Software Architecture, Principles, Design Patterns

Arslan Anwar | Senior Software Engineer

Remember, knowing concepts like abstraction, inheritance, and polymorphism do not make you a good object oriented designer. A design guru thinks about how to create flexible designs that are maintainable and that can cope with change.

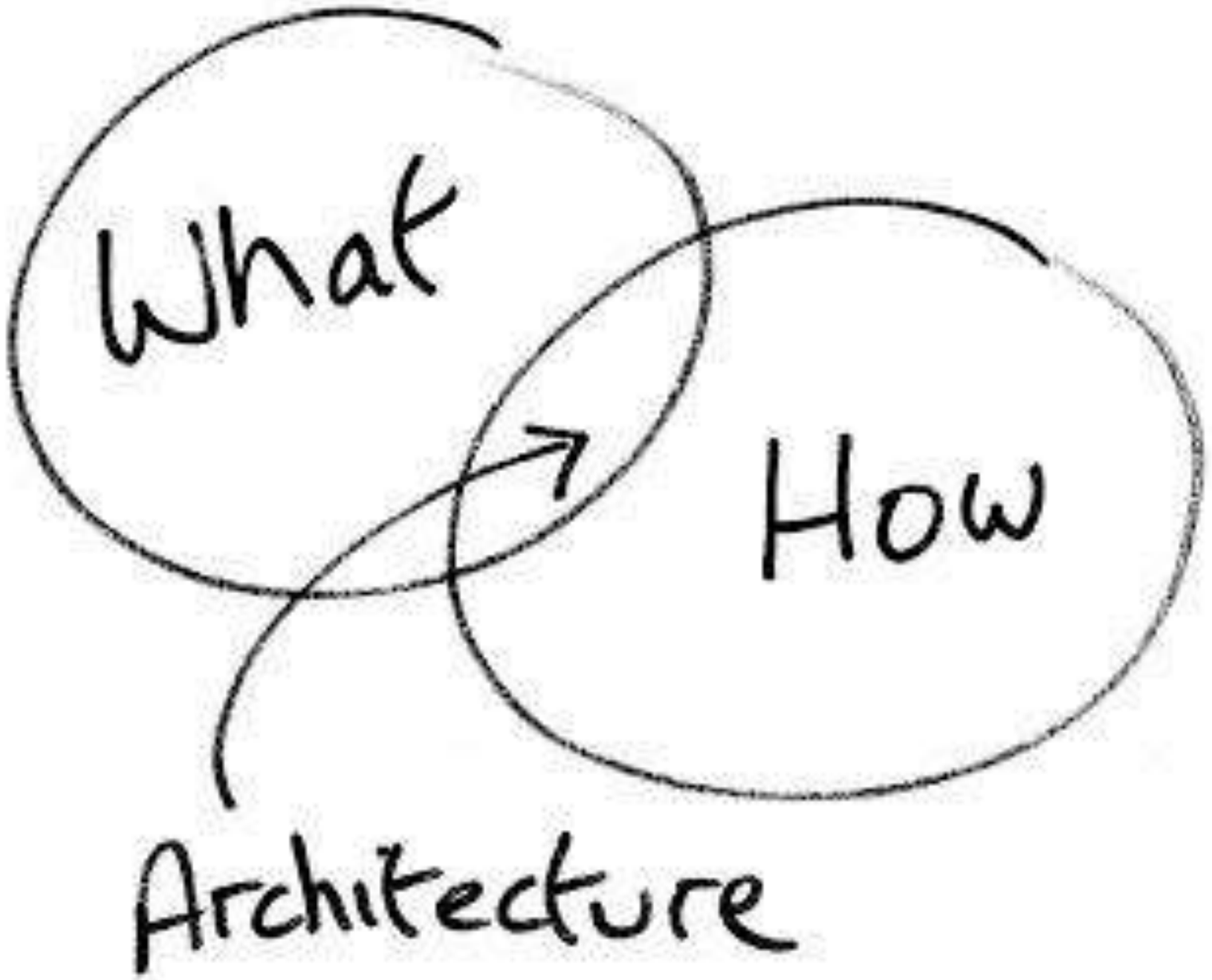
Agenda

- Software Architecture
- Software Design Principles

Software Architecture

- What is Software Architecture
- Architecture vs. Design
- Why Architecture is important
- MVC , MVP , MV VM, 3 Tier

What is Software Architecture



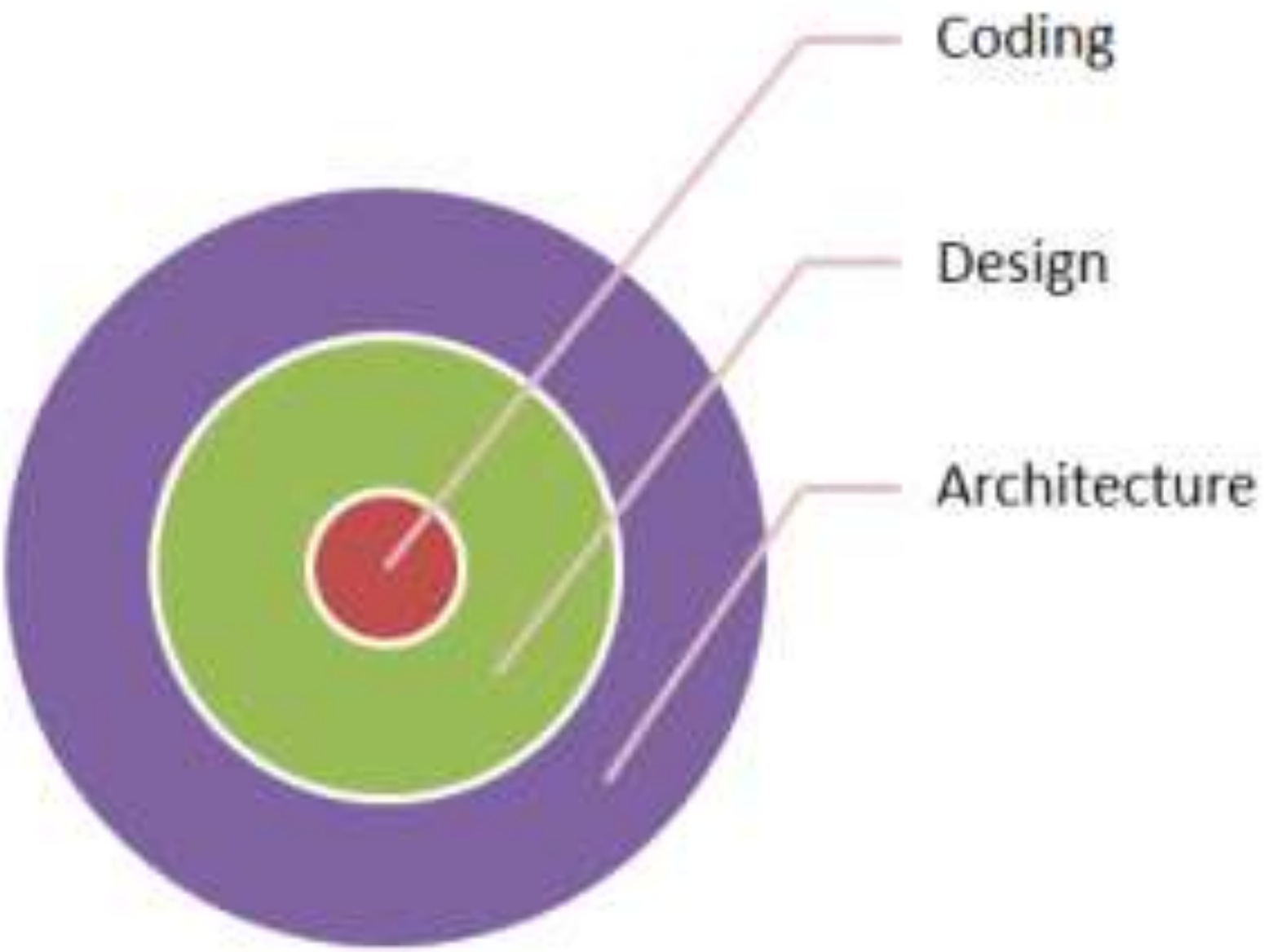
What is Software Architecture

o **IEEE: Architecture is defined by the recommended practice as the fundamental** organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution.

