Design Pattern Concepts

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

Outline

What’s a Pattern?

- A pattern is a way to capture expertise.
- Refer to a **best practice** documented as **a solution to a recurring problem** in **a given environment**.
- A pattern language refers to patterns working 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, for example, Evangelist, Fear Less, Personal Touch, Piggyback, Small Success, Step by Step, Time for reflection (from the book of Fearless Change), etc.

Examples of Patterns in the Real Life (I)

- Pattern: a best practice to a recurring problem

- Is the following question a recurring problem?
- Can I return a rental car to the car rental company at the airport during the midnight?

- Solution: Key drop off box and a parking space for the returnning rental car

Examples of Patterns in the Real Life (II)

- A recurring problem: Is there anything I can do to prove that I did return the car with no scratch and the car key was dropped in the box?

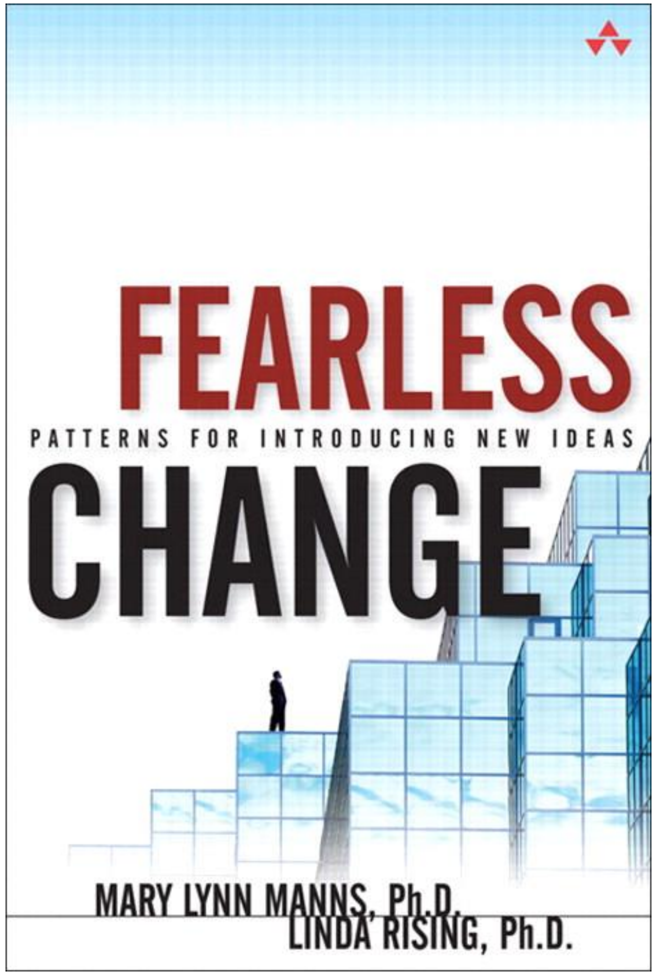
- Solution: Scenario (use case) analysis
- Take pictures of the four sides of the car (or video recording) and the key into the drop off box.

- Scenario (use case) analysis is also a good practice to come up with test cases for software testing

Examples of Patterns in the Real Life (III)

- I was trying to schedule an appointment to renew my driver's license online, but the online scheduler failed to show the renewal option.
- Solution:
  - (1) We should ask ourselves, am I the only person in the whole world encountering this situation alone?
  - (2) If not, then the situation must be a recurring problem.
  - (3) Google it on the web to find potential solutions.
- Because there are only the first name and last name to fill in the online application form, there is no place for the middle name; therefore, the scheduler won't be able to identify the applicant as an existing driver in the driver's license database resulting in pumping up the first-time DL application option all the time.
- Filling in the middle name in the first name field with a space separating them will do the trick.

Fearless Change



- Title: Fearless Change: Patterns for Introducing New Ideas
- Authors: Mary Lynn Manns Ph.D. and Linda Rising Ph.D.
- Publisher: Pearson Education, 2004

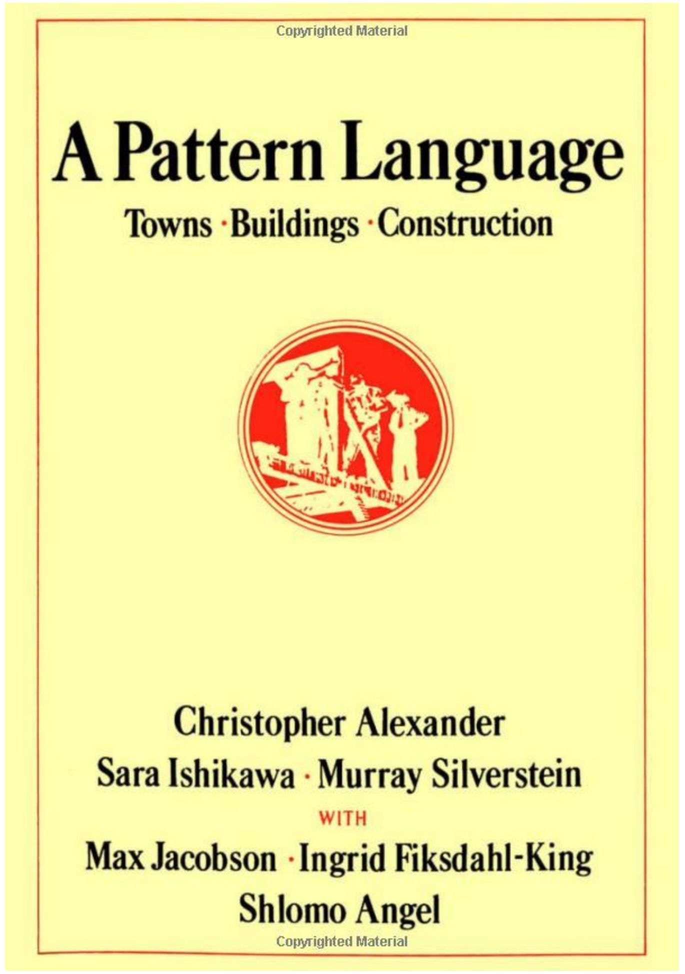
Fearless Change Patterns Description

- **Evangelist**: Introduce a new idea, let your passion for this new idea drive you.
- **Fear Less:** Ask for help from resistors.
- **Personal Touch:** Talk with individuals about the ways in which the new idea can be personally useful and valuable to them.
- **Piggyback**: Leverage the new idea on a well-accepted practices in the organization.
- **Small Success:** As you carry on **Step by Step**, take the time to recognize and celebrate successes, especially the small ones.
- **Step by Step**: Use an incremental approach in the change initiative, with short-term goals, while keeping your long-term vision.
- **Time for Reflection**: Pause in any activity to reflect on what is working well and what should be done differently.

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**."(from the book of A Pattern Language)

A Pattern Language


Copyrighted Material

A Pattern Language

Towns · Buildings · Construction

Christopher Alexander

Sara Ishikawa · Murray Silverstein

WITH

Max Jacobson · Ingrid Fiksdahl-King

Shlomo Angel

Copyrighted Material

- Title: A Pattern Language: Towns, Buildings, Construction
- Authors: Christopher Alexander, etc
- Publisher: Oxford University Press, New York, 1977

Essential Elements of a Pattern

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

Software System Ability

- System Flexibility: the ability of a system to cope with **variety (Encapsulate What Varies)**.

- System Extensibility: the ability for **future growth** without **impairing** existing system functions (**Open-Close Principle**).

- System Portability: the usability of a same software in different platforms or frameworks (**separate abstraction from its implementation**).

Design Patterns Category$_1$

- Design patterns are classified by two criteria:
  - Scope: specifies whether the pattern applies primarily to classes or to objects, including class pattern and object pattern.
  - Purpose: reflect what a pattern does, including creational, structural and behavioral.
- 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 are established via **composition** and can be changed at run-time. Example: 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, Facade, 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 **composite structure (example, 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.

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**
  - **design patterns**

Object-Oriented Concepts

- Inheritance (Generalization, Classification)
- Polymorphism

- Encapsulation
- Abstraction
- Delegation

- Association
- Composition (Aggregation)

Object-Oriented Design Principles

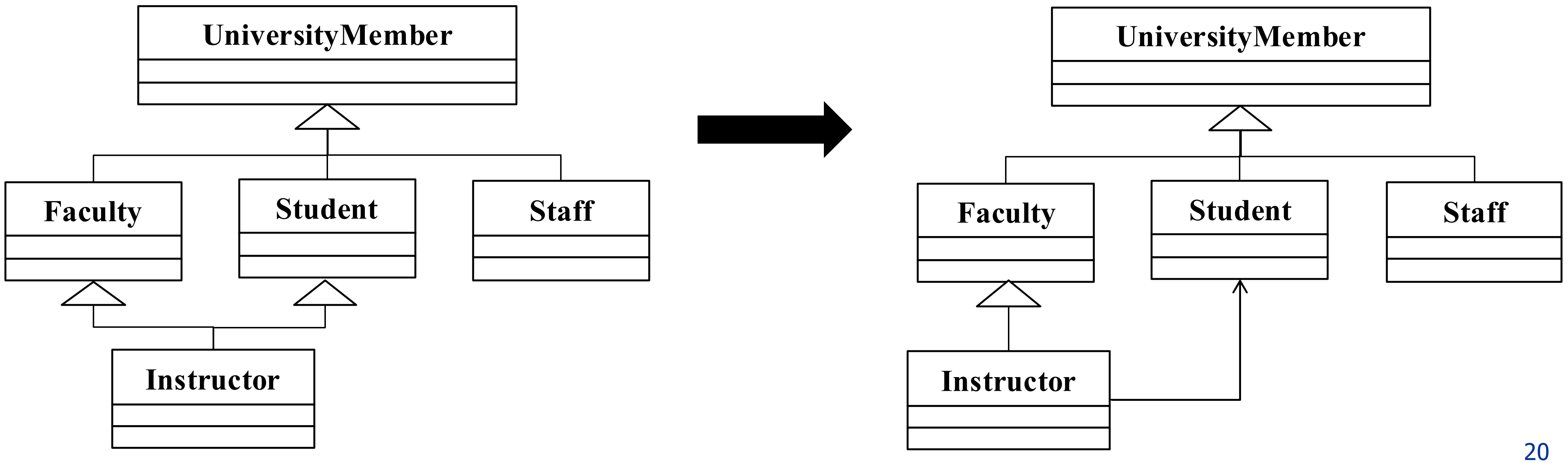
Prof. Jonathan Lee (李允中)
CSIE Department
National Taiwan University

Fundamental Design Principles

- Fundamental design principle: High cohesion and 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.

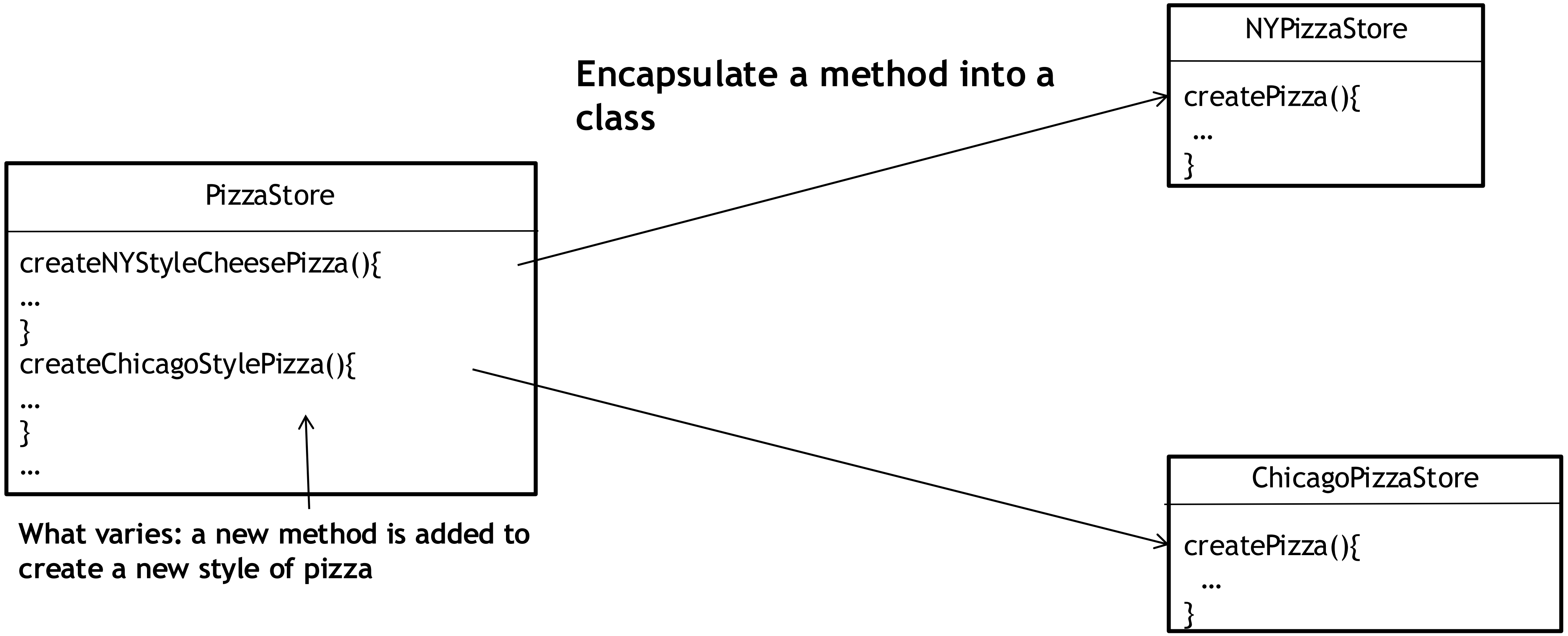
Inherit the most important features and delegate the rest

- 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.



