OO design principles

Jeszenszky, Péter
University of Debrecen, Faculty of Informatics
jeszenszky.peter@inf.unideb.hu

Kocsis, Gergely (English version)
University of Debrecen, Faculty of Informatics
kocsis.gergely@inf.unideb.hu

Last modified: 10.05.2017

PMD

- A static source code analyzer (programming language: Java, license: Apache License 2.0) https://pmd.github.io/
  - Supported programming languages: Java, JavaScript, Apex, PLSQL, Apache Velocity, XML, XSL
- Plugins:
  - *Apache Maven PMD Plugin* https://maven.apache.org/plugins/maven-pmd-plugin/
  - *Gradle: The PMD Plugin* https://docs.gradle.org/current/userguide/pmd_plugin.html
  - *Eclipse PMD Plug-in* http://acanda.github.io/eclipse-pmd/
  - *SQE (NetBeans)* https://github.com/sqe-team/sqe

DRY (1)

- *Don't Repeat Yourself*
  - *„Every piece of knowledge must have a single, unambiguous, authoritative representation within a system.”*
- Reference:
  - Andrew Hunt, David Thomas. *The Pragmatic Programmer: From Journeyman to Master*. Addison-Wesley, 1999.
- The contrary: WET.
  - *We enjoy typing, write everything twice*, *we edit terribly*, …

DRY (2)

- Types of duplication:
  - ***Imposed duplication***: the developers feel that they do not have other choice than duplication, since the environment requires it.
  - ***Inadvertent duplication***: developers do not realize that they are duplicating information.
  - ***Impatient duplication***: the reason is laziness. Duplication seems to be the more easy way of solving a problem.
  - ***Interdeveloper duplication***: more developers in a group or in different groups duplicate information.
- Related concept: *code duplication*

DRY (3)

- PMD support: *Copy/Paste Detector* (CPD)
  - *Finding duplicated code* https://pmd.github.io/pmd-5.5.4/usage/cpd-usage.html
  - Supported programming languages: C++, C#, ECMAScript (JavaScript), Fortran, Go, Java, Groovy, JSP, Matlab, Objective-C, Perl, PHP, PL/SQL, Python, Ruby, Scala, Swift
    - See: https://pmd.github.io/pmd-5.5.4/usage/cpd-usage.html#Supported_Languages

KISS

- *Keep it simple, stupid*
  - 1960's American navy.
  - By Kelly Johnson (1910–1990) aeronautical engineer.

Law of Demeter (1)

- *Law of Demeter*:
  - Ian M. Holland, Karl J. Lieberherr. *Assuring Good Style for Object-Oriented Programs*. IEEE Software, vol. 6, no 5, pp. 38– 48, 1989.
- *In other words: Don't Talk to Strangers*
  - Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. 3rd ed. Prentice Hall, 2005.

Law of Demeter (2)

- Limits the message passing structure of methods.
  - For every method only message passing for a limited set of objects is allowed.
  - The aim is to structure and reduce the amount of dependency between classes.

Law of Demeter (3)

- *Class form*:
  - The M method of a C class can use fields and methods of only the following classes and super classes
    - C
    - Classes of the fields of *C*
    - Classes of the parameters of *M*
    - Classes whose constructors are called from M
    - Classes of global variables used in M
- Can be checked in run-time.

Law of Demeter (4)

- The use of it increases maintainability and simplicity of the code.
  - Limits the set of callable methods thus it reduces the coupling of methods.
  - Enforcing information closure: the inner structure of an object is known only by itself.

Law of Demeter (5)

- Example:
  - All four hello() calls are allowed in the following code:

```
class C {
  private B b;
  void m(A a) {
    b.hello();
    a.hello();
    Singleton.INSTANCE.hello()
    new Z().hello();
  }
}
```

  - `a.x.hello()` is not allowed however.
  - See: Yegor Bugayenko. *The Law of Demeter Doesn't Mean One Dot*. 2016. http://www.yegor256.com/2016/07/18/law-of-demeter.html

Law of Demeter (6)

- Related PMD rule set:
  - *Coupling (java)* https://pmd.github.io/pmd-5.5.4/pmd-java/rules/java/coupling.html
    - See `LawOfDemeter` rule: https://pmd.github.io/pmd-5.5.4/pmd-java/rules/java/coupling.html#LawOfDemeter

Separation of Concerns (1)

- SoC – *Separation of Concerns*:
  - A software system is to be designed in a way that ensures that each component has its given role and these roles do not overlap each other..
    - Examples: model-view-controller (MVC) architectural pattern, TCP/IP protocol stack, HTML + CSS + JavaScript, …
  - Reference:
    - Ian Sommerville. *Software Engineering*. 10th ed. Pearson Education, 2015. http://iansommerville.com/software-engineering-book/

Separation of Concerns (2)

- Components of the programs (classes, methods...) should do only one thing.
  - The separated components can be edited without taking other components into account.
    - E.g. a part of the program can be understood without understanding other parts of the program.
    - When modifications are required, only a low number of components are to be affected.
    - Parts of the program can be developed and reused separately.

Separation of Concerns (3)

- A concern is something that may be important or interesting for a component or for a group of components.
  - Example: performance, providing a given functionality, maintainability, …
- They reflect the system requirements.

Separation of Concerns (4)

- Types of concerns:
  - ***Core concerns***: concerns related to the main functionalities of the system.
  - ***Secondary concerns***: e.g. functional concerns needed to fulfill non-functional requirements of the system.
  - ***Cross-cutting concerns***: Concerns describing system wide requirements.
    - Secondary concerns may be cross-cutting however thy o not always cross-cut the whole system
    - Example: security, logging.

Separation of Concerns (5)

- Related programming paradigm: AOP – *aspect-oriented programming*
  - Example: *AspectJ* https://eclipse.org/aspectj/
    - Aspect-oriented extension of the Java language.

GRASP (1)

- **GRASP** – ***General Responsibility Assignment Software Patterns***
  - Craig Larman. *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development*. 3rd ed. Prentice Hall, 2005.
- Design patterns expressing general responsibility assignment concepts.
  - Study tools that help to understand and apply oo planning.

GRASP (2)

- Responsibility: a contract of a classifier (UML).
- Types of responsibilities:
  - *Doing* sg.
  - *Knowing* sg.
- The assignment of responsibilities to classes is done during the design.

