Detailed Design & Design Principles

Tarek Alakmeh

Thomas Fritz

Many thanks to Reid Holmes and Elisa Baniassad

Agenda

1. Modular Design - Coupling & Cohesion
2. Design Principles – SOLID
3. Kahoot, Exercise & Quiz

Note:

- *Next lecture also Design Pattern recap*

Examinable skills

By the end of this lecture, you should be able to…

- Describe coupling and cohesion
- Describe the SOLID design principles
- Determine and fix design principle violations

Modular Design
Coupling & Cohesion

Learning Objectives

Be able to:

- Describe what coupling and cohesion is
- distinguish between better and worse forms of coupling/cohesion

Design

What is good modular design?

What are the benefits of a good design?

How do you achieve it?

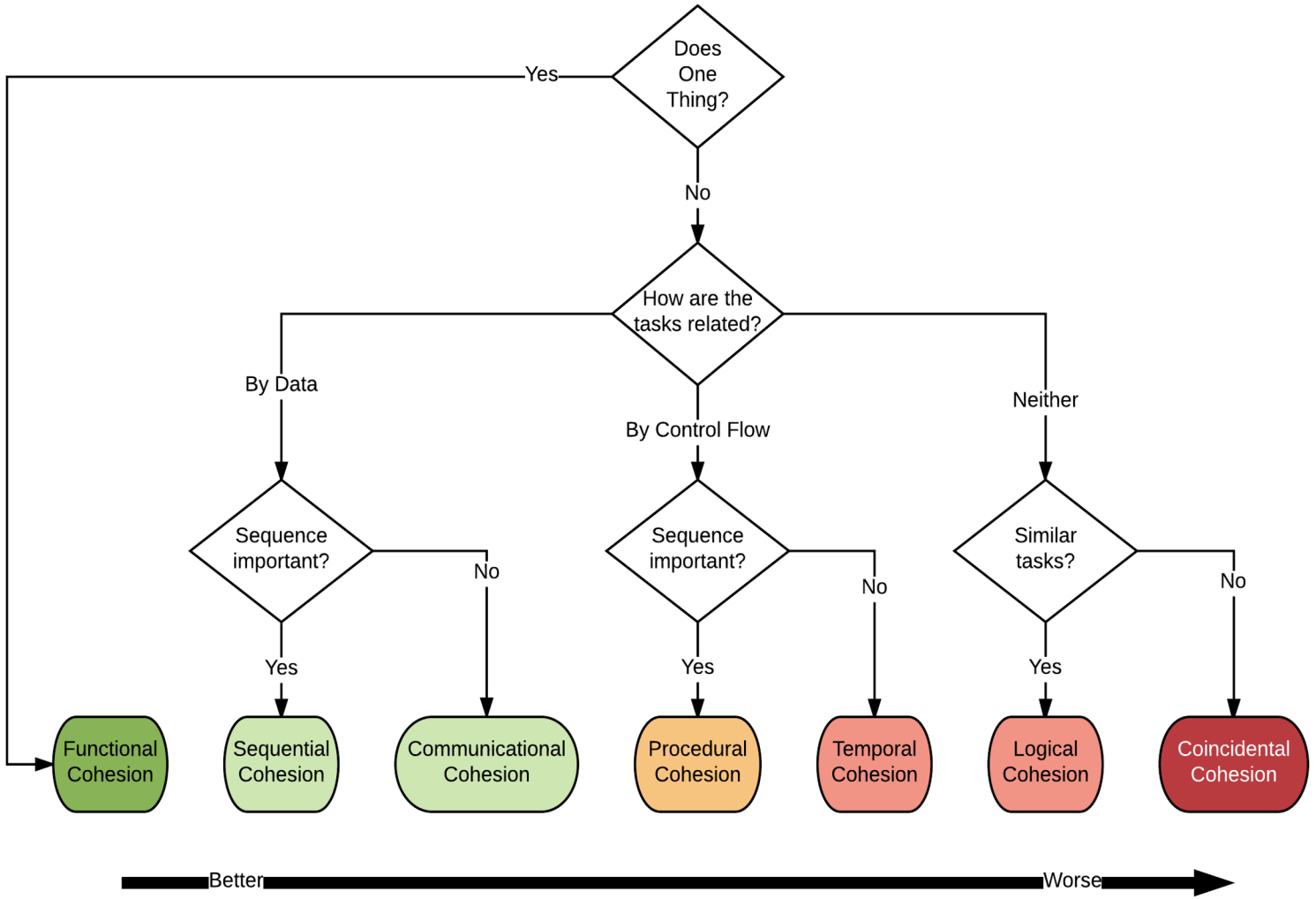
Modular design

“Interdependence within and independence across modules”

- High cohesion
  - Functions in a module should be closely related
- Low coupling
  - Modules should depend on as few modules as possible
- Encapsulation & information hiding
  - Restrict access to some of a module’s components: hide internal details / information

Cohesion

Many different kinds: best (green) to worst (red)



Cohesion Details

Functional cohesion (best) – Module does a single well-defined task

Sequential cohesion (very good) – output from one part is the input to next part (e.g. a function which reads data from a file and processes the data).

Communicational cohesion – operate on the same data (e.g. a module which operates on the same record of information)

Procedural cohesion – always follow a certain sequence of execution (e.g. a protocol).

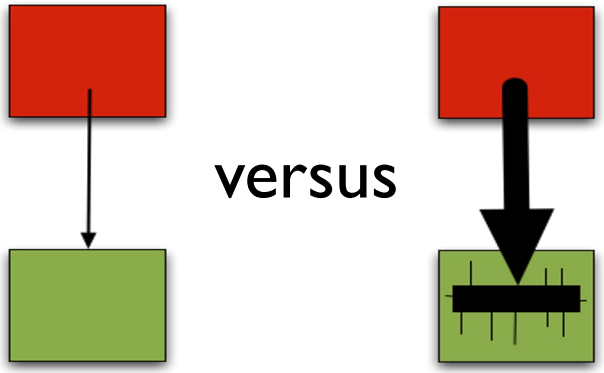
Temporal cohesion – processed at a particular time in program execution, but not the same thing (e.g. a function which is called after catching an exception which closes open files, creates an error log, and notifies the user).

Logical cohesion (bad) – logically related but otherwise different.