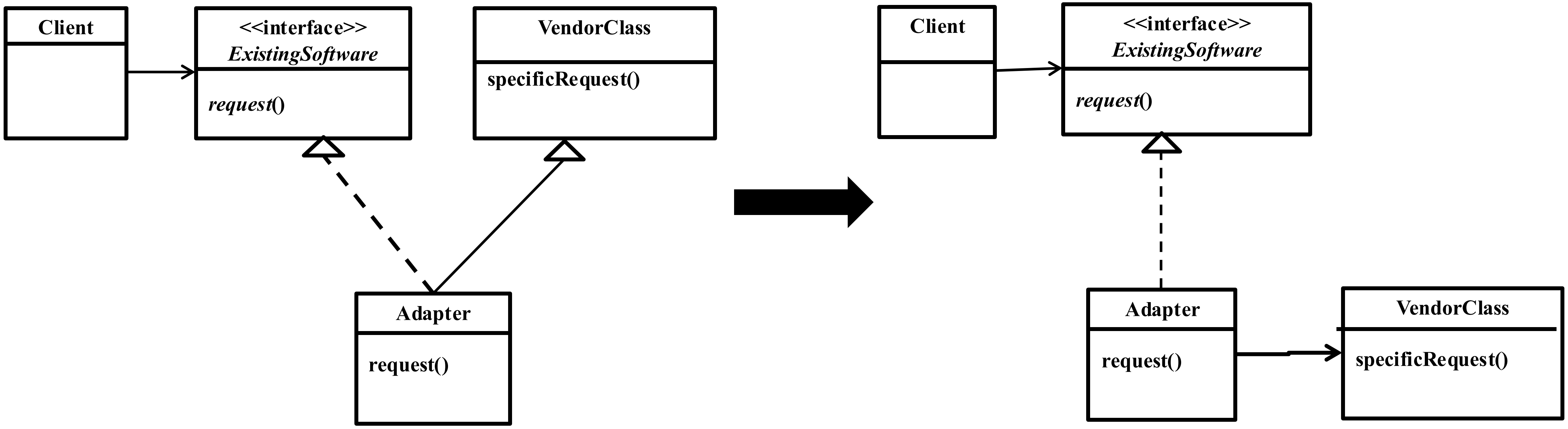
Encapsulate what varies

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


PizzaStore
createNYStyleCheesePizza(){
…
}
createChicagoStylePizza(){
…
}
…
Encapsulate a method into a class
NYPizzaStore
createPizza(){
…
}
ChicagoPizzaStore
createPizza(){
…
}
What varies: a new method is added to create a new style of pizza

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$_3$

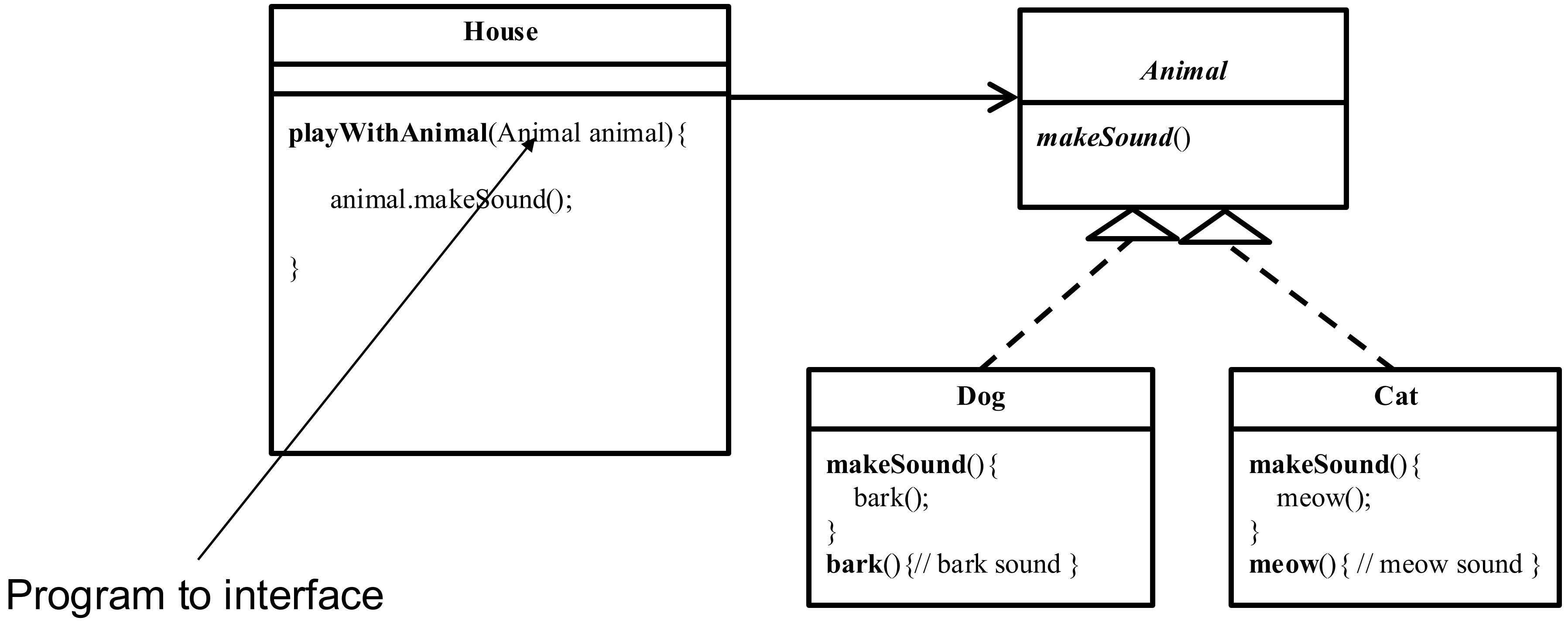
- Composition (Advantage)
  - Composition requires objects to respect each other’s interfaces, and therefore, we don’t break encapsulation.
  - Fewer implementation dependencies.
  - Classes and class hierarchies will remain small and will be less likely to grow into unmanageable monsters.
- Disadvantage: 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.
- **Extending functionality by sub-classing and composing its instances with existing ones.**

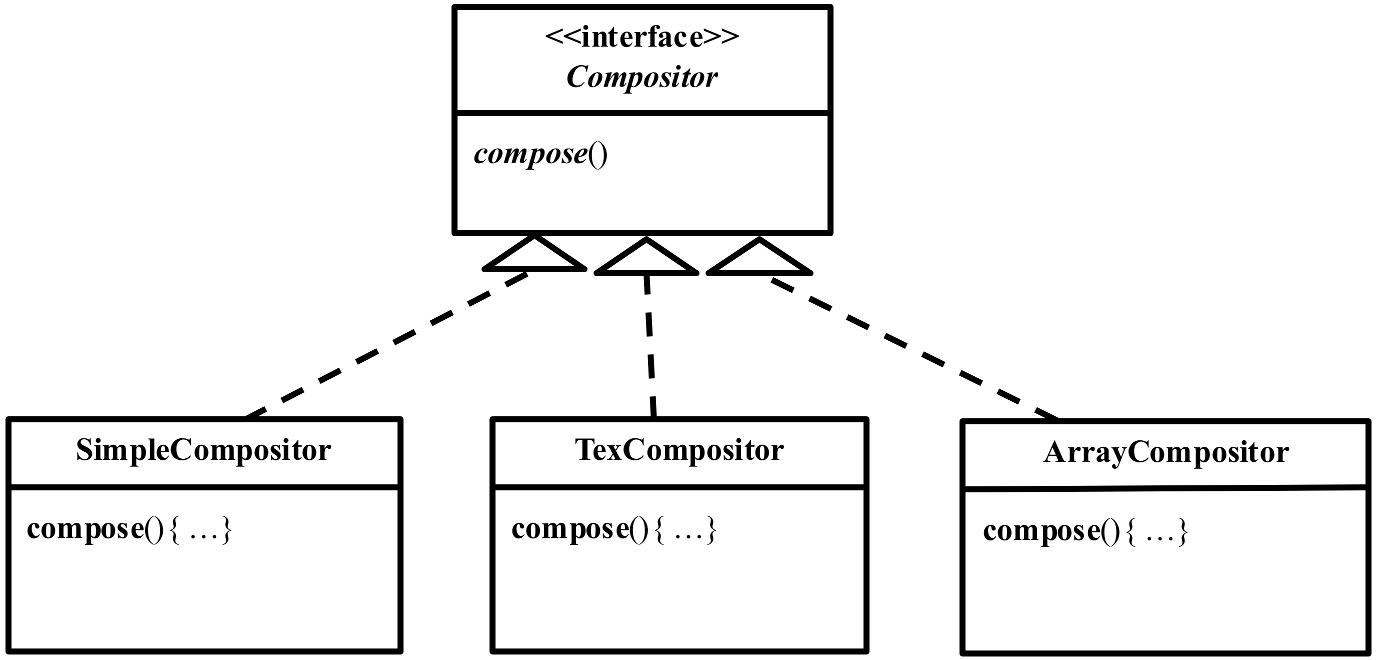
Program to Interface, not Implementation

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



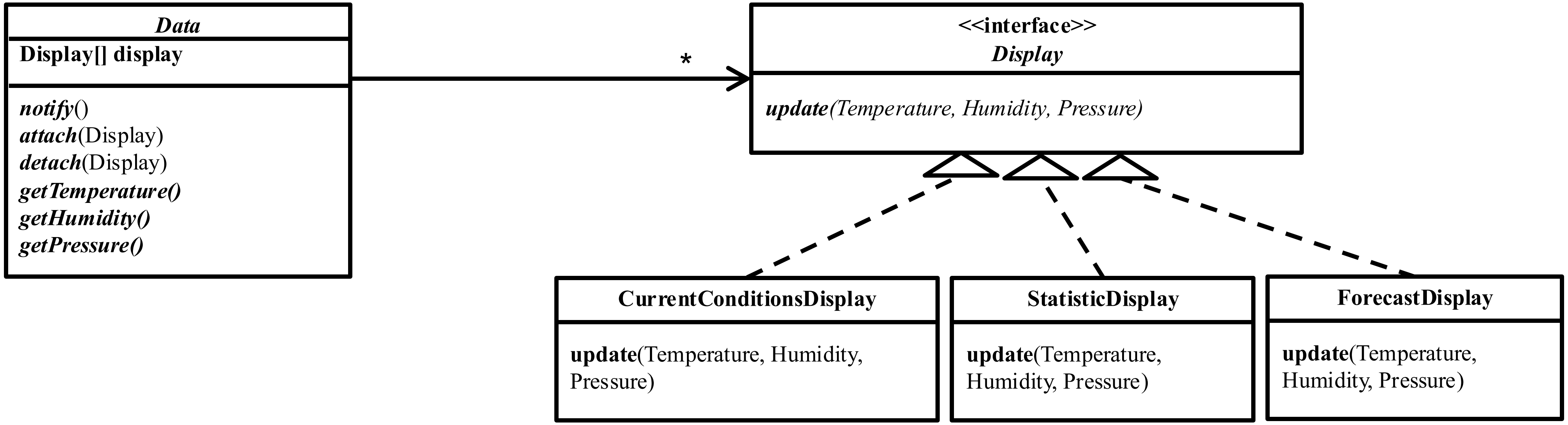
Open for Extension, Close for Modification

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



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.

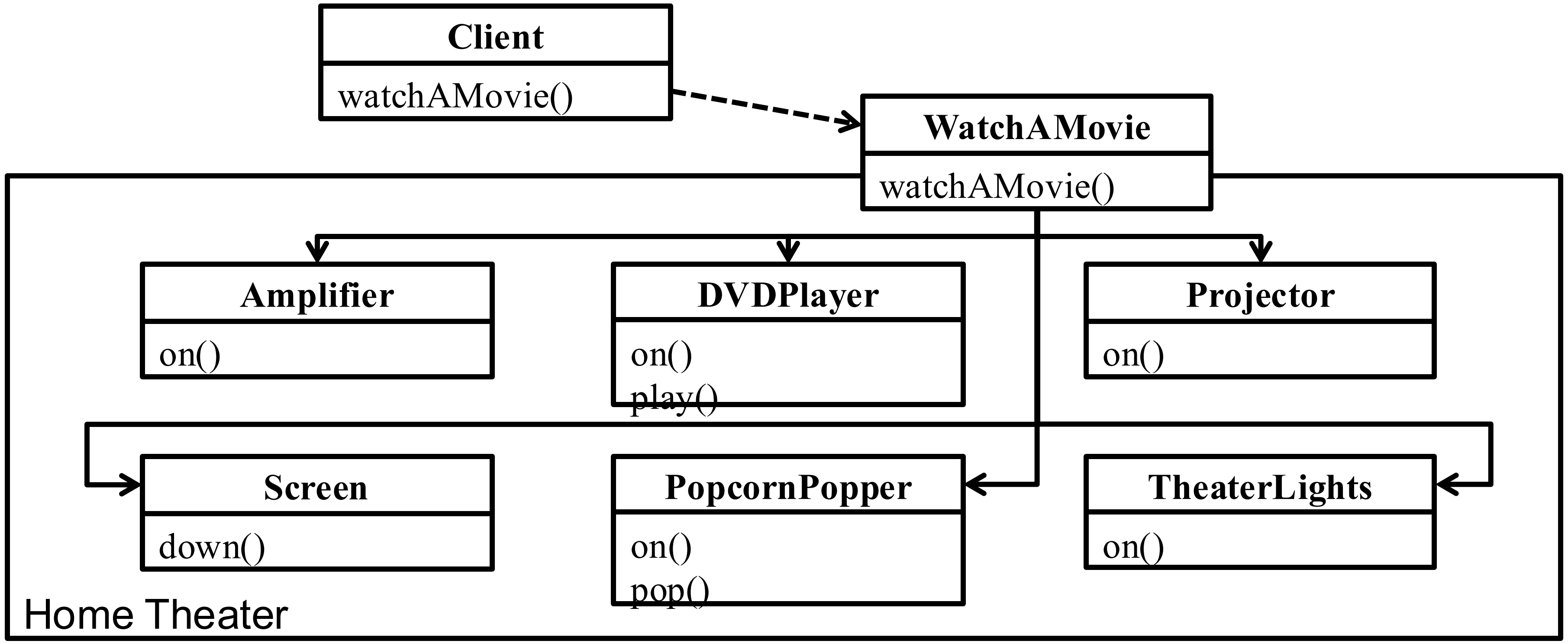


Law of Demeter (Principle of Least Knowledge)

- Each unit should have only limited knowledge of other units in the system.
- A unit should talk to its friends only.
- A unit should not know about the internal details of the object that it manipulates.
- Demeter is a Greek goddess of agriculture

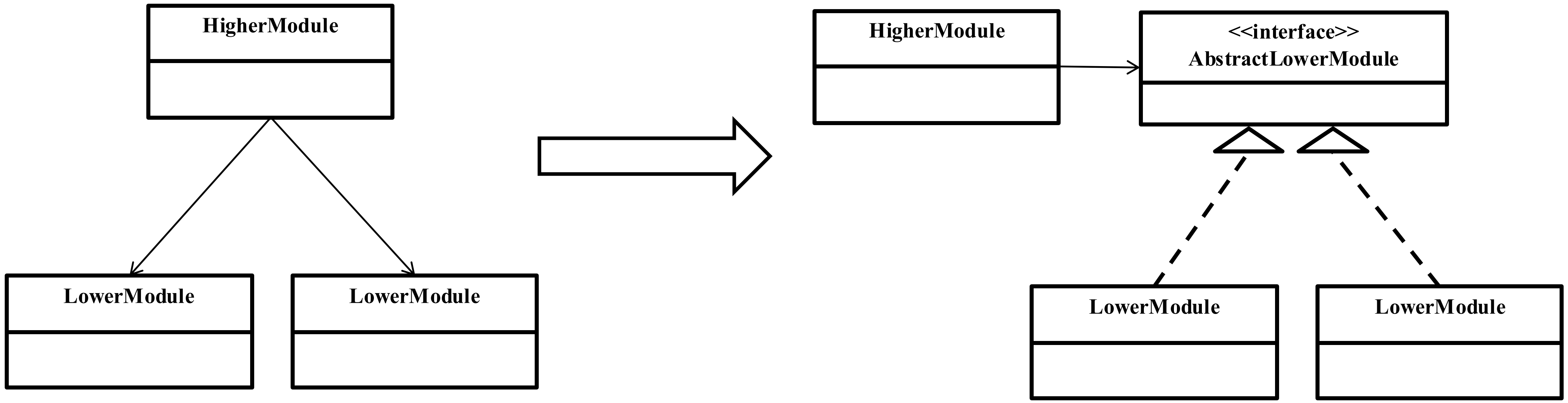
Only Talk to your Friends

- **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 (multiple interfaces are allowed).