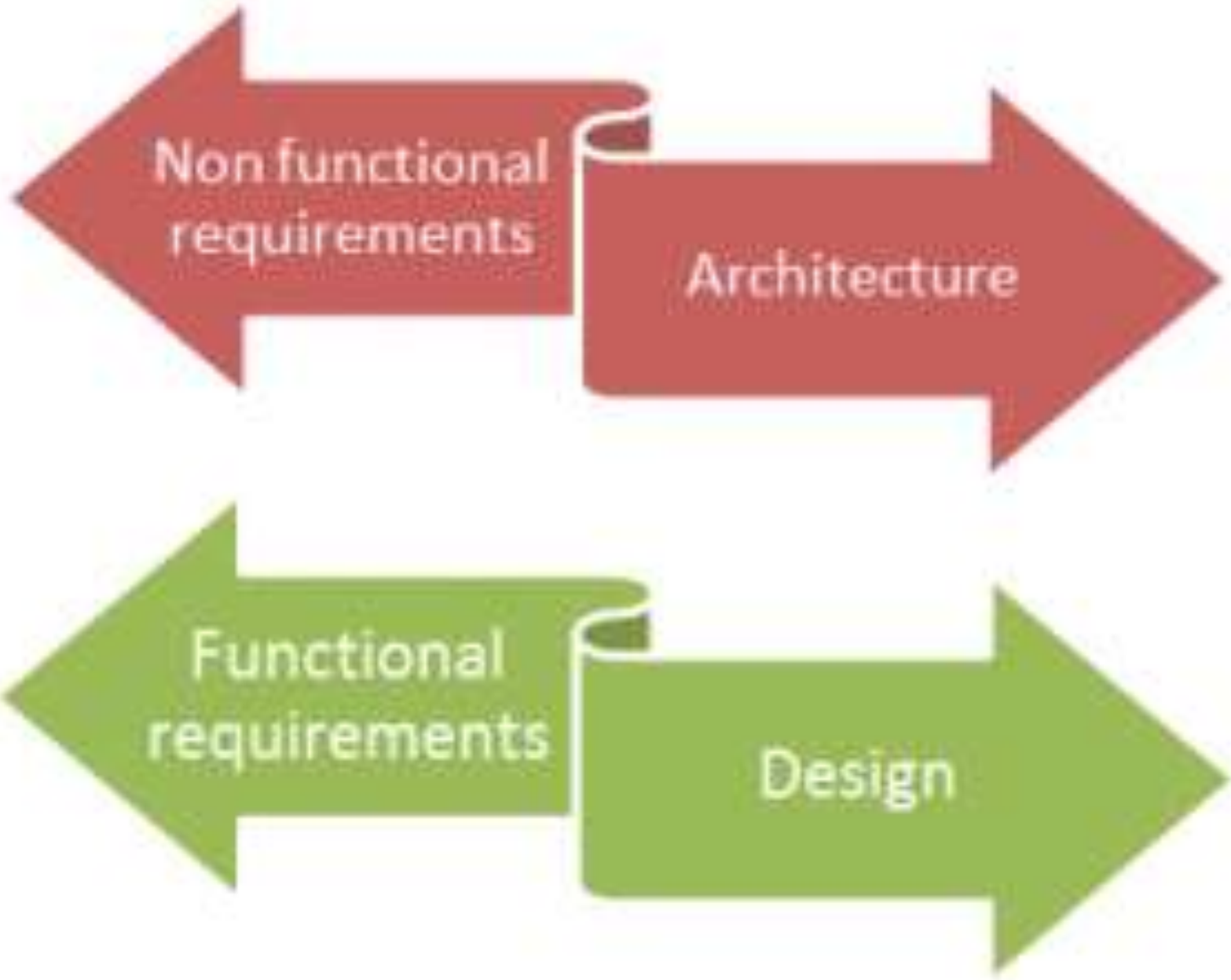
o Architecture **captures system structure in terms of components and how** they interact. It primarily focuses on aspects such as performance, reliability, scalability, testbility, maintainability and various other attributes, which can be key both structurally and behaviorally of a software system.

Software Architecture vs. Software Design

All architecture is design, but not all design is architecture.



Software Architecture vs. Software Design



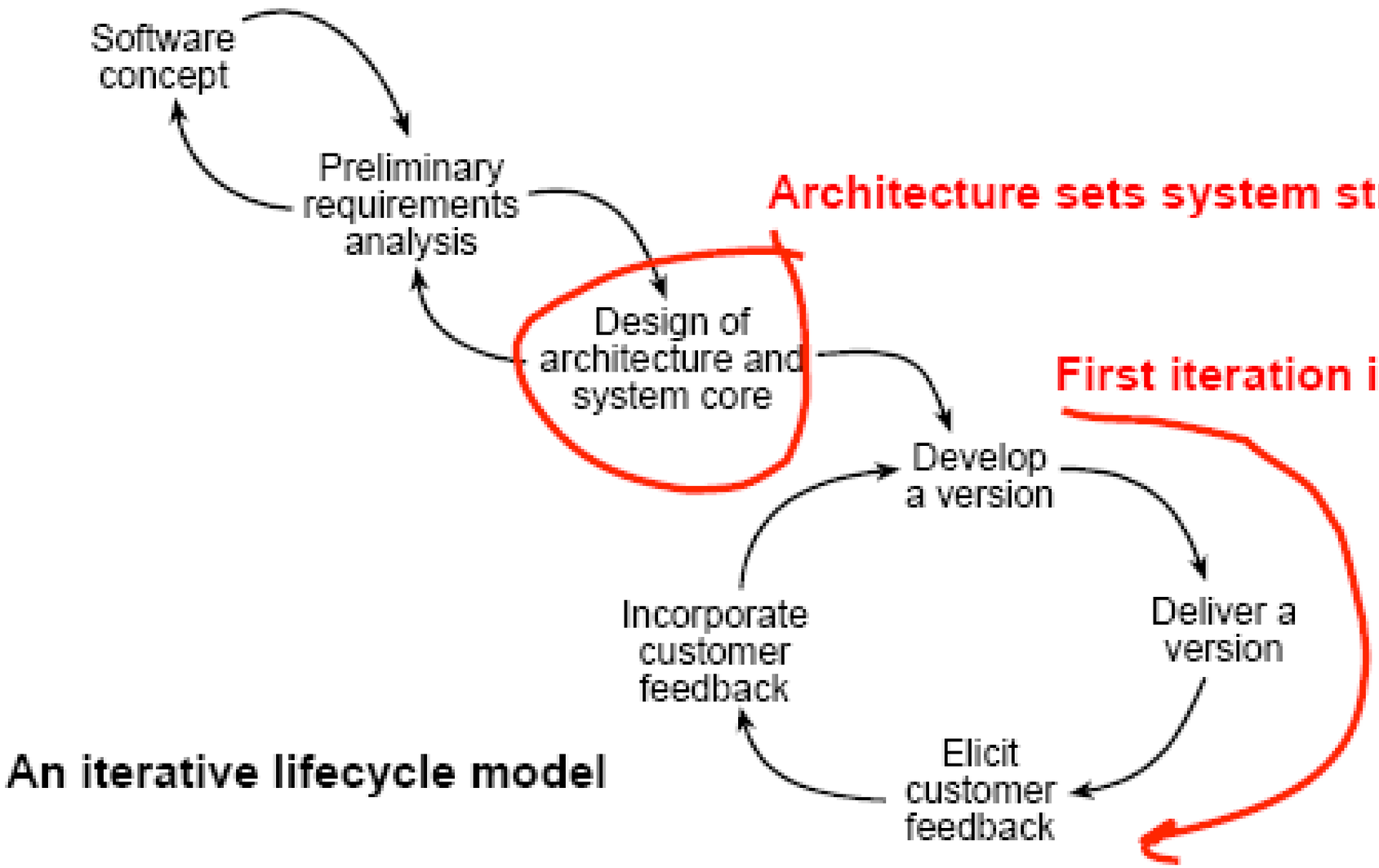
Software Architecture vs. Software Design