GRASP (3)

- 9 design patterns and concepts:
  - ***Information Expert***
  - ***Creator***
  - ***Low Coupling***
  - ***High Cohesion***
  - ***Controller***
  - ***Polymorphism***
  - ***Pure Fabrication***
  - ***Indirection***
  - ***Protected Variations***

GRASP (5)

- Pattern template:
  - **Name**:
  - **Problem**:
  - **Example**:
  - **Description**:
  - **Contraindications**:
  - **Advantages**:
  - **Background**:
  - **Related patterns**:

GRASP patterns (1)

- **Information expert**:
  - The responsibility is assigned to the information expert i.e. to the class that has the information to realize the responsibility.
    - Example: which class' responsibility is it to know the total cost of an order in a selling system?

GRASP patterns (2)

- **Creator**:
  - The responsibility of instantiating an instance of class A is for class B if something from the followings is true::
    - Instances *of* B aggregates A objects
    - Instances *of* B contains A objects
    - *Instances of B records A instances*
    - *Instances of B* closely uses *A instances*
    - Instances of B have the initializing information for instances of A and pass it on creation.
  - *B* is the creator of A instances.

GRASP patterns (3)

- **Low coupling**:
  - Coupling measures how much a component is connected to other elements, how much information it has about them, or how much it depends on them.
  - Responsibilities are to be assigned to components so that their coupling remain low.
    - The components are e.g. classes, subsystems, systems etc.
    - A low coupled components does not depend on too many other components, where the meaning of „too many” is based on the environment.
    - A high coupled class depends on many others. These class are to be avoided. They have the following problems:
      - Changes in the related classes imply the need of changes in the class.
      - It is more hard to understand them without the others.
      - It is more hard to reuse them since all the dependent classes are also needed for that.

GRASP patterns (4)

- **High cohesion**:
  - Cohesion measures how much the responsibilities of a given component connect to each other.
  - Try to keep high cohesion while assigning the responsibilities.
    - Possible components are classes, sub-systems, systems, etc.
    - The cohesion of a components is high if its responsibilities are closely connected and it does not do too much work.
    - Low cohesion components do a lot of not related things. These classes are to be avoided. They have the following problems:
      - They are hard to understand
      - It is hard to reuse them
      - It is hard to maintain them

GRASP patterns (5)

- **Controller**:
  - The responsibility of the control of system events are assigned to a class that
    - Describes the whole system, sub-system or tool
    - Or describes a scenario in which the event happens
  - The controller is a non-user interface object.

GRASP patterns (6)

- **Polymorphism**:
  - Responsibility of defining the variation of behaviors based on type is assigned to the type for which this variation happens. This is achieved using polymorphic operations. The user of the type should use polymorphic operations instead of explicit branching based on type.
- **Pure Fabrication**:
  - A set of closely related responsibilities is assigned to a class that realizes not a domain specific part of the problem, but is created for low coupling, high cohesion and reusability.
    - Example: DAO – Data Access Object classes

GRASP patterns (7)

- **Indirection**:
  - Too high coupling between objects can be lowered by the use of a central object that relays between them.
- **Protected Variations**:
  - Wrap the focus of instability with an interface and use polymorphism to create various implementations of this interface.

GoF principles (1)

- Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1994.

GoF principles (2)

- ***„Program to an interface, not an implementation."***
- See creational design patterns!

GoF principles (3)

- ***„Favor object composition over class inheritance."***
- The two most used ways of reuse are in object oriented systems::
  - inheritance (white-box reuse)
  - Object composition (black-box reuse)

GoF principles (4)

- Advantages of inheritance:
  - It is done in runtime. It is easy to use since it is supported by the programming language by default.
  - It simplifies the modification of the reused realization as well. If a class overrides some operations, the subclasses will also use this version unless they call explicitly the previous one.

GoF principles (5)

- Disadvantages if inheritance
  - The realizations inherit from the super class cannot be modified in runtime since it was already done at compilation.
  - Super classes usually specify the physical structure of the subclasses. Since inheritance allows subclasses to see the implementation of the super class it is often said that inheritance breaks the rules of encapsulation. The smallest modification of the super class affects all the subclasses as well.
  - Implementation dependencies may lead to problems at the time of reuse. If the inherited implementation is not proper for our needs we have to rewrite or substitute it with something else.

GoF principles (6)

- Object composition is done dynamically at runtime, through objects that get references for other objects.
- For the composition it is required that the objects know the interfaces of each other. For this it is required to plan the interfaces carefully.

GoF principles (7)

- Design patterns using object composition:
  - Structural patterns: (object) adapter, bridge, composite, decorator, facade, flyweight, proxy.
  - Behavioral object patterns: chain of responsibility, command, iterator, mediator, memento, observer, state, strategy, visitor.

GoF principles (8)

- Advantages of object composition
  - Since the objects can be reached only through their interfaces, encapsulation is not violated.
  - All objects can be replaced in runtime while their types are the same.
  - Since the implementation of objects is done through interfaces the amount of implementation dependency is reduced.
  - Contrary to inheritance it helps encapsulation.
  - The classes and class hierarchies remain small so it is less probable that they grow to be unmanageable.

GoF principles (9)

- Disadvantages of object composition
  - In case of object composition we will have more objects (even though we have less classes) and the behavior of the system will depend on the connections between them. (And not on one single class).

SOLID (1)

- Robert C. Martin's („Uncle Bob”) programming and design principles.
  - Homepage: http://cleancoder.com/
  - Uncle Bob. *Getting a SOLID start*. 2009. https://sites.google.com/site/unclebobconsultingllc/getting-a-solid-start
- References:
  - Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Pearson Education, 2002.
    - C++ and Java codes
  - Robert C. Martin, Micah Martin. *Agile Principles, Patterns, and Practices in C#*. Prentice Hall, 2006.

SOLID (2)

- ***S****ingle responsibility principle* (SRP)
- ***O****pen/closed principle* (OCP)
- ***L****iskov substitution principle* (LSP)
- ***I****nterface segregation principle* (ISP)
- ***D****ependency inversion principle* (DIP)

SOLID – *Single responsibility principle* (1)