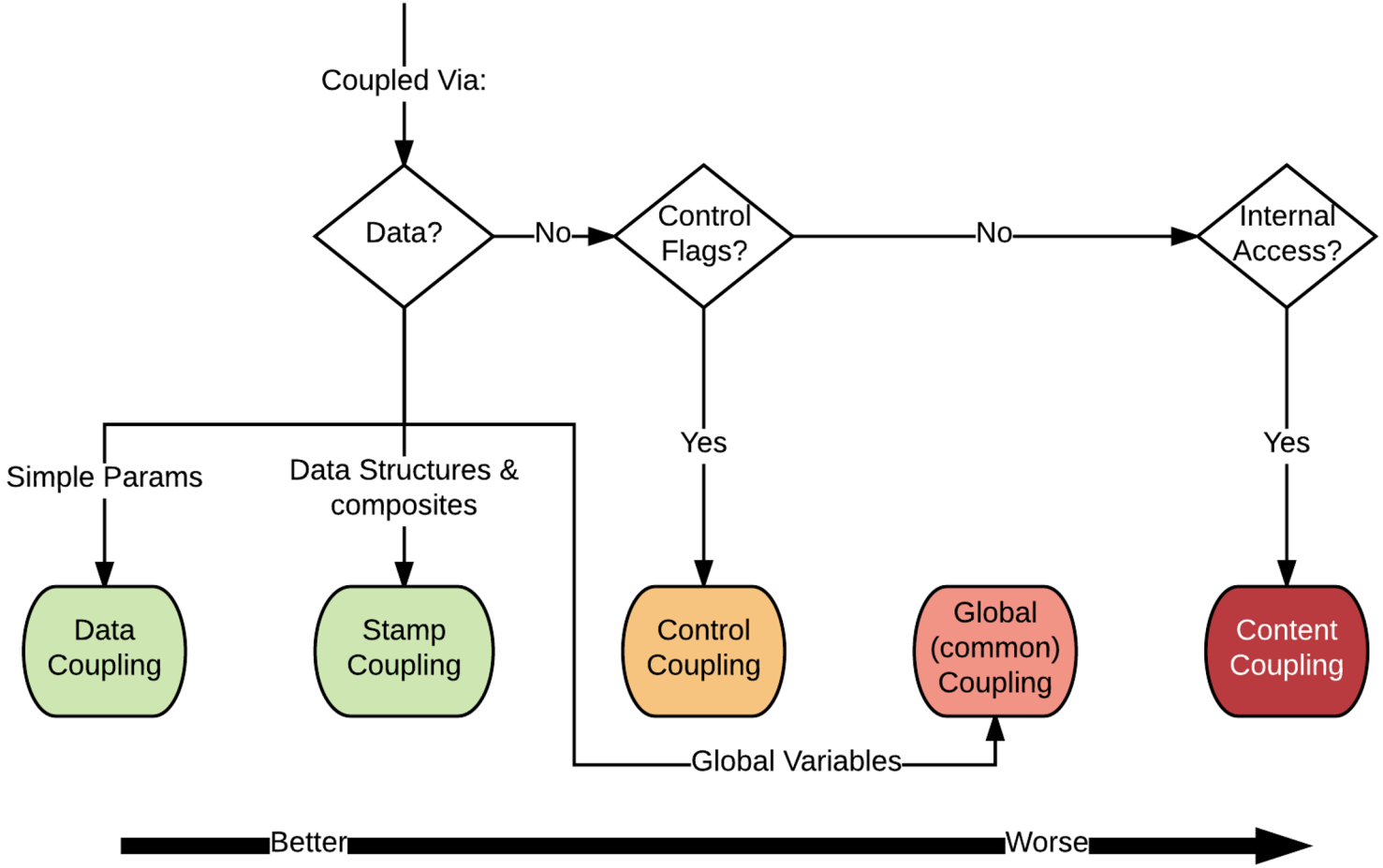
Coincidental cohesion (bad) – grouped arbitrarily; the only relationship between the parts is that they have been grouped together (e.g. a "Utilities" class).

More info: https://dzone.com/articles/cohesion-the-cornerstone-of-software-design
https://mrpicky.dev/tag/temporal-cohesion/

Coupling



Many different kinds: best (green) to worst (red)



Coupling

Data coupling – interacting through simple data types (e.g. parameters)

Stamp coupling – interacting through data structures or composite objects and only using parts of it (e.g. passing a whole record to a function that only needs one field of it)

Control coupling – modifying execution by sending control flags (e.g. passing a what-to-do flag)

Global/Common coupling – global variables (changing the shared resource implies changing all modules using it)

Content coupling – internal modification of other classes (e.g accessing local data of another module)

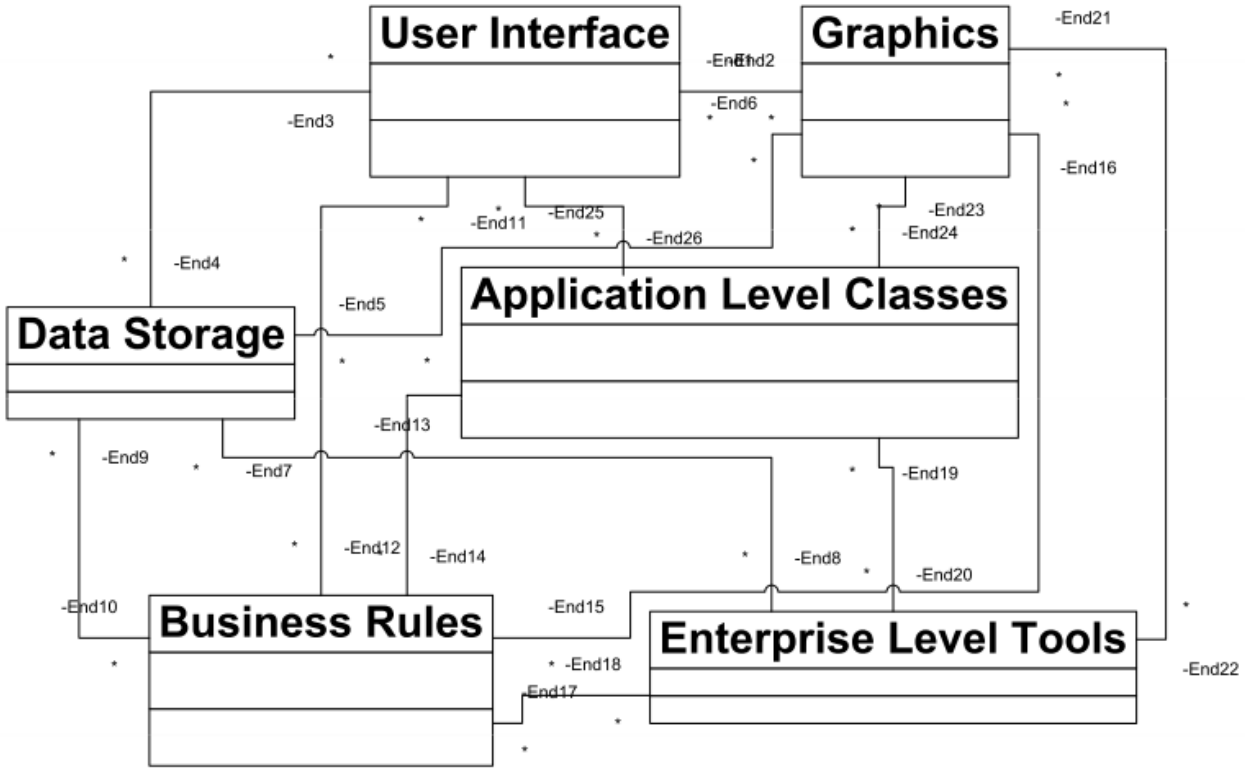
Can we avoid all coupling?

Good or bad design and why?

1)

```
public class EmailMessage {
    …
    public void sendMessage() {…}
    public void setSubject(String subj) {…}
    public void setSender(Sender sender) {…}
    public void login(String user, String passw) {…}
    ….
}
```

2)



Design Principles - SOLID

Learning Objectives

Be able to:

- Describe the SOLID design principles
- Determine and fix design principle violations

SOLID Design Principles

One set of principles

- S — Single responsibility
- O — Open/closed
- L — Liskov substitution
- I — Interface segregation
- D — Dependency Inversion

Single Responsibility Principle

Classes should do one thing and do it well.

