TDDE45 - Lecture 3: Design Principles

Adrian Pop and Martin Sjölund

Department of Computer and Information Science
Linköping University

2023-09-06

Part I

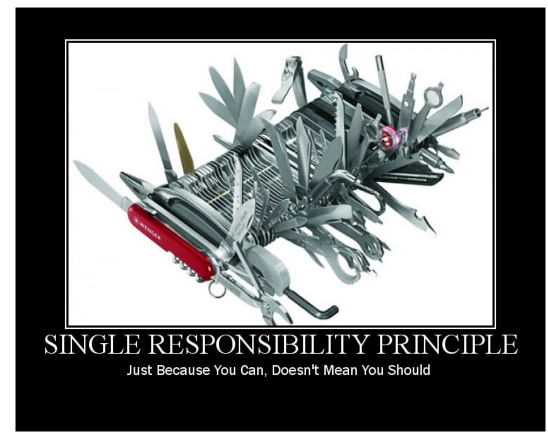
Single Responsibility Principle

Single Responsibility Principle - History

The term “Single responsibility principle” was made popular by *Agile Software Development, Principles, Patterns, and Practices* (Martin 2003). SRP was coined a few years earlier in the late 90s and he said “A class should have only one reason to change”. There is a clarification in *The Single Responsibility Principle* (Martin 2014) of what he meant by that.

Single Responsibility Principle

The single responsibility principle states that every module, class, or function should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class.



Violating the SRP using a God object

```
class Game {
  // The game's state
  State state;
  // Every object moves 1 frame
  void advanceGame() {
    foreach (Enemy e : state.enemies)
    // ...
  }
  // Heads-up display shows the player's life total, etc
  void renderHUD();
  // Render the game from the player's point of view
  void renderGame();
}
```

The God object has too many responsibilities. Basically a procedural program disguised in OOP clothing.

First step towards the SRP

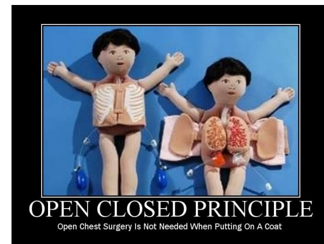
```
class Game {
  // The game's state
  State state;
  HUD hud;
  Scene scene;
  public Game() {
    state = new State();
    hud = new HUD(state);
    // Render the game from player's point of view
    scene = new Scene(state.getPlayer(), state);
  }
  void loop() {
    state.advanceGame();
    scene.render();
    hud.render();
  }
}
```

Part II

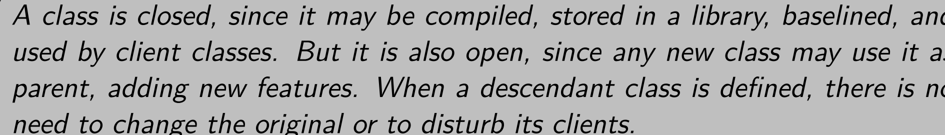
Open-Closed Principle

Open-Closed Principle – History

> - *A module will be said to be open if it is still available for extension. For example, it should be possible to add fields to the data structures it contains, or new elements to the set of functions it performs.*
> - *A module will be said to be closed if [it] is available for use by other modules. This assumes that the module has been given a well-defined, stable description (the interface in the sense of information hiding).*
>
> *Meyer 1988*



Open-Closed Principle



> *A class is closed, since it may be compiled, stored in a library, baselined, and used by client classes. But it is also open, since any new class may use it as parent, adding new features. When a descendant class is defined, there is no need to change the original or to disturb its clients.*
>
> *Meyer 1988*

In modern usage, code that adheres to the principle tends to use abstract classes or interfaces instead. At the time inheritance existed but the concept of abstract classes or interfaces had not quite been introduced in object-oriented programming.

Violating the Open-Closed Principle

```
class Shape {
  Shape(int t) : t(t) {}
  int t; // 1==Square, 2==Rectangle
}
class Rectangle : Shape {
  int width; int height;
  Rectangle(int width, int height)
    : Shape(RECTANGLE),
      width(width),
      height(height) {}
  int area() { return height * width; }
}
class Square : Shape {
  int length;
  Rectangle(int length)
    : Shape(SQUARE), length(length) {}
  int area() { return length * length; }
}
```

```
// Something using shapes
swicth (shape->t) {
case SQUARE:
  return ((Square*)shape)->area();
case RECTANGLE:
  return ((Rectangle*)shape)->area();
}
// Equally bad
Rectangle *r = dynamic_cast<Rectangle>(shape)
Square *s = dynamic_cast<Square>(shape)
r ? r->area() : s ? r->area() : 0;
// How will this handle a new shape?
```