Architecture	Design
Fundamental properties	Detailed properties
Define guidelines	Communicate with developers
Cross-cutting concerns	Details
High-impact	Individual components
Communicate with business stakeholders	Use guidelines
Manage uncertainty	Avoid uncertainty
Conceptual integrity	Completeness

Role of Software Architecture

Role of Software Architecture

Architecture plays a vital role in establishing the structure of the system, early in the development lifecycle



An iterative lifecycle model

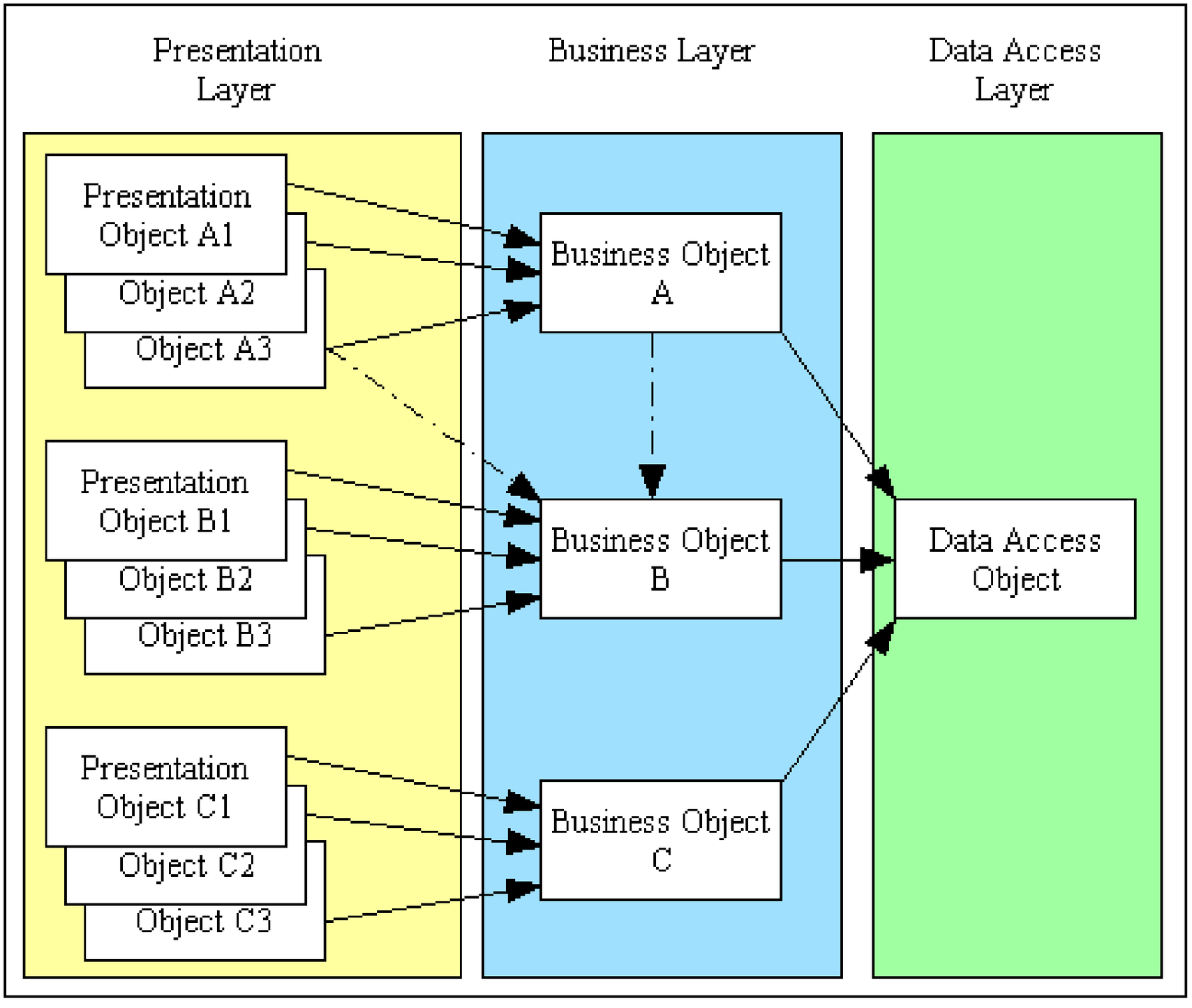
Multi Tier Architecture

In software engineering, multi-tier architecture (often referred to as n-tier architecture) is a client–server architecture in which

- **Presentation**
- **Application processing**
- **Data management**

functions are **physically separated**. The most widespread use of multi-tier architecture is the three-tier architecture.

Multitier Tier Architecture



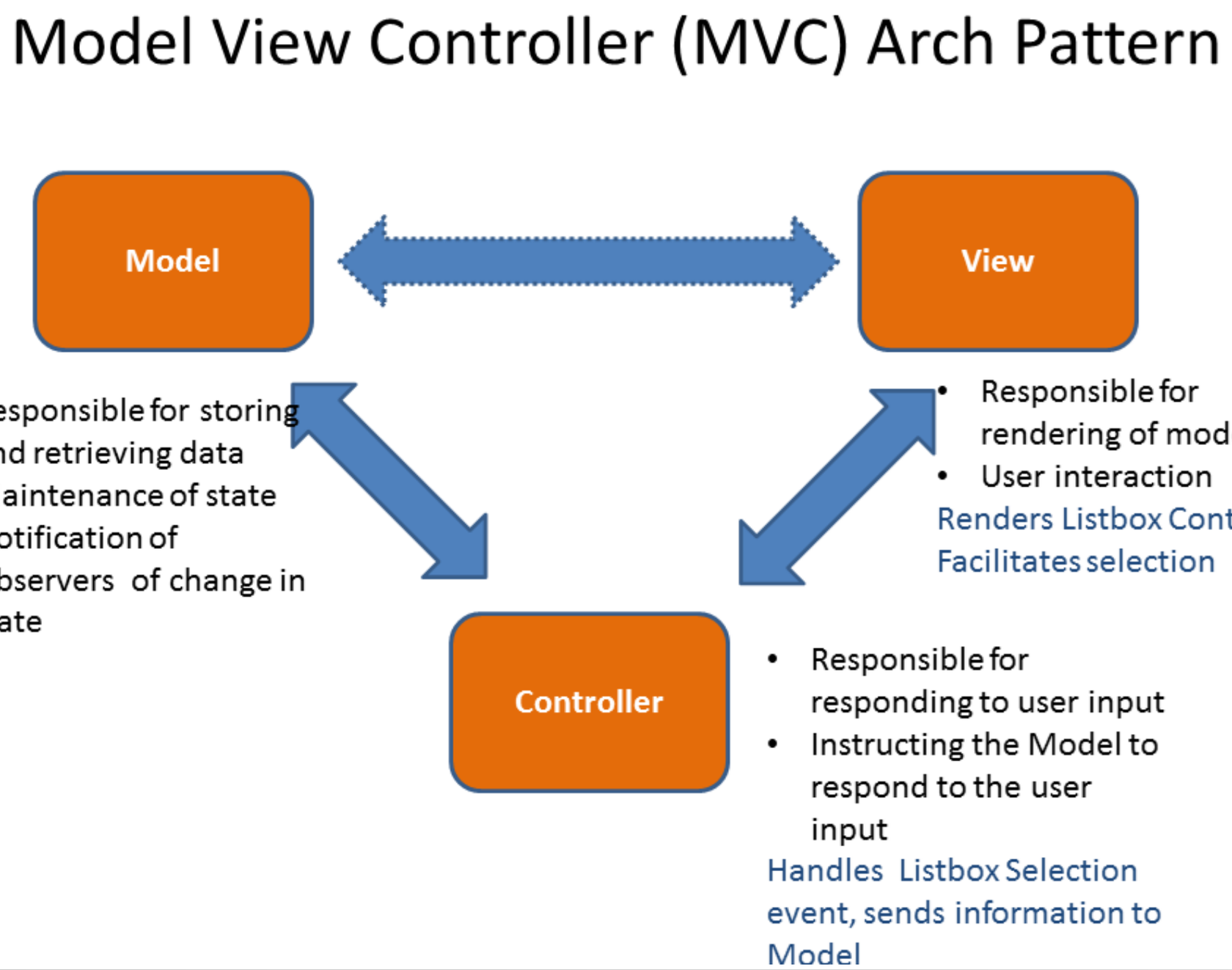
Model View Controller

▪ A **controller** can send commands to the model to update the model's state (e.g., editing a document). It can also send commands to its associated view to change the view's presentation of the model (e.g., by scrolling through a document).