Or check:

A description that describes a class in terms of alternatives is not one class, but a set of classes.

"A ClassRoom is a location where students attend lectures **or** labs."

Single Responsibility

```
class Book {
  public getTitle() { .. }
  public getAuthor() { .. }
  public getContent(..) { .. }
}
```

Single Responsibility

```
class Book {
  var currentPage : int;
  public getTitle() { .. }
  public getAuthor() { .. }
  public getContent(..) { .. }
  getCurrentPage() { .. }
}
```

Single Responsibility

```
class Book {
  var currentPage : int;
  public getTitle() { .. }
  public getAuthor() { .. }
  public getContent(..) { .. }
  getCurrentPage() { .. }
  save() { .. } // persist current page
}
```

Single Responsibility

```
class Book {
  var currentPage : int;
  public getTitle() { .. }
  public getAuthor() { .. }
  public getContent(..) { .. }
  getCurrentPage() { .. }
  save() { .. } // persist current page
  getRelatedBooks() { .. }
}
```

Single Responsibility

```
class Book {
  var currentPage : int;
  public getTitle() { .. }
  public getAuthor() { .. }
  public getContent(..) { .. }
  getCurrentPage() { .. }
  save() { .. } // persist current page
  getRelatedBooks() { .. }
  turnPage(..) { .. }
}
```

Class Exercise: Apply the single responsibility principle!

[https://t.ly/digr7]

```
class Book {
  var currentPage : int;
  public getTitle() { .. }
  public getAuthor() { .. }
  public getContent(..) { .. }
  getCurrentPage() { .. }
  save() { .. } // persist current page
  getRelatedBooks() { .. }
  turnPage(..) { .. }
}
```

Class Exercise: Apply the single responsibility principle!

```
class Book {
  public getTitle() { .. }
  public getAuthor() { .. }
  public getContent(..) { .. }
}

class BookReader {
  var currentPage: int;
  public setBook(book: Book) { .. }
  getCurrentPage() { .. }
  turnPage(..) { .. }
  goToPage(..) { .. }
}

class BookRecommender {
  public getRelatedBooks(book: Book) { .. } }
```

```
class BookPersistence {
  public save(state: BookReader) {
    .. }
}
```

Open/Closed Principle

Classes should be open to extension and closed to modification.

When designing classes, do not plan for brand new functionality to be added by modifying the core of the class.

Instead, design your class so that extensions can be made in a modular way, to provide new functionality by leveraging the power of the inheritance facilities of the language, or through pre-accommodated addition of methods.

Good/bad design, why and how to improve? [https://t.ly/o_szF]

```
class Drawing {
  public void drawAllShapes(List<IShape> shapes) {
     for (IShape shape : shapes) {
        if (shape instanceof Square()) {
           drawSquare((Square) shape);
        } else if (shape instanceof Circle) {
           drawCircle((Circle) shape));
  } } }

  private void drawSquare(Square square) {  …// draw the square…  }
  private void drawCircle(Circle square) {  …// draw the circle…  }
}
```

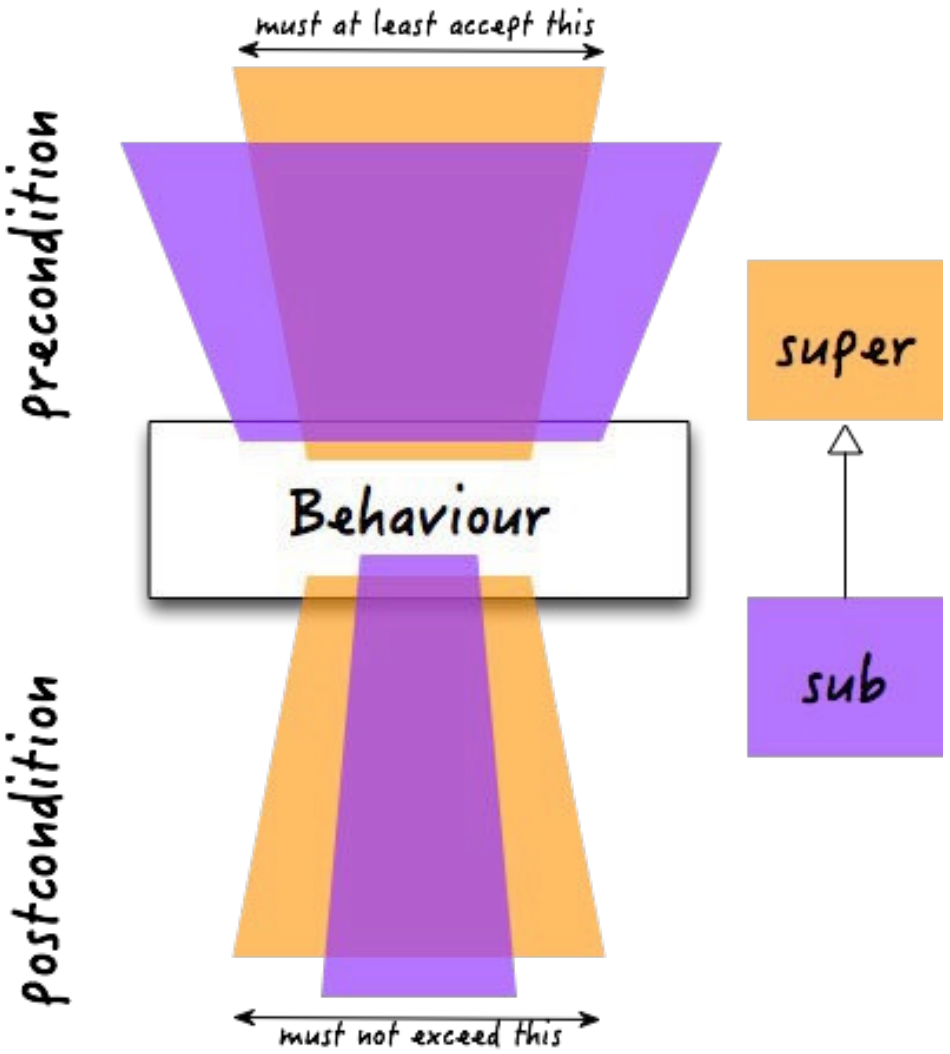
Good/bad design, why and how to improve?

```
class Drawing {
    public void drawAllShapes(List<IShape> shapes) {
        for (IShape shape : shapes) {
            shape.draw();
} } }

interface IShape {
    public void draw();
}

class Square implements IShape {
    public void draw() {  // draw the square  }
}
```

Liskov Substitution Principle

An object of a superclass
should always be substitutible
by an object of a subclass.

