Welcome

Design Principles

DevelopIntelligence
A PLURALSIGHT COMPANY

discussion

What is "Software Design"?

Design Principles

- Best practices or suggested practices to increase code quality.
- There are easy and hard ones.
- Programming language and frameworks can help you.
- *No rule without exception.*
- **Experience comes with time.**

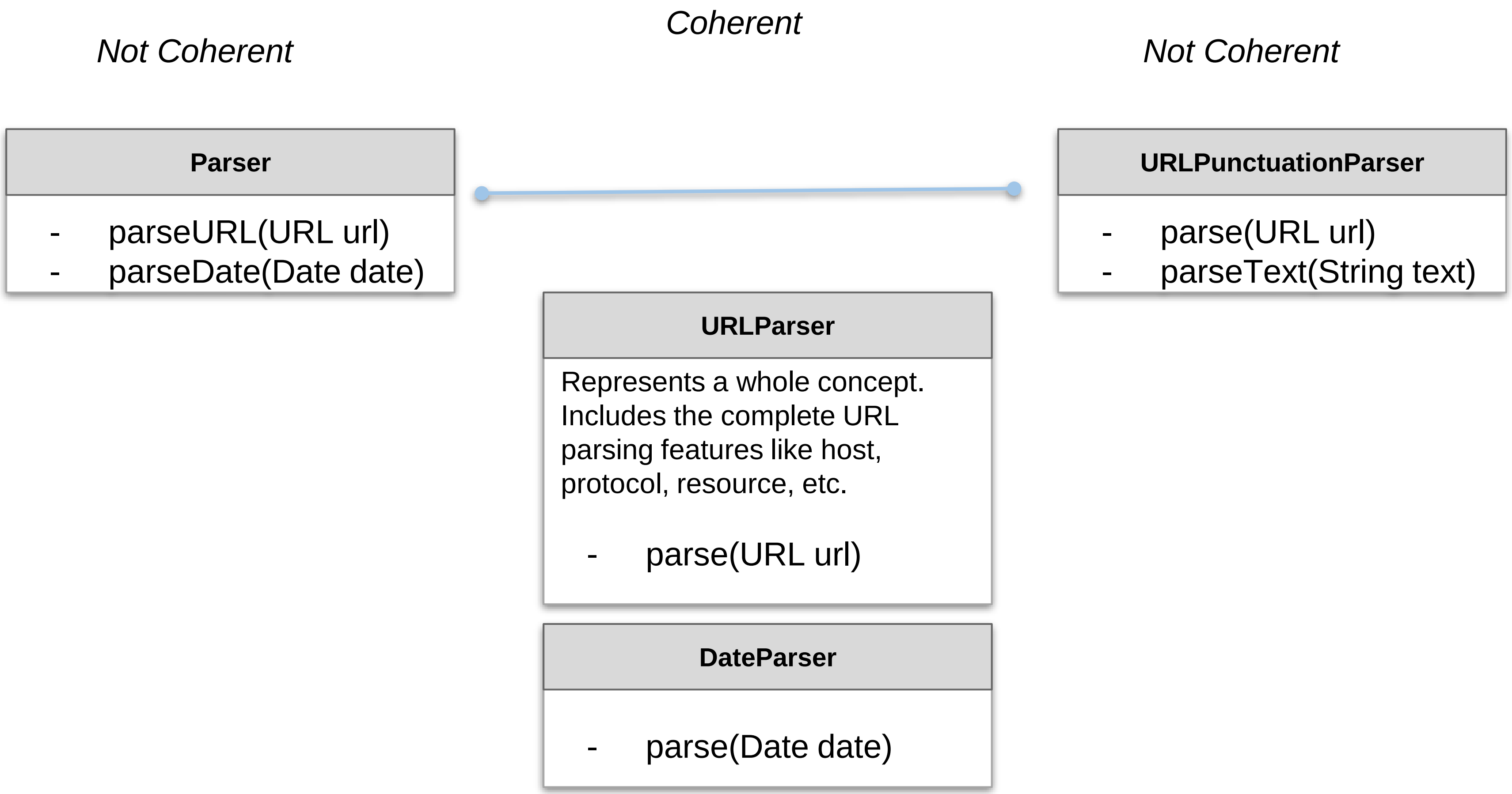
Coupling & Cohesion

These metrics describe how easy it will be to change the behavior of some code.