▪ A **model** notifies its associated views and controllers when there has been a change in its state. This notification allows the views to produce updated output, and the controllers to change the available set of commands. In some cases an MVC implementation might instead be "passive," so that other components must poll the model for updates rather than being notified.

▪ A **view** requests information from the model that it uses to generate an output representation to the user.
▪1 - * relationship between controller and view

Model View Controller



Model View Presenter

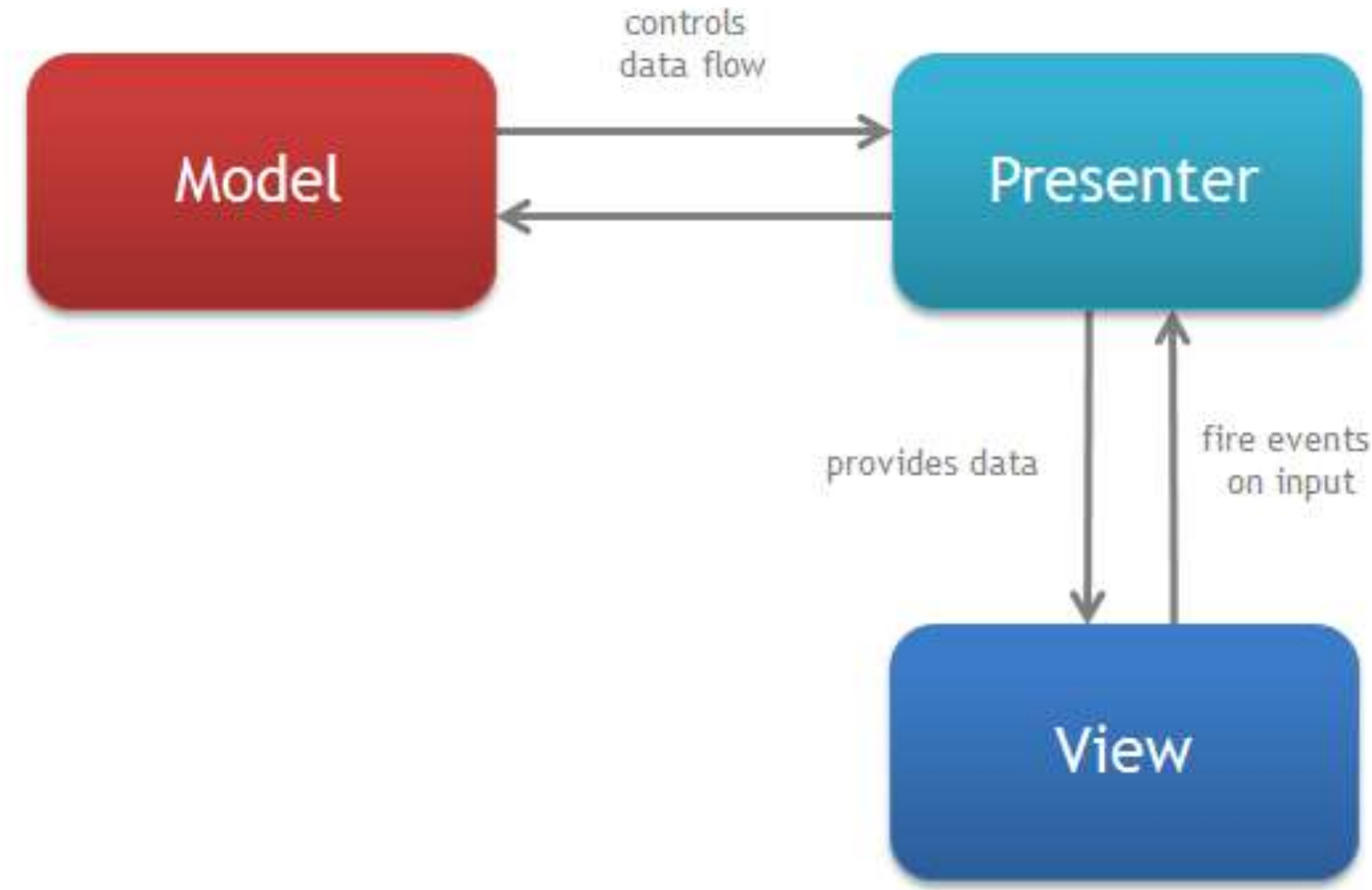
MVP is a user interface architectural pattern engineered to facilitate automated unit testing and improve the separation of concerns in presentation logic:

- The **model** is an interface defining the data to be displayed or otherwise acted upon in the user interface.

- The **view** is a passive interface that displays data (the model) and routes user commands (events) to the presenter to act upon that data.

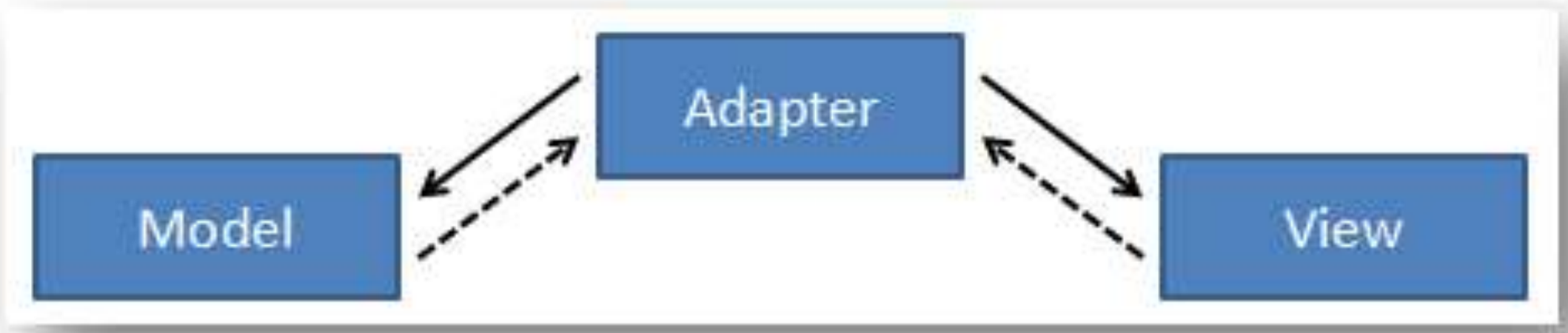
- The **presenter** acts upon the model and the view. It retrieves data from repositories (the model), and formats it for display in the view.
- 1 - 1 relationship between presenter and view

Model View Presenter



Model View Adapter

Model–view–adapter (MVA) or mediating-controller MVC is an architectural pattern and multitier architecture, used in software engineering. In complex computer applications that present large amounts of data to users, developers often wish to separate data (model) and user interface (view) concerns so that changes to the user interface will not affect data handling and that the data can be reorganized without changing the user interface.