Adhering to the Open-Closed Principle

```
// Could be abstract class or interface
class Shape {
  /* virtual */ int area();
}
class Rectangle : Shape {
  int width; int height;
  Rectangle(int width, int height)
    : width(width),
      height(height) {}
  int area() { return height * width; }
}
class Square : Shape {
  int length;
  Rectangle(int length)
    : length(length) {}
  int area() { return length * length; }
}
```

```
// Something using shapes
shape->area();
// Or filtering
foreach (auto shape : shapes) {
  if (dynamic_cast<Square>(shape) {
    sum += shape->area();
  }
}
```

Open-Closed Principle without Object-Orientation?

```
abstract type Shape end
struct Rectangle <: Shape
  width::Int
  height::Int
end
struct Square <: Shape
  length::Int
end
area(s::Rectangle)::Int = s.width * s.height
area(s::Square)::Int = s.length ^ 2
```

Multiple dispatch makes it easy to extend code. In Julia there typically exists an informal interface you need to add functions for (which is not checked by the compiler).

Part III

Liskov Substitution Principle

Liskov Substitution Principle - History

Initially introduced in a keynote address by Liskov 1987.
The original text talks about type systems in object-oriented programming at the time (SmallTalk) and is rather hard to decipher what is the actual principle is. A later paper has a rather short description of what is now called the Liskov Substitution Principle:

> *Subtype Requirement:*
> *Let $\phi(x)$ be a property provable about objects x of type T.*
> *Then $\phi(y)$ should be true for objects y of type S where S is a subtype of T.*
>
> *Liskov and Wing 1994*

Note that this property cannot be automatically checked by a compiler since it is related to semantics and expected behaviour of the object.

Liskov Substitution Principle

When we inherit from a class that is quite similar to what we want, we need to decide if we need a common ancestor (abstraction) to the two or if the subtype has proper subtype semantics of the parent.

Violating the Liskov Substitution Principle (1)

```
public class Rectangle {
  public virtual int height { get; set; }
  public virtual int width { get; set; }
  int area() { return height * width; }
}
```

```
public class Square : Rectangle {
  private int _length;
  public override int height {
    get {
      return _length;
    }
    set {
      _length = value;
    }
  }
  public override int width {
    get {
      return _length;
    }
    set {
     _length = value;
    }
  }
}
```

Violating the Liskov Substitution Principle (2)

```java
Rectangle r = new Square();
r.height = 6;
r.width = 4;
// What is the area?
```

Violating the Liskov Substitution Principle (2)

```
Rectangle r = new Square();
r.height = 6;
r.width = 4;
// What is the area?
```

16

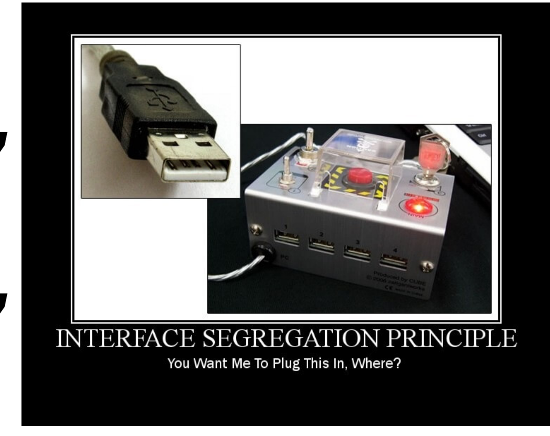
Part IV

Interface Segregation Principle

Interface Segregation Principle

> *Clients should not be forced to depend upon interfaces that they do not use.*
>
> *Martin 1996*

> *Make fine grained interfaces that are client specific.*
>
> *Robert C. Martin*



Interface Segregation Principle – History

Xerox had created a new printer with many functions. This had resulted in a huge God class which supported these functions.
See also, Single Responsibility Principle.

Violation of the ISP

Document;

AbstractPrinter

void Print(Document d);
Document Scan();
void Fax(Document d);

MultiFunctionPrinter

void Print(Document d);
Document Scan();
void Fax(Document d);

BasicPrinter

void Print(Document d);

"Needs to add dummy Scan and Fax
functions that are not supported"

According to the ISP

Document;

AbstractPrinter

void Print(Document d);

AbstractScanner

Document Scan();

AbstractFax

void Fax(Document d);

BasicPrinter

void Print(Document d);

AbstractMultiFunction

Perhaps even extending the basic printer

MultiFunctionPrinter

void Print(Document d);
Document Scan();
void Fax(Document d);

Isn't this (ISP) the same thing as the SRP?

