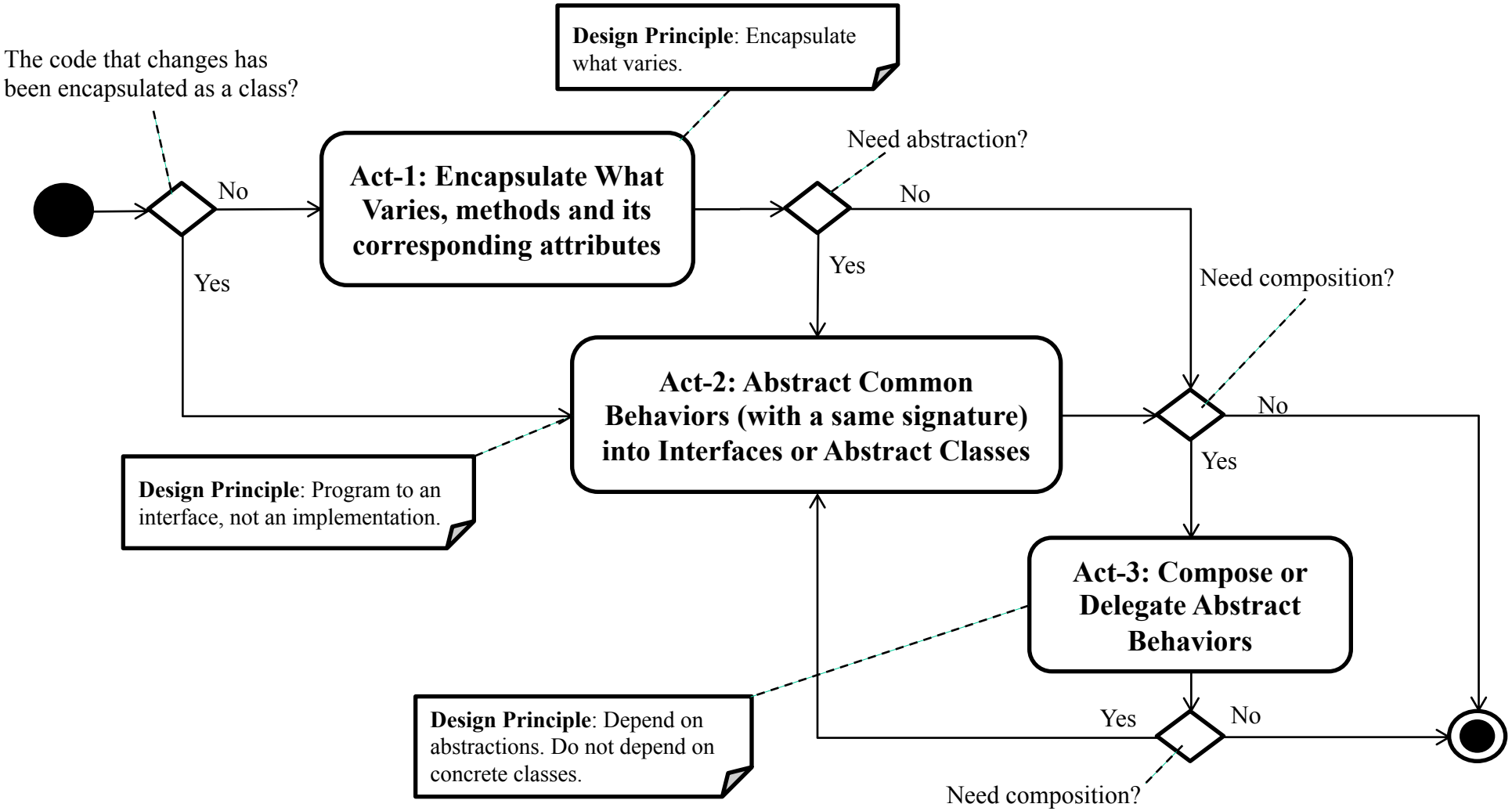
Various Kinds of Patterns

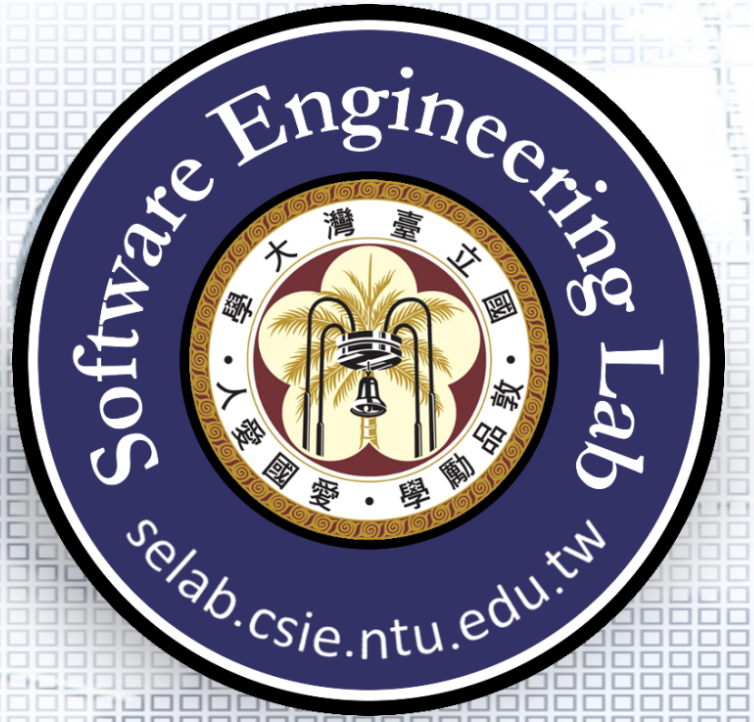
- Application pattern: patterns for creating system level architecture.
  - Client server architecture, 3-tier architecture, Model-View-Control, and etc.
- Domain-specific pattern: concern problems in specific domains.
  - J2EE
- User interface design pattern
  - Mobile User Interface patterns

Software Design with Design Patterns?

- Focus on modeling first from the requirements statement
- Follow object-oriented concepts where they are applicable such as encapsulation, abstraction, composition, polymorphism and etc.
- Identify design aspects for various kinds of design patterns.

Design Process for Changes





An Order Handling System

Prof. Jonathan Lee (李允中)

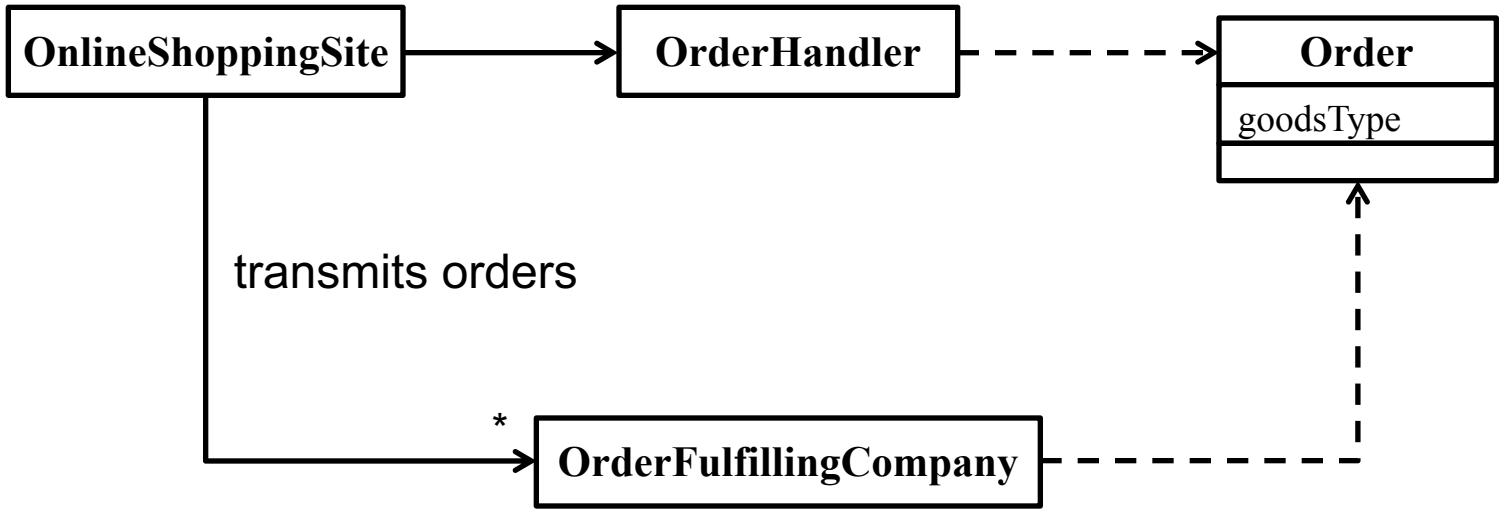
Department of Computer Science and Information Engineering
National Taiwan University

