Polymorphism and Object-Oriented Design

Hui Chen [a]

[a]CUNY Brooklyn College, Brooklyn, NY, USA

April 20, 2023

Outline

Outline

Software Design

- Design starts mostly from/with requirements – evolving mostly from functionalities and other non-functional characteristics
  - In the waterfall model Design generally occurs after Requirements
  - In agile, design is performed during in each iteration
- To answer: How is the software solution going to be structured?
  - What are the main components – (functional composition) often directly from requirements' functionalities (e.g., use cases, user stories, scenarios)
  - How are these components related? – Possibly re-organize the components (composition/decomposition)
- Two main levels of design:
  - Architectural (high level) design
  - Detailed design
  - Different design concerns at different abstraction levels (e.g. classes vs. modules vs. entire system)
- How should we depict design – what notation/language?

Review: High-level and Low-level Designs

Architectural design (high-level design) patterns and styles

- MVC, Layered, Pipeline, Client-Server, SOA, . . .

Detailed design (low-level design)

- Functional decomposition, database design, Object-Oriented design, user-interface design, . . .
- Object-Oriented Design and UML – focused on modeling
- To discuss more about Object-Oriented design

Outline

Object-Oriented Design

- Design principles, mechanisms, and techniques
  - Encapsulation, information hiding, abstraction, immutability, interface, ...
- Design patterns
  - Visitor, Observer, Strategy, ...

In this lesson, we shall discuss several concepts about Object-Oriented design principles, mechanisms, and techniques

- Encapsulation, interface, and polymorphism

Use Martin Robillard's "Software Design" and his course materials as the main source

Outline

Encapsulation

Discussed it in the context of SOA, now as a concept in Object-Oriented Design

- encapsulate both data and computation to protect them from corruption, and to simplify the design

Principle of Information Hiding

- "The principle generally states that you only show a client that part of the total information that is really necessary for the client's task and you hide all remaining information."

Design Class – Encapsulation

Design is about making decisions – what decisions make when we design a class?

Class Encapsulation Guidelines

1. Make all fields private (almost all the time) – unless you have a strong argument to make a field non-private
2. Do not automatically supply a class with a “getter” and “setter” for every field
3. Make your classes immutable whenever possible (meaning of immutable? how?)
   - Try to avoid defining methods that both change (“mutate”) the state of an object and return ("access") a value
   - Define your instance variables as final whenever possible
4. Ensure your accessor methods do not return a reference to a mutable instance variable

Source:
Robillard – Module 01

Let's examine the design of the Deck class

How does it violate the *Encapsulation* class design guidelines?

```java
public class Deck
{
    public Stack<Card> aCards = new Stack<>();

    public Stack<Card> getCards()
    { return aCards; }

    public void setStack(Stack<Card> pCards)
    { aCards = pCards; }

    public void applyAll( List<Stack<Card>> pTaskList )
    { pTaskList.add(aCards); }
}
```

Let's examine the design of the Deck class

How does it violate the *Encapsulation* class design guidelines?

```
public class Deck
{
    // violates 1. public -> no door to guide the data field
    public Stack<Card> aCards = new Stack<>();

    // violates 4. return reference to a class variable -> font door is open
    public Stack<Card> getCards()
    { return aCards; }

    /* violates 2 and 3. set a reference to a class variable; but caller
       keeps a reference -- back door open because caller has a reference
       to containing object */
    public void setStack(Stack<Card> pCards)
    { aCards = pCards; }

    /* violates 3. set a reference to a class variable; but caller
       keeps a reference -- back door open because caller has a reference
       to containing object */
    public void applyAll( List<Stack<Card>> pTaskList )
    { pTaskList.add(aCards); }
}
```

Let's redesign the Deck class

Refactor: Improving the design of code without changing its functionality

Let's redesign the application

Perhaps, the Deck class was ill-conceptualized ...

Outline

Java Interface

- In Java, *interfaces* provide a specification of the methods that it should be possible to invoke on the objects of a class
- For instance the interface Icon specifies three method signatures and documents their expected behavior
- What problem does it help solve?

```
interface Icon {
  public int getIconWidth();
  public int getIconHeigth();
  public void painIcon();
  }

public class ImageIcon implements Icon { // ... }
```

Java Interface

What problem does it help solve? Consider the following

```
class Game {
   Icon aIcon = ...;

   public void showIcon() {
      if(aIcon.getIconWidth() > 0 && aIcon.getIconHeight() > 0
        ) {
         aIcon.paintIcon(...);
      }
   }
   ...
```

- In practice, Icon can be in different formats and even computed on-the-fly. How can we represent that in an Object-Oriented language like Java?
- Use Java interface
- Can we also solve it using subclass?