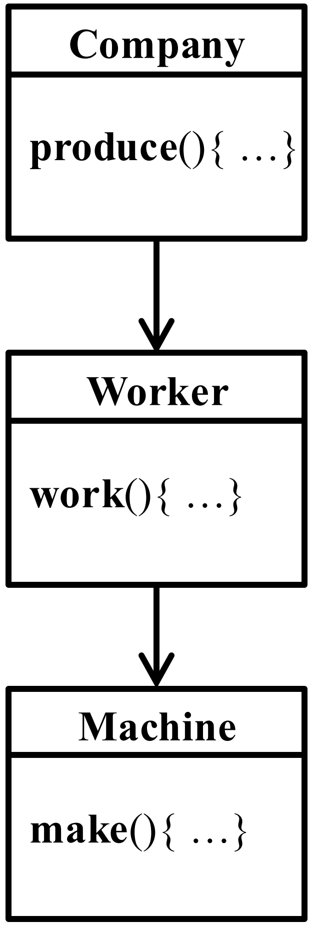
Depend on Abstractions not on Concrete Classes

- **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.



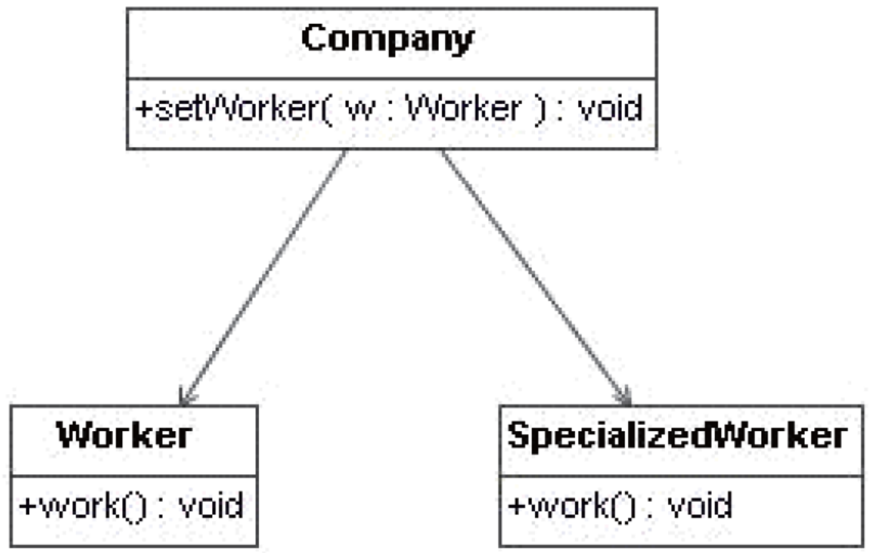
Dependency Inversion Principle (DIP)