Elements are coupled if a change in one forces a change in another. For e.g., if two classes inherit from a common parent, then a change in one class might require a change in the other.

Think of a combo audio system: It’s tightly coupled because if we want to change from analog to digital radio, we must rebuild the whole system. If we assemble a system from independent components, it would have low coupling and we could just swap out the receiver. “Loosely” coupled features are easier to maintain.

An element’s *cohesion* is a measure of whether its responsibilities form a meaningful unit. For e.g., a class that parses both dates and URLs is not coherent, because they’re unrelated concepts.



Parser

- parseURL(URL url)
- parseDate(Date date)

URLParser

Represents a whole concept. Includes the complete URL parsing features like host, protocol, resource, etc.

- parse(URL url)

DateParser

- parse(Date date)

URLPunctuationParser

- parse(URL url)
- parseText(String text)

SOLID

The SOLID principles were introduced by Robert C. Martin in his 2000 paper “Design Principles and Design Patterns.” These concepts were later built upon by Michael Feathers, who introduced us to the SOLID acronym. And in the last 20 years, these five principles have revolutionized the world of object-oriented programming, changing the way that we write software.

Why SOLID?

Design principles encourage us to create more maintainable, understandable, and flexible software*. Consequently, *as our applications grow in size, we can reduce their complexity and save ourselves a lot of headaches further down the road!

Single Responsibility Principle

Based on the principle of *cohesion*, as described by Tom DeMarco in his book *Structured Analysis and System Specification (1979)*, and Meilir Page-Jones in *The Practical Guide to Structured Systems Design (1988)*.

“A class should have one and only one reason to change”

Open-Closed Principle

Bertrand Meyer coined the open-closed principle in 1988. It means simply we should write our modules so that they can be extended, without requiring them to be modified. In other words, we want to be able to change what the modules do, without changing the source code of the modules.

"Open for extension and closed for modification"

OCP violation in example code

```
public class Alarm
{
// …
        private Sensor sensor = new Sensor();
```

Want to use new type of sensor?
Must modify code; cannot extend it

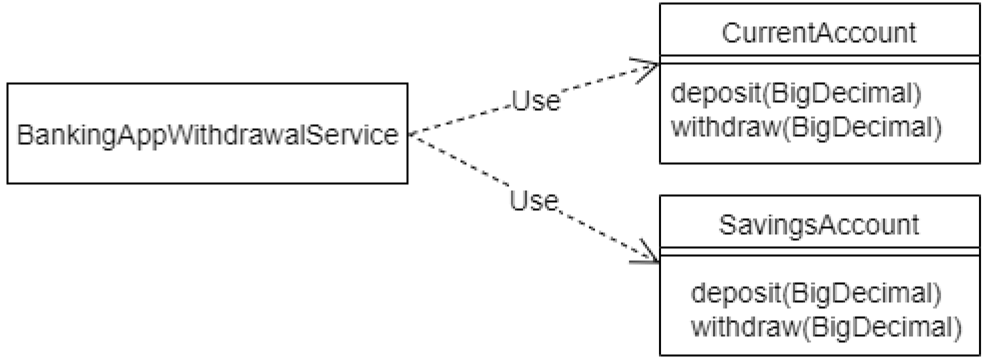
Liskov Substitution Principle

Introduced by Barbara Liskov in a 1987 conference keynote address titled *Data abstraction and hierarchy*

- Subclasses should substitute the parent class
- Any child type of a parent type should be able to stand in for that parent without things blowing up