Requirements Statement

- Design the order handling functionality for a different type of an online shopping site that transmits orders to different order fulfilling companies based on the type of the goods ordered.
- Suppose that the group of order processing companies can be classified into three categories based on the format of the order information they expect to receive.
- These formats include comma-separated value (CSV), XML and a custom object. When the order information is transformed into one of these formats, appropriate header and footer information that is specific to a format needs to be added to the order data. The series of steps required for the creation of an Order object can be summarized as follows:
  - Create the header specific to the format
  - Add the order data
  - Create the footer specific to the format

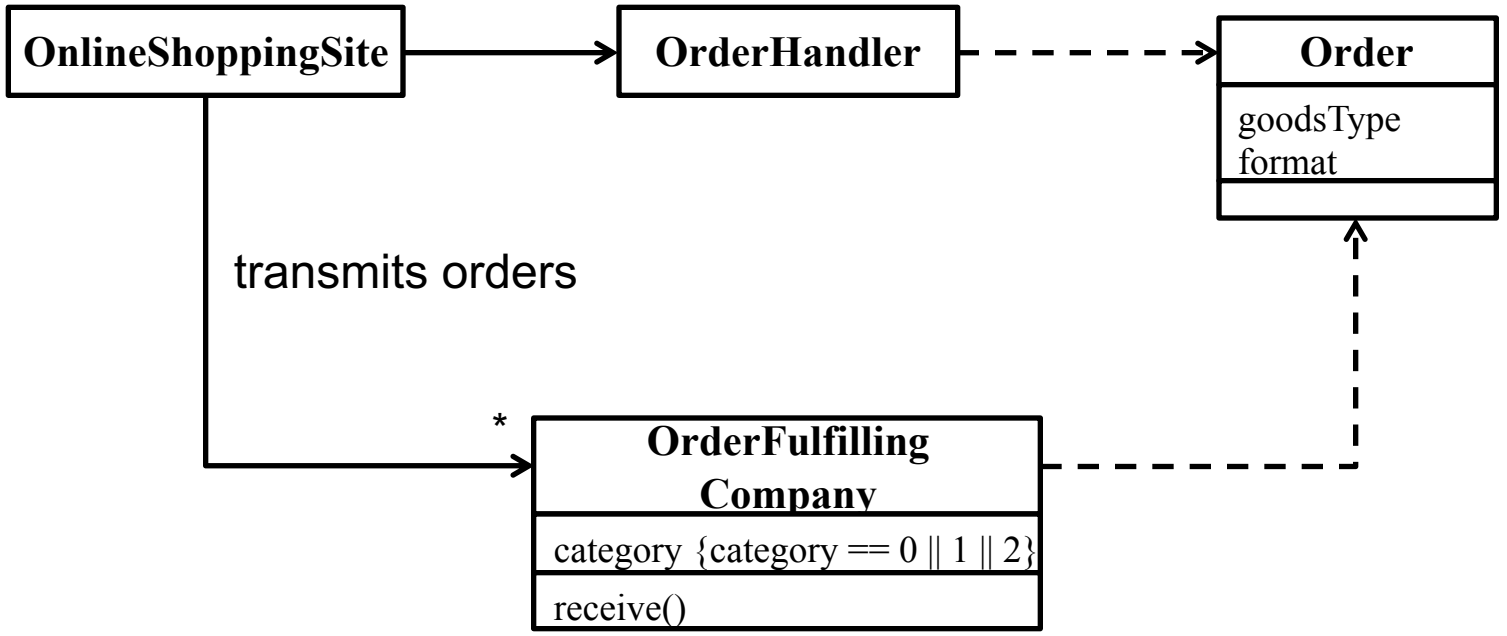
Requirements Statement$_1$

- Design the order handling functionality for a different type of an online shopping site that transmits orders to different order fulfilling companies based on the type of the goods ordered.



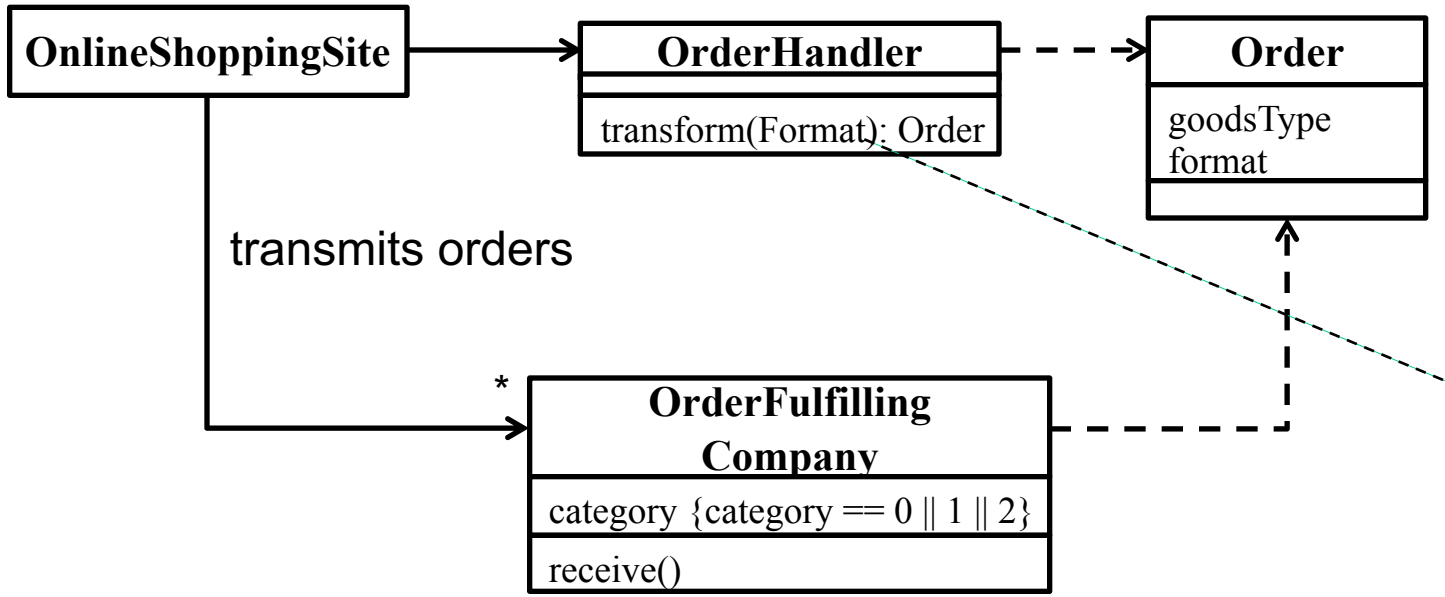
Requirements Statement$_2$

- Suppose that the group of order processing companies can be classified into three categories based on the format of the order information they expect to receive.