- 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: An Example

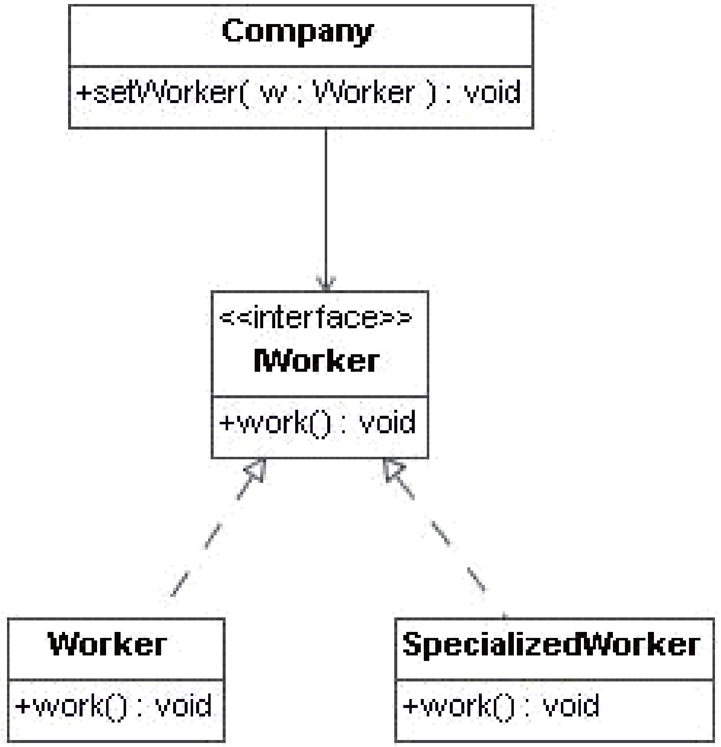
- Company NTU has established a new business which requires a new kind of workers, called SpecializedWorker. In order to reflect this change, the Company Class (high-level component) has been revised to include SpecializedWorker class. This is a clear violation of Dependency Inversion Principle.



```
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

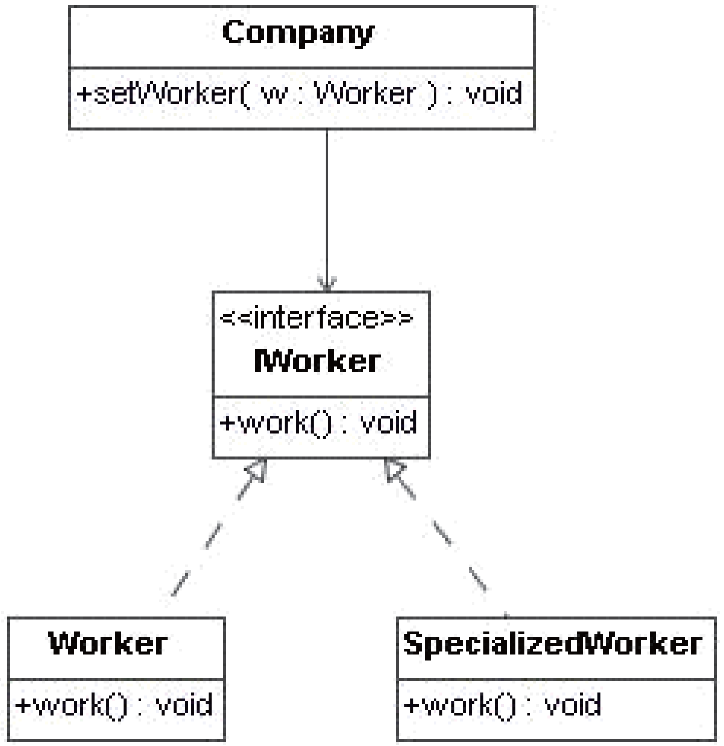
- Insert an isolation layer (Abstract layer) between the high-level component and low-level components to make both types of components depend on this layer.



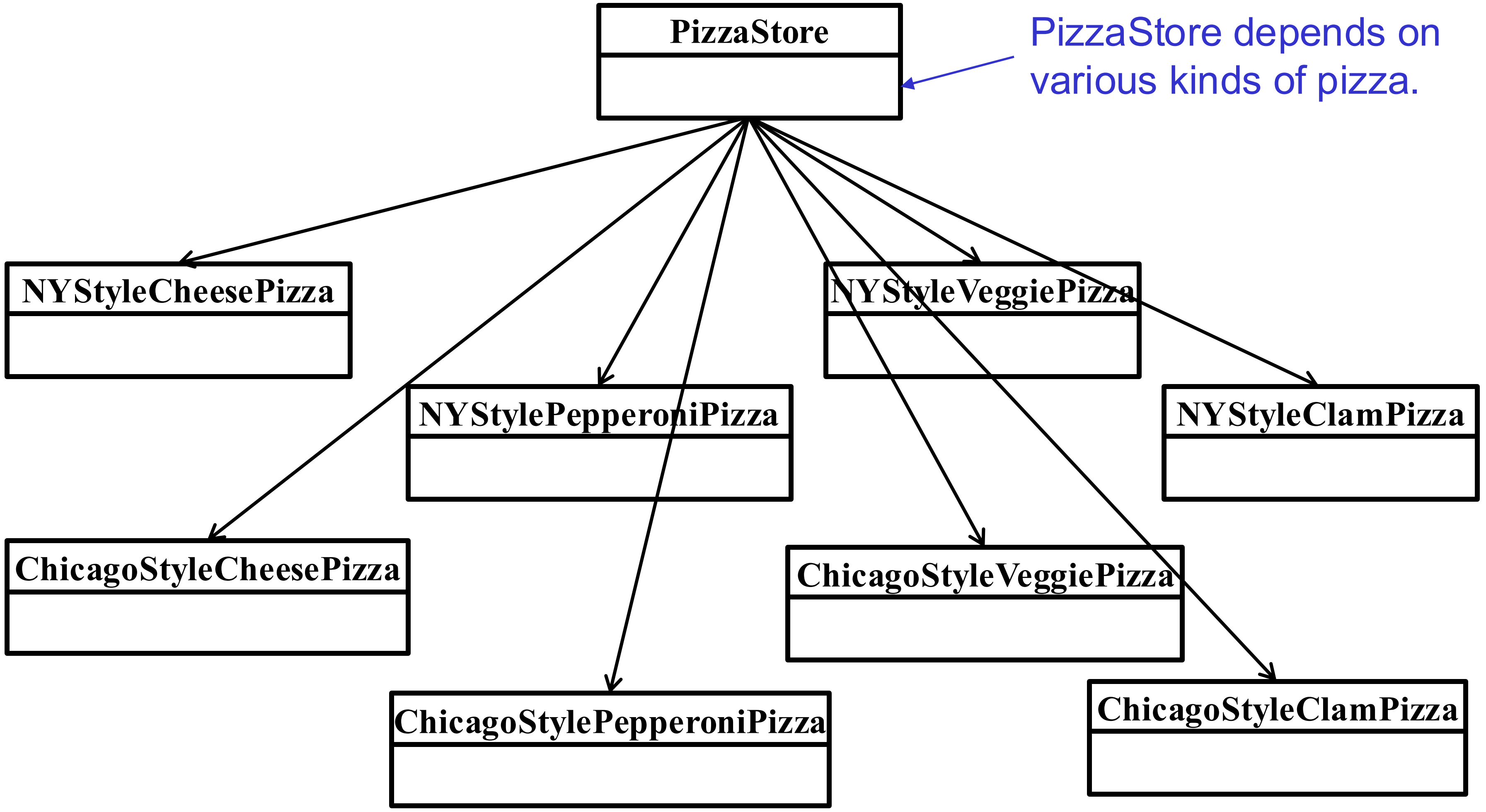
```
class Company {
            IWorker worker;
publich void setWorker (IWorker w) {
            worker = w;
}
public void produce () {
            worker.work (); }
}
```

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.

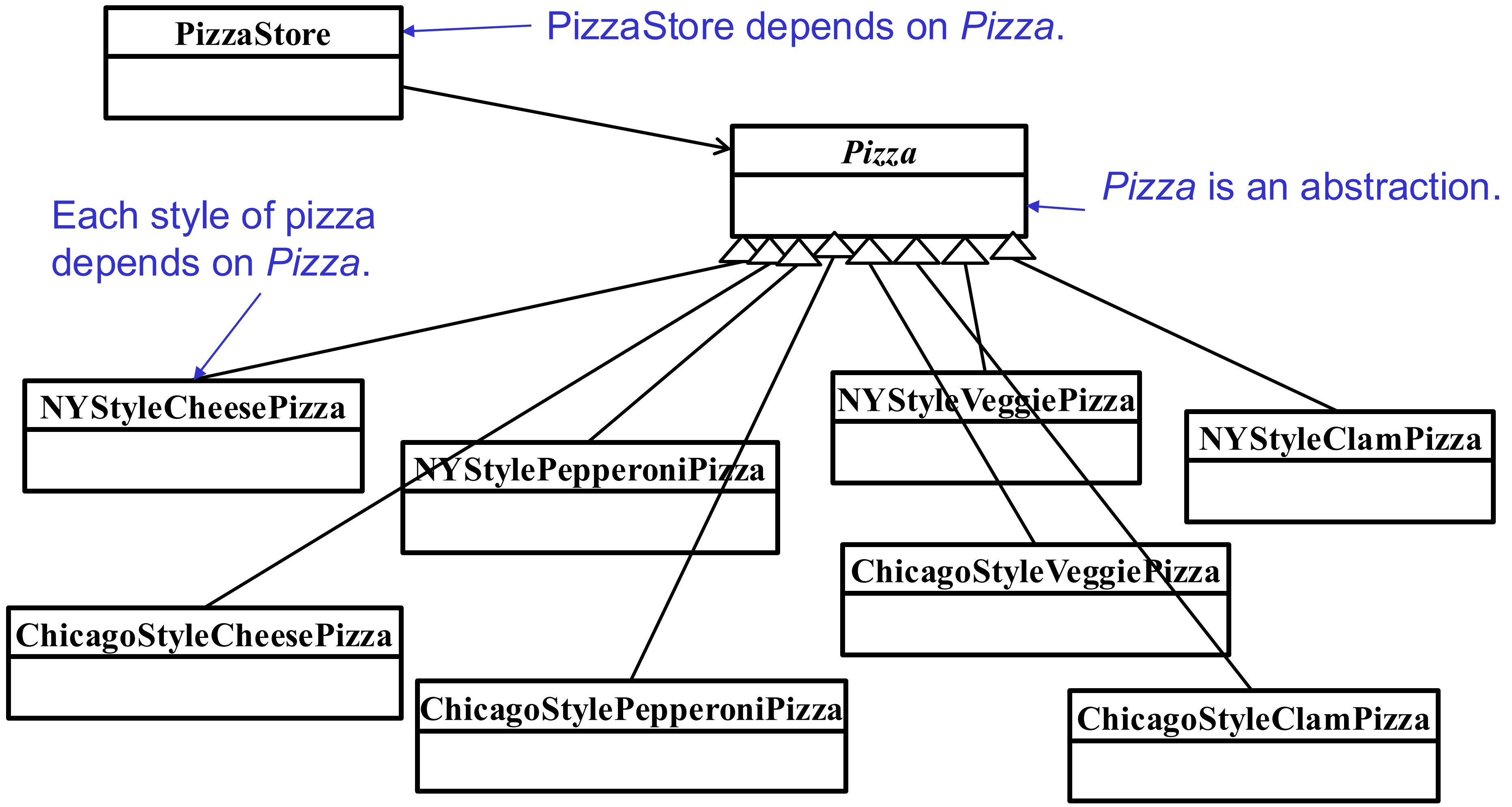


Violate DIP: PizzaStore



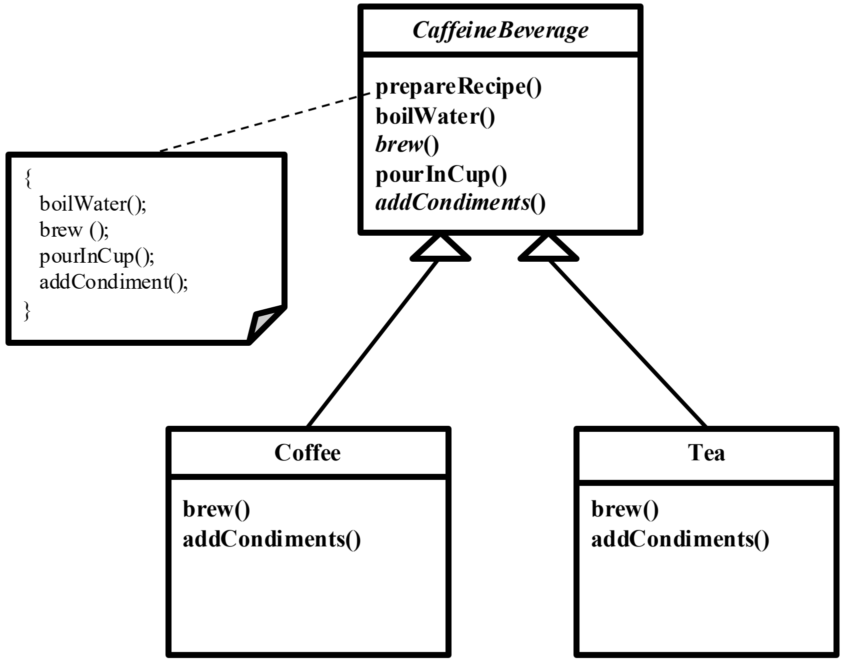
PizzaStore depends on various kinds of pizza.

Enforce DIP: Insert Isolation Layer



Don’t Call us, we’ll Call you.

- **The Hollywood Principle:** Film production houses call actors if there is any role for the actor, but not the other way around. In the software design, High level components call low level components, but not vice verse.



The high-level components control when and how, that is, arrange the steps.

A low-level component is called on to define the concrete implementation for the steps, and 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, *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.
- Dependency Inversion Principle: Depend on abstractions. Do not depend on concrete classes. 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)

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

OO Design Principles (IV)

- Least Knowledge Principle: Only talk to your friends. The least we know about the subsystem components, the better.
  - Provide a unified interface to a set of interfaces in a subsystem.
- The Hollywood Principle: Don’t call us, we’ll call you. 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)

- Single Responsibility Principle: A class should have only one reason to change. 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** in designing your systems so that they can **evolve** accordingly.
- A design that doesn't **take changes 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$_2$

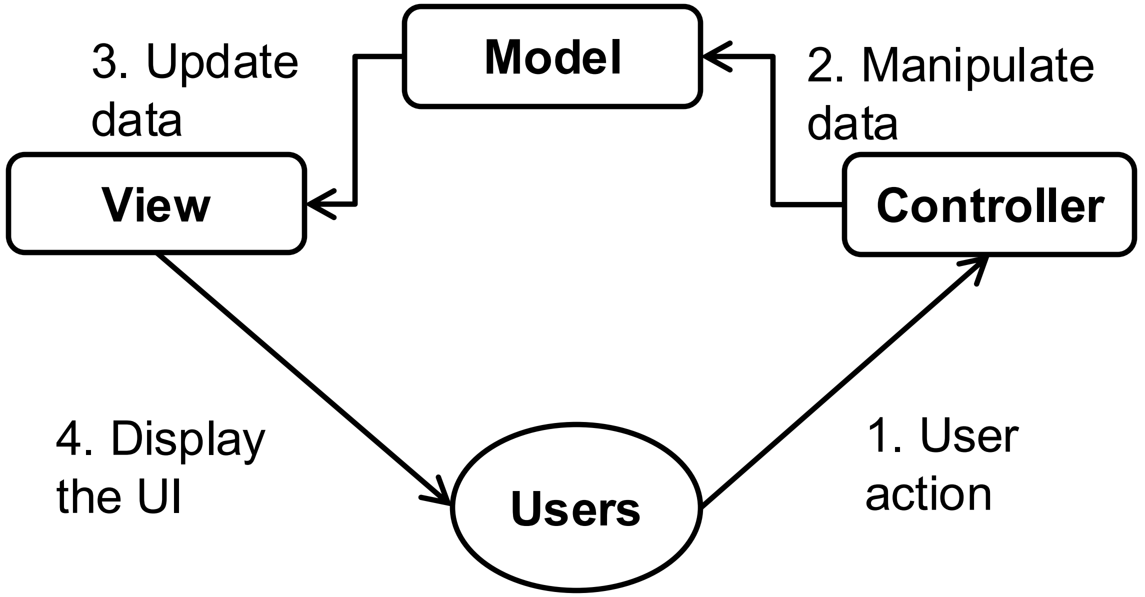
- Create object indirectly.
- Avoid hard-coded requests.
- Limit platform dependencies.
- Hide object representation from clients.
- Algorithms likely to change should be isolated.
- Avoid tight coupling.
- **Extend functionality by subclassing and composing its instances with existing ones.**

Design Pattern vs Framework

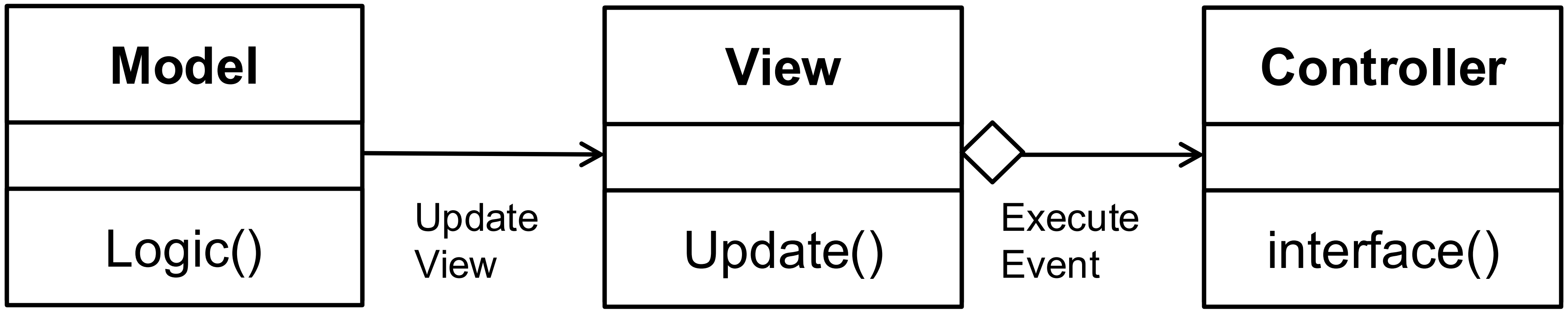
- A typical framework contains several design patterns, but the reverse is never true.
- Framework always have a particular application domain. Whilst design patterns can be used in nearly any kind of application.
- Compound pattern: combine two or more patterns into a solution that solves a recurring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

MVC Pattern: Model-View-Controller

- Model: store and manipulate data.
- View: build user interfaces and data display.
- Controller: connect the model and view.
- Users: request for results based on the corresponding actions.

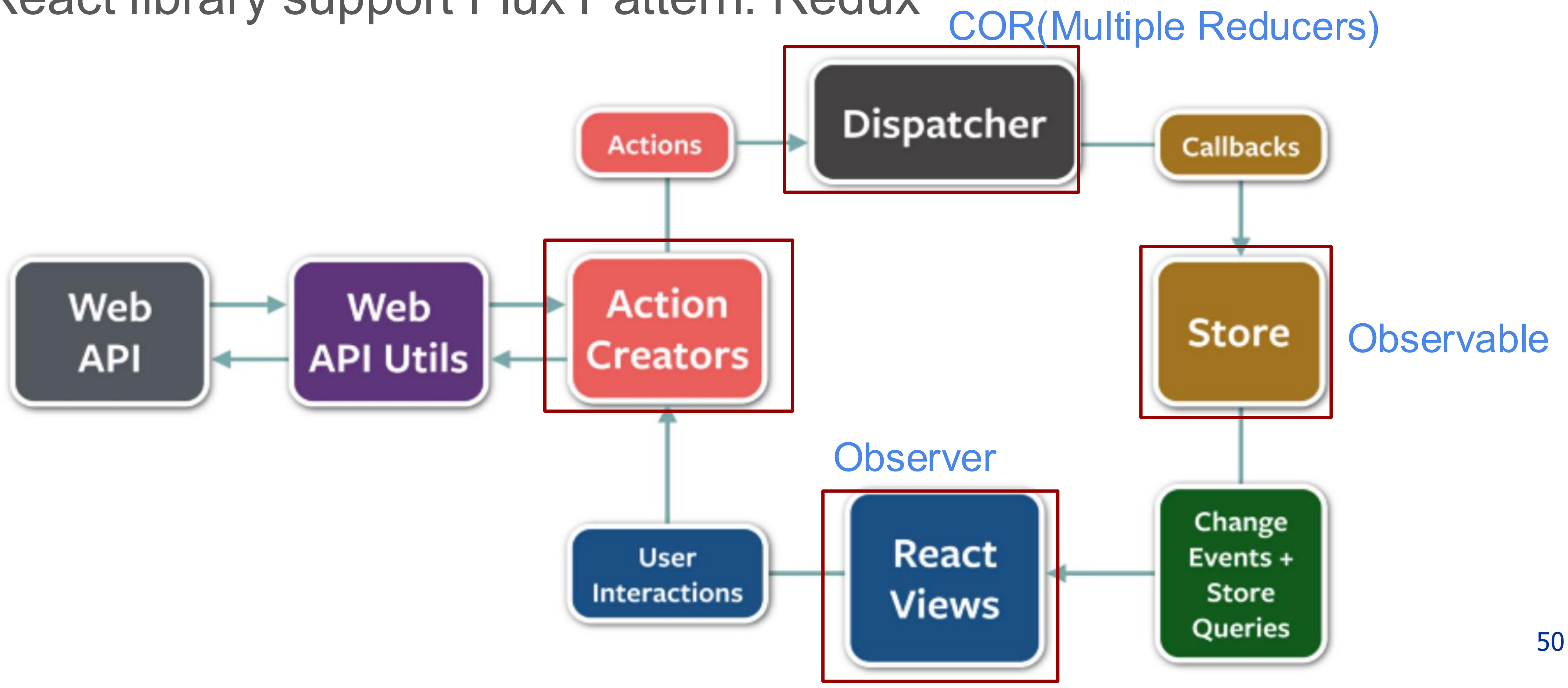


MVC Class Diagram



FLUX Compound Pattern