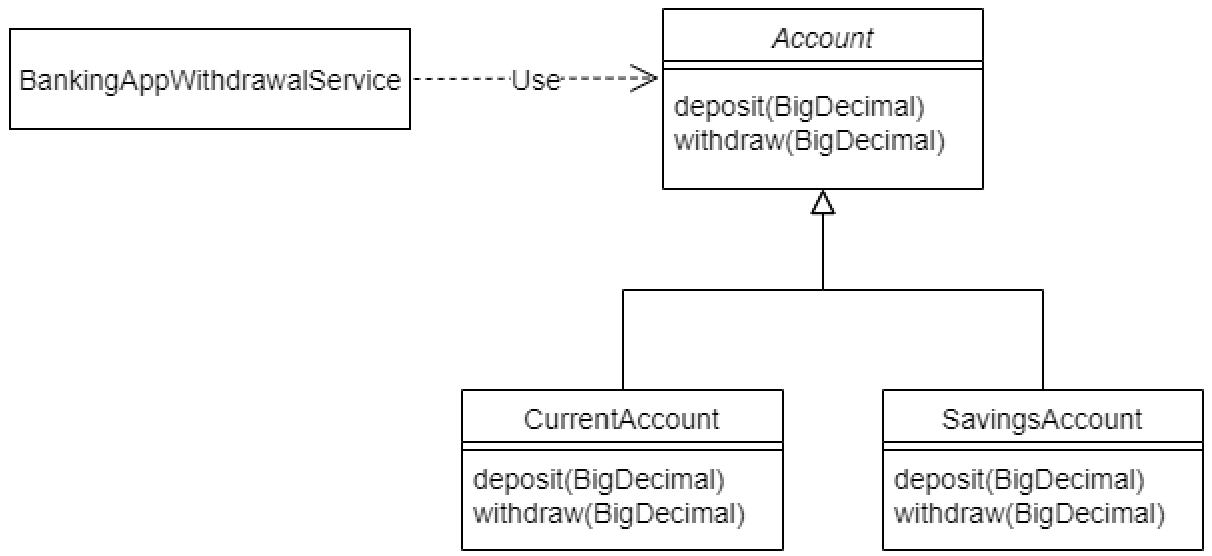
"Derived classes should be substitutable for their base classes"

Banking application



Our banking application supports two account types – “current” and “savings”. These are represented by the classes *CurrentAccount* and *SavingsAccount* respectively.

Liskov Substitution Principle



Using the Open/Closed Principle to Make the Code Extensible

A new account type

```java
public class FixedDepositAccount extends Account {
    // Overridden methods…
}
```

```java
    @Override
    protected void deposit(BigDecimal amount) {
        // Deposit into this account
    }

    @Override
    protected void withdraw(BigDecimal amount) {
        throw new UnsupportedOperationException("Withdrawals are not 
supported by FixedDepositAccount!!");
    }

  }
```

What went wrong ?

The `WithdrawalService` is a client of the `Account` class. It expects that both `Account` and its subtypes guarantee the behavior that the `Account` class has specified for its `withdraw` method

However, by not supporting the `withdraw` method, the `FixedDepositAccount` violates this method specification. Therefore, we cannot reliably substitute `FixedDepositAccount` for Account.

In other words, the `FixedDepositAccount` has violated the ***Liskov Substitution Principle***

Can't We Handle the Error in `BankingAppWithdrawalService`?

We could amend the design so that the client of `Account’s withdraw` method has to be aware of a possible error in calling it. However, this would mean that clients have to have special knowledge of unexpected subtype behavior. ***This starts to break the Open/Closed principle.***

In other words, for the Open/Closed Principle to work well, ***all subtypes must be substitutable for their supertype without ever having to modify the client code***. Adhering to the Liskov Substitution Principle ensures this substitutability.

When Is a Subtype Substitutable for Its Supertype?

A subtype doesn't automatically become substitutable for its supertype. **To be substitutable, the subtype must behave like its supertype**.

An object's behavior is the contract that its clients can rely on. The behavior is specified by the public methods and:

- any constraints placed on their inputs
- any state changes that the object goes through
- and the side effects from the execution of methods.

When Is a Subtype Substitutable for Its Supertype?

Subtyping in Java requires the base class's properties and methods are available in the subclass.

However, behavioral subtyping means that not only does a subtype provide all of the methods in the supertype, but it **must adhere to the behavioral specification of the supertype**. This ensures that any assumptions made by the clients about the supertype behavior are met by the subtype.