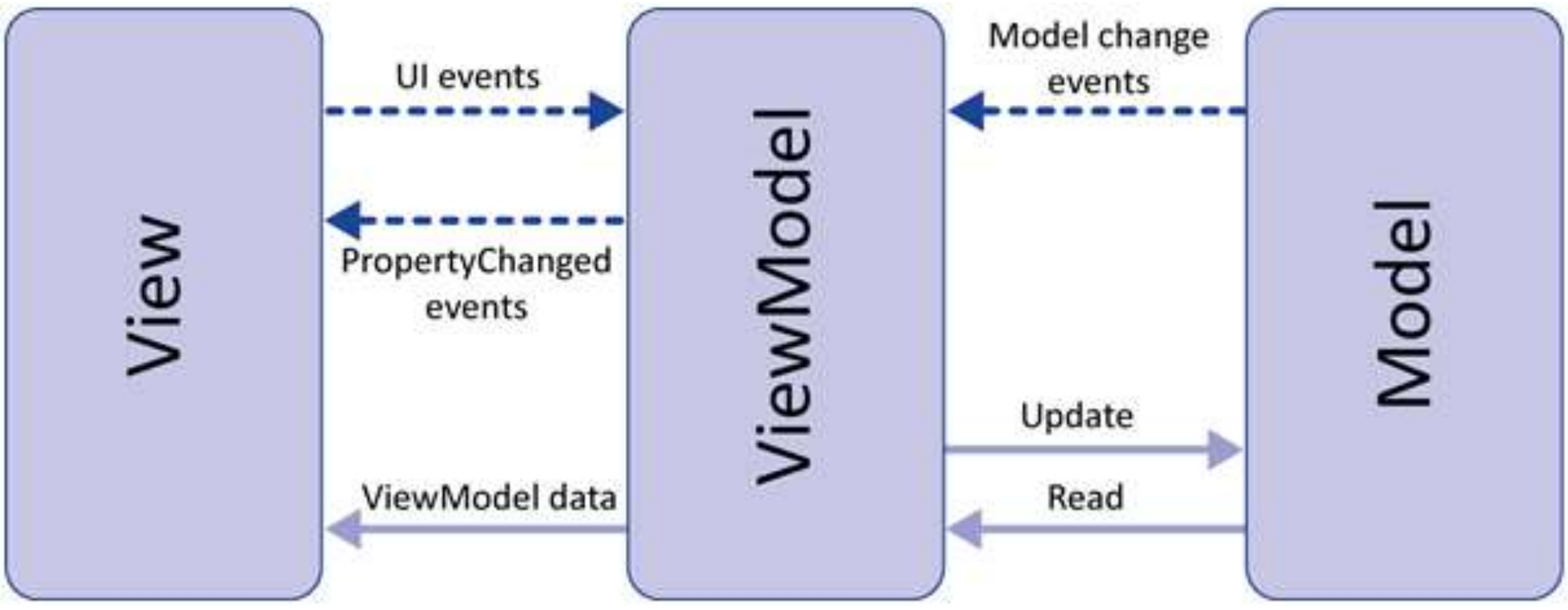


Model View ViewModel

Model: domain model which or the data access layer that represents that content
View: View refers to all elements displayed by the GUI such as buttons, labels, and other controls.
View model: “model of the view” meaning it is an abstraction of the view that also serves in mediating between the view and the model which is the target of the view data bindings. It could be seen as a specialized aspect of what would be a controller (in the MVC pattern) that acts as a converter that changes model information into view information and passes commands from the view into the model.
Controller: some references for MVVM also include a controller layer to illustrate that the view model is a specialized functional set in parallel with a controller, while others do not.

Binder: the use of a declarative databinding and command bind technology is an implicit part of the pattern.
Relationship: Many views can use one ViewModel

Model View ViewModel



Problems - Android side

- Tight coupling in business logic and view
- Difficult to segregate responsibilities
- Difficult to test
- A lot of boilerplate code

Solution: Follow an architectural pattern!

Solution - Android side

- MVC in Android
- MVP in Android
- MVVM in Android

RoboBinding vs AndroidBinding

- Binding - a look
- Basic approach of implementation
- Maintenance and stability
- Reduction of boilerplate code
- Ease of use

RoboBinding - Sample Project

Talk is cheap, show me the code!

Conclusion

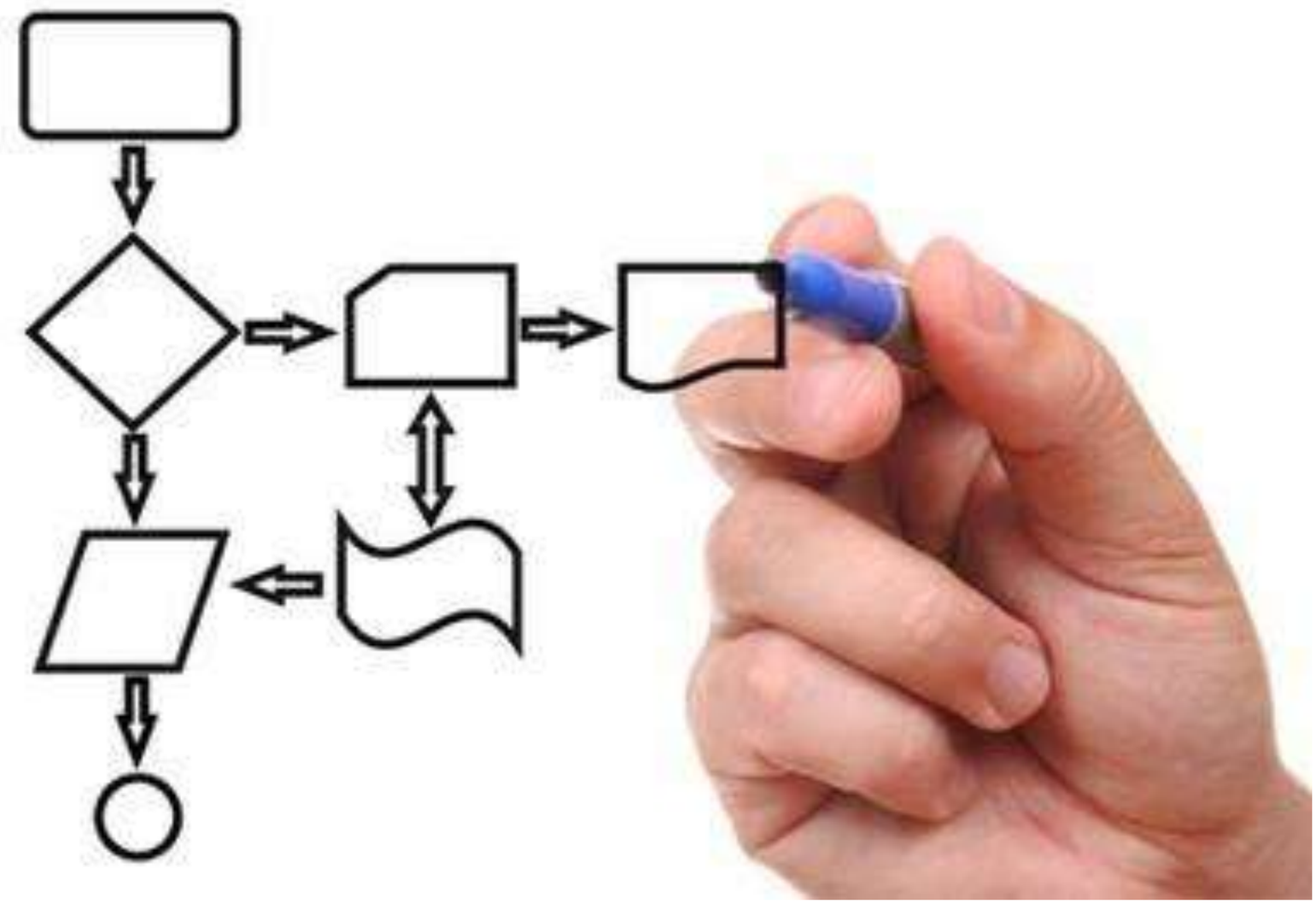
- RoboBinding solves our problems:
  - reduced a lot of boilerplate code
  - decouples view and logic
  - responsibilities segregated
  - easy to test code

Software Design and Principles

- What is Software Design
- Design Smell
- Software Design Principles
  - SOLID (object-oriented design)
  - DRY principle
  - YAGNI principle
  - KISS principle

What is Software Design

o Software design is the process of defining software methods, functions, objects, and the overall structure and interaction of your code so that the resulting functionality will satisfy your users requirements.



7 Design Smell

1. Rigidity
2. Fragility
3. Immobility
4. Viscosity
5. Needless Complexity
6. Needless repetition
7. Opacity

7 Design Smell

- **Rigidity**

  The system is hard to change because every change forces many other changes to other parts of the system.