- ❑ Flux is an architectural pattern proposed by Facebook for building SPAs.
- ❑ Angular library support Flux Pattern: NgRx
- ❑ React library support Flux Pattern: Redux



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
  - Design patterns for JavaScript

Designs to Avoid: Anti-Patterns

- Anti-pattern is an outcome of an solution to recurring problems that is ineffective and counterproductive.

- Two kinds of anti-patterns:
  - Software development antipatterns: code-level
  - Software architecture antipatterns: architecture-level

Development Anti-Patterns

- Spaghetti code: no modularity
- Golden hammer: a hammer for all the nails possible
- Lava flow: lava turning into a hard rock, a piece of code turing to dead code. That is, an unusable piece of code lying in a program for the fear of breaking the program.
- Copy-and-paste programming

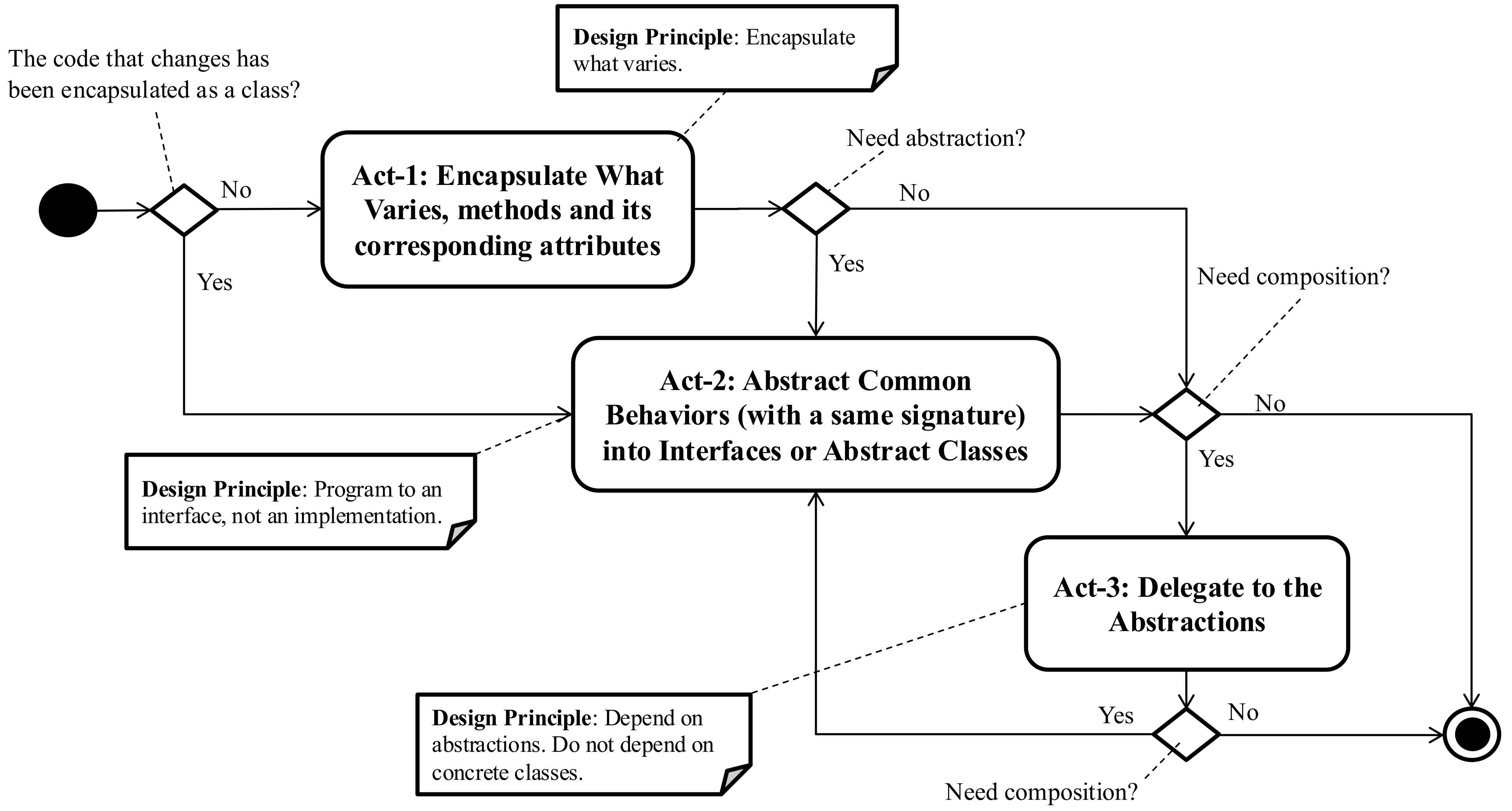
Arcchitecture Anti-Patterns

- Re-inventing the wheel: Reuse a well-thought architecture or design, don’t reinvent the wheel.
- Vendor lock-in: Vendor’s technology are too glued in the system to move away.
- Design by committee: a design invloves too many parties with their own processes and technologies that may not have the right skillset or expereience in the design.

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 etc.
- ❑ Identify the **design aspects** for various kinds of design patterns.

Design Process: Encapsulation, Abstraction, Delegation


The code that changes has been encapsulated as a class?
No
Yes
Design Principle: Encapsulate what varies.
Act-1: Encapsulate What Varies, methods and its corresponding attributes
Need abstraction?
No
Yes
Need composition?
Act-2: Abstract Common Behaviors (with a same signature) into Interfaces or Abstract Classes
Design Principle: Program to an interface, not an implementation.
No
Yes
Act-3: Delegate to the Abstractions
Design Principle: Depend on abstractions. Do not depend on concrete classes.
Yes
No
Need composition?

Design Process Act-1: Encapsulation

- Design Principle: Encapsulate What Varies
- Varies: methods, attributes, part of a method body, request, iteration, dependency
- Encapsulate to concrete classes

Design Process Act-2: Abstraction

- Design Principle: Program to an interface, not to an implementation

- Abstract common behaviors (with a same signature) into interfaces or abstract classes.

Design Process Act-3: Delegation

- Design Principle: Depend on abstractions, do not depend on concrete classes.
- Delegate to the abstraction: interfaces or abstract classes

An Order Handling System

Prof. Jonathan Lee (李允中)

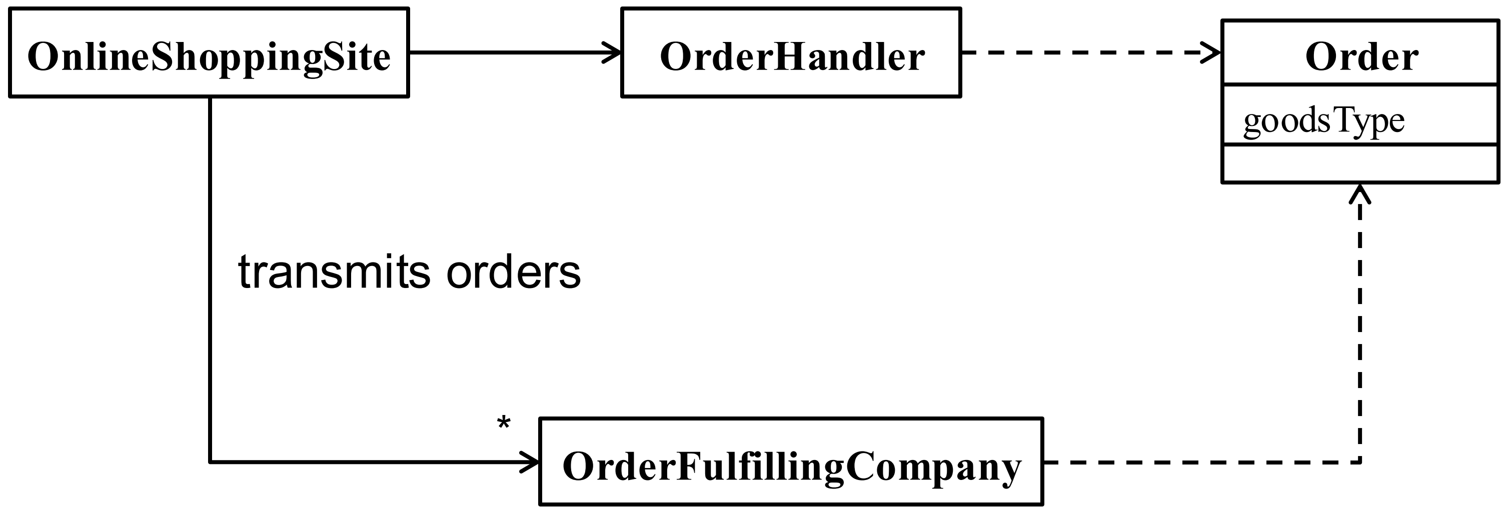
Department of Computer Science and Information Engineering
National Taiwan University

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

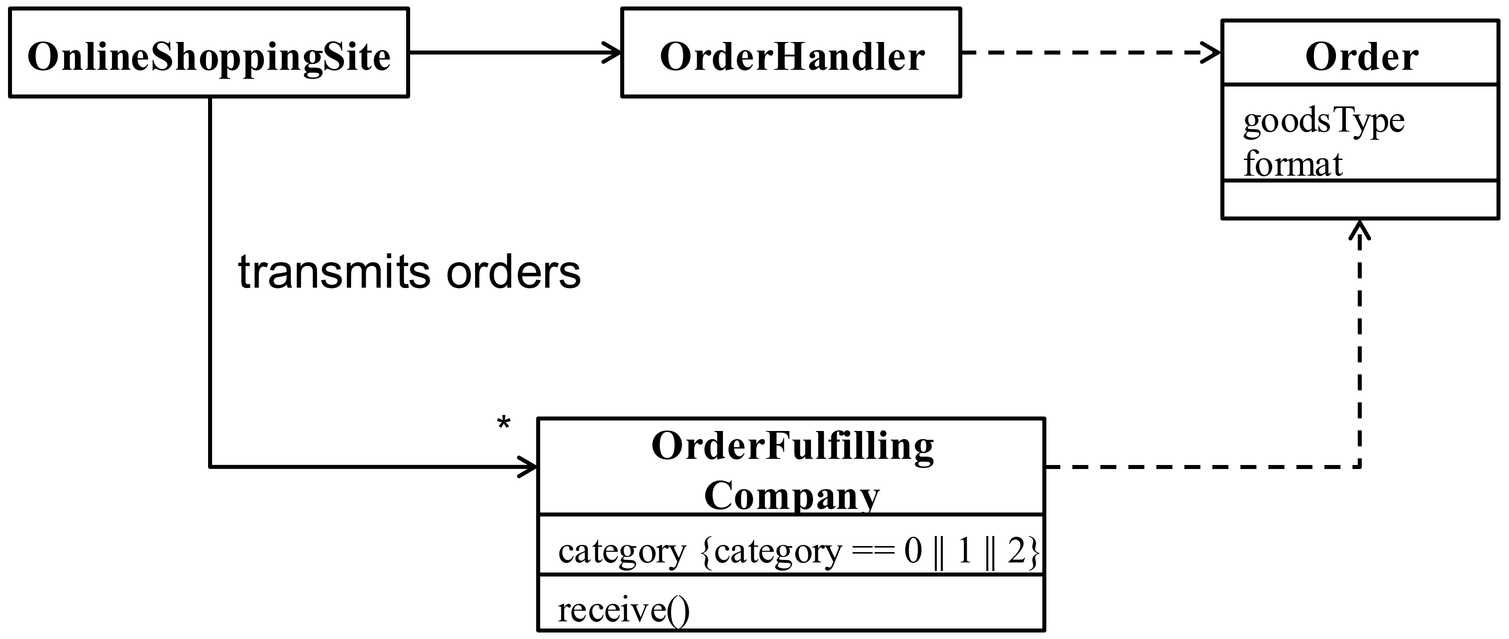
Requirements Statement$_1$

- 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.



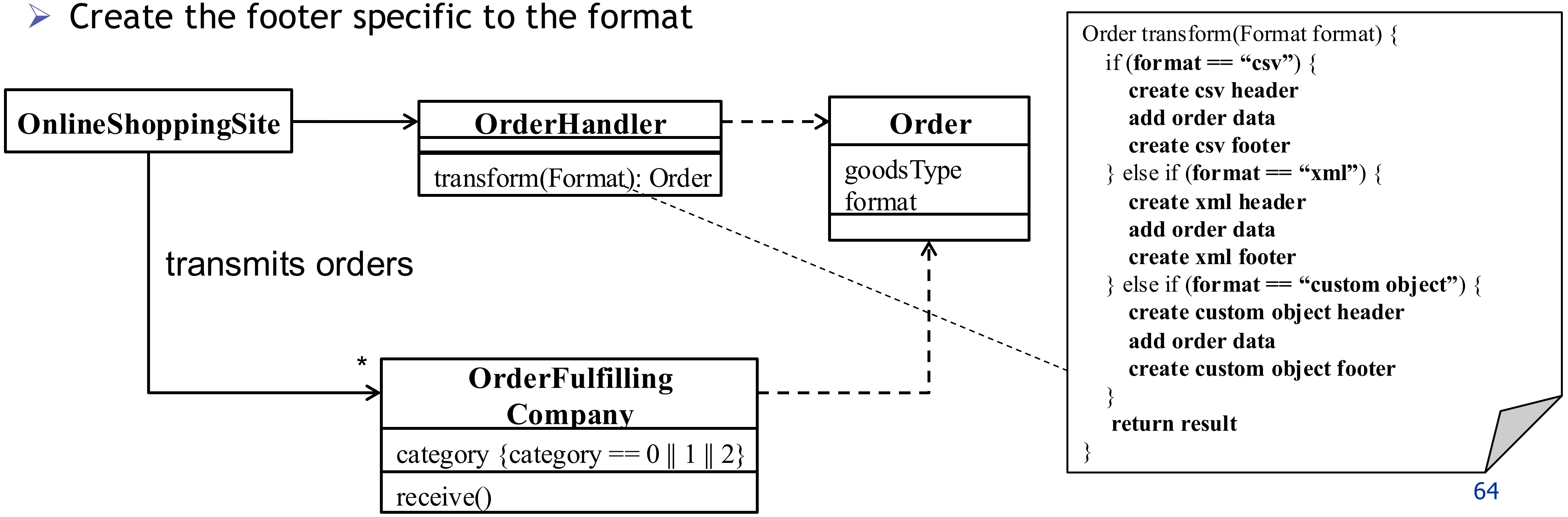
Requirements Statement$_2$

- 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.



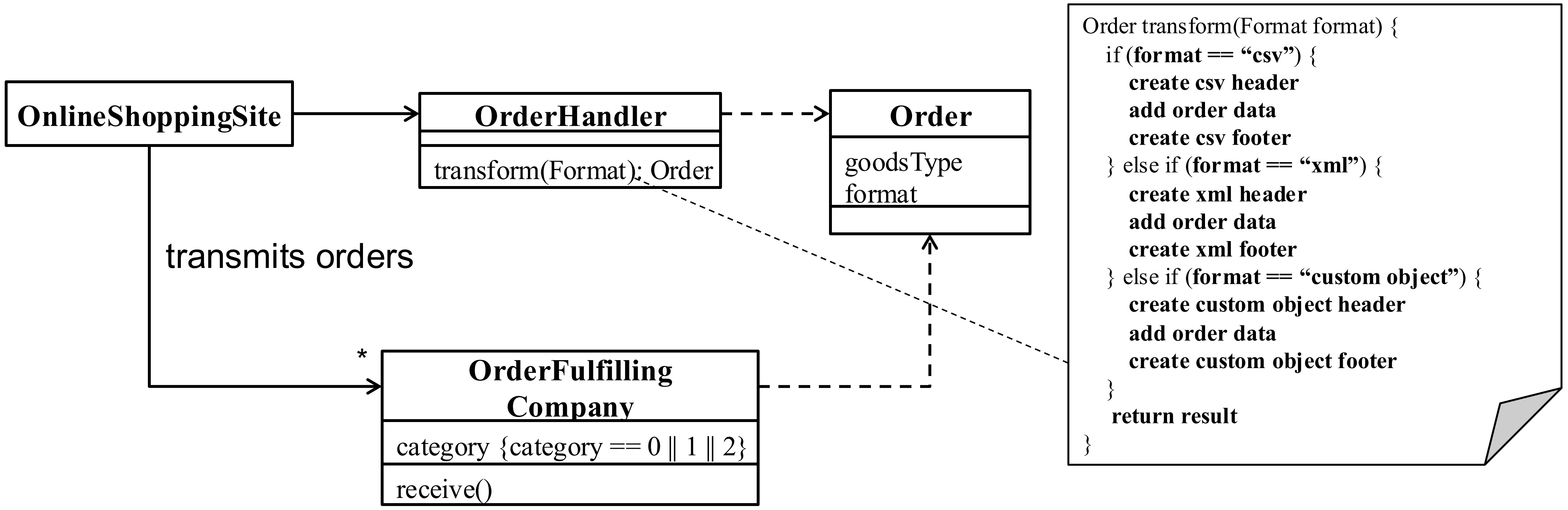
Requirements Statement$_3$

- 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



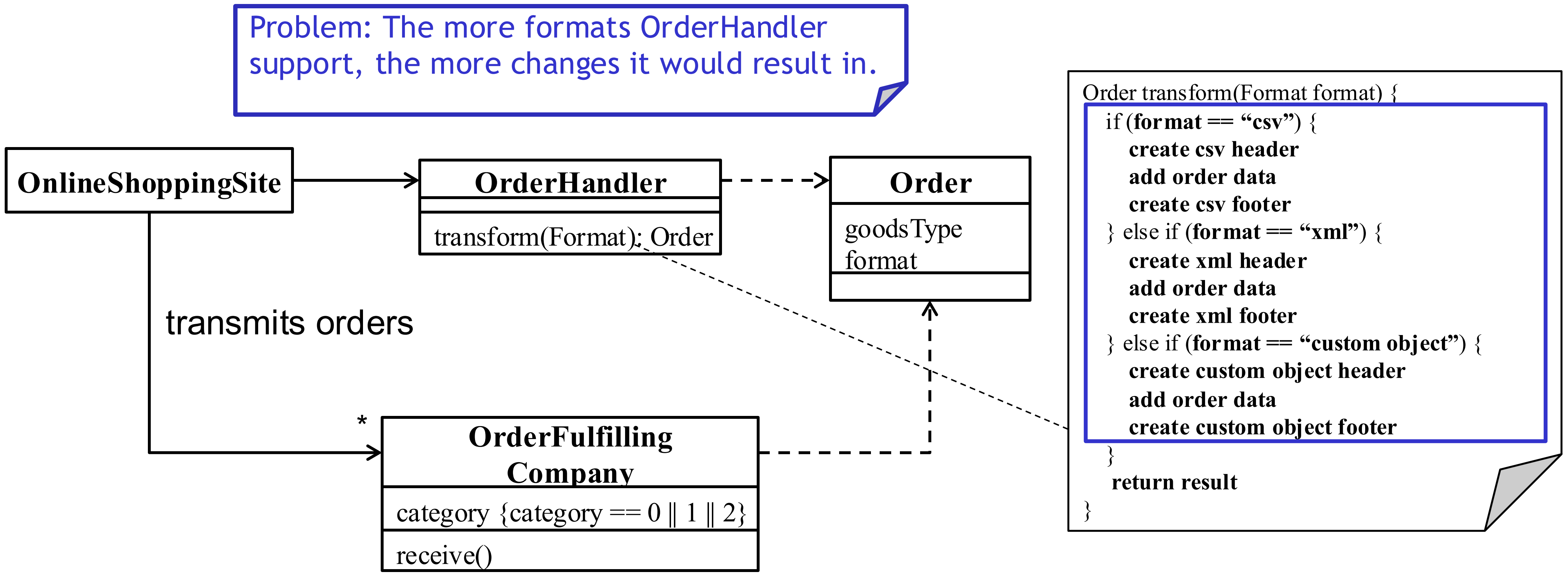
```
Order transform(Format format) {
  if (format == "csv") {
    create csv header
    add order data
    create csv footer
  } else if (format == "xml") {
    create xml header
    add order data
    create xml footer
  } else if (format == "custom object") {
    create custom object header
    add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