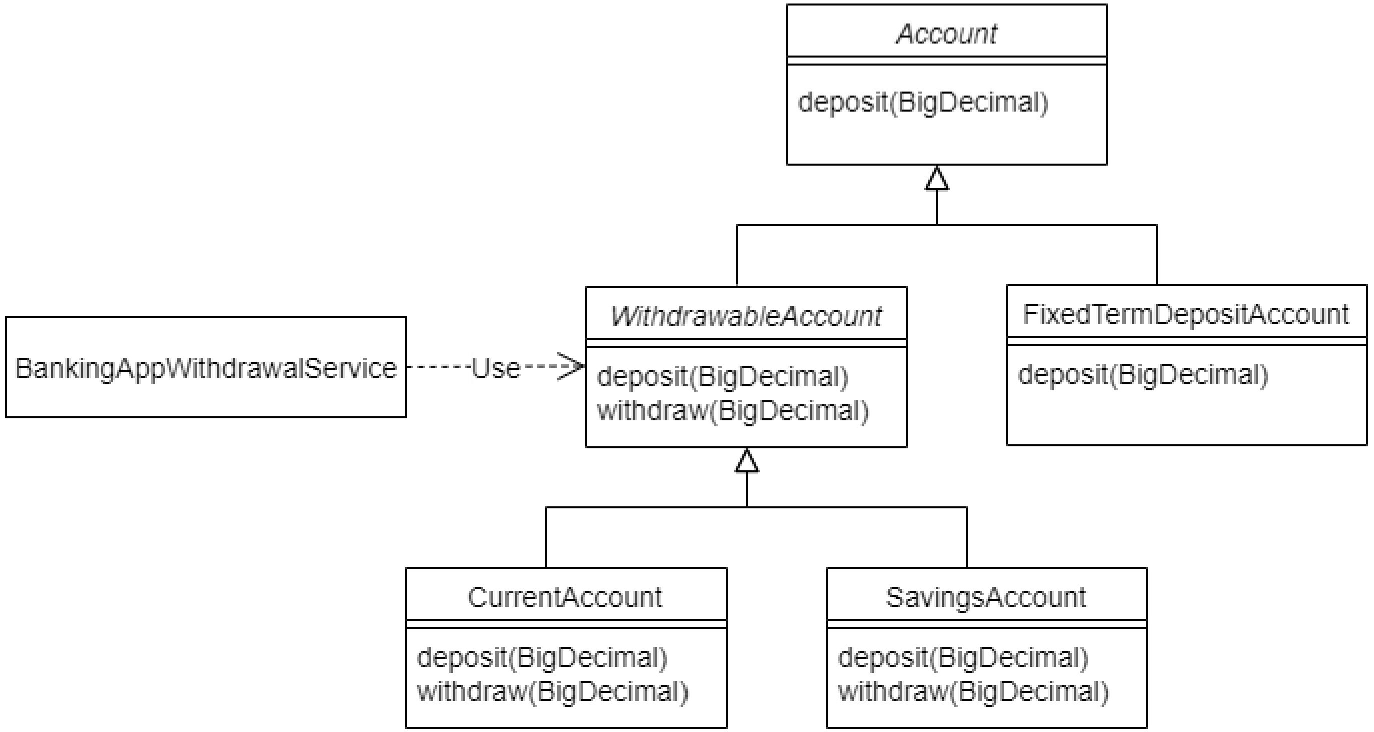
This is the additional constraint that the *Liskov Substitution Principle* brings to *object-oriented design*.

Root cause of the problem?

`FixedTermDepositAccount` is not a behavioral subtype of `Account`.

The design of `Account` incorrectly assumed that all Account types allow withdrawals. Consequently, all subtypes of *Account,* including `FixedTermDepositAccount` which doesn't support withdrawals, inherited the `withdraw` method.

Revised class diagram



Refactored *BankingAppWithdrawalService*

```
public class BankingAppWithdrawalService {
    private WithdrawableAccount withdrawableAccount;

    public BankingAppWithdrawalService(WithdrawableAccount withdrawableAccount) {
        this.withdrawableAccount = withdrawableAccount;
    }

    public void withdraw(BigDecimal amount) {
        withdrawableAccount.withdraw(amount);
    }

  }
```

Interface Segregation Principle

Robert C Martin while consulting for Xerox

Make fine grained interfaces that are client specific

The goal of this principle is to **reduce the side effects of using larger interfaces by breaking application interfaces into smaller ones**. It's similar to the *Single Responsibility Principle*, where each class or interface serves a single purpose.