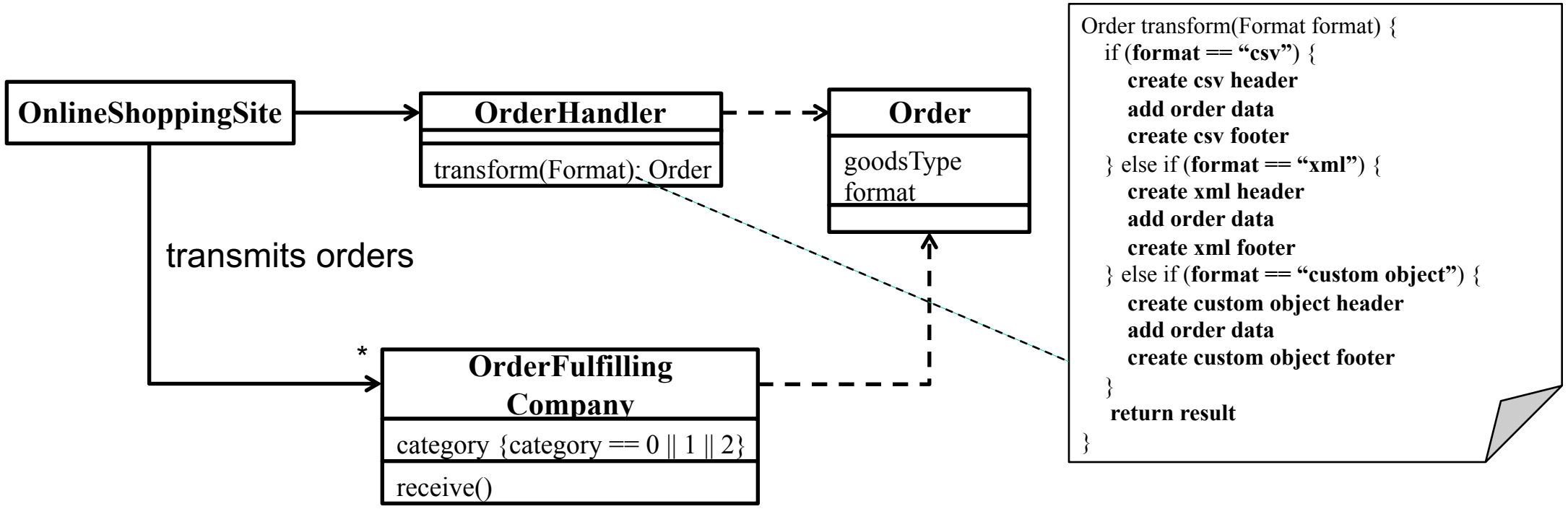
Requirements Statement$_3$

- These formats include comma-separated value (CSV), XML and a custom object. When the order information is transformed into one of these formats, appropriate header and footer information that is specific to a format needs to be added to the order data. The series of steps required for the creation of an Order object can be summarized as follows:
  - Create the header specific to the format
  - Add the order data
  - Create the footer specific to the format



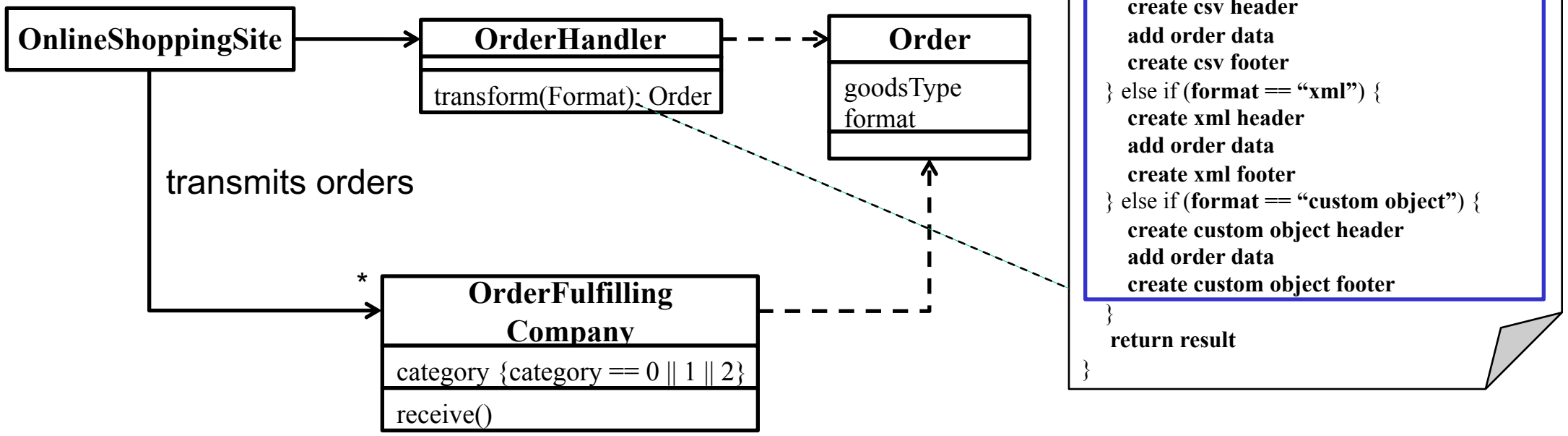

```
Order transform(Format format) {
   if (format == "csv") {
      create csv header
      add order data
      create csv footer
   } else if (format == "xml") {
      create xml header
      add order data
      create xml footer
   } else if (format == "custom object") {
      create custom object header
      add order data
      create custom object footer
   }
    return result
}
```

Initial Design

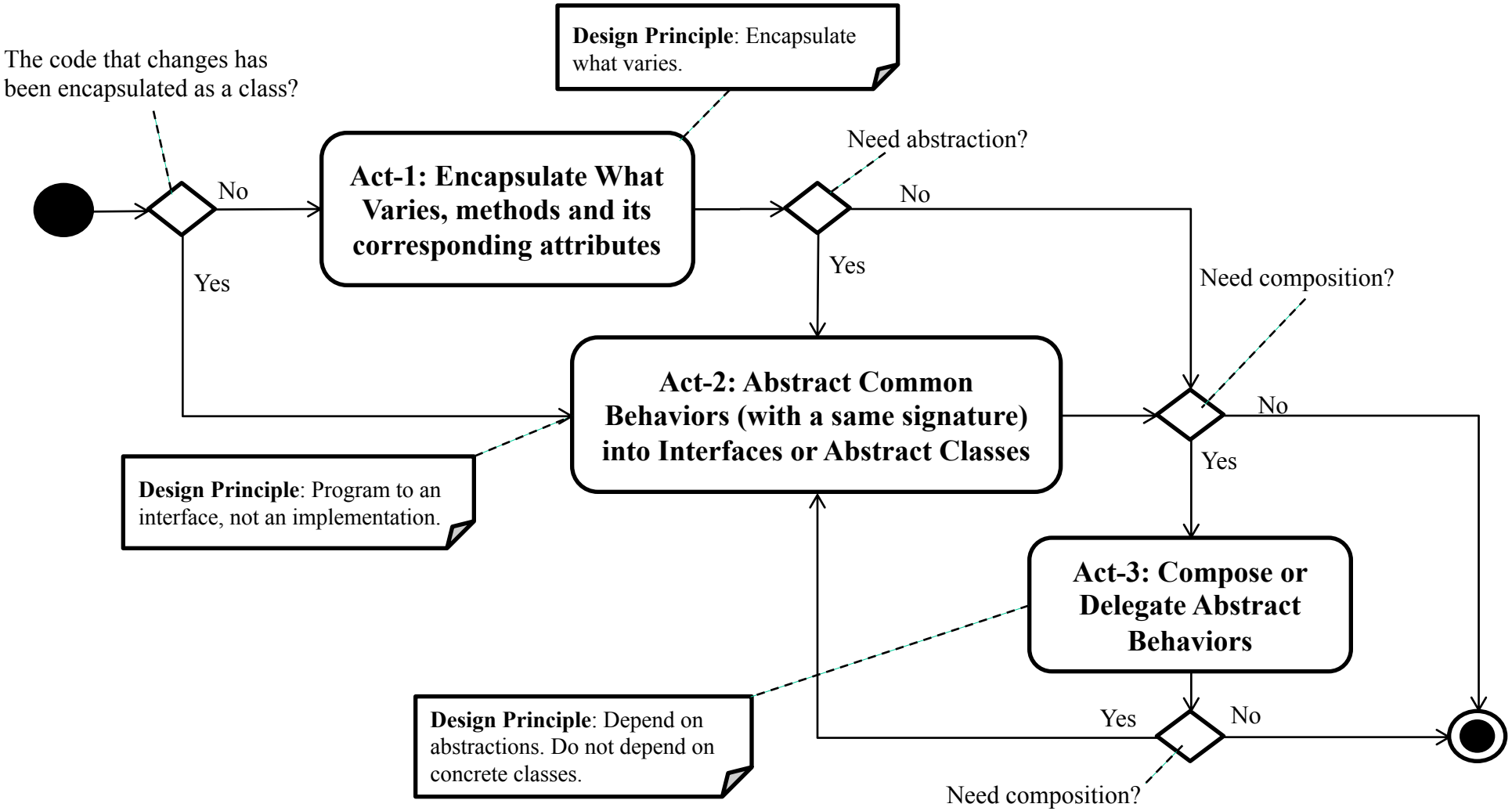


Problem with Initial Design

Problem: The more formats OrderHandler support, the more changes it would result in.