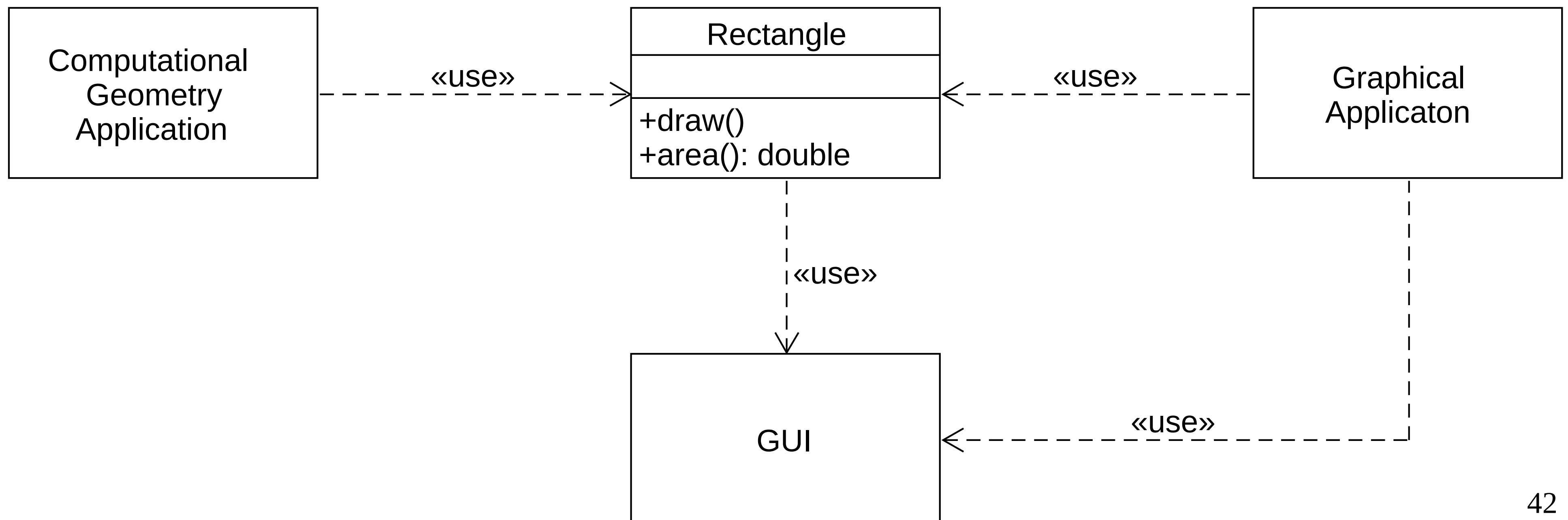
- Robert C. Martin:
  - *„A class should have only one reason to change.”*
- Related patterns: decorator, chain of responsibility

SOLID – *Single responsibility principle* (2)

- A responsibility is a reason of change.
- Every responsibility is an axis of change. When the requirements change the change shows up as a change of responsibilities.
- If a class has more than one responsibilities it has more than one reasons to change.
- If there are more responsibilities they can be coupled meaning that changes in a responsibility may hinder the capability of the class to fulfill others.

SOLID – *Single responsibility principle* (3)

- An example of violating the principle:
  - The two responsibilities of the Rectangle class:
    - Model the geometry of a rectangle
    - Show a rectangle on the GUI

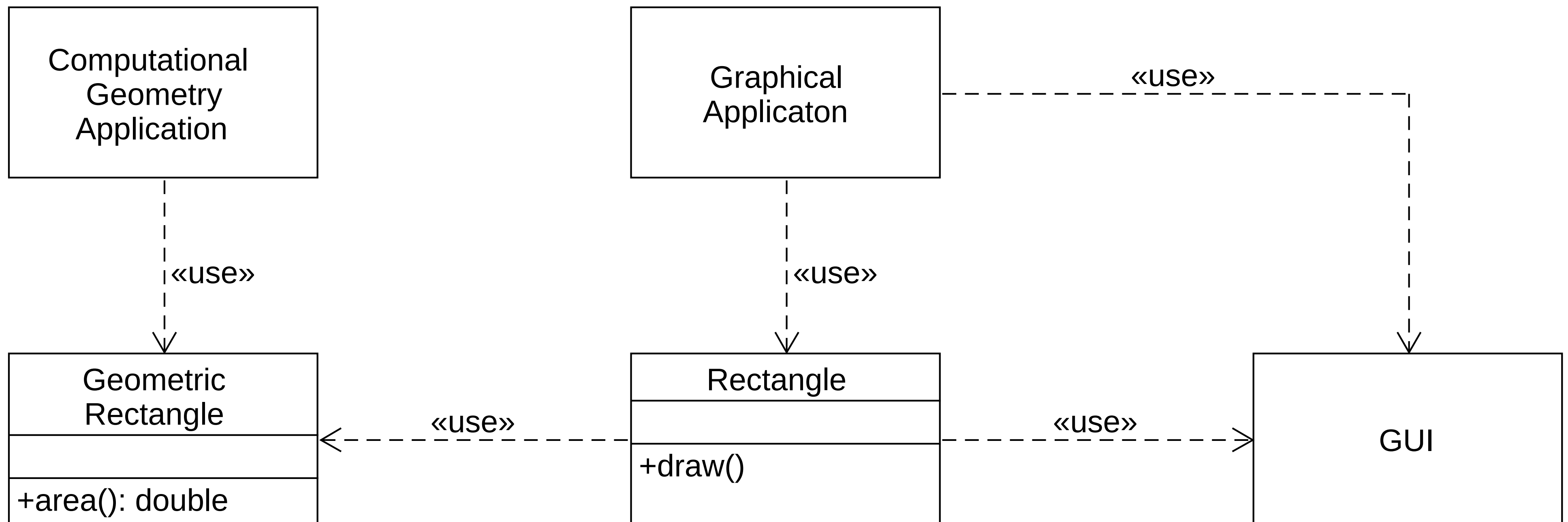


SOLID – *Single responsibility principle* (4)

- An example of violating the principle (2):
  - The computational geometry application has to contain the GUI.
  - If the `Rectangle` class is changed because of the GUI, the repeated building, testing and deploying of the computational geometry application is required.

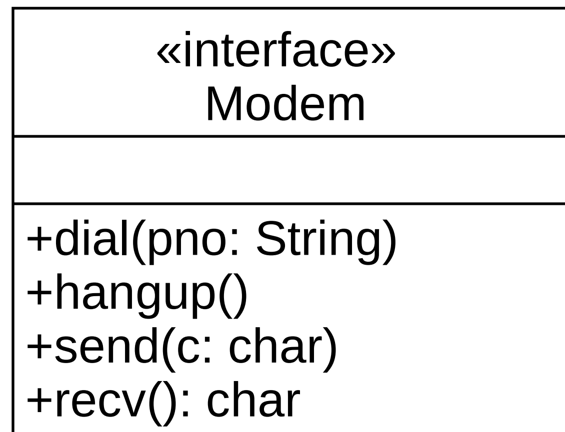
SOLID – *Single responsibility principle* (5)

- The same example fulfilling the principle:



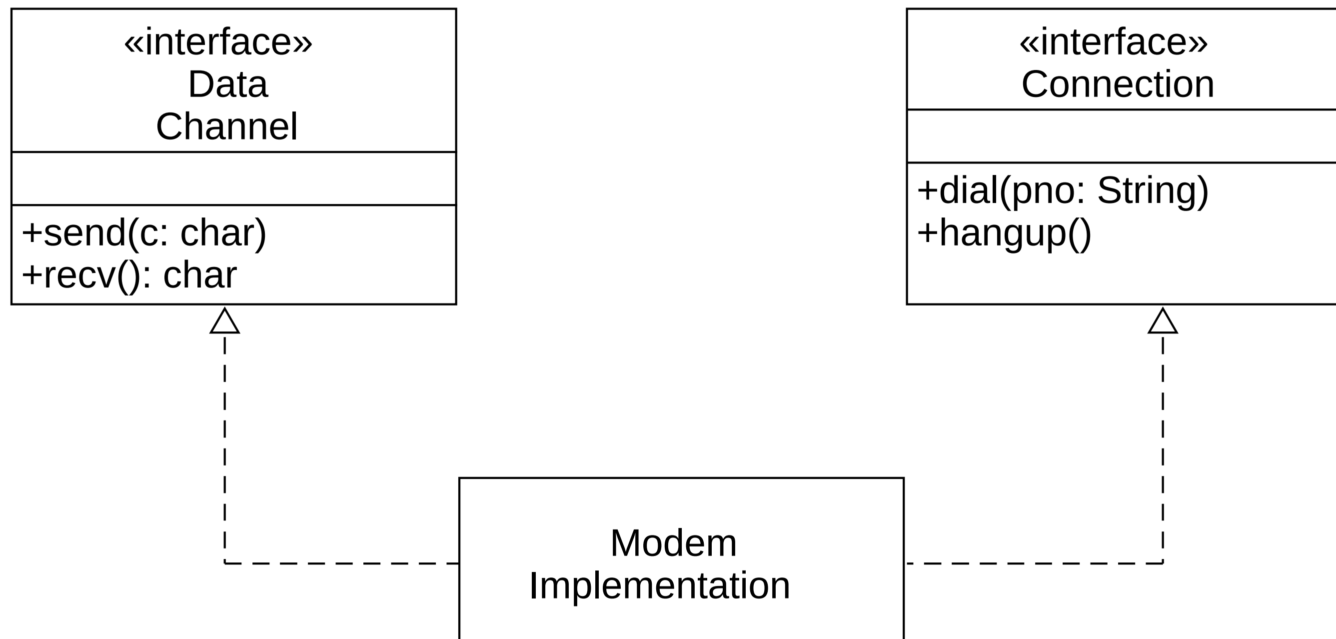
SOLID – *Single responsibility principle* (6)

- Example: What is a responsibility?
  - In the case of the `Modem` interface below there are two responsibilities: connection management and data communication
  - Based on the changes of the application it may worth or not to separate them.



SOLID – *Single responsibility principle* (7)

- Example: What is a responsibility?
  - If the application sages in a way that changes the signatures of connection management methods we have to separate the responsibilities.



SOLID – *Single responsibility principle* (8)

- A more precise definition of the principle:
  - *„A* ***class*** *should have only one reason to change.”*
    - Robert C. Martin. *Agile Software Development: Principles, Patterns, and Practices*. Pearson Education, 2002. p. 95.
  - *„… a* ***class or module*** *should have one, and only one, reason to change.”*
    - Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, 2008. p. 138.

SOLID – *Single responsibility principle* (9)

- Separation of concerns and single responsibility principles are closely related.
- In an ideal case each concern is described by a responsibility, but in one responsibility usually more concerns are mixed.
- Separation of concerns does not mean that one responsibility can describe only one concern. It just says that the concerns have to be separated and it has to be clear if there are more than one concerns.

SOLID – *Single responsibility principle* (10)

- Example code that fulfills single responsibility principle but violates separation of concerns:
  - Artur Trosin. *Separation of Concern vs Single Responsibility Principle (SoC vs SRP)*. 2009. https://weblogs.asp.net/arturtrosin/separation-of-concern-vs-single-responsibility-principle-soc-vs-srp

SOLID – *Open/closed principle* (1)

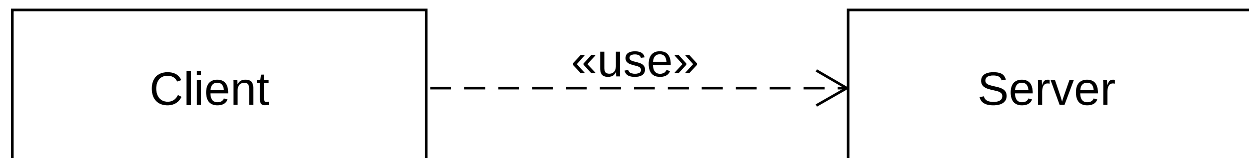
- A principle by Bertrand Meyer.
  - Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988.
- Software entities (classes, modules, functions, …) should be open for extension, but closed for modification.
- Related design pattern: factory method, proxy, strategy, template function, visitor

SOLID – *Open/closed principle* (2)

- A module fulfilling the principle has two major properties:
  - Open for extension: the behavior of the module can be extended
  - Closed for modification: The extension of the behavior of the module does not result in the modification of the source or binary code.

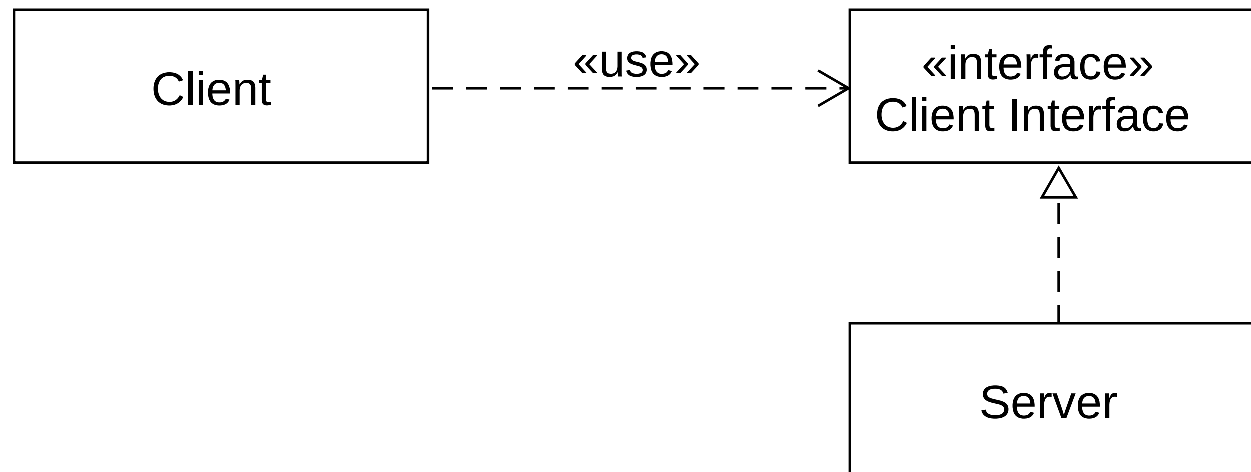
SOLID – *Open/closed principle* (3)

- Example of violating the principle:
  - `Client` and `Server` are exact classes. The `Client` class uses the `Server` class. If we want the `Client` object to use a different `Server` object we have to change the name of the `Server` class in the `Client`.