“Clients should not be forced to depend upon interfaces that they do not use”

Example

```
public interface Payment {
    void initiatePayments();
    Object status();
    List<Object> getPayments();
}
```

Example

```
public class BankPayment implements Payment {
  @Override
  public void initiatePayments() {
    // ...
  }

  @Override
  public Object status() {
    // ...
  }
  @Override
  public List<Object> getPayments() {
    // ...
  }
}
```

Polluting the interface

Now, as we move ahead in time, and more features come in, there's a need to add a *LoanPayment* service. This service is also a kind of *Payment* but has a few more operations.

```
public interface Payment {

  // original methods
    ...
  void intiateLoanSettlement();
  void initiateRePayment();

 }
```

LoanPayment implementation

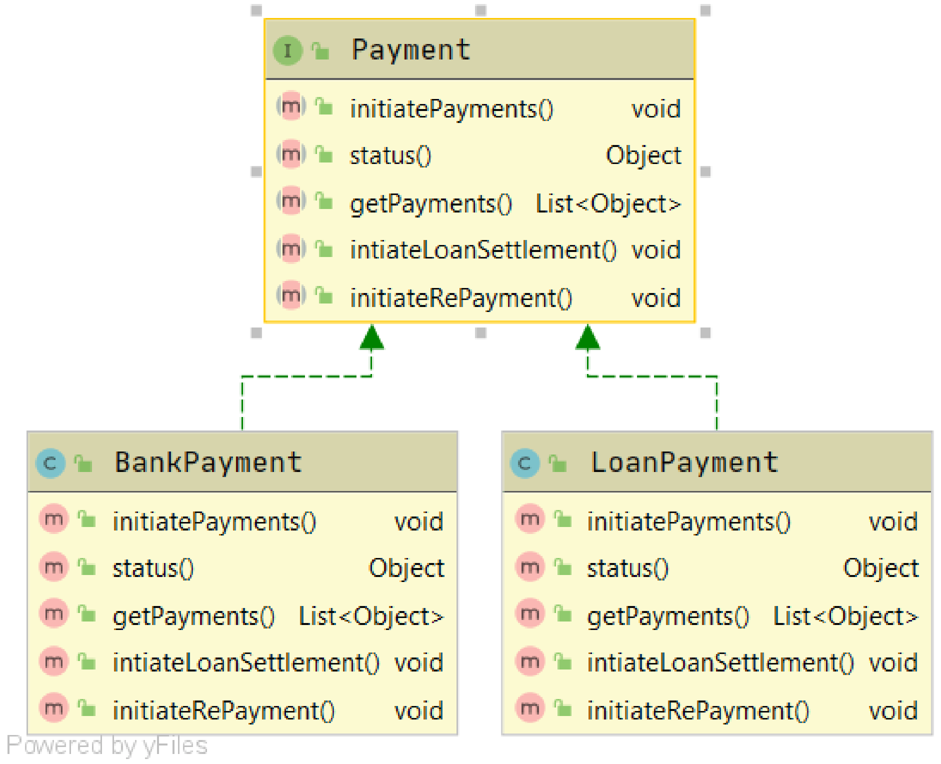
```
public class LoanPayment implements Payment {
  @Override
  public void initiatePayments() {
    throw new UnsupportedOperationException("This is not a bank payment");
  }

  @Override
  public Object status() { // ... }

  @Override
  public List<Object> getPayments() {// ... }

  @Override
  public void intiateLoanSettlement() { // ...}

  @Override
  public void initiateRePayment() {// ...}
}
```

Implementing unwanted functions could lead to many side effects. Here, the *LoanPayment* implementation class has to implement the *initiatePayments()* without any actual need for this.