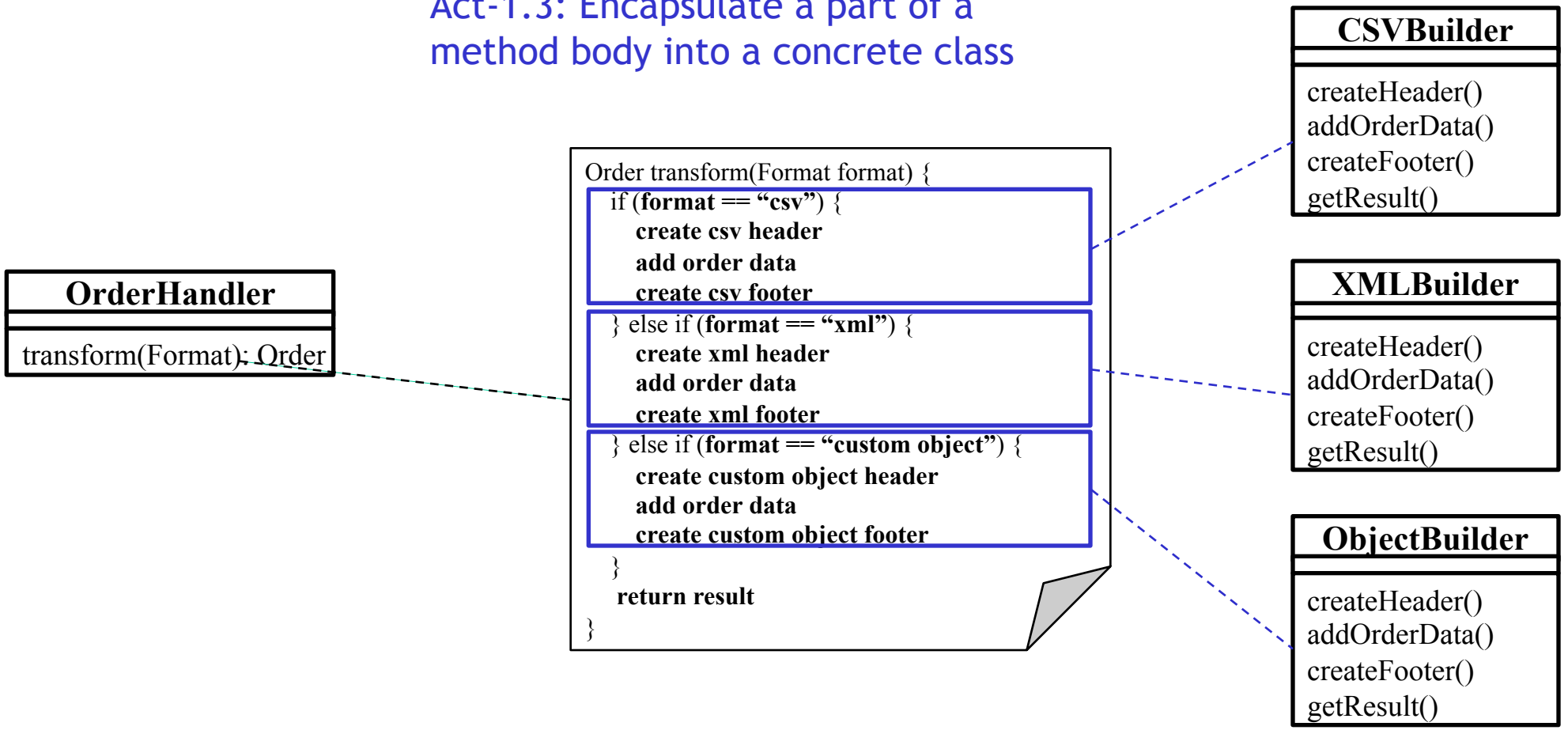


Software Design Process

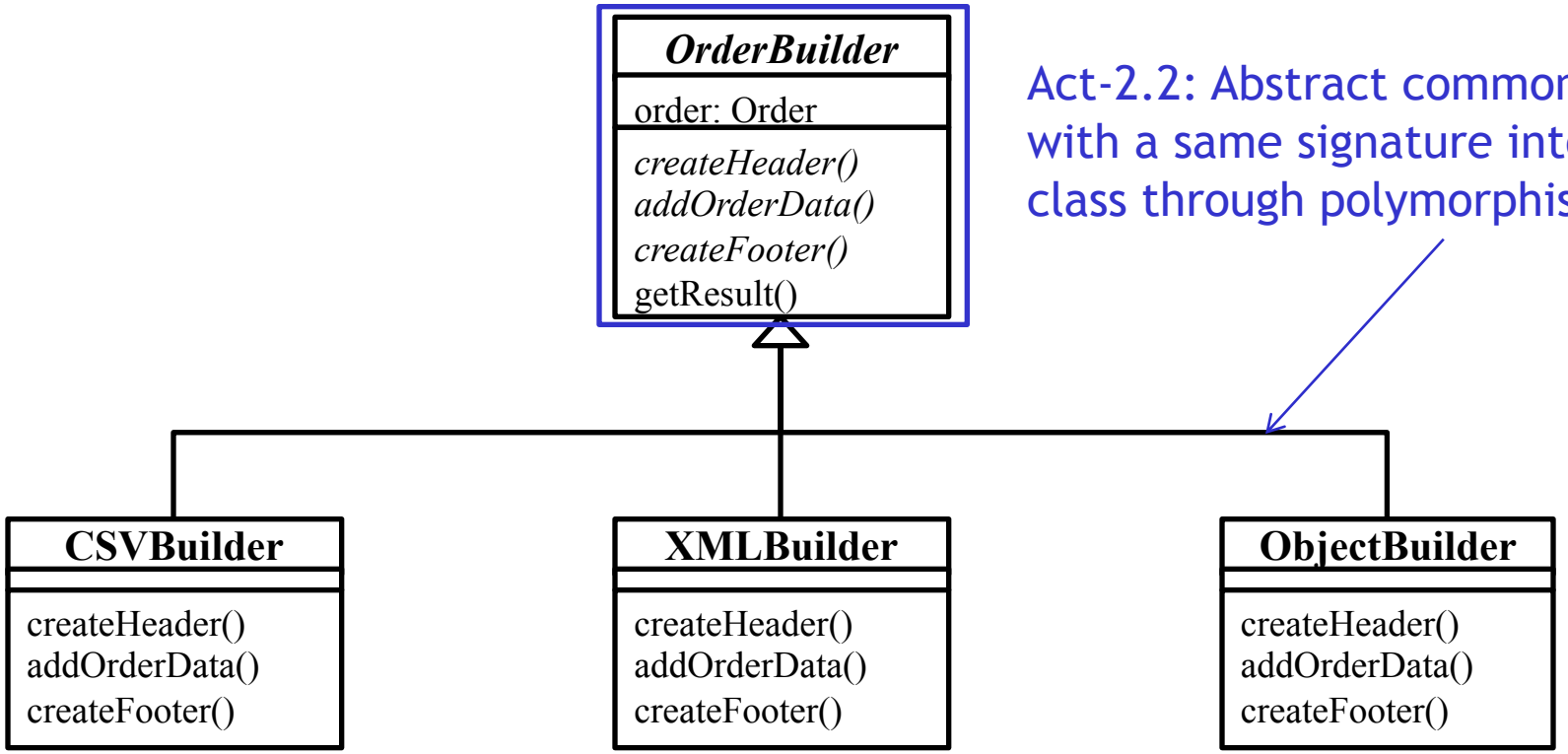


Act-1: Encapsulate What Varies

Act-1.3: Encapsulate a part of a method body into a concrete class

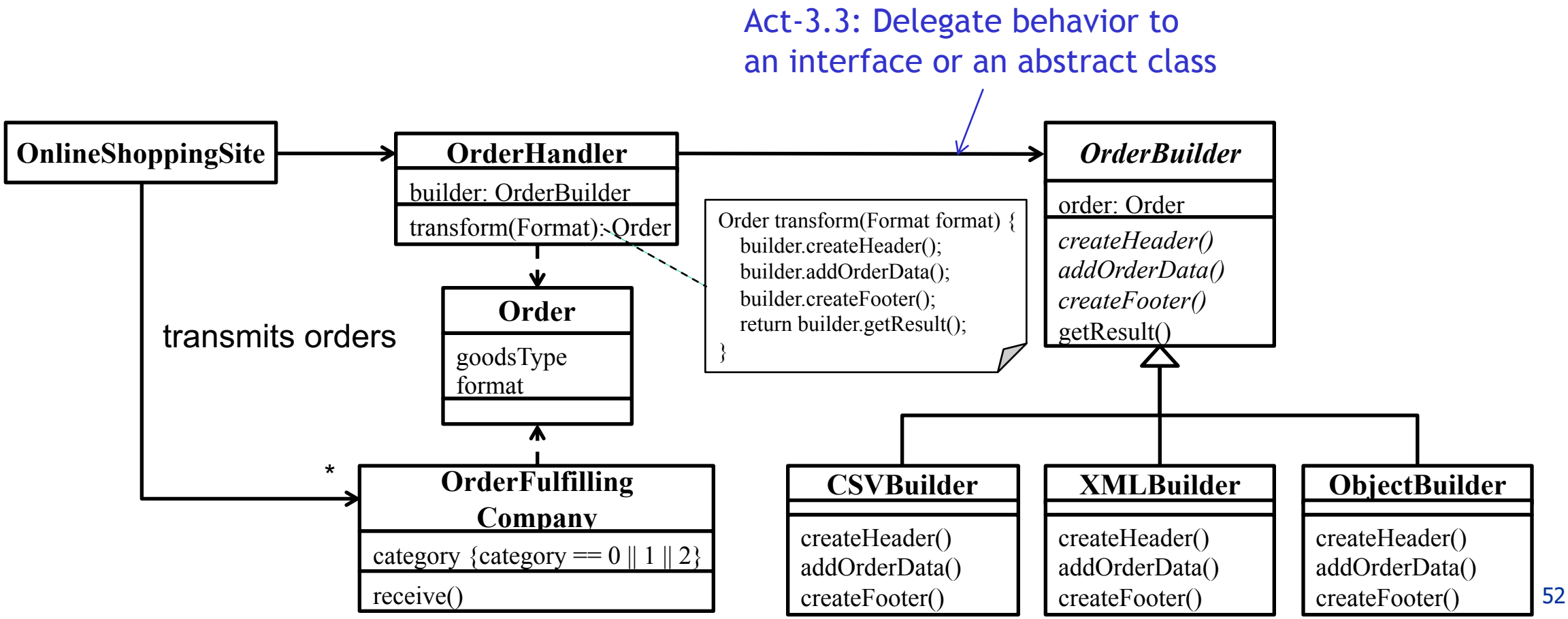


Act-2: Abstract Common Behaviors

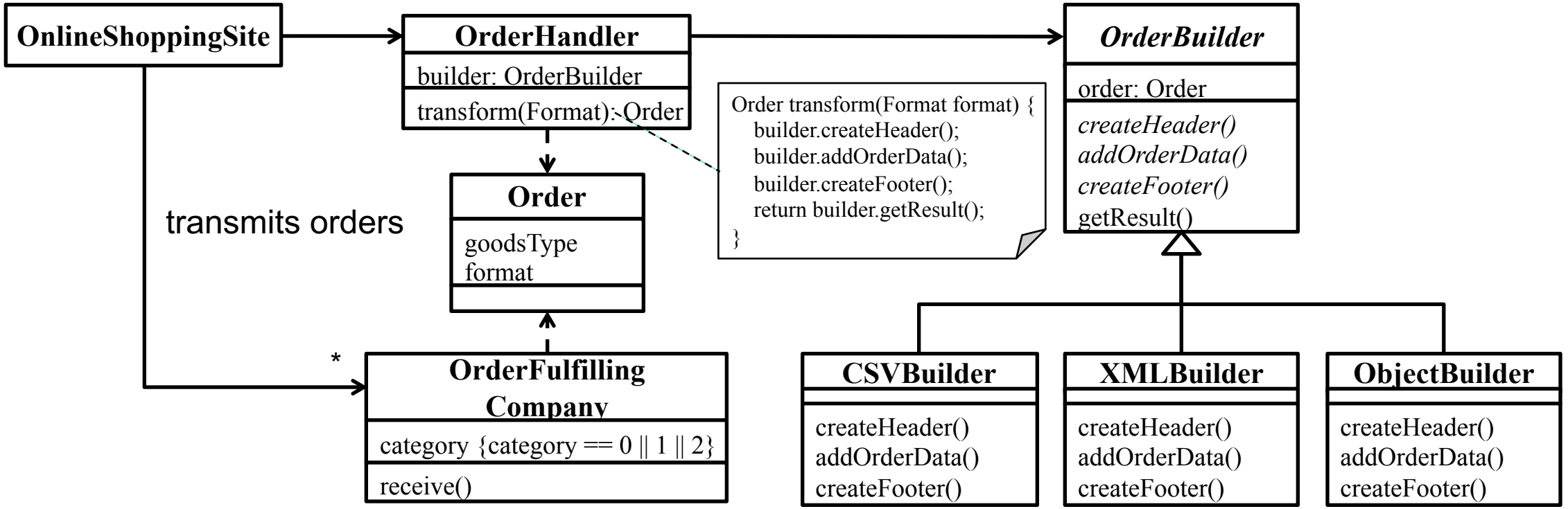


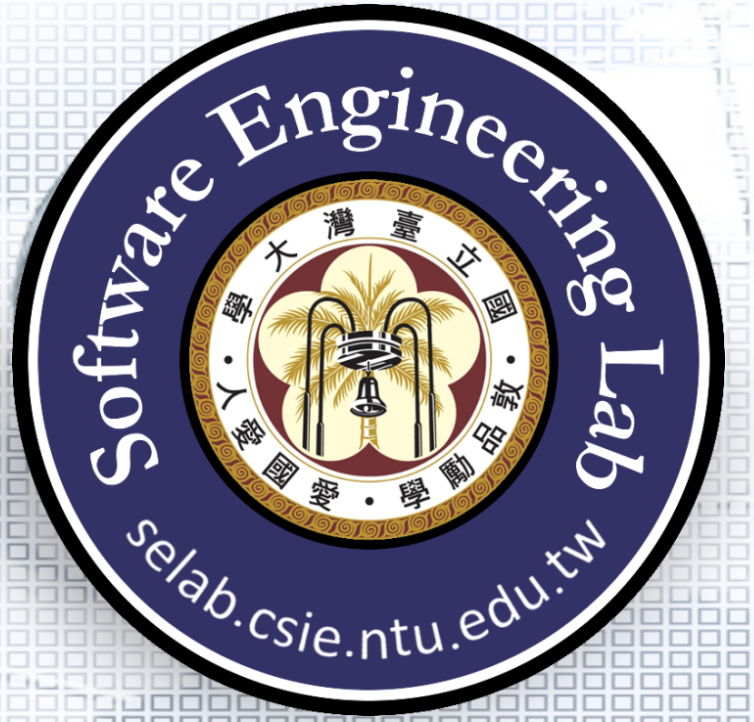
Act-2.2: Abstract common behaviors with a same signature into abstract class through polymorphism

Act-3: Compose Abstract Behaviors



Refactored Design after Design Process





A Sales Reporting Application

Prof. Jonathan Lee (李允中)

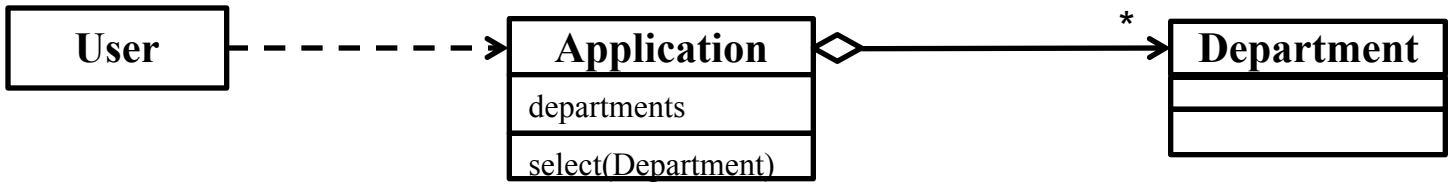
Department of Computer Science and Information Engineering
National Taiwan University

Requirements Statements

- Build a sales reporting application for the management of a store with multiple departments.
- Users should be able to select a specific department they are interested in.
- Upon selecting a department, two types of reports are to be displayed:
  - Monthly report - A list of all transactions for the current month for the selected department.
  - YTD sales chart - A chart showing the year-to-date sales for the selected department by month.
- Whenever a different department is selected, both of the reports should be refreshed with the data for the currently selected department.