- **Fragility**

  Changes cause the system to break in places that have no conceptual relationship to the part that was changed.

- **Immobility**

  It is hard to disentangle the system into components that can be reused in other systems.

7 Design Smell

- **Viscosity**

  Doing things right is harder than doing things wrong.

- **Needless Complexity**

  The design contains infrastructure that adds no direct benefit.

- **Need Repetition**

  The design contains repeating structures that could be unified under a single abstraction.

- **Opacity**

  It is hard to read and understand. It does not express its intent well.

Software Design Principles

Software design principles represent a set of guidelines that helps us to avoid having a bad design. The software designs helps to make a software well designed, well read and best maintainable.

S.O.L.I.D Principles

S.O.L.I.D. is a collection of best-practice, object-oriented design principles which can be applied to your design, allowing you to accomplish various desirable goals such as loose-coupling, higher maintainability, intuitive location of interesting code, etc.

Single Responsibility
Open Closed
Liskov Substitution
Interface Segregation
Dependency Inversion

S.O.L.I.D Principles

Single Responsibility Principle

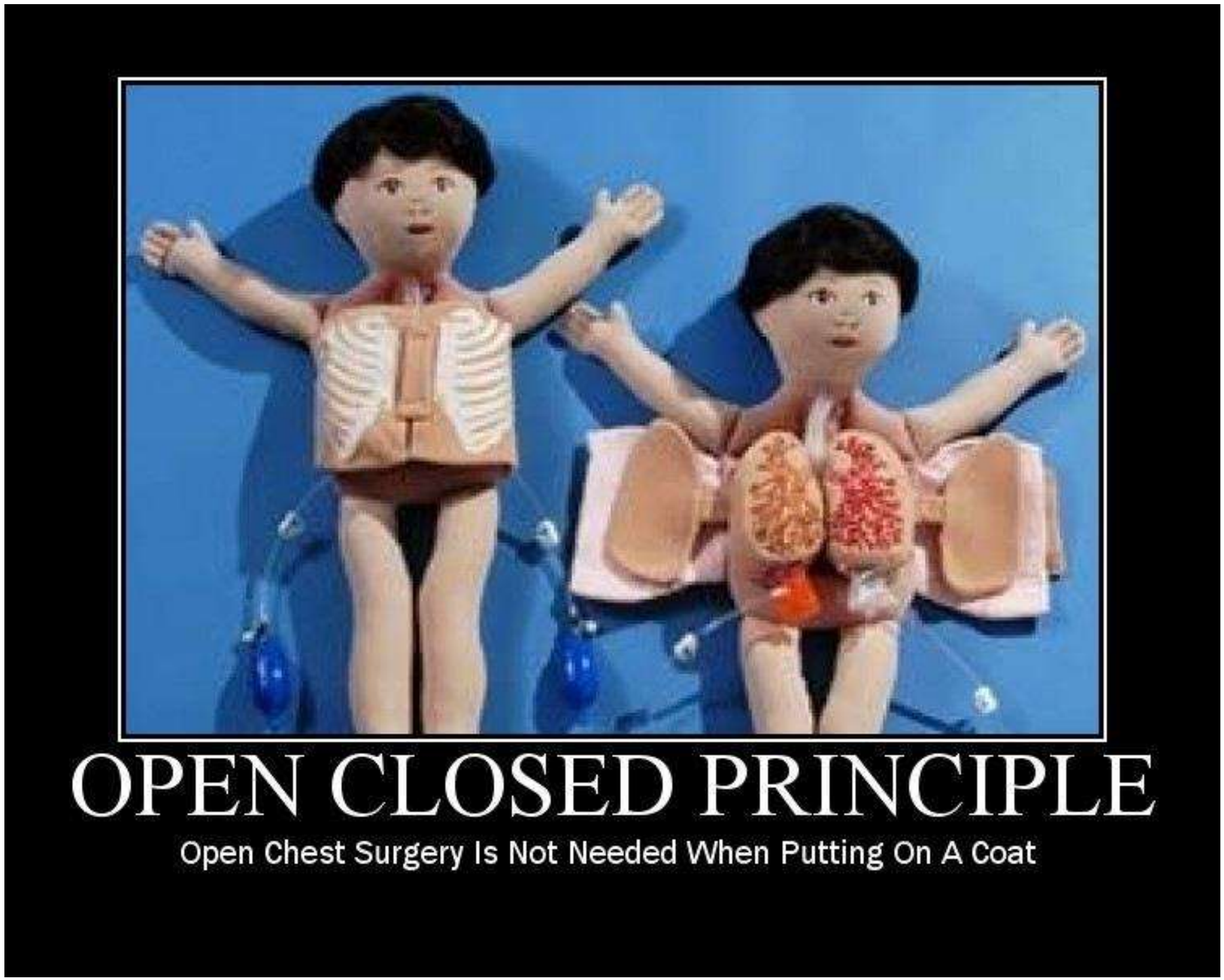
Single Responsibility Principle

The Single Responsibility Principle (SRP) states that there should never be more than one reason for a class to change. This means that every class, or similar structure, in your code should have only one job to do.

How many responsibilities?

```
class Employee {
   public Pay calculatePay() {...}
   public void save() {...}
   public String describeEmployee() {...}
}
```

Open Closed Principle



Open Closed Principle

The Open-Closed Principle (OCP) states that classes should be open for extension but closed for modification.

“Open to extension” means that you should design your classes so that new functionality can be added as new requirements are generated. “Closed for modification” means that once you have developed a class you should never modify it, except to correct bugs.

```
void checkOut(Receipt receipt) {
  Money total = Money.zero;
  for (item : items) {
    total += item.getPrice();
    receipt.addItem(item);
  }
  Payment p = acceptCash(total);
  receipt.addPayment(p);
}
```

Open Closed Principle

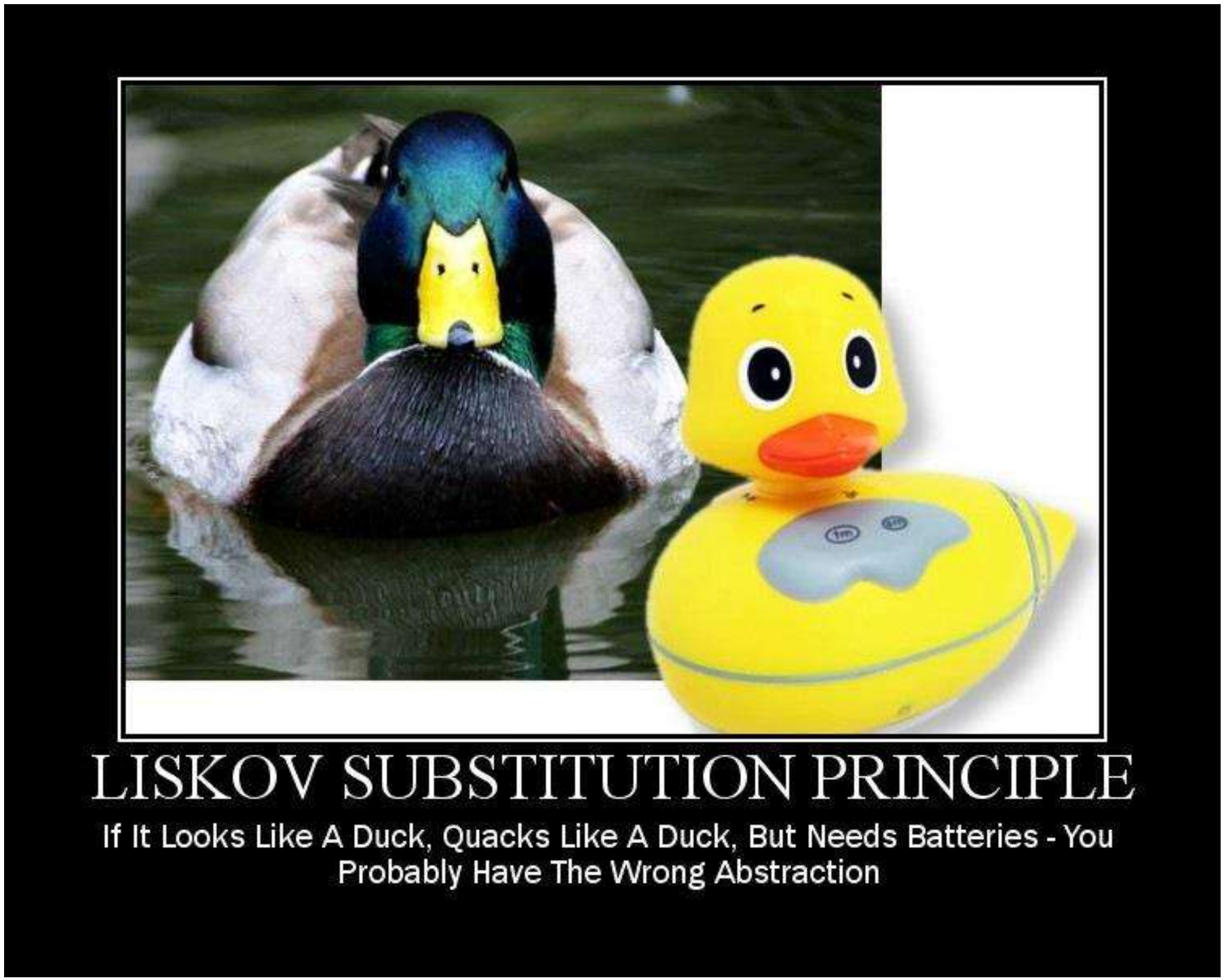
So how do we add credit card support? You could add an “if” statement like this, but then that would be violation of OCP.

```
Payment p;
if (credit)
  p = acceptCredit(total);
else
  p = acceptCash(total);
receipt.addPayment(p);
```