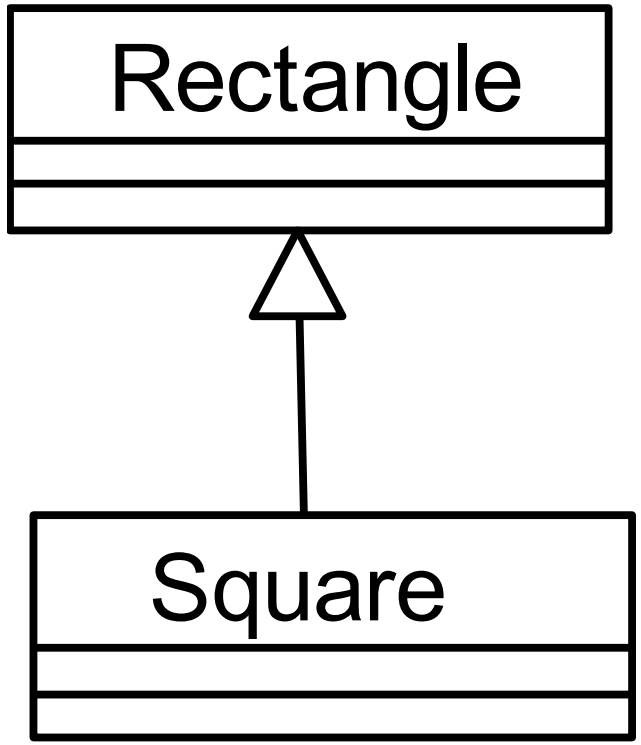
Liskov Substitution Principle



- Subclass has same or weaker preconditions
- Subclass has same or stronger postconditions

Derived methods should assume less deliver more

Liskov Subsitution Principle Question



Reasonable to derive a square from a rectangle?

Liskov Subsitution Principle

```
class Rectangle {

  private double fWidth, fHeight;

  public void setWidth(double w) { fWidth = w; }
  public void setHeight(double h) { fHeight = h; }
  public double getWidth() { return fWidth; }
  public double getHeight() { return fHeight; }
}
```

```
class Square extends Rectangle {

  public void setWidth(double w) {
      super.setWidth(w);
      super.setHeight(w);
  }

 public void setHeight(double h) {
      super.setHeight(h);
      super.setWidth(h);
  }
}
```

```
// somewhere else
public void calculate(Rectangle r) {
   r.setWidth(5);
   r.setHeight(6);
   assert(r.getWidth() * r.getHeight() == 30);
}
```

LSP Example – Rectangle & Square

- Postcondition for Rectangle setWidth(…) method

  assert((fWidth == w) && (fHeight == old.fHeight));

- Square setWidth(…) has weaker postcondition
  - does not conform to (fHeight == old.fHeight)
- Square has stronger preconditions
  - Square assumes fWidth == fHeight

- In other words
  - Derived methods assume more and deliver less.

LSP

LSP shows that a design can be structurally consistent (A Square ISA Rectangle)

But behaviourally **inconsistent**

So, we must verify whether the pre and postconditions in properties will hold when a subclass is used.

“It is only when derived types are completely substitutable for their base [/super] types that functions which use those base types can be reused with impunity, and the derived types can be changed with impunity.”

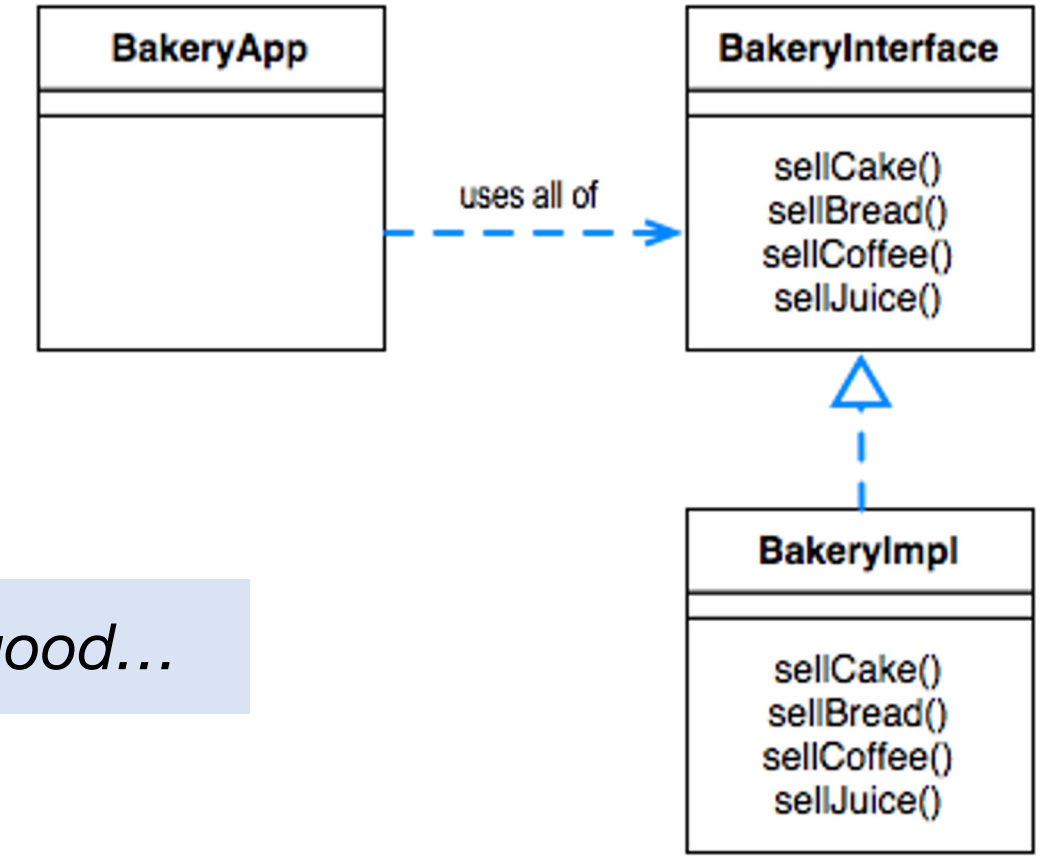
Interface Segregation Principle

Clients should not be forced to depend on interfaces they do not use.

Many client-specific interfaces are better than one general-purpose interface.