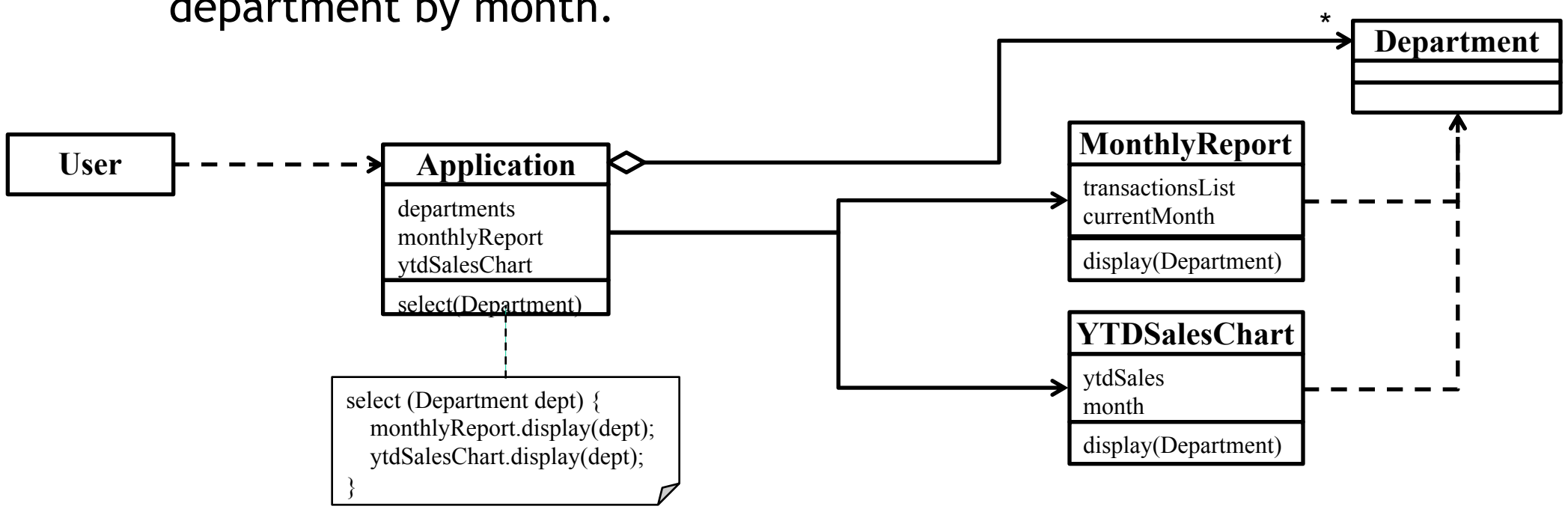
Requirements Statement$_1$

- Build a sales reporting application for the management of a store with multiple departments.
- Users should be able to select a specific department they are interested in.



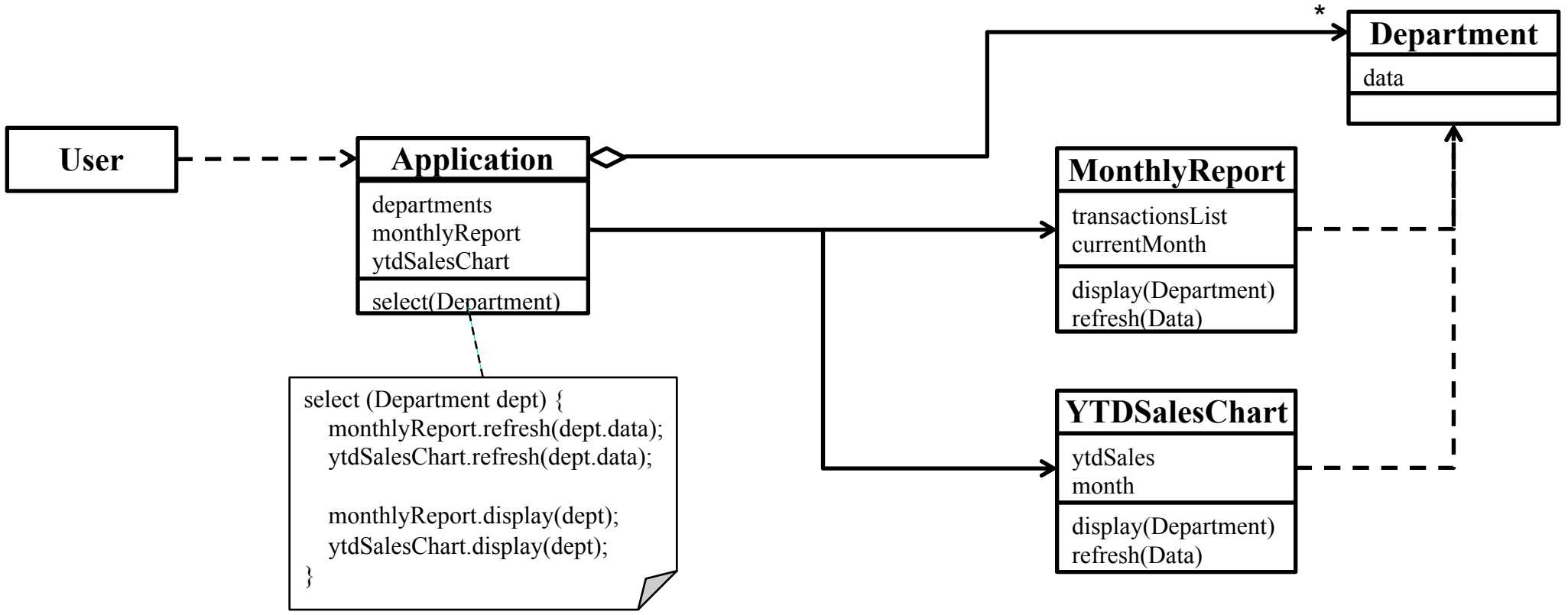
Requirements Statement$_2$

- Upon selecting a department, two types of reports are to be displayed:
  - Monthly report - A list of all transactions for the current month for the selected department.
  - YTD sales chart - A chart showing the year-to-date sales for the selected department by month.

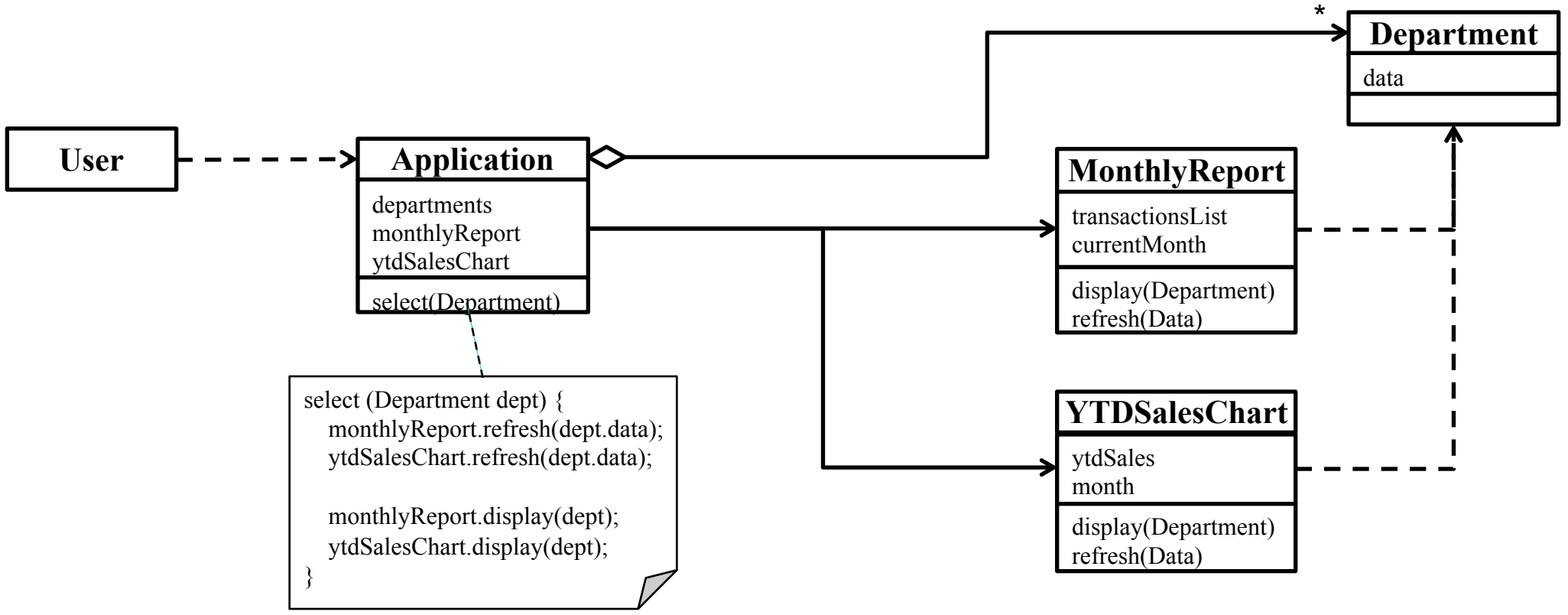


Requirements Statement$_3$

- Whenever a different department is selected, both of the reports should be refreshed with the data for the currently selected department.