Initial Design



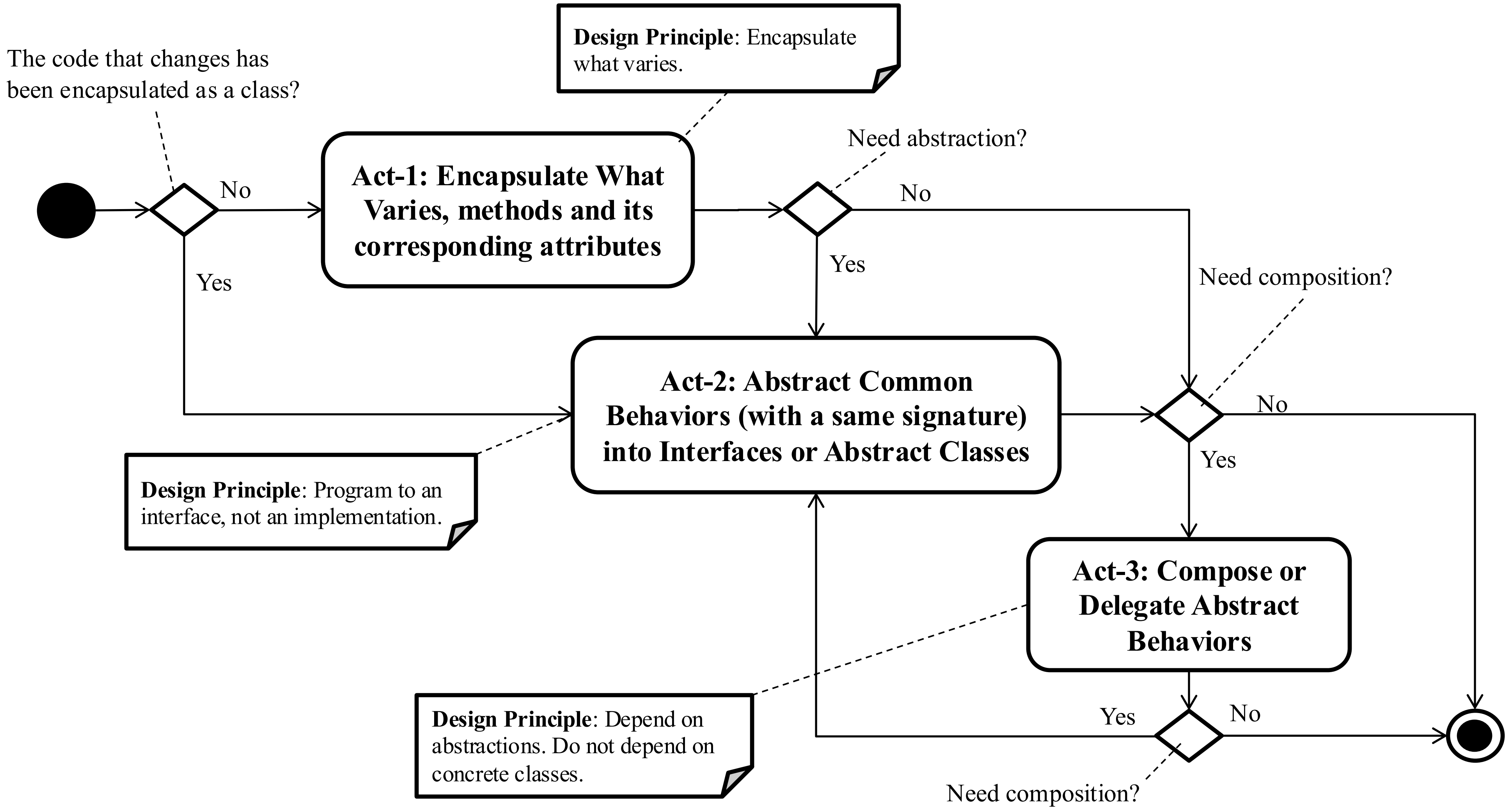
```
Order transform(Format format) {
   if (format == “csv”) {
      create csv header
      add order data
      create csv footer
   } else if (format == “xml”) {
      create xml header
      add order data
      create xml footer
   } else if (format == “custom object”) {
      create custom object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

Problem with Initial Design



```
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
}
```

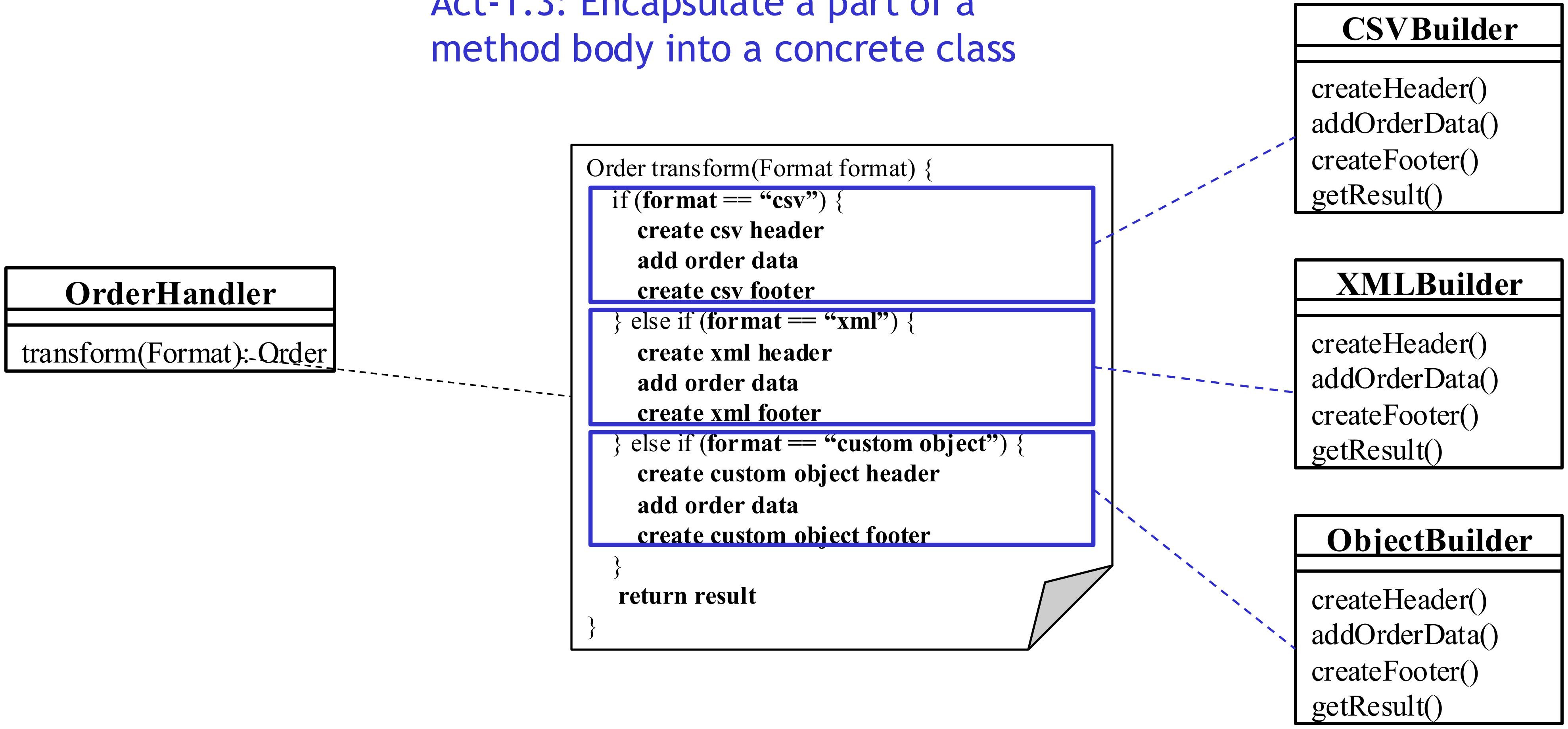
Software Design Process



The code that changes has been encapsulated as a class?
Design Principle: Encapsulate what varies.
No
Act-1: Encapsulate What Varies, methods and its corresponding attributes
Need abstraction?
No
Yes
Yes
Need composition?
Act-2: Abstract Common Behaviors (with a same signature) into Interfaces or Abstract Classes
No
Yes
Design Principle: Program to an interface, not an implementation.
Act-3: Compose or Delegate Abstract Behaviors
Design Principle: Depend on abstractions. Do not depend on concrete classes.
Yes
No
Need composition?

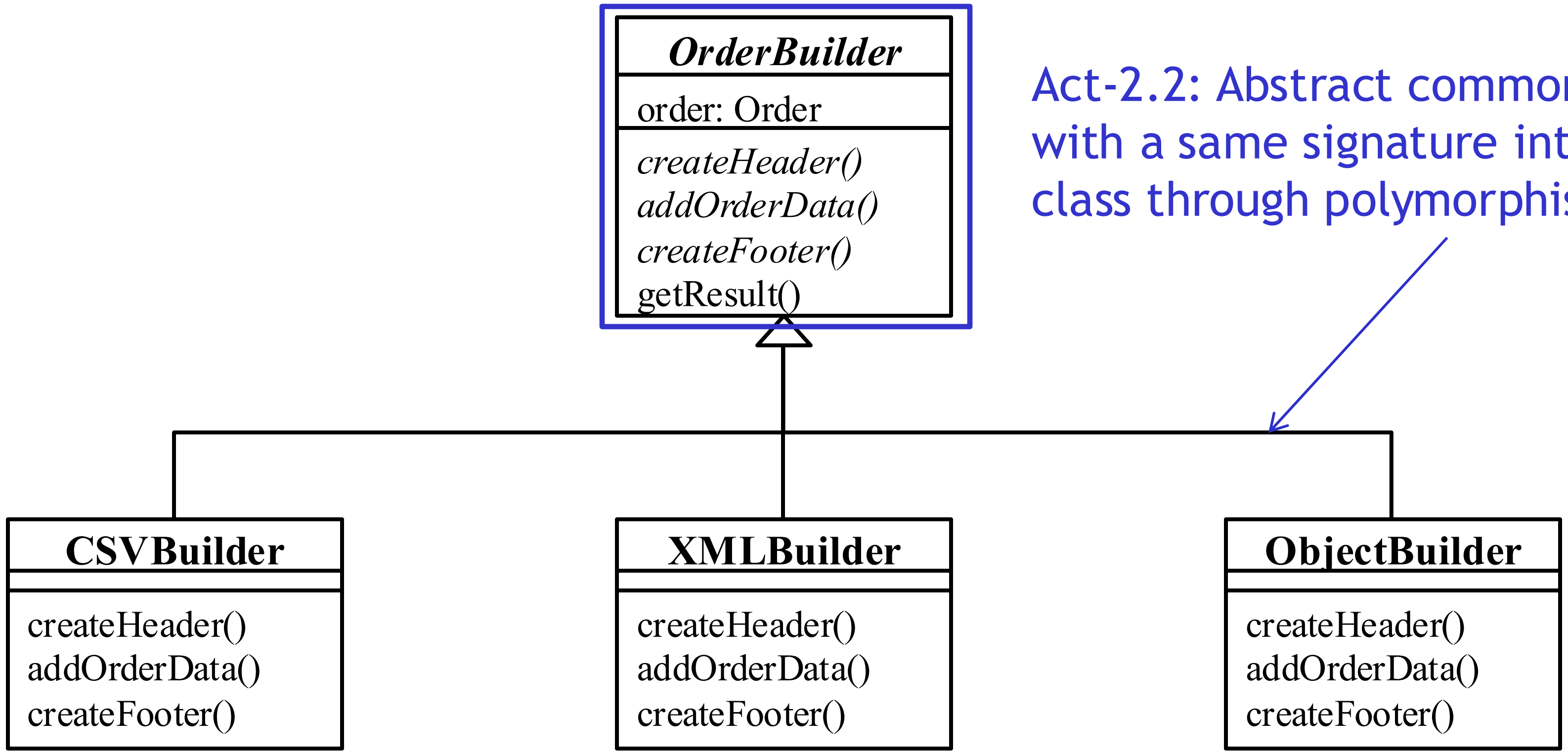
Act-1: Encapsulate What Varies

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



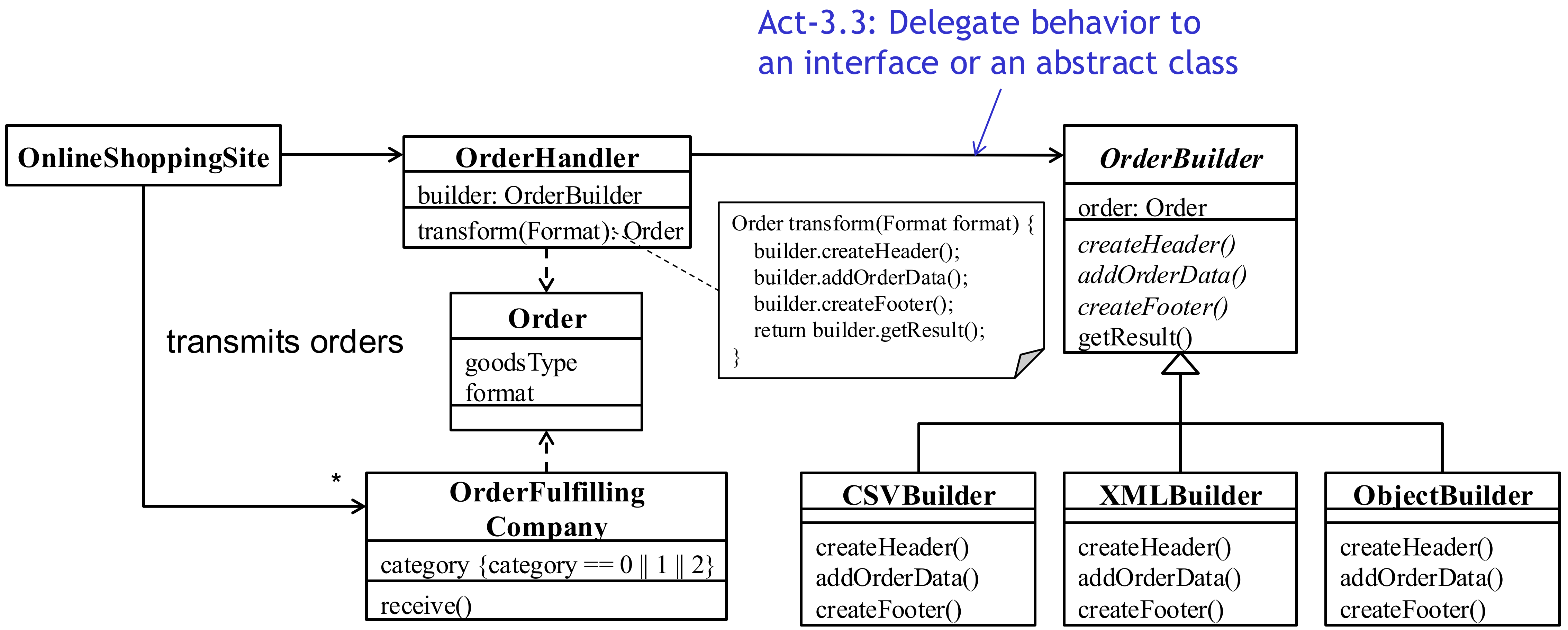
```
Order transform(Format format) {
  if (format == "csv") {
     create csv header
     add order data
     create csv footer
  } else if (format == "xml") {
     create xml header
     add order data
     create xml footer
  } else if (format == "custom object") {
     create custom object header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