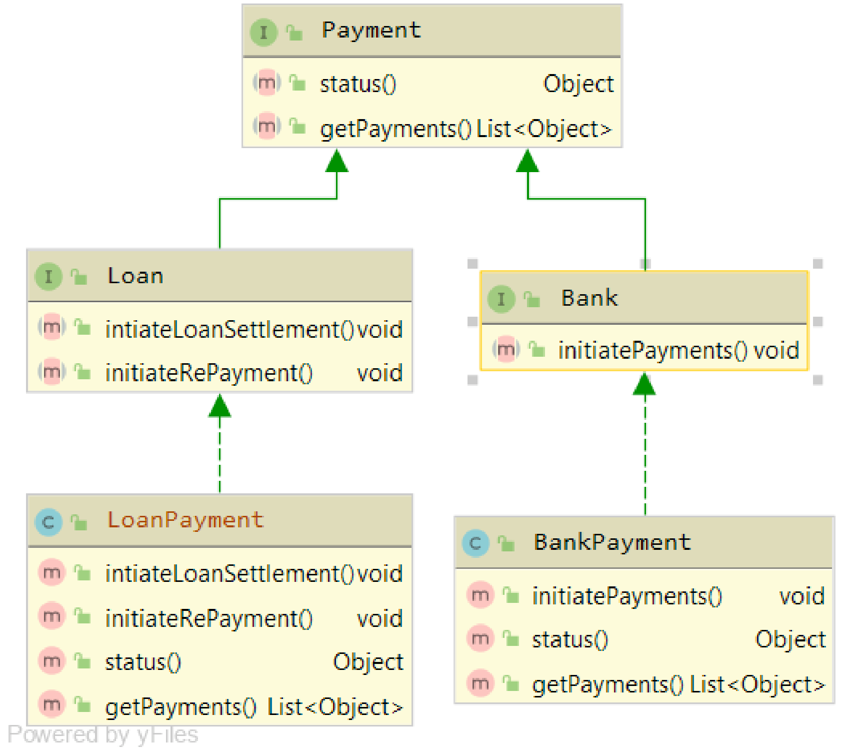
```java
public class BankPayment implements Payment {
  @Override
  public void initiatePayments() { // ...}

  @Override
  public Object status() { // ... }

  @Override
  public List<Object> getPayments() { // ...}

  @Override
  public void intiateLoanSettlement() {
    throw new UnsupportedOperationException("This is not a loan payment");
  }

  @Override
  public void initiateRePayment() {
    throw new UnsupportedOperationException("This is not a loan payment");
  }
}
```

Implementing unwanted functions could lead to many side effects. Here, the *BankPayment* implementation class has to implement the *initiateLoanSettlement()* without any actual need for this.

Applying the principle



Applying the principle



LoanPayment implementation

```
public class LoanPayment implements Payment {
  @Override
  public void initiatePayments() {
    throw new UnsupportedOperationException("This is not a bank payment");
  }

  @Override
  public Object status() { // ... }

  @Override
  public List<Object> getPayments() {// ... }

  @Override
  public void intiateLoanSettlement() { // ...}

  @Override
  public void initiateRePayment() {// ...}
}
```

Dependency Inversion Principle

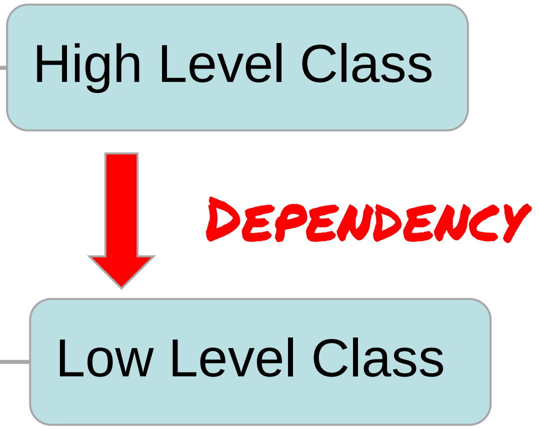
The dependency inversion principle was postulated by Robert C. Martin

- 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.