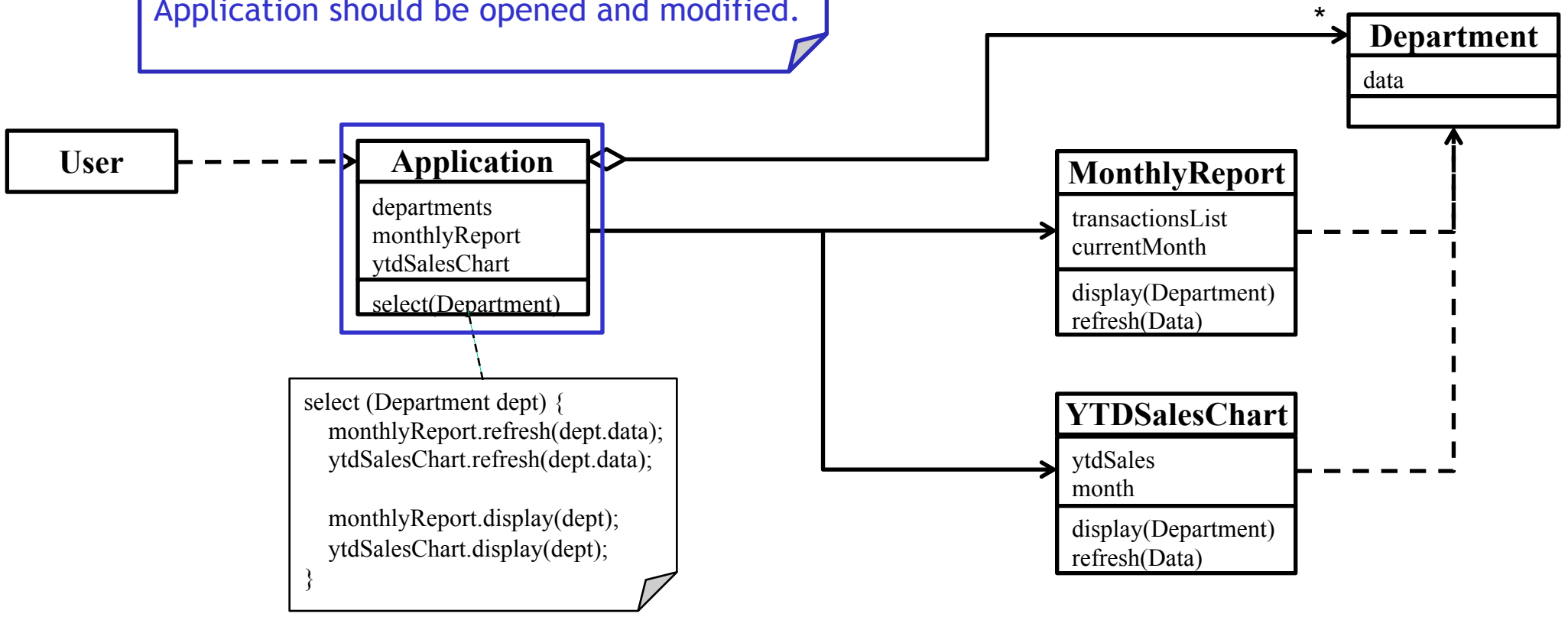


Initial Design

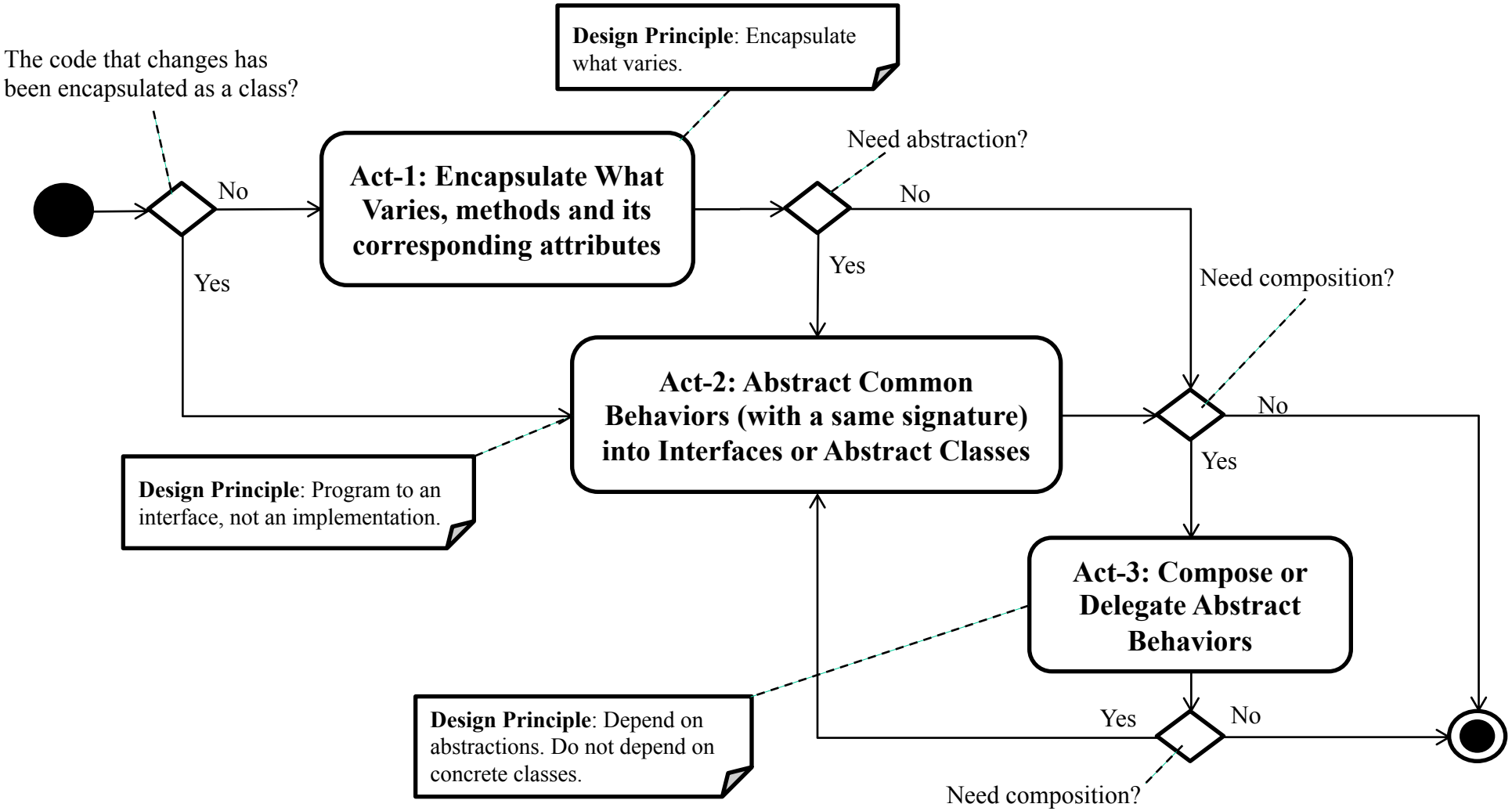


Problems with Initial Design

Problem: If different kind of report is added, Application should be opened and modified.