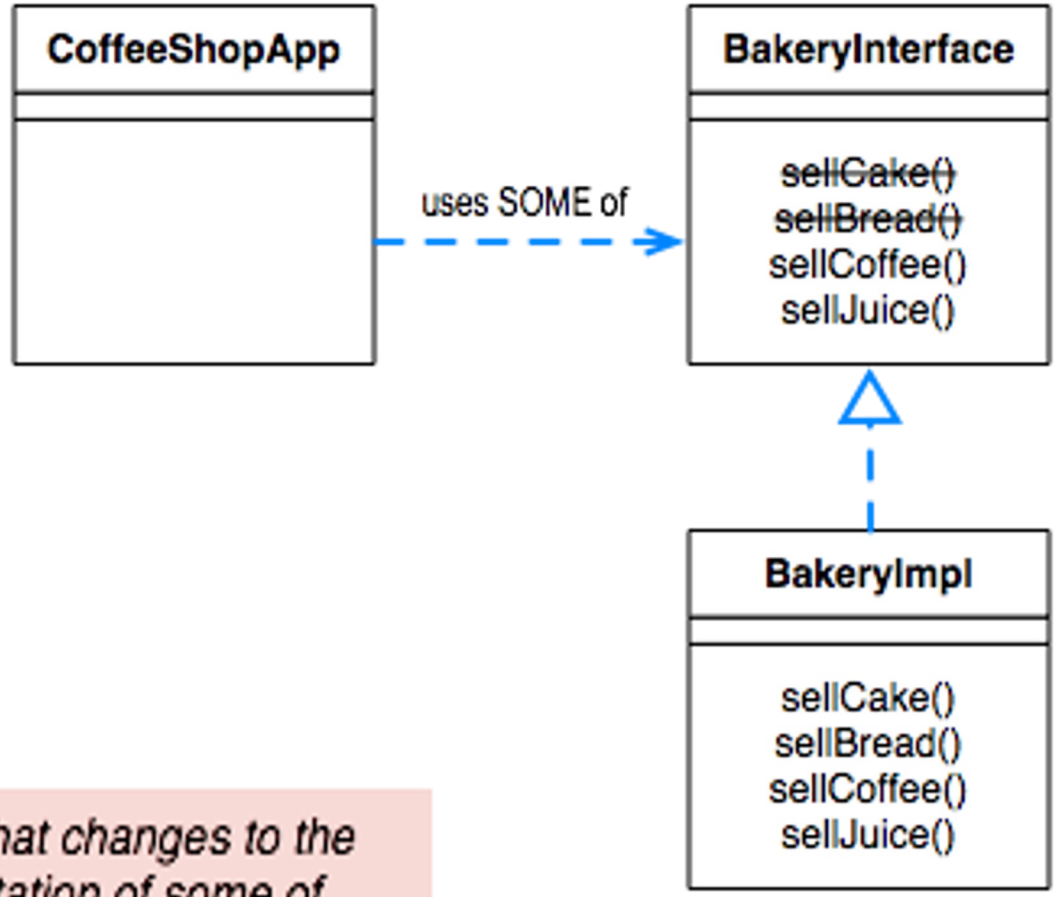
Depending on irrelevant interfaces causes needless coupling. This causes classes to change even when interfaces they do not care about are modified.

Interface Segregation Principle – Example



So far so good…

What if we want a coffee shop?

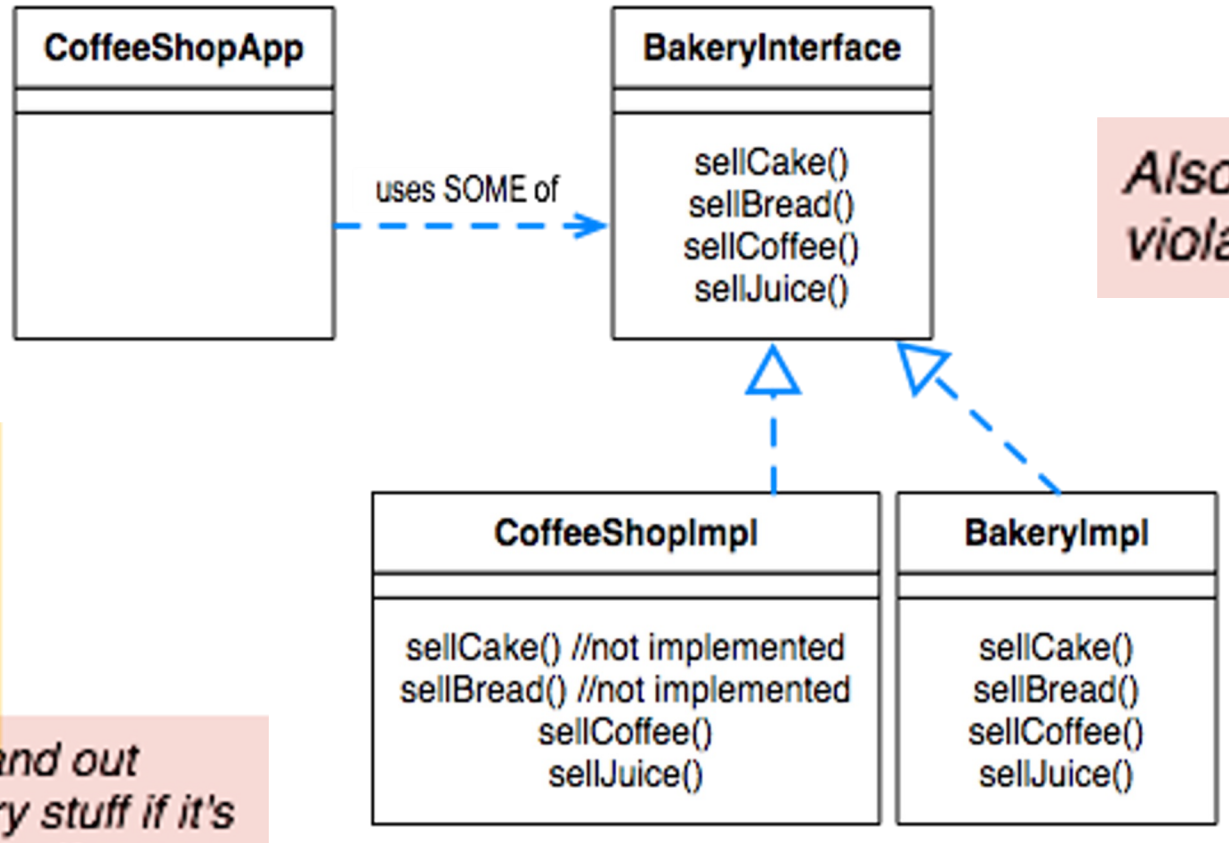


This is not as good!

Now the CoffeeShopApp client is only using part of the BakeryInterface. But it has to "know about" all of it.

it's likely that changes to the implementation of some of those methods will affect the implementations of the others.

How about a special implementation?