SOLID – *Open/closed principle* (4)

- Solution:



SOLID – *Liskov substitution principle*

- A principle by Barbara Liskov.
  - Barbara Liskov. *Keynote Address – Data Abstraction and Hierarchy*. 1987.
- If type *S* is a subtype of *T*, the functionality of the program must not change if we substitute the type *T* objects to type *S* objects in the code

SOLID – *Interface segregation principle* (1)

- A principle
  - *„Classes should not be forced to depend on methods they do not use.”*

SOLID – *Interface segregation principle* (2)

- ***Fat interface*** (Bjarne Stroustrup) http://www.stroustrup.com/glossary.html#Gfat-interface
  - *„An interface with more member functions and friends than are logically necessary.”*

SOLID – *Interface segregation principle* (3)

- The interface segregation principle deals with fat interfaces.
- Classes of fat interfaces are not coherent. The methods can be organize to groups where each group serves different clients.
- ISP accepts that there are objects that require non-coherent interfaces, but it advises that the clients should know these interfaces not as a single class.

SOLID – *Interface segregation principle* (4)

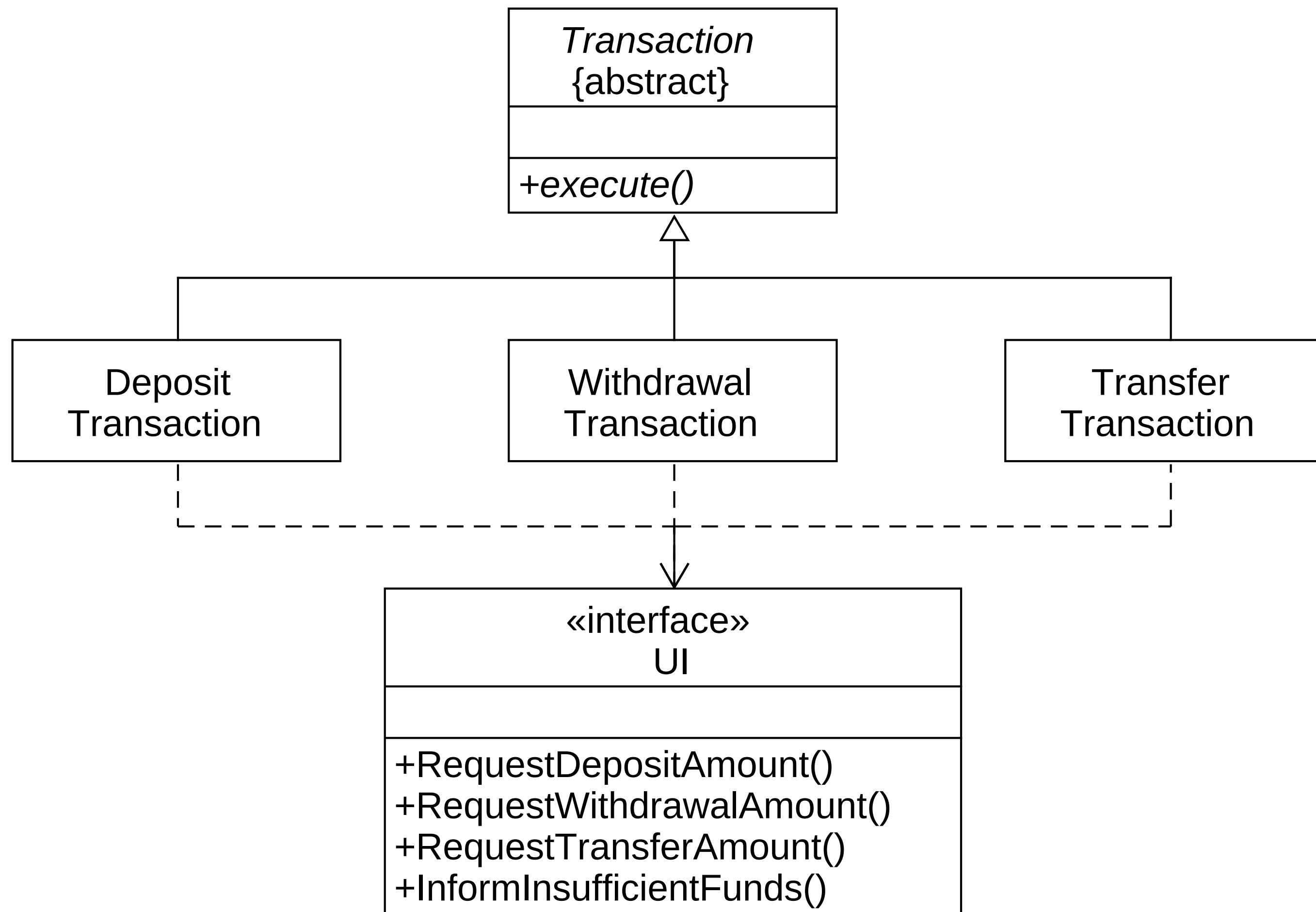
- **I*nterface pollution***:
  - Pollution of an interface with unneeded methods.