Here is a better solution:

```
public interface PaymentMethod {void acceptPayment(Money total);}

void checkOut(Receipt receipt, PaymentMethod pm) {
  Money total = Money.zero;
  for (item : items) {
    total += item.getPrice();
    receipt.addItem(item);
  }
  Payment p = pm.acceptPayment(total);
  receipt.addPayment(p);
}
```

Liskov Substitution Principle



Liskov Substitution Principle

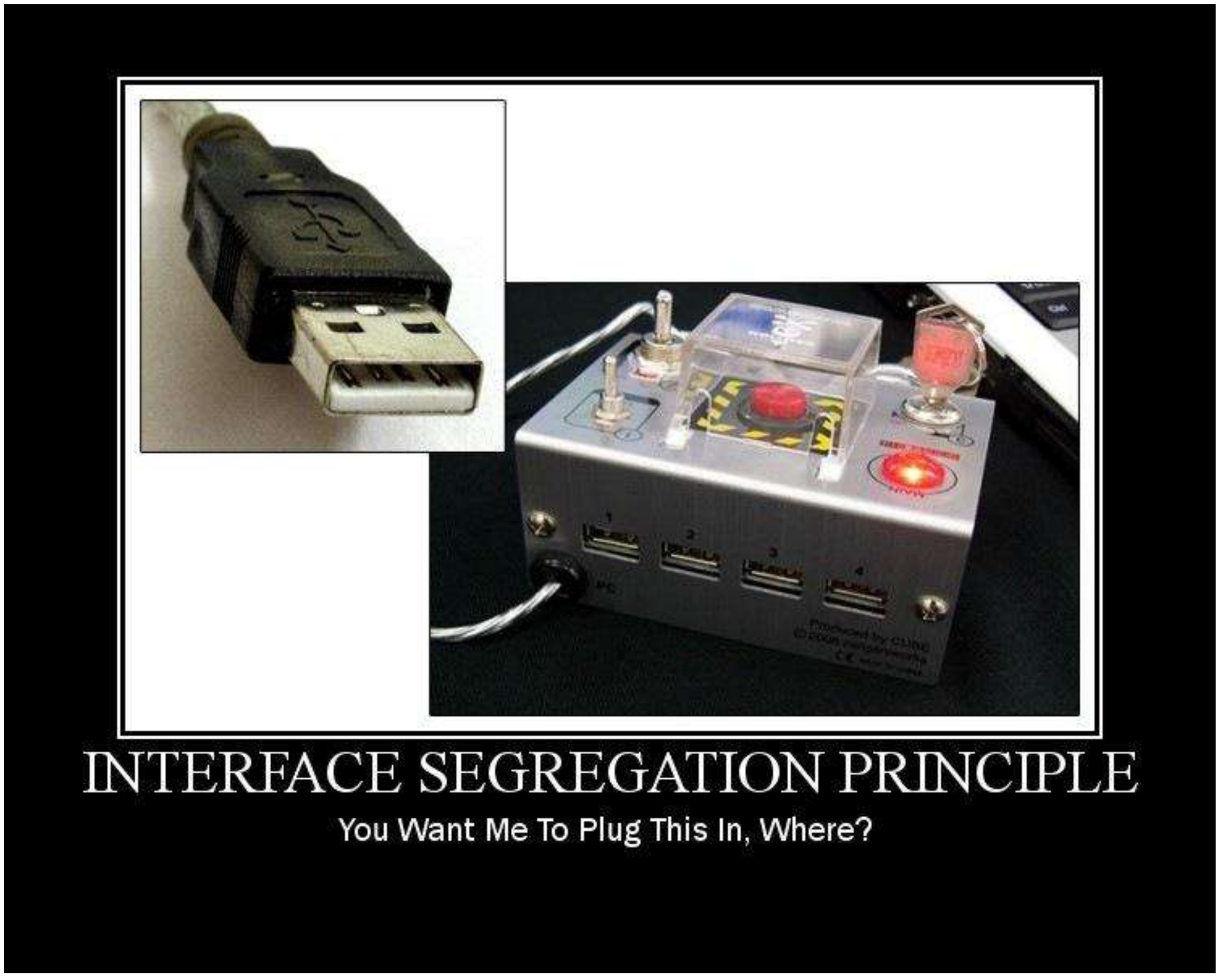
"Derived types must be completely substitutable for their base types"

```
public class Rectangle {
   private double height;
   private double width;

   public double area();

   public void setHeight(double height);
   public void setWidth(double width);
}
```

Liskov Substitution Principle

```
public class Square extends Rectangle {
   public void setHeight(double height) {
      super.setHeight(height);
      super.setWidth(height);
   }

   public void setWidth(double width) {
      setHeight(width);
   }
}
```