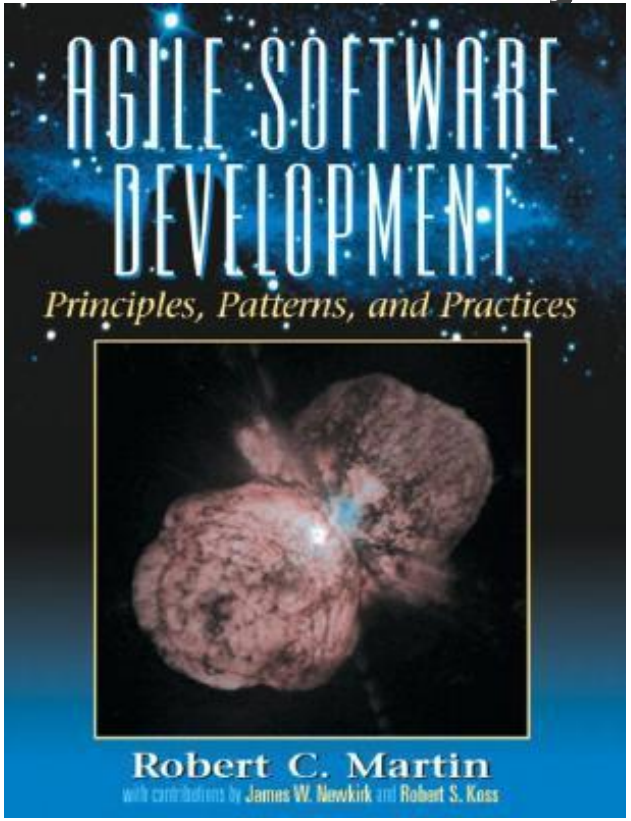
“Depend on abstractions, not on concretions.”

DIP violation in example code

```
public class Alarm
{
// …
      private Sensor sensor = new Sensor();
```



~~Recommended~~ Mandatory reading


AGILE SOFTWARE DEVELOPMENT
Principles, Patterns, and Practices
Robert C. Martin
with contributions by James W. Newkirk and Robert S. Koss

User-centered design

User-Centered Design

- An approach to designing products, systems, and services that places the needs, preferences, and behaviors of the users at the forefront of the design process.
- The goal of user-centered design is to create products that are effective, efficient, and satisfying for the users, ultimately leading to better usability and user experience.

User-Centered Design - key principles

- **User Involvement**: Users are actively involved throughout the design and development process. Their input, feedback, and insights are integral to making design decisions.
- **Iterative Process**: UCD is an iterative process, with frequent cycles of design, testing, and refinement. This allows for continuous improvement based on user feedback.
- **Focus on User Needs**: Design decisions are driven by a deep understanding of user needs, goals, and tasks. Research methods like surveys, interviews, and observations are used to gather this information.

User-Centered Design - key principles

- **Usability and Accessibility**: UCD emphasizes creating products that are easy to use and accessible to a diverse range of users, including those with disabilities.
- **Prototyping and Testing**: Prototypes are created early in the design process and are tested with real users to identify issues and gather feedback. This helps validate design decisions and refine the product.
- **Consistency and Standards**: UCD aims to create a consistent and predictable user experience by following established design principles and best practices.

User-Centered Design - key principles

- **Clear Communication**: UCD encourages clear and effective communication among design teams, stakeholders, and users to ensure everyone's needs and expectations are addressed.
- **Context of Use**: Designs are developed in consideration of the context in which the product will be used, taking into account the environment, goals, and tasks of the users.
- **Flexibility and Adaptability**: UCD recognizes that user needs and technology can change over time. Designs should be flexible and adaptable to accommodate these changes.
- **Evidence-Based Design**: Design decisions are based on evidence and data gathered from user research, testing, and analysis rather than assumptions or intuition.

User-Centered Design

- Overall, user-centered design aims to create products that provide meaningful and valuable experiences for users, leading to increased user satisfaction, engagement, and loyalty.
- It is widely used in various fields, including software development, web design, product design, and service design.

Design Patterns

Design patterns

- Design patterns are reusable solutions to common problems that arise during software design and development.
- They are general, proven solutions that provide a blueprint for solving specific design and architectural challenges in a consistent and effective way.
- Design patterns help developers create more maintainable, modular, and scalable software by providing well-defined templates for structuring code and interactions between components.
- Design patterns can be categorized into three main types.

Creational patterns

These patterns deal with object creation mechanisms, trying to create objects in a manner suitable to the situation. Common creational patterns include:

- **Singleton**: Ensures a class has only one instance and provides a global point of access to it.
- **Factory Method**: Defines an interface for creating objects but allows subclasses to alter the type of objects that will be created.
- **Abstract Factory**: Provides an interface for creating families of related or dependent objects without specifying their concrete classes.
- **Builder**: Separates the construction of a complex object from its representation, allowing the same construction process to create various representations.