SOLID – *Interface segregation principle* (5)

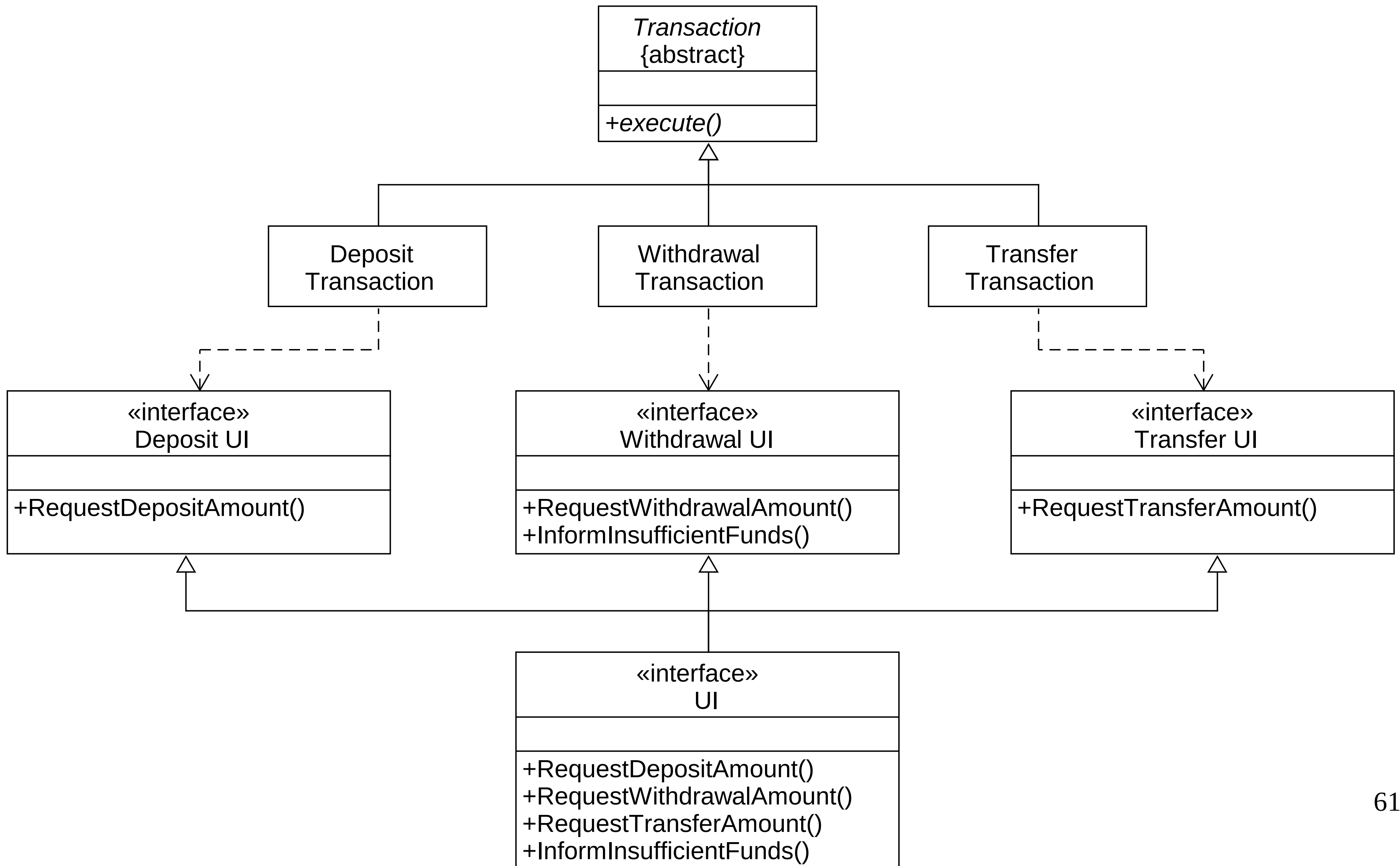
- If a client depends on a class that has methods not used by the client, but used by other clients the changes enforced by these others may affect the original client as well.
- This leads to unintentional coupling of clients.

SOLID – *Interface segregation principle* (6)

- Example: ATM (Robert C. Martin)



SOLID – *Interface segregation principle* (7)



SOLID – *Dependency inversion principle* (1)

- A principle 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.

SOLID – *Dependency inversion principle* (2)

- The name comes from the phenomenon that the classical software developing methods usually produce software in which high-level modules depend on low-level modules
- Related pattern: adapter

SOLID – *Dependency inversion principle* (3)

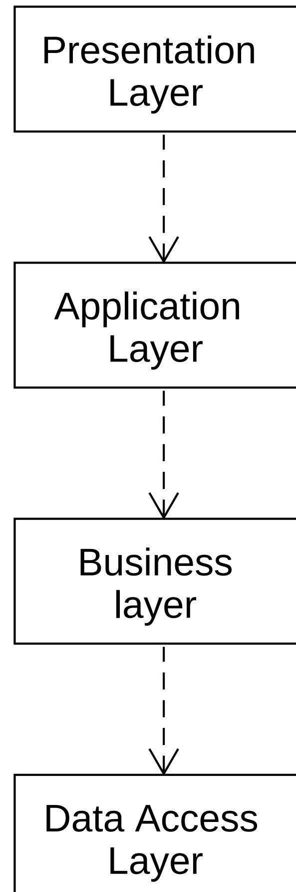
- High-level modules contain the business logic of the application. They describe the identity of the application. If these modules depend on low-level modules than changes in low-level modules may affect high-level ones implying the need of changes of them.
- While in contrary the high-level modules are those who should specify low-level ones.

SOLID – *Dependency inversion principle* (4)

- We would like to reuse the high-level modules. The reuse of low-level modules is already solved by program libraries.
- If high-level modules depend on low-level modules, the reuse of them is really hard in different situations.
- In the opposite case however this is fairly simple.