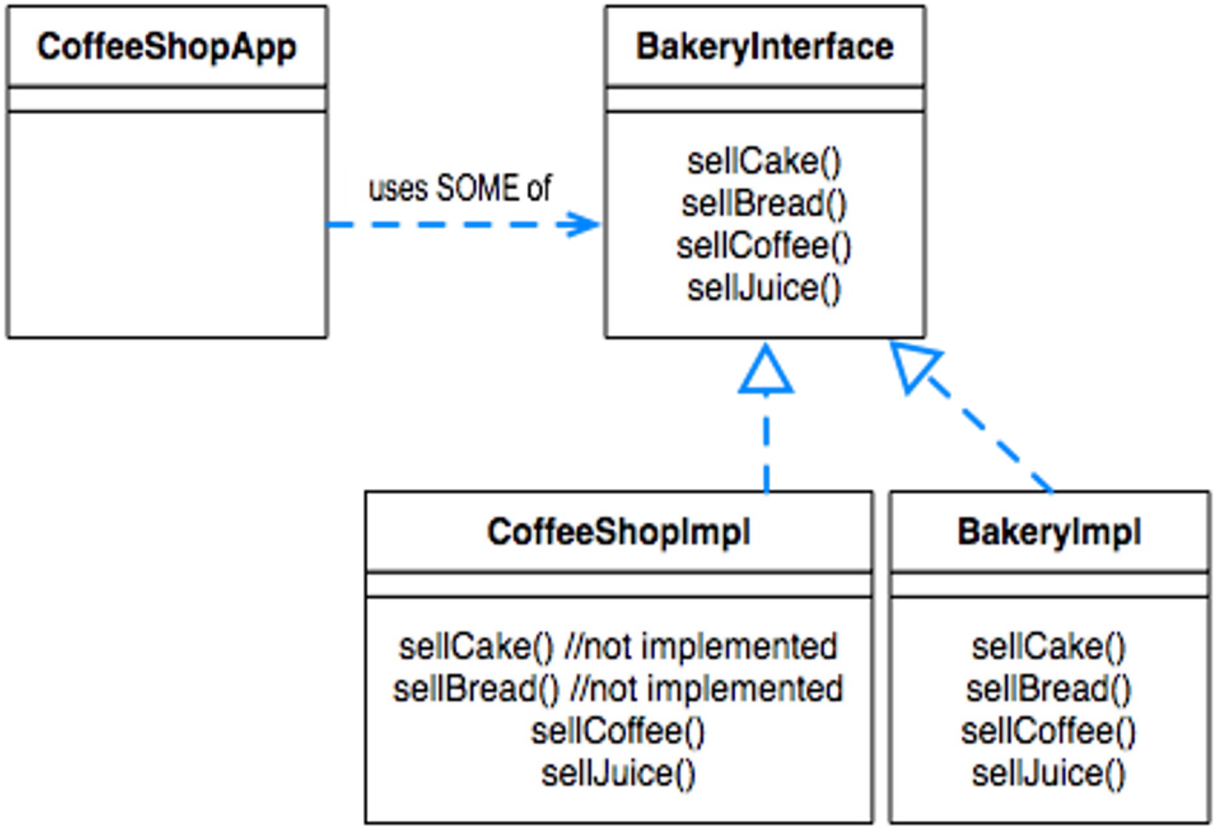


Also, myCoffeeShop app violates LSP!

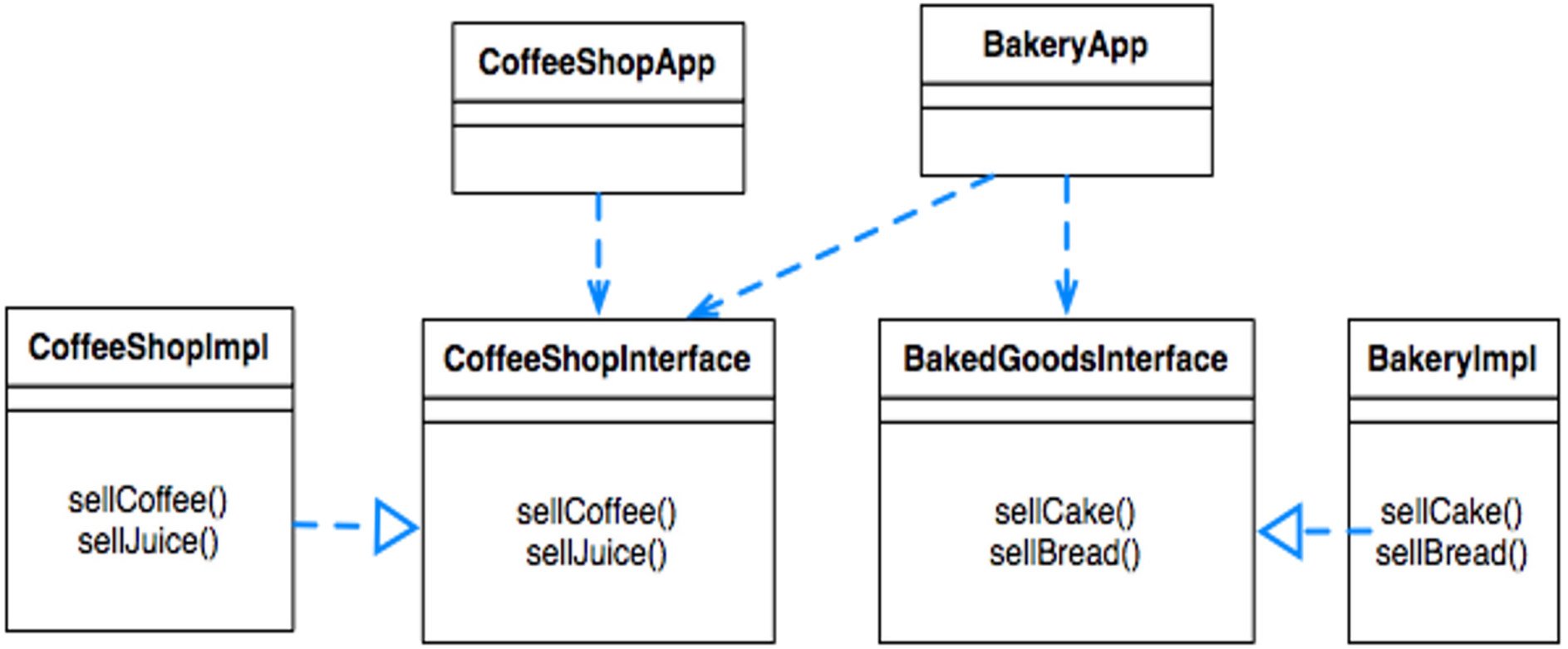
This is still bad. Now we have a CoffeeShop implementation that focuses on coffee shop stuff,

but now it has to hand out errors for the Bakery stuff if it's ever accidentally used!

How can this be redesigned using interface segregation principle?



Interface Segregation



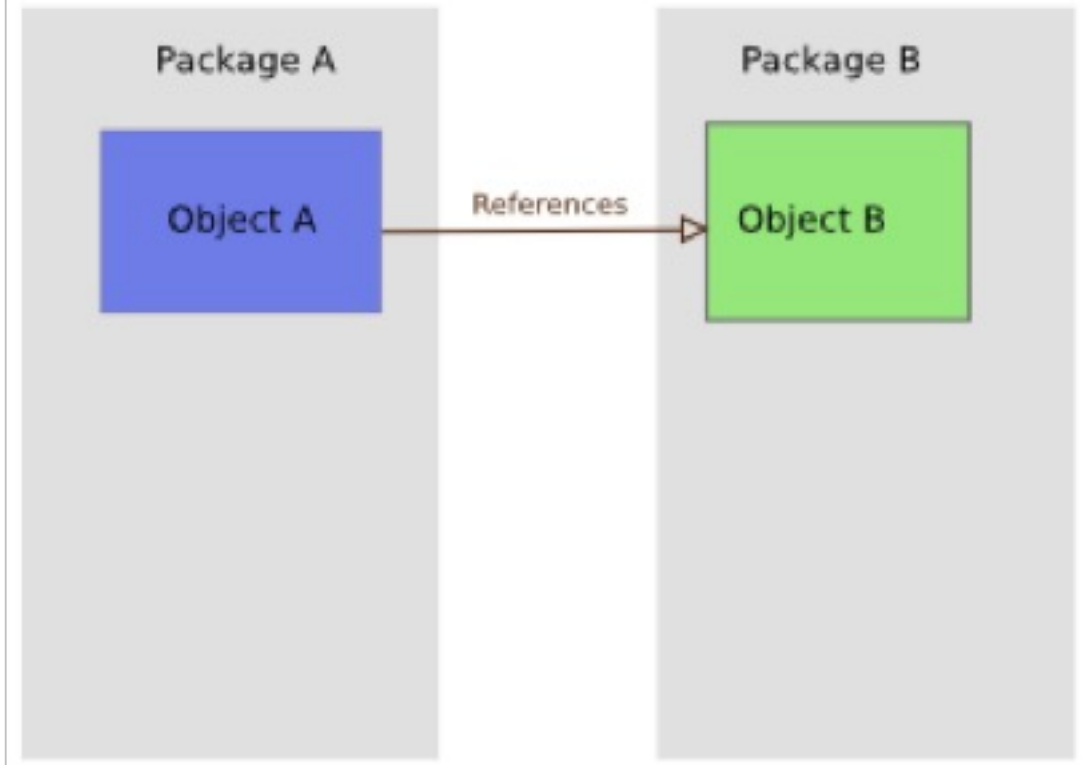
Dependency Inversion

Depend on abstractions not implementations.

