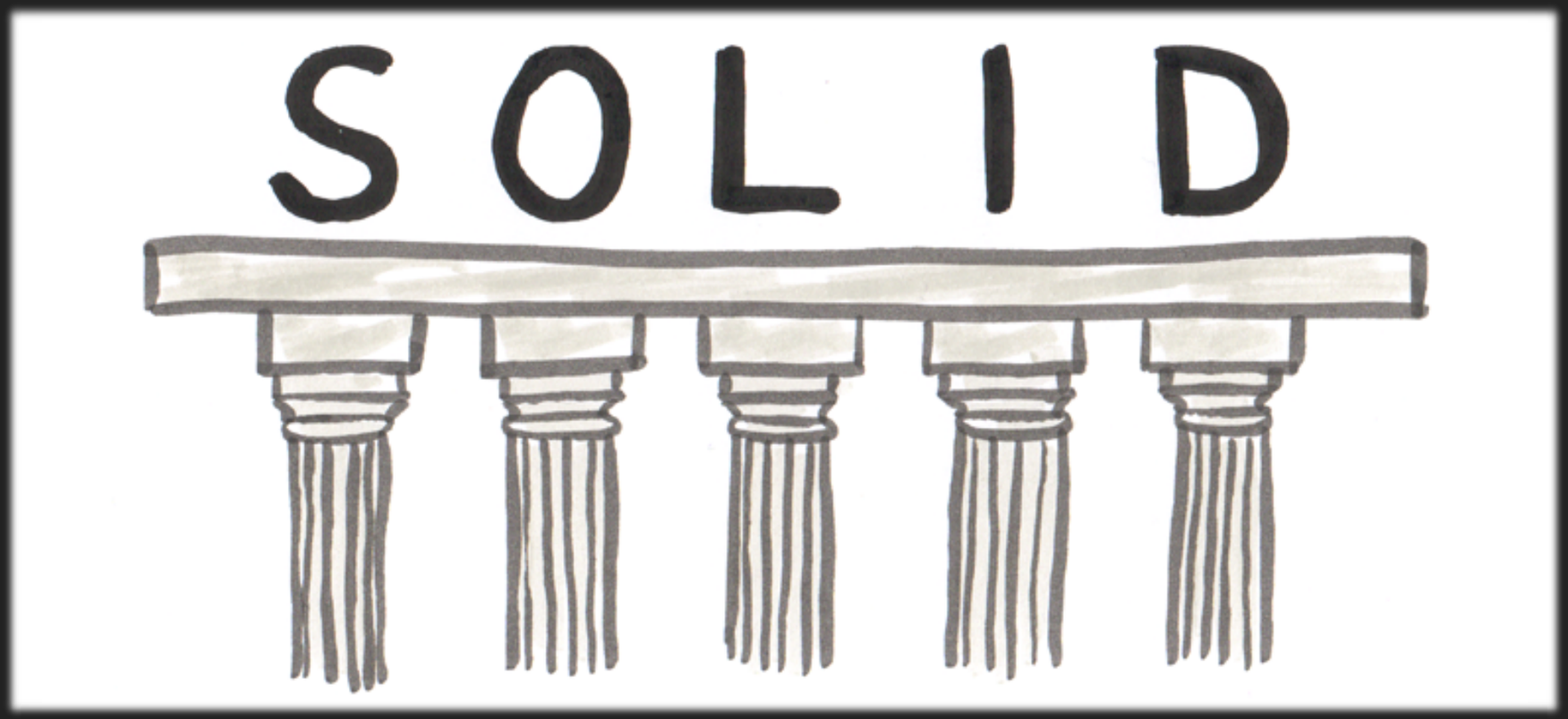
The SOLID principles of Software Design

Presenter:
Pavel Sulimau

HISTORY

- In the early 2000s Robert C. Martin (“Uncle Bob”) came up with a list of 11 principles of good Object Oriented Design (OOD).
- The first five principles are principles of what makes good class design.
- The five principles are what have become known by the acronym “SOLID” which Michael Feathers helped coin.



THE SOLID ACRONYM

- Single Responsibility Principle
- Open/Closed Principle
- Liskov Substitution Principle
- Interface Segregation Principle
- Dependency Inversion Principle

What for?

TO DEVELOP SOFTWARE THAT IS EASY TO MAINTAIN AND EXTEND

MOTIVATION: MAINTAINABILITY & EXTENSIBILITY

- Maintainability refers to the degree to which a code base handles updates without generating regression and new issues.
- Usually the notion of adding features (extensibility) is included in the idea of maintenance.