Interface Segregation Principle



Interface Segregation Principle

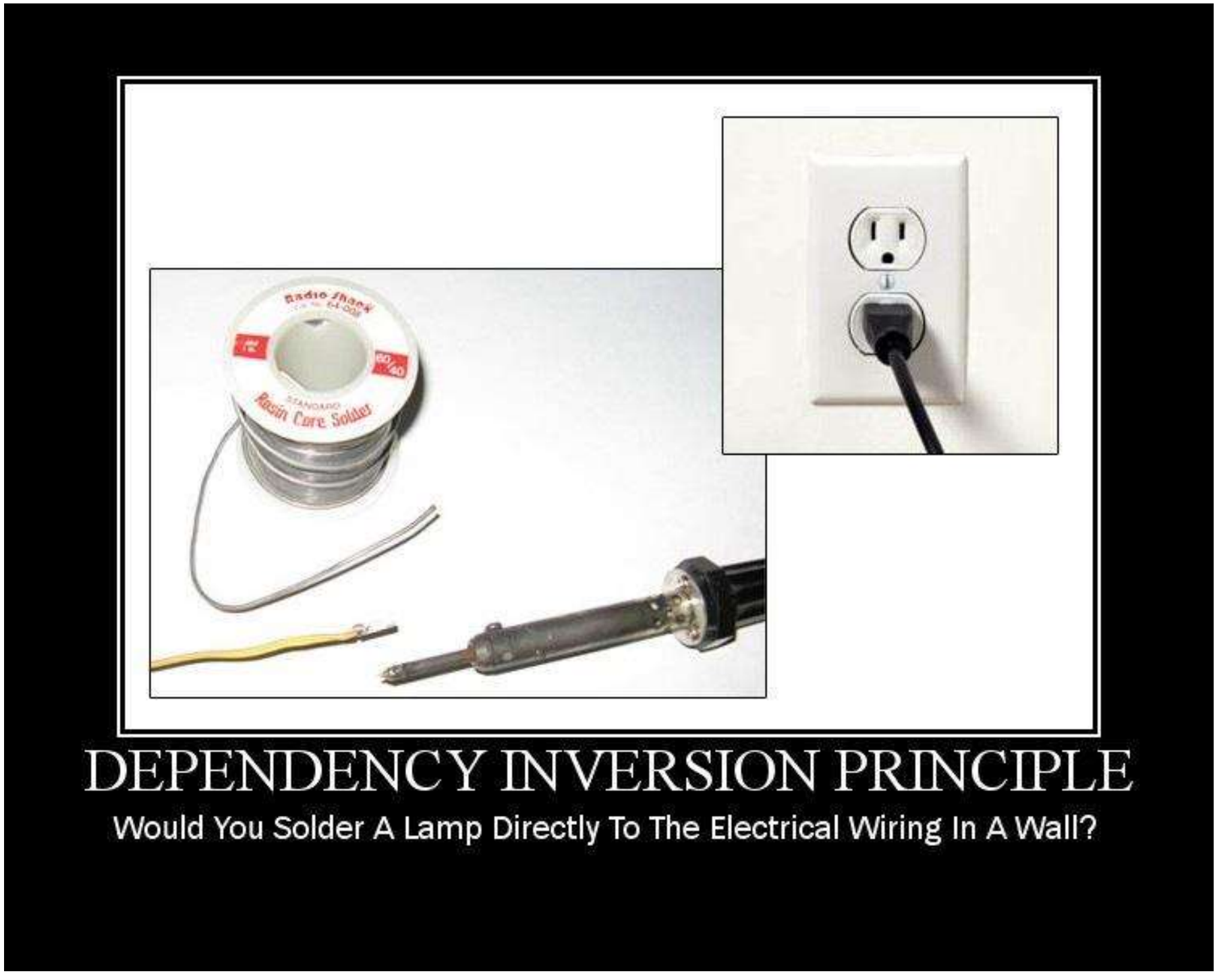
The Interface Segregation Principle (ISP) states that clients should not be forced to depend upon interface members they do not use. When we have non-cohesive interfaces, the ISP guides us to create multiple, smaller, cohesive interfaces.

```
public interface Messenger {
  askForCard();
  tellInvalidCard();
  askForPin();
  tellInvalidPin();
  tellCardWasSiezed();
  askForAccount();
  tellNotEnoughMoneyInAccount();
  tellAmountDeposited();
  tellBalance();
}
```

Interface Segregation Principle

```
public interface LoginMessenger {
  askForCard();
  tellInvalidCard();
  askForPin();
  tellInvalidPin();
}

public interface WithdrawalMessenger {
  tellNotEnoughMoneyInAccount();
  askForFeeConfirmation();
}

publc class EnglishMessenger implements LoginMessenger, WithdrawalMessenger {
```

Dependency Inversion Principle



Dependency Inversion Principle

The Dependency Inversion Principle (DIP) states that high-level modules should not depend upon low-level modules; they should depend on abstractions. Secondly, abstractions should not depend upon details; details should depend upon abstractions.

```
public interface Reader { char getchar(); }
public interface Writer { void putchar(char c)}

class CharCopier {

  void copy(Reader reader, Writer writer) {
    int c;
    while ((c = reader.getchar()) != EOF) {
      writer.putchar();
    }
  }
}

public Keyboard implements Reader {...}
```

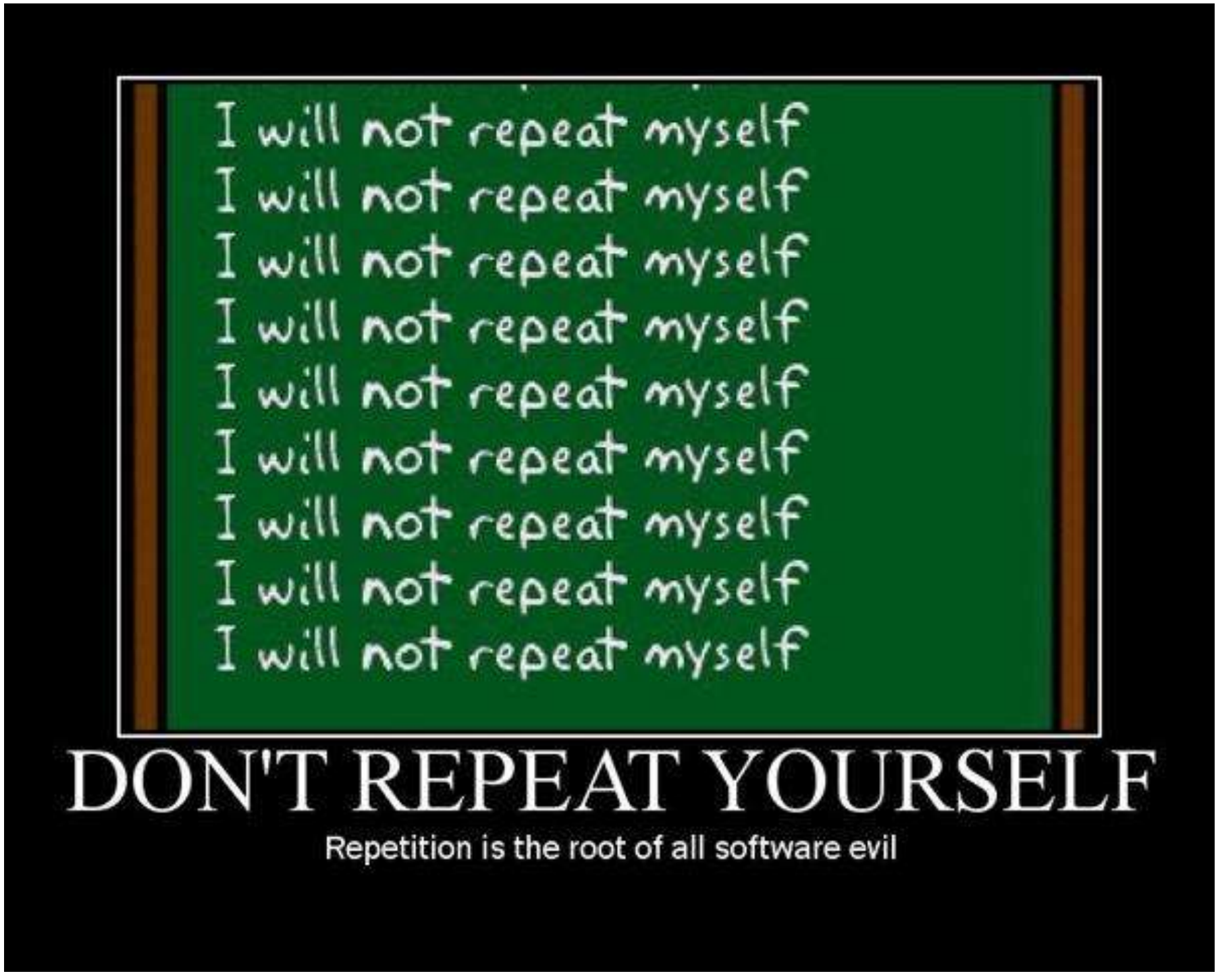
Miscellaneous Principles which are worth mentioning

- Program to Interface Not Implementation.
- Depend on Abstractions, Not Concrete classes.
- YAGNI (You aren’t going to need it. So don’t implement it).

- Least Knowledge Principle. (Only talk to your immediate friends. Classes that they inherit from, objects that they contain, objects passed by argument)

- Hollywood Principle. (Don’t call use, we will call you)
- Keep it Simple and Sweet / Stupid. (KISS)
- Minimize Coupling
- Maximize Cohesion
- Apply Design Pattern wherever possible.

Don’t Repeat Yourself



Thank You