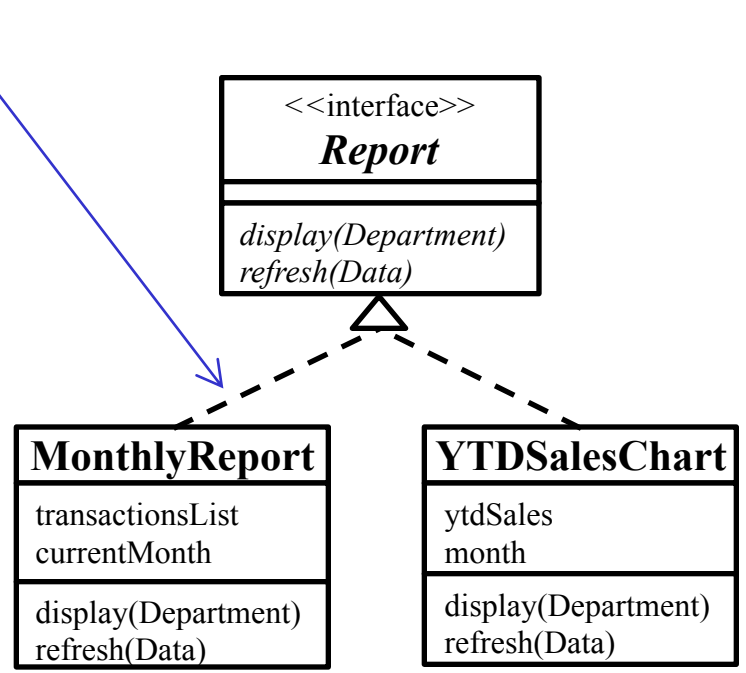


Software Design Process



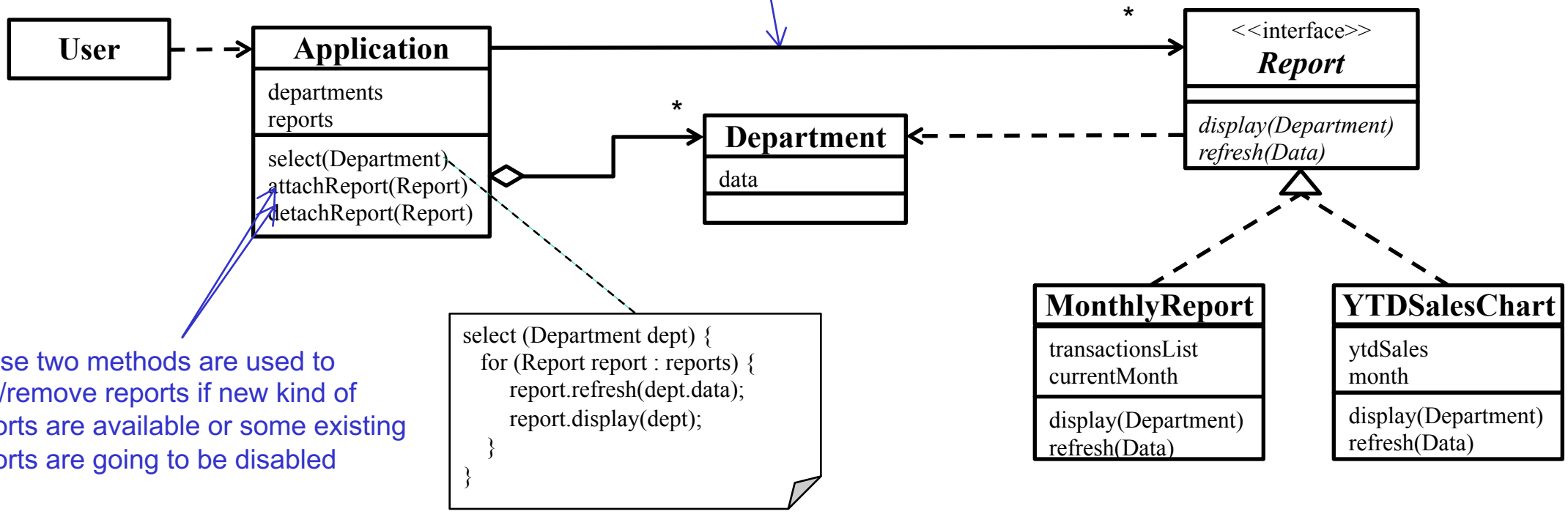
Act-2: Abstract Common Behaviors

Act-2.2: Abstract common behaviors with a same signature into interface through inheritance

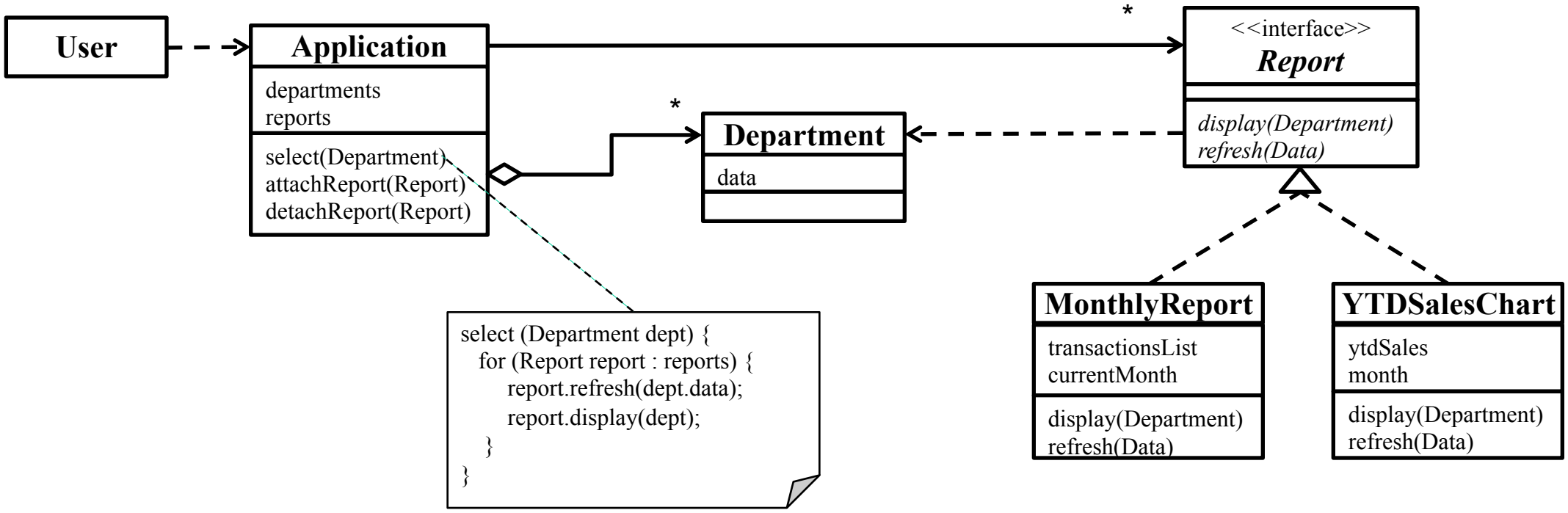


Act-3: Compose Abstract Behaviors

Act-3.1: Compose behaviors of an interface or an abstract class



Redesign after Refactoring



Design Aspects (Creational)

- Abstract Factory: families of product objects
- Builder: how a composite object gets created
- Factory Method: subclass of object that is instantiated
- Prototype: class of object that is instantiated
- Singleton: the sole instance of a class

Design Aspects (Structural)

- Adapter: interface to an object
- Bridge: implementation of an object
- Composite: structure and composition of an object
- Decorator: responsibilities of an object without sub-classing
- Façade: interface to a subsystem
- Flyweight: storage costs of objects
- Proxy: how an object is accessed; its location

Design Aspects (Behavioral)

- Chain of Responsibility: object that can fulfill a request
- Command: when and how a request is fulfilled
- Interpreter: grammar and interpretation of a language
- Iterator: how an aggregate's elements are accessed, traversed
- Mediator: how and which objects interact with each other
- Memento: what private information is stored outside an object, and when
- Observer: number of objects that depend on another object; how the dependent objects stay up to date
- State: states of an object
- Strategy: an algorithm
- Template Method: steps of an algorithm
- Visitor: operations that can be applied to objects without changing their classes

Homework: Requirements Statement

- The Composition class maintains a collection of Component instances, which represent text and graphical elements in a document.
- A composition arranges component objects into lines using a linebreaking strategy.
- Each component has an associated natural size, stretchability, and shrinkability.
- The stretchability defines how much the component can grow beyond its natural size; shrinkability is how much it can shrink.
- When a new layout is required, the composition calls its compose method to determine where to place linebreaks.

- There are 3 different algorithms for breaking lines:
  - Simple Composition: A simple strategy that determines line breaks one at a time.
  - Tex Composition: This strategy tries to optimize line breaks globally, that is, one paragraph at a time.
  - Array Composition: A strategy that selects breaks so that each row has a fixed number of items. It's useful for breaking a collection of icons into rows, for example.