- High-level modules should not depend on low-level modules. Both should depend on abstractions.
- Abstractions should not depend on details. Details should depend on abstractions.

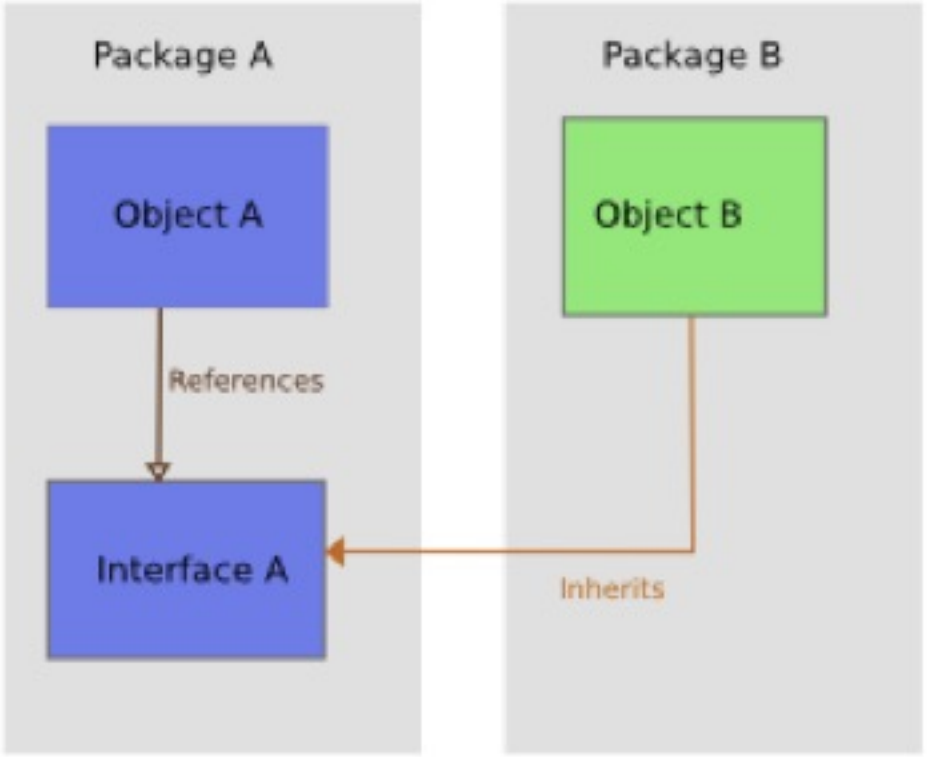
Dependency Inversion

From this:

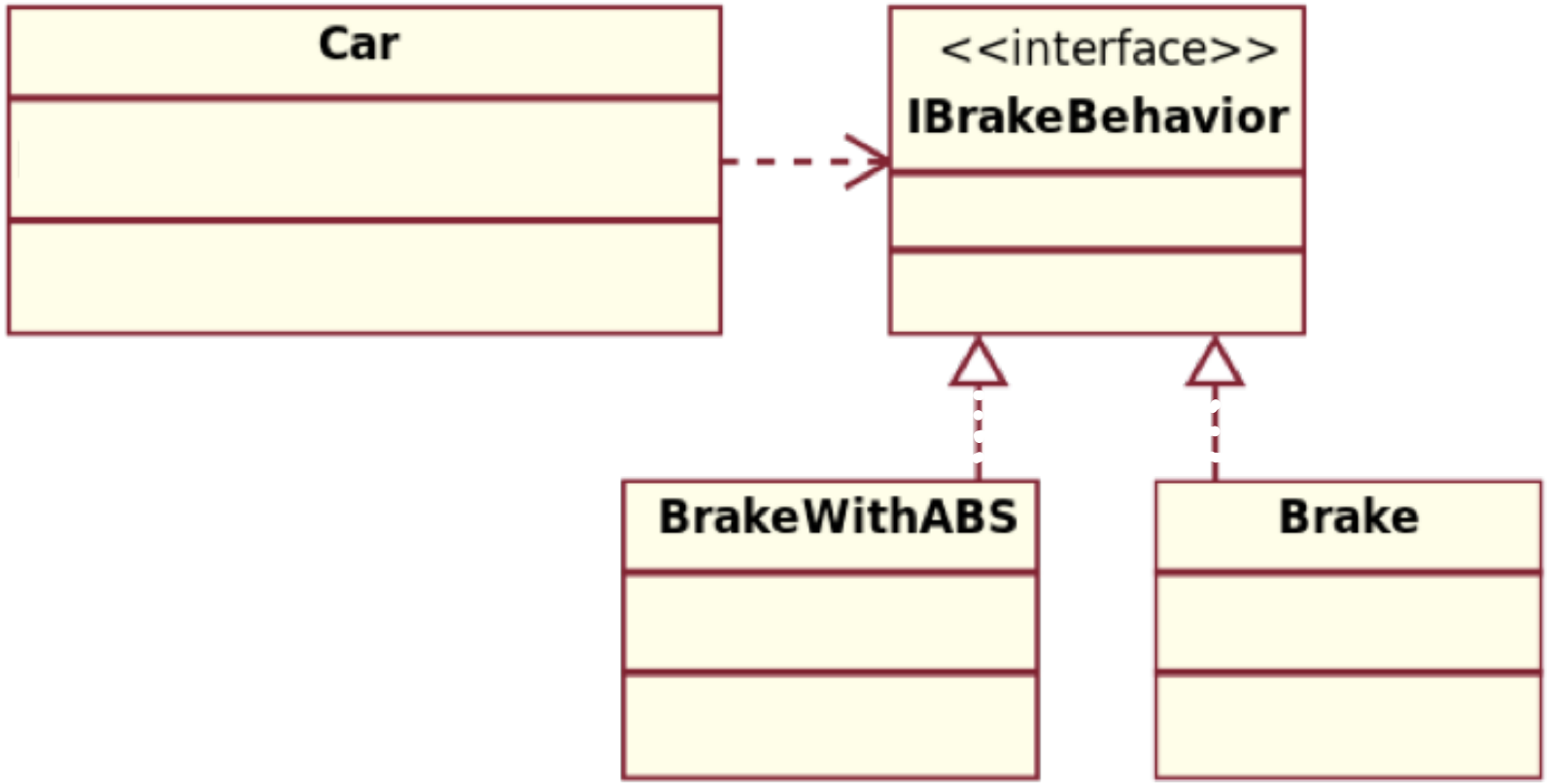


In the original version, reusing ObjectA requires reusing ObjectB. In the second, reusing A only requires an implementation of InterfaceA.