Act-2: Abstract Common Behaviors



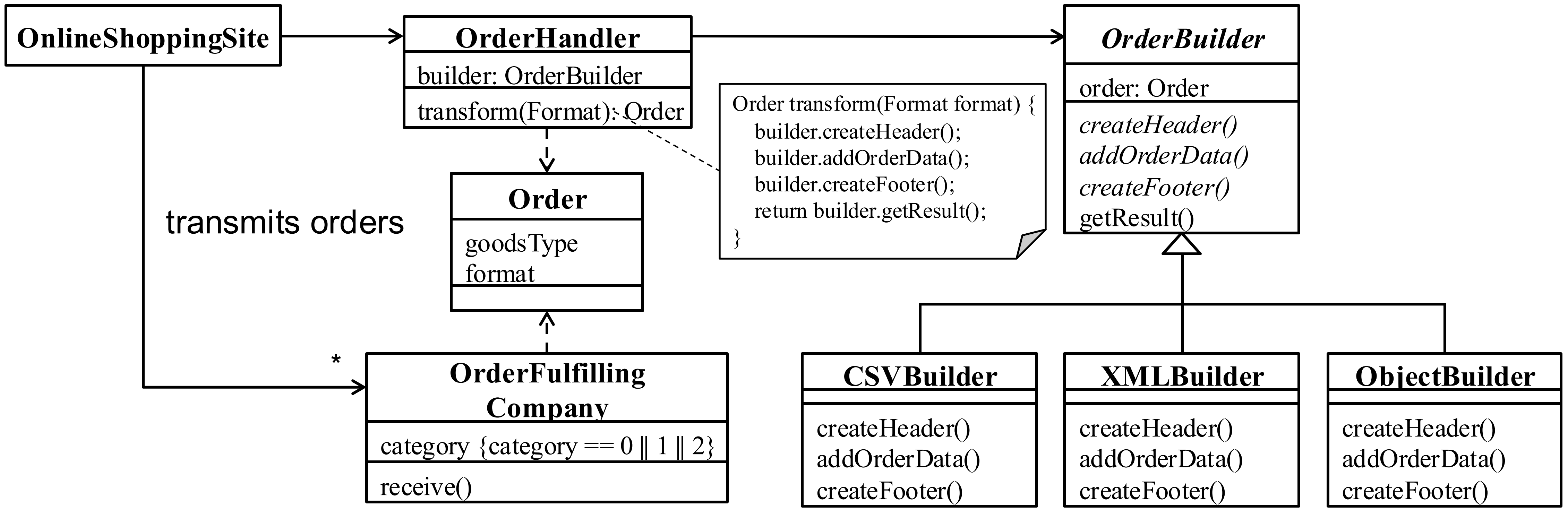
Act-2.2: Abstract common behaviors with a same signature into abstract class through polymorphism

Act-3: Compose Abstract Behaviors



```
Order transform(Format format) {
   builder.createHeader();
   builder.addOrderData();
   builder.createFooter();
   return builder.getResult();
}
```

Refactored Design after Design Process



```
Order transform(Format format) {
    builder.createHeader();
    builder.addOrderData();
    builder.createFooter();
    return builder.getResult();
}
```

A Sales Reporting Application

Prof. Jonathan Lee (李允中)

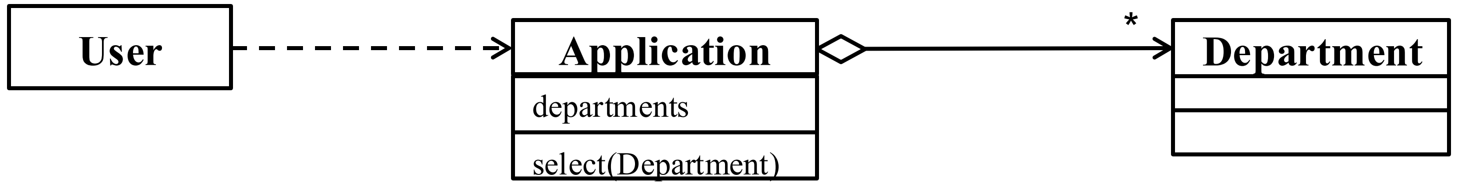
Department of Computer Science and Information Engineering
National Taiwan University

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.

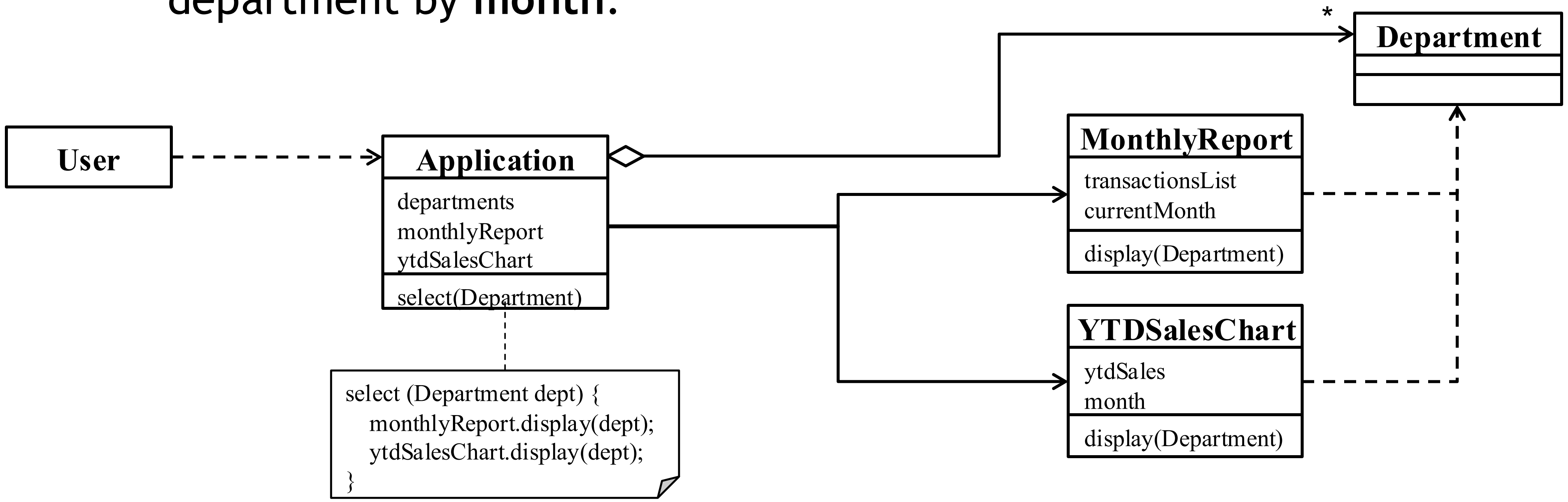
Requirements Statement$_1$

- 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$_2$

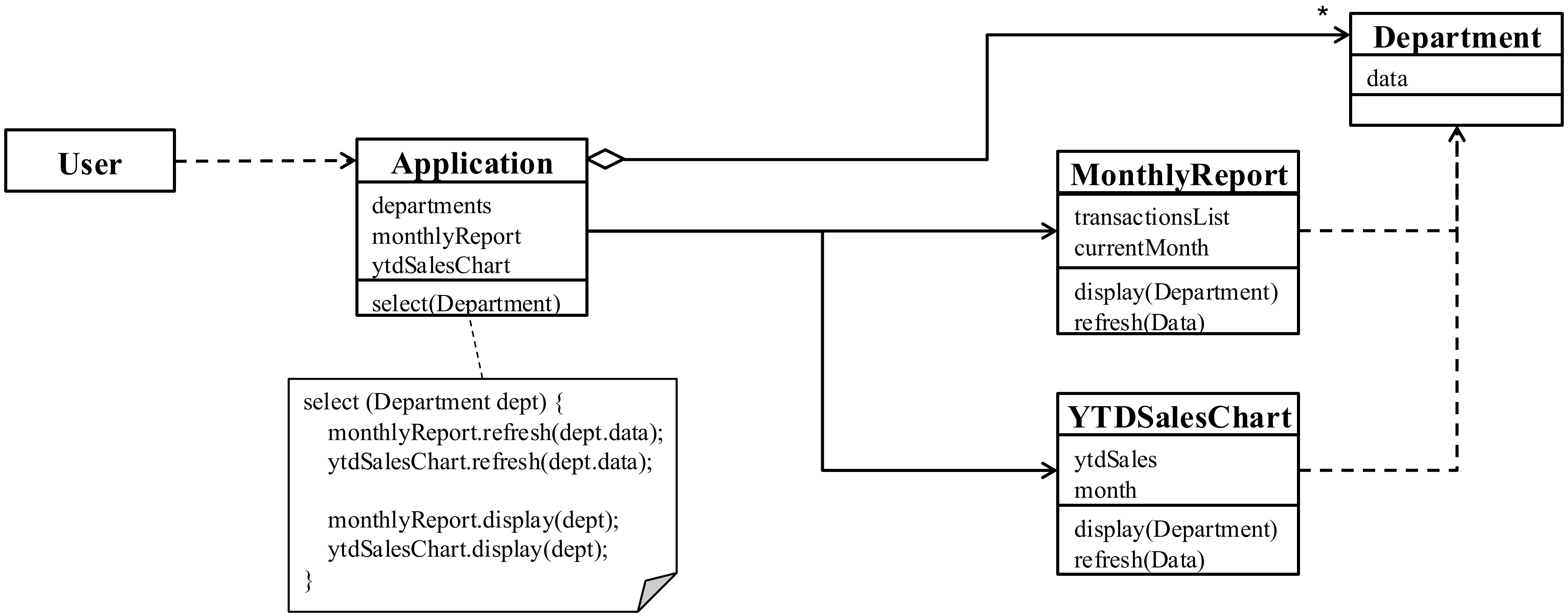
- Upon selecting a department, **two types of reports** are to be **displayed:**
  - Monthly report - A **list of all transactions** for the **current month** for the selected department.
  - YTD sales chart - A chart showing the **year-to-date sales** for the selected department by **month.**



```
select (Department dept) {
    monthlyReport.display(dept);
    ytdSalesChart.display(dept);
}
```