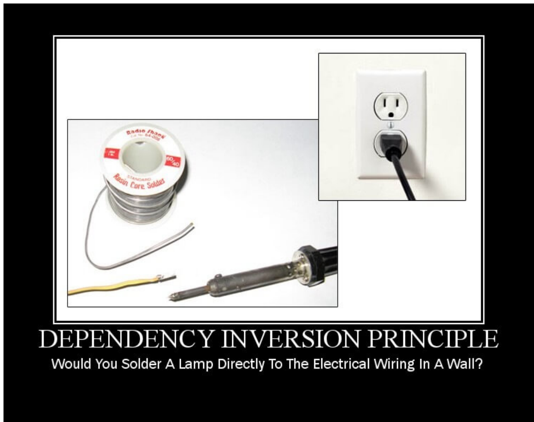
Not quite. Go back and look at the solutions.
The ISP adds interfaces for the smaller parts so that clients do not see the parts they are not interested in. Adhering to it is mostly a structural thing (moving code around).
For the SRP, the underlying problem may be bigger and you will tend to change the code somewhat.

Part V

Dependency Inversion Principle

Dependency Inversion Principle

Martin 1995 is an early work describing dependency inversion.
The inversion here being inverting how you write code: instead of creating client code that uses a concrete implementation such as a KeyboardReader, create a general Reader interface and tie that to the concrete implementation later on.



Dependency Inversion Principle

It also generally says that interfaces having few to no dependencies is good. This means dependencies will rarely need to be updated (because they do not depend on implementation details).
Which in turn means your code which depends on interfaces will not need to be updated when the implementations of this interface changes.

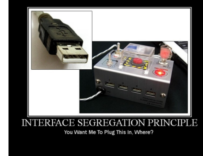
```
class Writer {
public:
  virtual void Write(char) = 0;
};
class Reader {
public:
  virtual char Read() = 0;
};
```

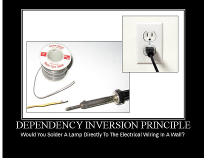
Example of an interface that is probably not going to change.

Part VI

SOLID

Design Principles: SOLID





SOLID Motivational Posters by Derick Bailey, used under CC BY-SA 3.0 US

Part VII

Example

Example

```
class Employee {
  String name;
  int id;
  double salary;

  Employee(String name, int id, double salary) {
    this.name = name; this.id = id; this.salary = salary;
  }

  void calculateBonus() {
    if (name.equals("Manager")) { salary += 1000;}
    else if (name.equals("Developer")) { salary += 500; }
  }

  void printDetails() {
    System.out.println("Employee ID: " + id);
    System.out.println("Employee Name: " + name);
    System.out.println("Employee Salary: " + salary);
  }
}
```

Part VIII

Remarks

Principles + Problem = Pattern

Principles = SOLID + Some general tips

SOLID

1. Encapsulate what varies (S)
2. Program to an interface, not to an implementation (I, D)
3. Favor Composition over Inheritance (L)
4. Don't call us, we'll call you (O)

Some more tips

5. Depend upon abstractions, not upon concrete classes (see 2).
6. Strive for loosely coupled designs between objects that interact (see 4).
7. Only talk to your friends.
8. Avoid global variables (constants can be fine), static methods (thread-safe code).
9. Simple, readable code is often favorable over strictly adhering to the design principles.

Practice makes perfect

Design is very open-ended problem with as many solutions as there are programmers. You tend to learn what good design is after a few years of programming in a language. Otherwise there are plenty of books available.

Note: These slides have a lot of references to works by Robert C. Martin since SOLID is a part of the course. Other design principles and hints are also valid if you want a different opinion.

What is clean code? *Clean Code: A Handbook of Agile Software Craftsmanship* (Martin 2008):

- Elegant (Bjarne Stroustrup)
- Readable (Grady Booch)
- Readable by other people (Dave Thomas)
- Care of the code (Michael Feathers)
- No duplication, one thing, expressivness, tiny abstractions (Ron Jeffries)
- “when each routine you read turns out to be pretty much what you expected” (Ward Cunningham)

Sometimes classes are unsuitable for some design patterns

Effective Java (Bloch 2017) chapter 4, item 19: Design and document for inheritance or else prohibit it says that:

> *...*
> *By now it should be apparent that designing a class for inheritance requires great effort and places substantial limitations on the class. This is not a decision to be undertaken lightly.*
> *...*
> *The best solution to this problem is to prohibit subclassing in classes that are not designed and documented to be safely subclassed*

Dependency injection

Not the same as (but easily confused with) dependency inversion, but follows good design principles and allows for testability (see future lecture).