To this:



Dependency Inversion



Kahoot & Exercise

Which property / principle is violated affected most?

Questions to discuss

1. At which point of designing software should one check the design for design symptoms, is there a specific slot?
2. How to measure coupling or cohesion?
3. Can following the SOLID principles lead to a cluttered codebase since it encourages heavy use of interfaces / abstract classes?
4. What is the right number of interfaces (trade-off between abstraction and complexity)? What are dangers of over-minimizing interfaces? How small should modules be?
5. Does an increase in the number of interfaces, also lead to an increase in coupling?
6. Should global and content coupling be avoided completely?
7. How to find a balance between adhering to the principles and other factors (time, cost, complexity...)?

Good exercise: revisiting your designs

- Which design principles did you violate? How can you improve the design?
- Which principles did you apply and why?

Class Exercise – Designing a social networking app (~10-20mins if time allows) [https://bit.ly/3KKM0ql]

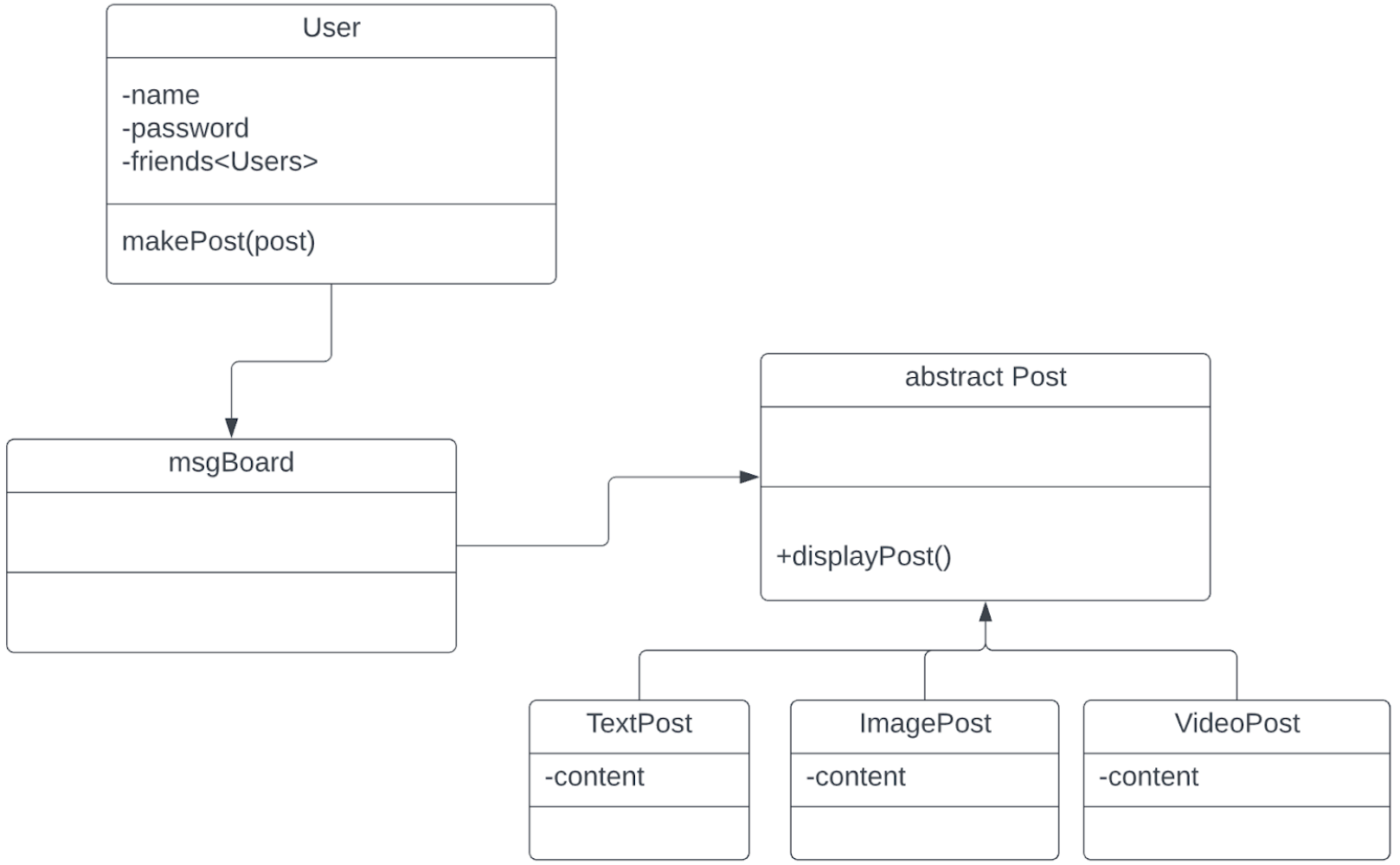
In groups design the model for a social networking app that allows its users to

- create a user account with name and password,
- befriend other users of the app to see their messaging board, and
- create and share different types of posts (for now, text posts, posts with an image and a text, posts with a video) that are displayed on the messaging board.

In addition, the app should allow for two different types of users: priority users (the ones that pay) that are able to also share their current status on their messaging board and regular users that are not able to share their status.

→ Sketch out the class diagram as well as the code for the Messaging Board. Once you sketched it out, make sure to go over the design principles and examine whether you violate any as well as which ones you applied and why

Class Exercise – Designing a social networking app (~10-20mins if time allows)



Quiz

Quiz

1. Which of the following statements accurately describes the consequences of high coupling in software development? (Choose all that apply)
   A. It enhances the system's maintainability and evolvability by making code changes more straightforward
   B. Changes in one class can lead to errors in others, complicating bug fixes and feature additions

2. Which type of cohesion is considered the most detrimental out of the following three? [Select only one]
   A. Functional cohesion, because it limits the scope of the class to a single functionality
   B. Sequential cohesion, due to the strict order in which operations must be performed, reducing flexibility
   C. Coincidental cohesion, because it involves arbitrary grouping of class functionality, leading to poor class design

Quiz

3. Select whether the following statement is true or false:

   “The Interface Segregation Principle suggests that having a single, comprehensive interface for all functionalities in a software system is the most efficient way to ensure flexibility and reusability across various components.”.
   [True / False]

Quiz

4. Given the following sketched out code of a class Dog that extends the class Animal, which of the following design principles is clearly violated?
   A. Single responsibility
   B. Interface segregation
   C. Liskov substitution

```
class Dog extends Animal {
  private String name;
  private User owner;
  private boolean hasEaten;
  public String getName() {return name;}
  public void adjustOwnerName(String newOwnerName) {
                owner.setUserName(newOwnerName);}
  public void setHasEaten(boolean ate) {hasEaten = ate; }
  public Date getCurrentDate() {return
Calendar.getInstance().getTime();}
}
```