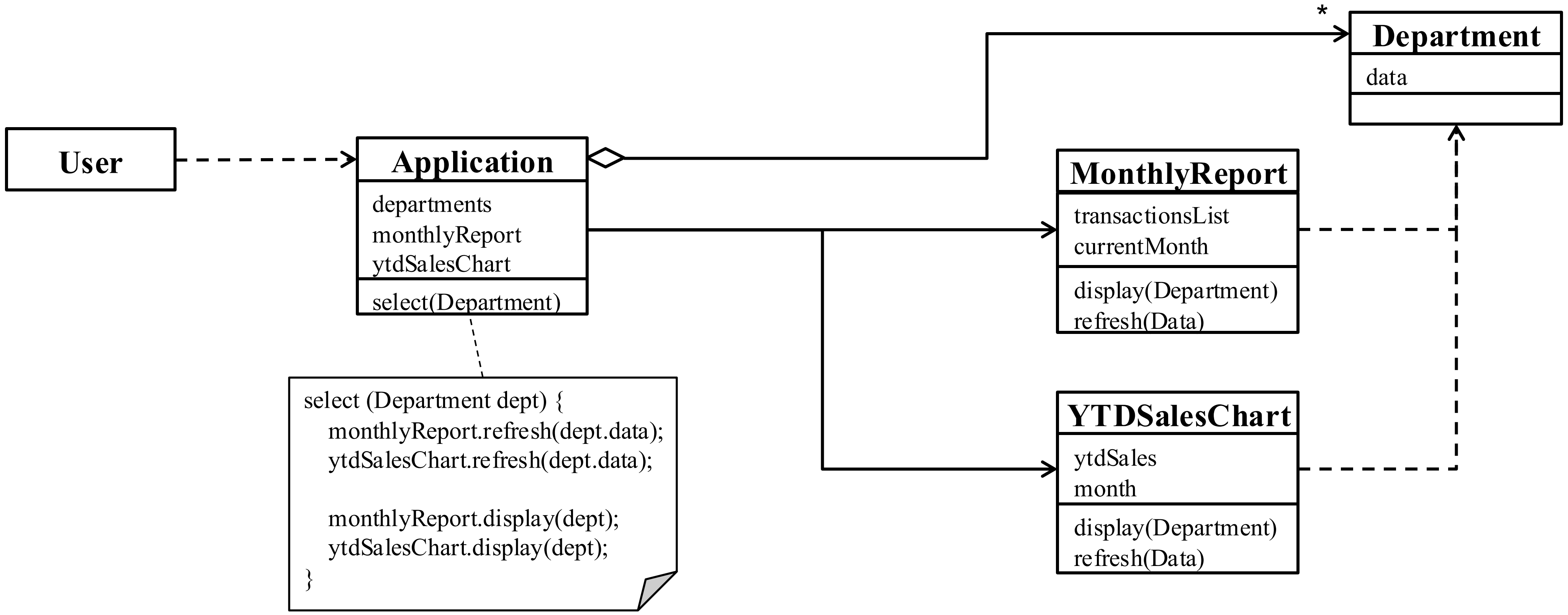
Requirements Statement$_3$

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



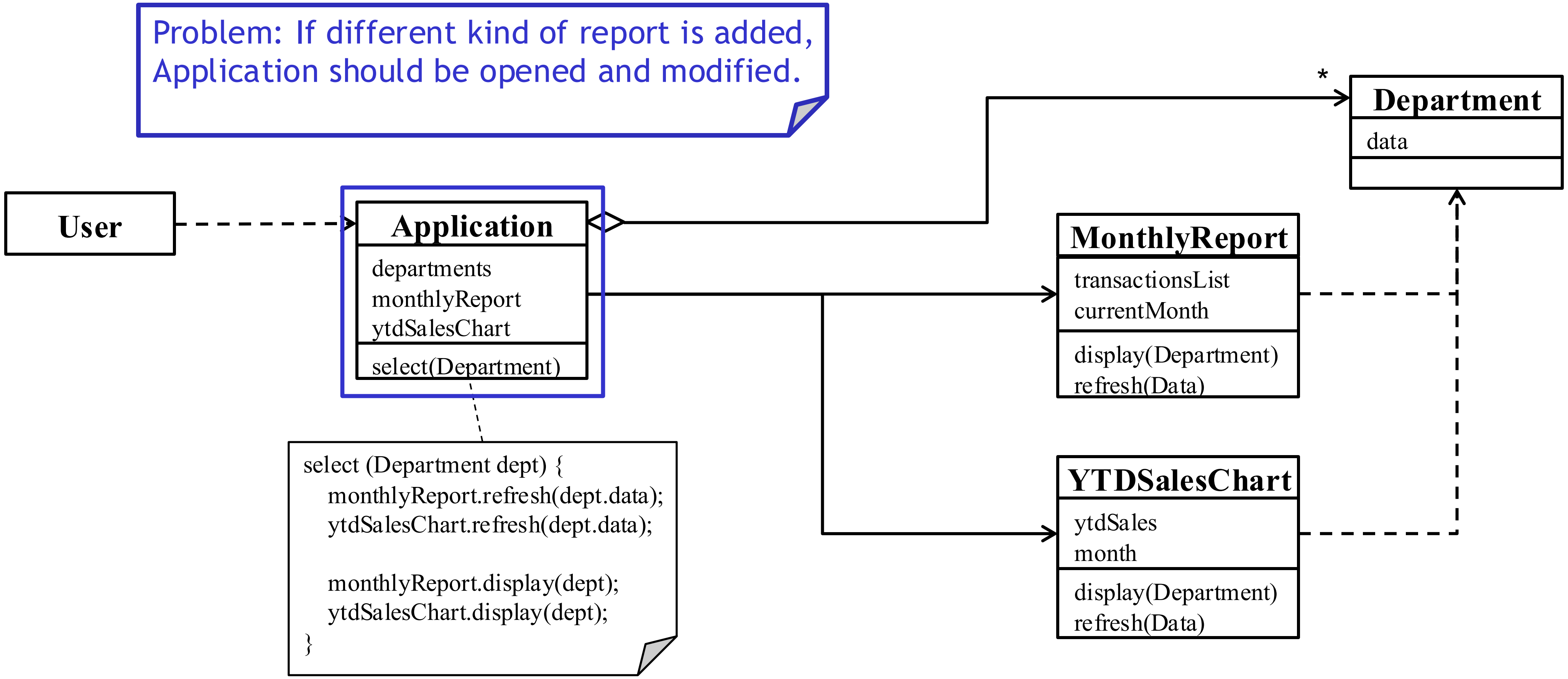
```
select (Department dept) {
    monthlyReport.refresh(dept.data);
    ytdSalesChart.refresh(dept.data);

    monthlyReport.display(dept);
    ytdSalesChart.display(dept);
}
```

Initial Design



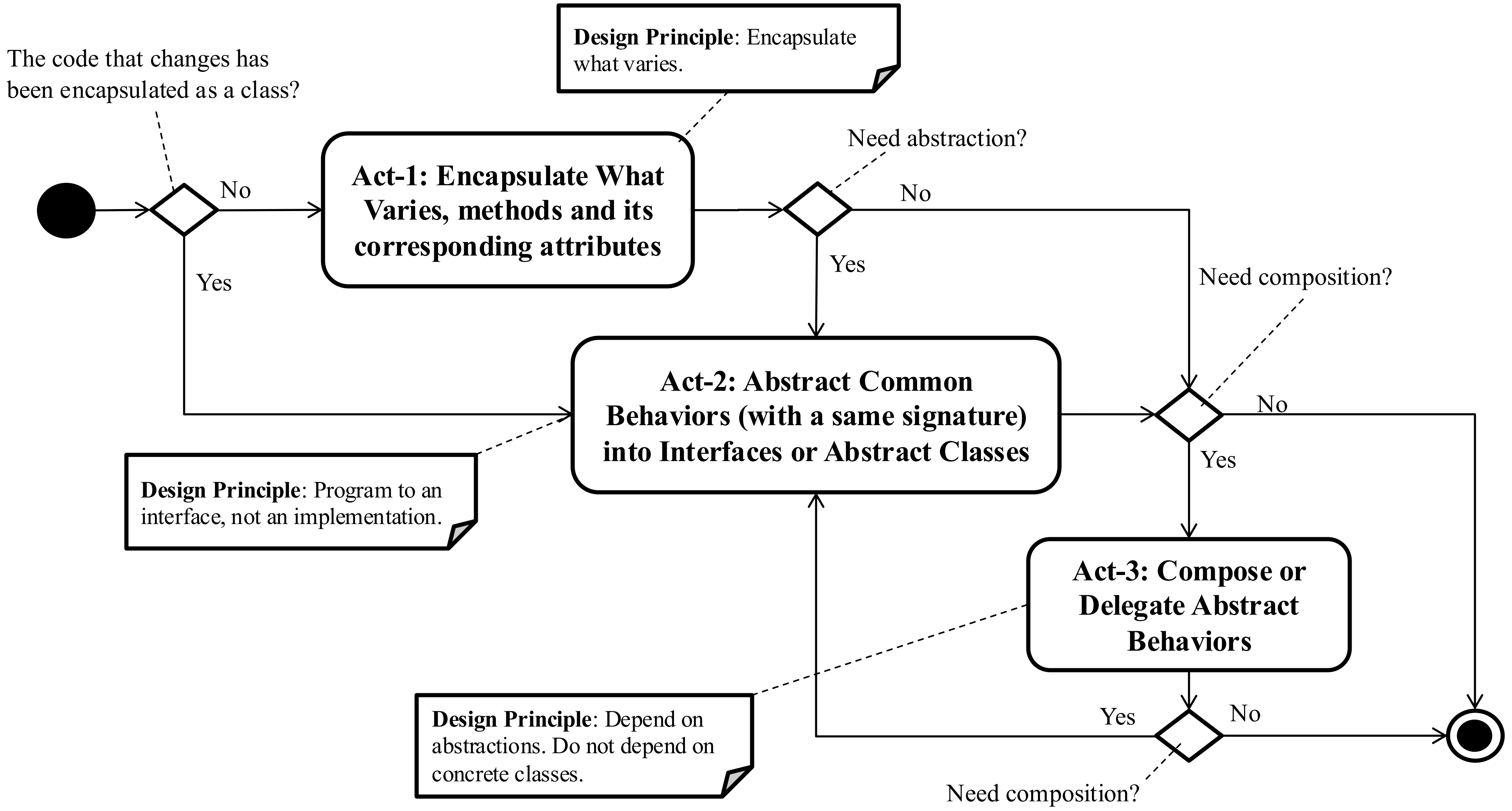
```
select (Department dept) {
   monthlyReport.refresh(dept.data);
   ytdSalesChart.refresh(dept.data);

   monthlyReport.display(dept);
   ytdSalesChart.display(dept);
}
```

Problems with Initial Design



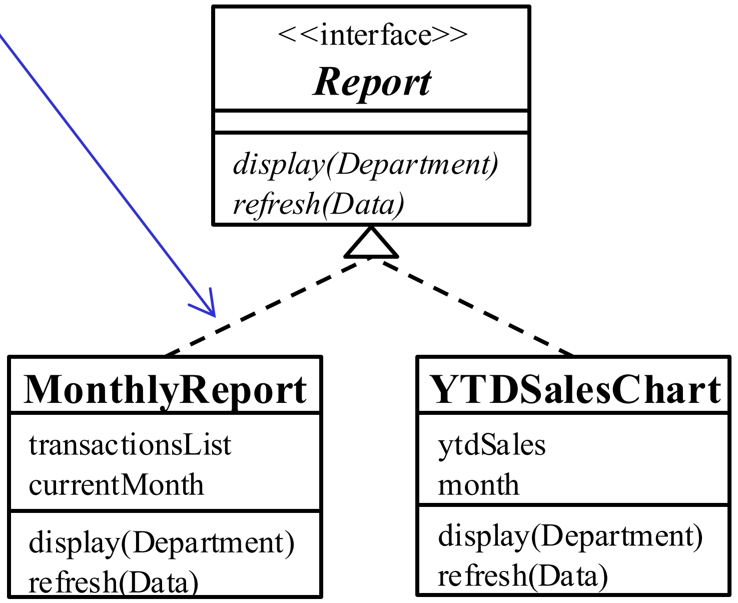
```
select (Department dept) {
    monthlyReport.refresh(dept.data);
    ytdSalesChart.refresh(dept.data);

    monthlyReport.display(dept);
    ytdSalesChart.display(dept);
}
```

Software Design Process


The code that changes has been encapsulated as a class?
Design Principle: Encapsulate what varies.
No
Yes
Act-1: Encapsulate What Varies, methods and its corresponding attributes
Need abstraction?
No
Yes
Need composition?
Act-2: Abstract Common Behaviors (with a same signature) into Interfaces or Abstract Classes
No
Yes
Design Principle: Program to an interface, not an implementation.
Act-3: Compose or Delegate Abstract Behaviors
Design Principle: Depend on abstractions. Do not depend on concrete classes.
Yes
No
Need composition?

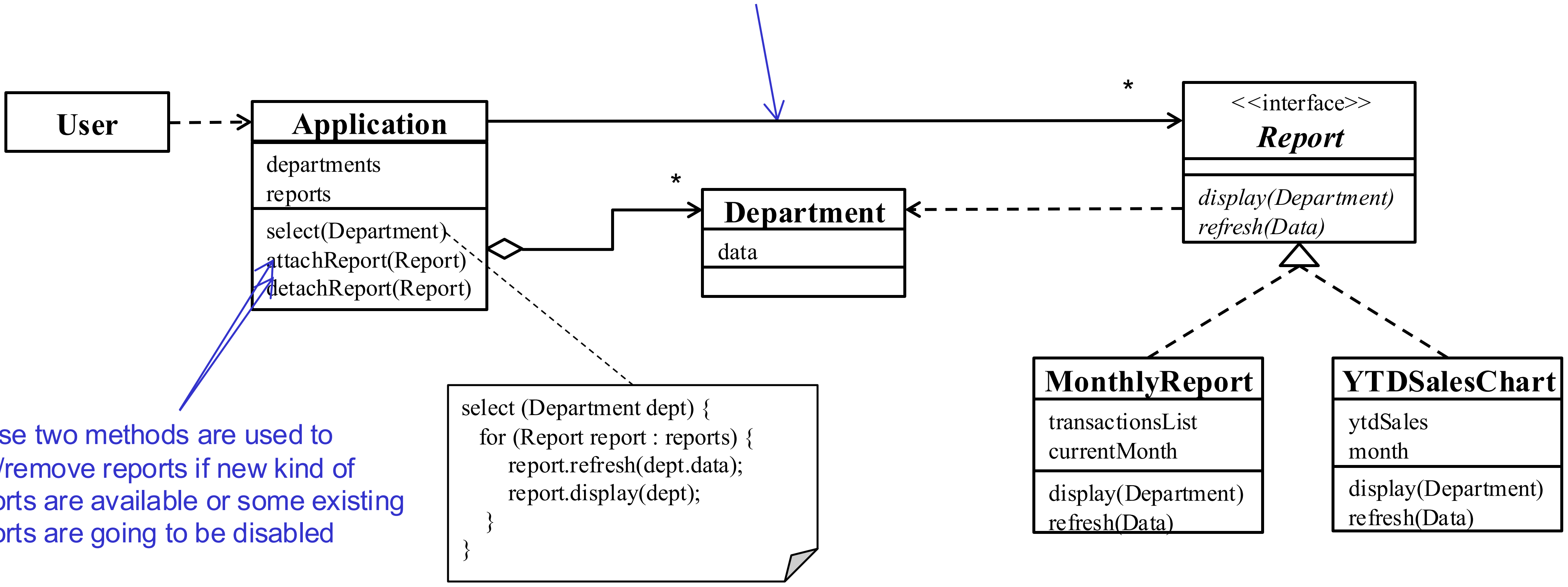
Act-2: Abstract Common Behaviors

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



Act-3: Compose Abstract Behaviors

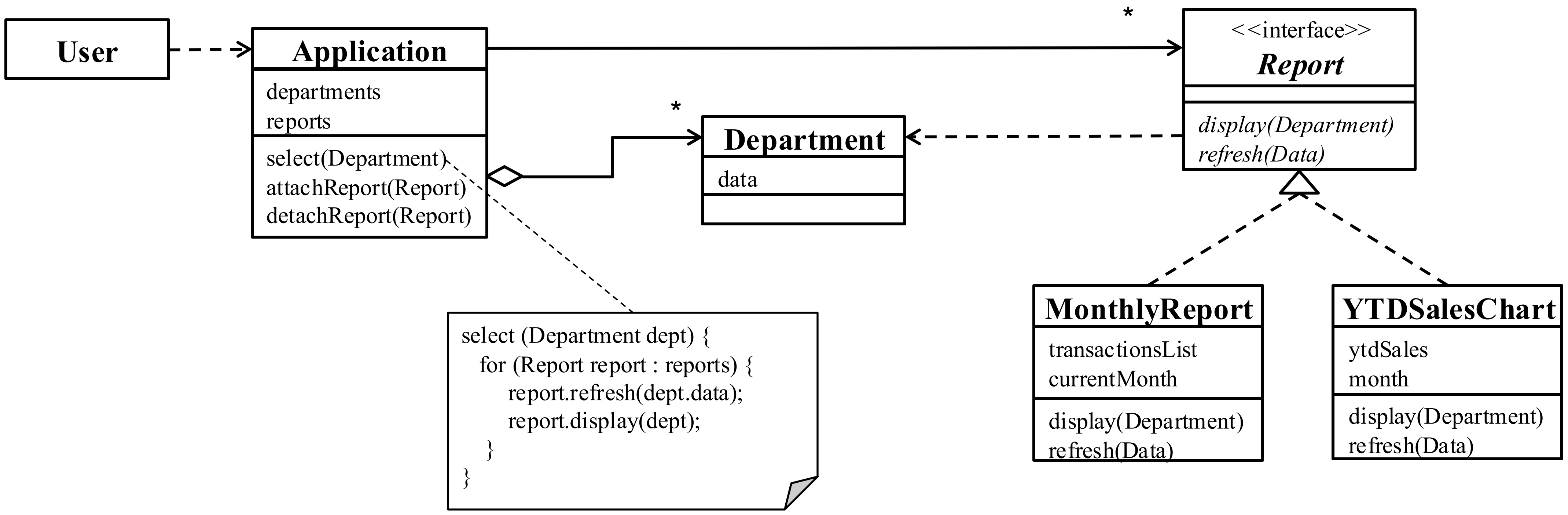
Act-3.1: Compose behaviors of an interface or an abstract class



```
select (Department dept) {
  for (Report report : reports) {
      report.refresh(dept.data);
      report.display(dept);
   }
}
```

These two methods are used to add/remove reports if new kind of reports are available or some existing reports are going to be disabled

Redesign after Refactoring



```
select (Department dept) {
   for (Report report : reports) {
        report.refresh(dept.data);
        report.display(dept);
    }
}
```

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
- Facade: 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 or a family of algorithms
- Template Method: steps of an algorithm
- Visitor: operations that can be applied to objects without changing their classes