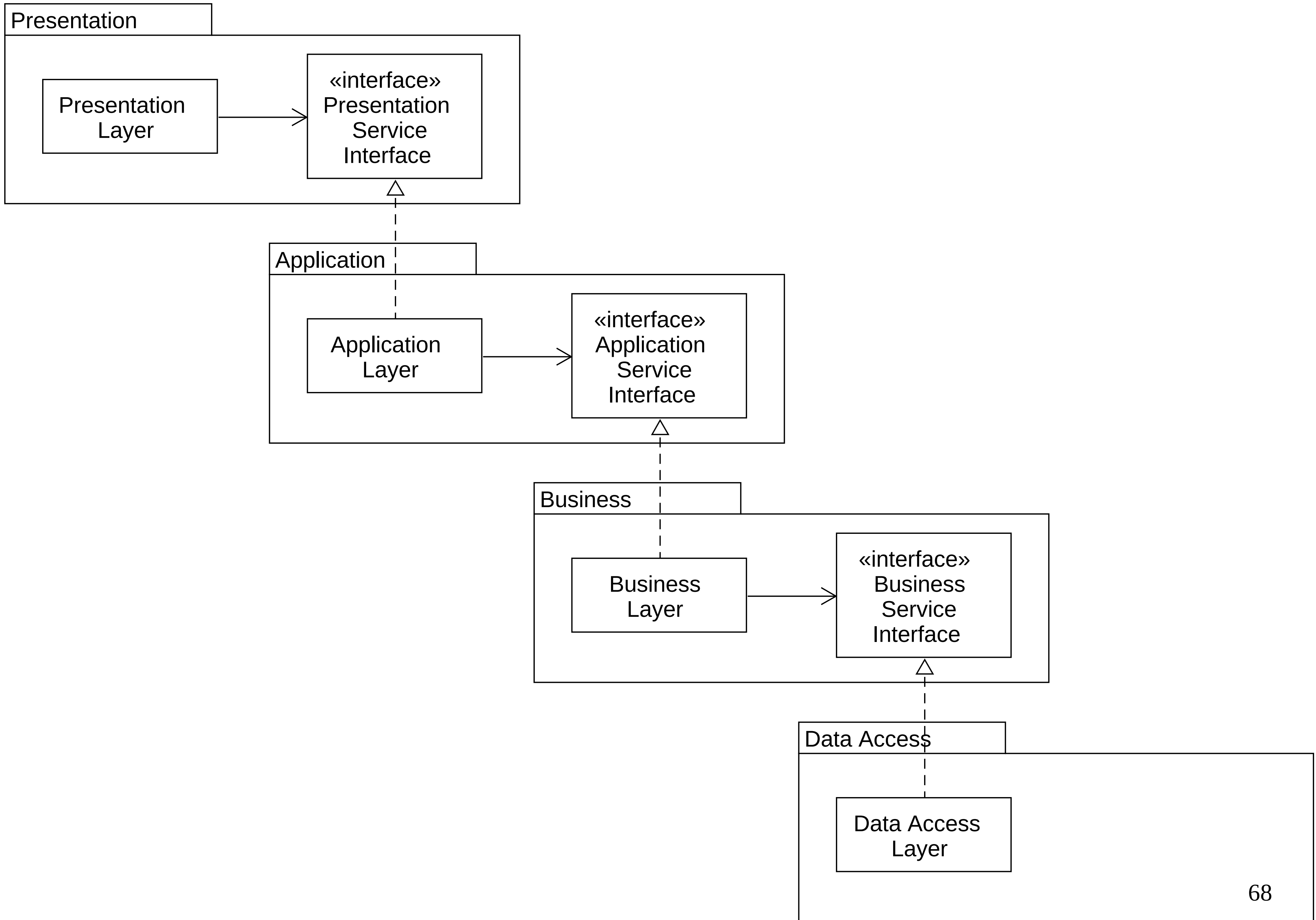
SOLID – *Dependency inversion principle* (5)

- Example of the application of the layered architectural pattern:



SOLID – *Dependency inversion principle* (6)

- The same pattern when fulfilling the principle:
  - Every high-level interface defines an interface for the services it requires.
  - The implementation of the lower level layers are based on these interfaces.
  - As a result high-level layers will not depend on low-level layers.



Presentation
Presentation Layer
«interface» Presentation Service Interface
Application
Application Layer
«interface» Application Service Interface
Business
Business Layer
«interface» Business Service Interface
Data Access
Data Access Layer

SOLID – *Dependency inversion principle* (8)

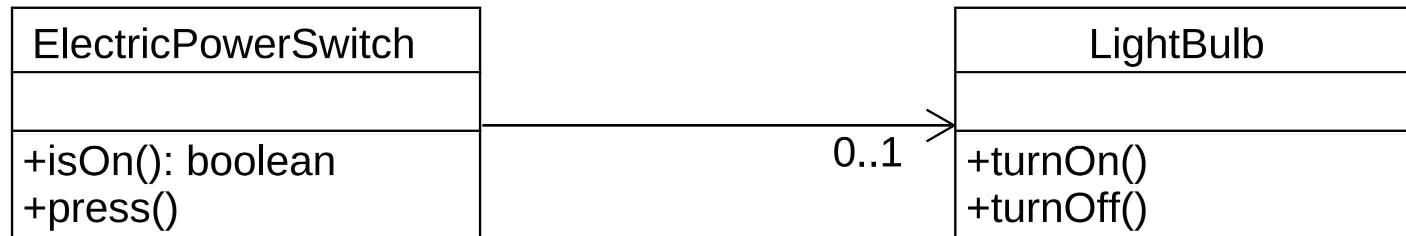
- The same pattern when fulfilling the principle:
  - Not only the dependencies are inverted but also the ownership of interfaces (*inversion of ownership*).
    - Hollywood principle: *Don't call us, we'll call you.*

SOLID – *Dependency inversion principle* (9)

- Depending on abstraction:
  - The program should not depend on exact classes. It should depend only on abstract classes and on interfaces.
    - No variable should refer to an exact class.
    - No class should extend an exact class.
    - No class should override methods of the parent class.
  - Most of thee programs usually violate the above rules at least once.
  - In case of classes that changes rarely (e.g. `String`) dependency may be accepted