Polymorphism

- In plain language, polymorphism is the ability to have different shapes
  - In the context of the Icon example, it is the ability of the abstractly specified Icon to have different implementations
- Polymorphism as supported by Java interfaces supports two very useful quality features in software design:
  - Loose coupling, because the code using a set of methods is not tied to a specific implementation of these methods
  - Extensibility, because we can easily add new implementations of an interface (new "shapes" in the polymorphic relation)

The Interface Segregation Principle (ISP)

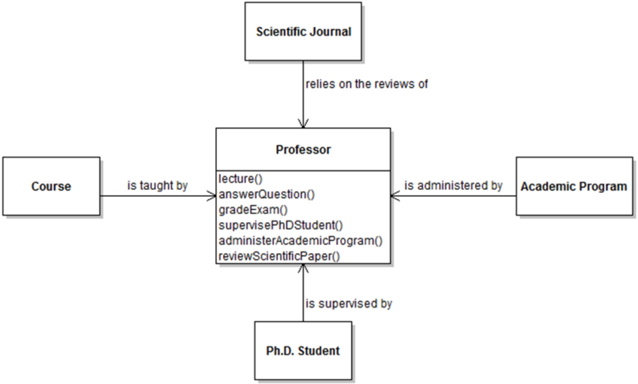
When designing multiple classes, we ought to consider how these classes interact.

- terminology: client and server classes/objects – the client class/object invokes the server class/object's method

The ISP: clients should not be forced to depend on interfaces they do not need.

Violation of ISP: Example

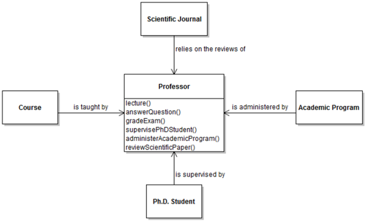
How do the following design violate the ISP?



Source:
Robillard – Module 02

Violation of ISP: Example

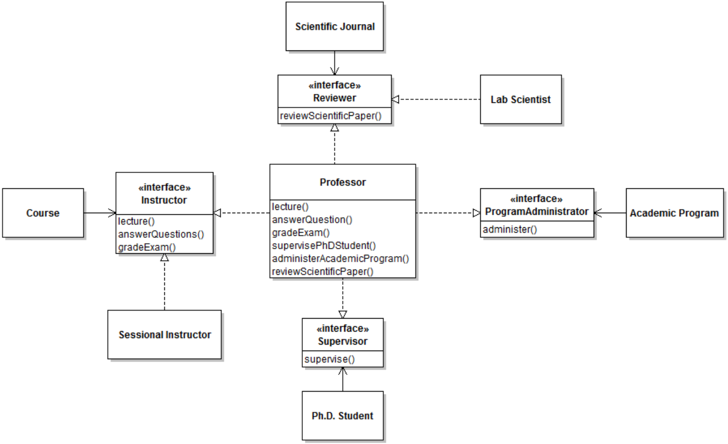
How do the following design violate the ISP?



- Clients depend on services they do not need, e.g., Course depends on a class that supplies a service reviewScientificPaper.
- With this design it is not possible to have any object besides an instance of Professor provide the lecture functionality.

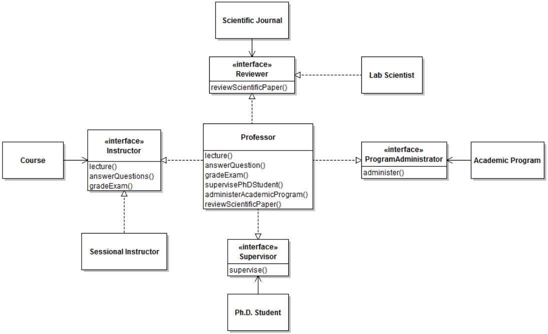
Source:
Robillard – Module 02

Improved Design



Improved Design

With what mechanism is the design improved?



- decoupling behavior from implementation
- clients depend on *interfaces* that represent specific roles directly relevant to each client.
- benefits: louse coupling and extensibility

Outline

"Introduction to Software Design with Java" by Martin P. Robillard
"Engineering Software as a Service" by Armando Fox and David Patterson (2nd Edition)
"Essentials of Software Engineering" by Frank Tsui, Orlando Karam, and Barbara Bernal(4th Edition)