MOTIVATION: IMPORTANCE OF MAINTAINABILITY

> All programming is maintenance programming because you are rarely writing original code. It's only the first 10 minutes that the code's original, when you type it in the first time. That's it.

Dave Thomas and Andy Hunt

SINGLE RESPONSIBILITY PRINCIPLE

> There should never be more than one reason for a class to change (more than one responsibility).

Robert C. Martin

Responsibilities are axes of changes:

- Requirements typically map to responsibilities.
- The more responsibilities a class has,
  the more likelihood of a change in that class.
- The more code the change affects,
  the more likely the change will introduce errors.

Examples of responsibilities:

- persistence;
- validation;
- notification;
- logging;
- parsing;
- mapping;
- object construction;
- serialization/deserialization;
- business logic;
- interaction with a user.

SINGLE RESPONSIBILITY PRINCIPLE

Demo

OPEN/CLOSED PRINCIPLE

Software entities should be open for extension, but closed for modification.

Bertrand Meyer

"Open for extension". This means that the behavior of the module can be extended.

"Closed for modification." Extending the behavior of a module does not result in changes to the source or binary code of the module.

Robert C. Martin

Approaches to achieve OCP:

- Parameters.
  Allow client to control behavior specifics via parameter (combined with lambda can be a very powerful approach).
- Inheritance/Template Method pattern.
- Composition/Strategy pattern.
  Client code depends on abstraction.
  Provides “plug in” model.
  Implementation utilize inheritance, clients utilize composition.

Demo

When do we apply OCP:

- Experience will tell you.
- Otherwise: “Fool me once, shame on you; fool me twice shame on me”:
  Don’t apply OCP at first.
  If the module changes once, accept it.
  If it changes a second time, refactor to achieve OCP.
- Remember TANSTAAFL (There Ain’t No Such Thing As A Free Lunch):
  OCP adds complexity to design.
  No design can be closed against all changes.

LISKOV SUBSTITUTION PRINCIPLE

> Derived classes must be substitutable for their base classes.

Robert C. Martin

> A subclass should behave in such a way that it will not cause problems when used instead of superclass.

Chris Klug

LSP rules on method signatures:

- Contravariance of method arguments in the subtype.
- Covariance of return types in the subtype.
- No new exceptions should be thrown by methods of the subtype, except where those exceptions are themselves subtypes of the exceptions thrown by the methods of the supertype.

LSP behaviors conditions:

- Preconditions cannot be strengthened in a subtype.
- Postconditions cannot be weakened in a subtype.
- Invariants of a supertype must be preserved in a subtype.

LISKOV SUBSTITUTION PRINCIPLE

Demo

INTERFACE SEGREGATION PRINCIPLE

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

Robert C. Martin

ISP smells:

- Unimplemented interface methods (remember, it violates LSP).
- Client references a class but uses only a small portion of if.

When do we fix ISP:

- Once there is pain.
  If there is no pain, there is no problem to address.
- If your find yourself depending of a“fat” interface of your own.
  Create a smaller interface with just what you need.
  Have the “fat” interface implement your new interface.
  Reference the new interface with you code.
- If you find “fat” interfaces problematic but you do not own them.
  Create a smaller interface with just what you need.
  Implement this interface using a Facade pattern.

INTERFACE SEGREGATION PRINCIPLE

Demo

DEPENDENCY INVERSION PRINCIPLE

> **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.**

Robert C. Martin

What are dependencies:

- Framework.
- Third party libraries.
- Database.
- File system.
- Email.
- Web service.
- System resources (clock).
- Configuration.
- The new keyword.
- Static methods.
- Random.

Achieve DIP by

- designing components whose external dependencies are expressed in terms of an interface for which an implementation must be provided by the consumer of the component. In other words, the defined interfaces express what is needed by the component, not how you use the component (e.g. "INeedSomething", not “IDoSomething").

What DIP does not refer to is

- the simple practice of abstracting dependencies through the use of interfaces e.g.
  MyService → [ILogger ⇐ Logger].

While this decouples a component from the specific implementation detail of the dependency, it does not invert the relationship between the consumer and dependency e.g.
[MyService → IMyServiceLogger] ⇐ Logger.

DEPENDENCY INVERSION PRINCIPLE

Demo

CONCLUSION

- Object modeling is neither easy not is an exact science. The principles are not silver bullets, they exist for the most part to show you the way to go – to give guidance and possibly to point you in the right direction.
- Be diligent in applying the principles, every one of them has its price. You should understand which problem a principle solves and whether the problem is relevant to you in your circumstances.