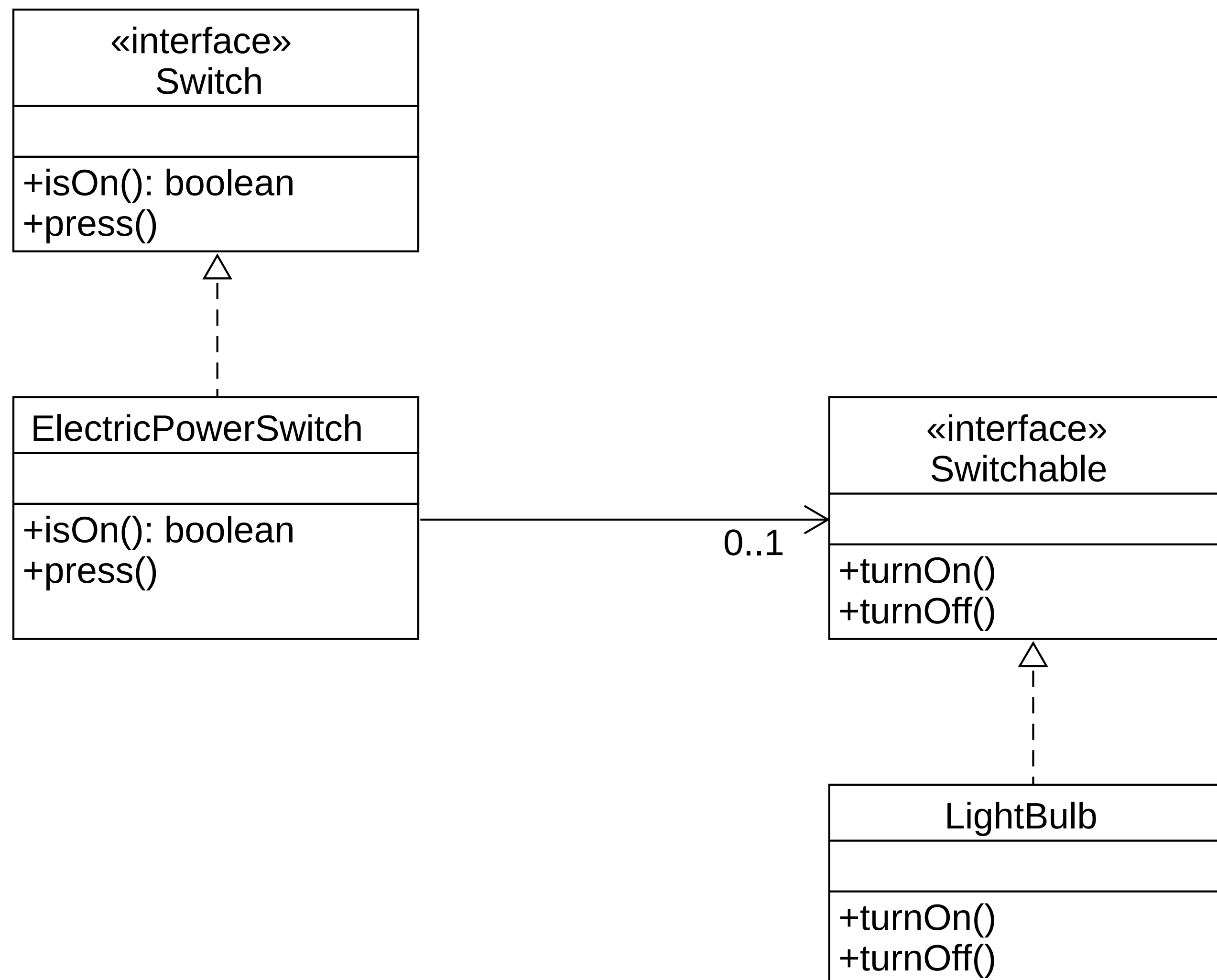
SOLID – *Dependency inversion principle* (10)

- Example of violating the principle:
  - Reference: https://springframework.guru/principles-of-object-oriented-design/dependency-inversion-principle/



SOLID – *Dependency inversion principle* (11)

- The same example fulfilling the principle:



Dependency Injection (1)

- References:
  - Dhanji R. Prasanna. *Dependency Injection Design patterns using Spring and Guice*. Manning, 2009. https://www.manning.com/books/dependency-injection
  - Mark Seemann. *Dependency Injection in .NET*. Manning, 2011. https://www.manning.com/books/dependency-injection-in-dot-net

Dependency Injection (2)

- DI – *dependency injection is a concept from* Martin Fowler.
  - Martin Fowler. *Inversion of Control Containers and the Dependency Injection pattern*. 2004. https://martinfowler.com/articles/injection.html
- A special case of IoC – *inversion of control* architectural pattern
  - Martin Fowler. *InversionOfControl*. 2005. https://martinfowler.com/bliki/InversionOfControl.html

Dependency Injection (3)

- Definition (Seemann):
  - *„Dependency Injection is a set of software design principles and patterns that enable us to develop loosely coupled code.”*

Dependency Injection (4)

- N object is handled as a service that is used as a client by other objects.
- The client-server connection between object is called dependency. This connection is transitive.

Dependency Injection (5)

- ***Dependency***: a service required by a client that is required for its job to be done.
- ***Dependent***: a client object that need dependency of dependencies for its job to be done.
- **O*bject graph***: a collection of dependents and their dependencies.
- ***Injection***: providing the dependencies of a client.
- ***DI container***: a library providing dependency injection functionality.
  - We also use ***Inversion of Control (IoC) containe*r** for them

Dependency Injection (6)

- Dependency injection is about planning object graphs in an efficient way using its best practices and patterns .
- DI containers let the clients to outsource the creation and injection of their dependencies to outer source codes.

Dependency Injection (7)

- Example: no dependency injection

```
public interface SpellChecker {

  public boolean check(String text);

}

public class TextEditor {

  private SpellChecker spellChecker;

  public TextEditor() {
    spellChecker = new HungarianSpellChecker();
  }

  // …

}
```

Dependency Injection (8)

- *Constructor injection* (Dependency injection by constructor):

```
public class TextEditor {

  private SpellChecker spellChecker;

  public TextEditor(SpellChecker spellChecker) {
    this.spellChecker = spellChecker;
  }

  // …

}
```

Dependency Injection (9)

- *Setter injection* (Dependency injection by setter)*:*

```
public class TextEditor {

  private SpellChecker spellChecker;

  public TextEditor() {}

  public void setSpellChecker(SpellChecker spellChecker) {
    this.spellChecker = spellChecker;
  }

  // …

}
```

Dependency Injection (10)

- *Interface injection* (Dependency injection by interface)*:*

```
public interface SpellCheckerSetter {

  void setSpellChecker(SpellChecker spellChecker);

}

public class TextEditor implements SpellCheckerSetter {

  private SpellChecker spellChecker;

  public TextEditor() {}

  @Override
  public void setSpellChecker(SpellChecker spellChecker) {
    this.spellChecker = spellChecker;
  }

  // …

}
```

Dependency Injection (11)

- C++ frameworks:
  - *[Boost].DI* (license: *Boost Software License*)
    http://boost-experimental.github.io/di/
  - *Fruit* (license: Apache License 2.0)
    https://github.com/google/fruit
  - *Hypodermic* (license: MIT License)
    https://github.com/ybainier/Hypodermic
  - …

Dependency Injection (12)

- Java:
  - *JSR 330: Dependency Injection for Java* https://www.jcp.org/en/jsr/detail?id=330
    - From Java EE 6
    - `javax.inject` package.
    - Dependency injection through annotations

Dependency Injection (13)

- Java frameworks:
  - *Dagger* (license: Apache License 2.0) https://google.github.io/dagger/
  - *Guice* (license: Apache License 2.0) https://github.com/google/guice
  - *HK2* (license: CDDL + GPLv2) https://hk2.java.net/
  - *Java EE 7 CDI* https://docs.oracle.com/javaee/7/tutorial/cdi-basic.htm
  - *Spring Framework* (license: Apache License 2.0) https://projects.spring.io/spring-framework/ http://www.vogella.com/tutorials/SpringDependencyInjection/article.html
  - …

Dependency Injection (14)

- .NET frameworks:
  - *Castle Windsor* (license: Apache License 2.0) https://github.com/castleproject/Windsor
  - *Ninject* (license: Apache License 2.0) http://www.ninject.org/
  - *StructureMap* (license: Apache License 2.0) http://structuremap.github.io/
  - …