Structural patterns

These patterns focus on how objects are composed to form larger structures while keeping the structures flexible and efficient. Common structural patterns include:

- **Adapter**: Allows objects with incompatible interfaces to work together by providing a common interface.
- **Decorator**: Adds behavior to objects dynamically, without altering their existing code.
- **Facade**: Provides a simplified interface to a set of interfaces in a subsystem.
- **Composite**: Composes objects into tree structures to represent part-whole hierarchies.
- **Proxy**: Provides a surrogate or placeholder for another object to control its access.

Behavioral patterns

These patterns deal with communication between objects and responsibilities among objects. Common behavioral patterns include:

- **Observer**: Defines a dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.
- **Strategy**: Defines a family of algorithms, encapsulates each one, and makes them interchangeable.
- **Command**: Turns a request into a stand-alone object, allowing parameterization of clients with different requests.
- **State**: Allows an object to change its behavior when its internal state changes.
- **Iterator**: Provides a way to access elements of a collection without exposing its underlying representation.

Design patterns

- Design patterns provide a shared vocabulary for developers to communicate and discuss solutions to common design problems.
- They are not specific pieces of code but rather high-level concepts that can be implemented in different programming languages and frameworks.
- By using design patterns, developers can benefit from proven solutions, reduce development time, improve code quality, and make their software more maintainable and adaptable.

KISS

KISS

- The KISS principle, which stands for "Keep It Simple, Stupid," is a design principle that emphasizes simplicity and clarity in various fields, including software development, design, engineering, and management.
- The principle suggests that simplicity should be a key goal and that unnecessary complexity should be avoided.

KISS

- The core idea of the KISS principle is to favor straightforward, uncomplicated solutions over convoluted or intricate ones.
- This approach is based on the belief that simpler designs are easier to understand, maintain, and troubleshoot.
- When applied, the KISS principle can lead to more efficient processes, reduced chances of errors, and improved user experiences.
- In software development, the KISS principle encourages developers to write clean, concise, and easily understandable code.

KISS

- While the KISS principle advocates for simplicity, it doesn't mean ignoring necessary complexities or sacrificing functionality.
- Rather, it encourages developers to strike a balance between meeting the requirements and avoiding unnecessary intricacies that can lead to confusion, bugs, and maintenance challenges.

System Architecture

System architecture

- System architecture refers to the high-level structure and organization of a complex software or hardware system.
- It encompasses the design, layout, components, interactions, and relationships between various parts of a system to achieve specific goals and functionalities.
- System architecture serves as a blueprint that guides the development, implementation, and maintenance of the system.

System architecture: key aspects

- **Components and Modules**: Identifying the major components, modules, or subsystems that make up the system. These components could include software applications, databases, servers, communication interfaces, and more.
- **Interactions and Interfaces**: Defining how the different components interact with each other. This includes specifying communication protocols, data exchange formats, and APIs (Application Programming Interfaces).
- **Data Flow and Processing**: Describing how data flows through the system, how it is processed, stored, and retrieved. This involves designing data structures, databases, and data synchronization methods.

System architecture: key aspects

- **Scalability and Performance**: Planning for the system's ability to handle increased loads and user interactions. This may involve considerations for load balancing, caching, and distributed processing.
- **Security and Privacy**: Incorporating measures to protect the system and its data from unauthorized access, breaches, and vulnerabilities.
- **Reliability and Availability**: Designing the system to be reliable and available, even in the face of failures. This may involve redundancy, failover mechanisms, and backup strategies.

System architecture: key aspects

- **Deployment and Infrastructure**: Determining the hardware, network, and infrastructure requirements for deploying the system. This includes considerations for hosting, cloud services, and server configurations.
- **User Experience**: Ensuring that the architecture supports a positive user experience by designing intuitive user interfaces, responsive design, and efficient user interactions.
- **Maintenance and Upgrades**: Planning for the ongoing maintenance, updates, and upgrades of the system. This involves designing the architecture to be flexible and adaptable to changes.
- **Cost and Resources**: Considering the budget, resources, and constraints for developing and maintaining the system.

System architecture

- System architecture can take different forms based on the specific domain and requirements.
- It can range from software systems like web applications and mobile apps to hardware systems like computer networks, embedded systems, and IoT devices.
- Effective system architecture is crucial for building systems that are scalable, maintainable, and capable of meeting user needs while aligning with business goals.

Thank you!

DevelopIntelligence
A PLURALSIGHT COMPANY