Part IX

Overusing Design Patterns

FizzBuzz

> *The rules of FizzBuzz are as follows:*
> *For numbers 1 through 100,*
> *if the number is divisible by 3 print Fizz; if the number is divisible by 5 print Buzz; if the number is divisible by 3 and 5 (15) print FizzBuzz; else, print the number.*

Take a few minutes to think how you would solve this.

FizzBuzz

> *The rules of FizzBuzz are as follows:*
> *For numbers 1 through 100,*
> *if the number is divisible by 3 print Fizz; if the number is divisible by 5 print Buzz; if the number is divisible by 3 and 5 (15) print FizzBuzz; else, print the number.*

We will now have a look at:
`https://github.com/EnterpriseQualityCoding/FizzBuzzEnterpriseEdition`

Your solution is probably straight-forward Python code

```
for each number 1 to 100:
    if number % 15 == 0:
        print number, "fizzbuzz"
    else if number % 5 == 0:
        print number, "buzz"
    else if number % 3 == 0:
        print number, "fizz"
    else:
        print number
```

Or perhaps something interesting half-readable written in C

```c
#include <stdio.h>
char const * template[] = {
  "%i",
  "Buzz",
  "Fizz",
  "FizzBuzz"
};
const int __donotuseme3[] = { 2, 0, 0 };
const int __donotuseme5[] = { 1, 0, 0, 0, 0 };
#define TEMPLATE(x) (template[__donotuseme3[(x) % 3] | __donotuseme5[(x) % 5]])
int main(void) {
  int i;
  for (i = 1; i <= 100; i++) {
    printf(TEMPLATE(i), i);
    putchar('\n');
  }
  return 0;
}
```

src/main/java/com/seriouscompany/business/java/fizzbuzz/packagenamingpackage

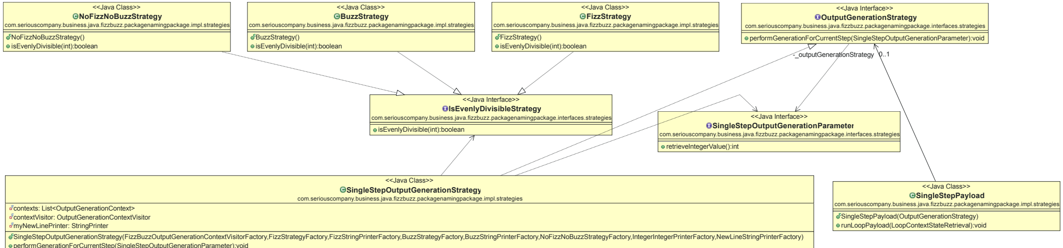
```
…├── impl
│   ├── ApplicationContextHolder.java
│   ├── Constants.java
│   ├── factories
│   │   ├── BuzzStrategyFactory.java
│   │   ├── BuzzStringPrinterFactory.java
│   │   ├── BuzzStringReturnerFactory.java
│   │   ├── EnterpriseGradeFizzBuzzSolutionStrategyFactory.java
│   │   ├── FizzBuzzOutputGenerationContextVisitorFactory.java
│   │   ├── FizzStrategyFactory.java
│   │   ├── FizzStringPrinterFactory.java
│   │   ├── FizzStringReturnerFactory.java
│   │   ├── IntegerIntegerPrinterFactory.java
│   │   ├── IntegerIntegerStringReturnerFactory.java
│   │   ├── LoopComponentFactory.java
│   │   ├── NewLineStringPrinterFactory.java
│   │   ├── NewLineStringReturnerFactory.java
│   │   ├── NoFizzNoBuzzStrategyFactory.java
│   │   └── SystemOutFizzBuzzOutputStrategyFactory.java
│   ├── loop
│   │   ├── LoopCondition.java
│   │   ├── LoopContext.java
│   │   ├── LoopFinalizer.java
│   │   ├── LoopInitializer.java
│   │   ├── LoopRunner.java
│   │   └── LoopStep.java
│   ├── Main.java
│   ├── math
│   │   └── arithmetics
│   │       ├── IntegerDivider.java
│   │       └── NumberIsMultipleOfAnotherNumberVerifier.java
│   ├── parameters
│   │   └── DefaultFizzBuzzUpperLimitParameter.java
│   ├── printers
│   │   ├── BuzzPrinter.java
│   │   ├── BuzzStringPrinter.java
│   │   ├── FizzPrinter.java
│   │   ├── FizzStringPrinter.java
│   │   ├── IntegerIntegerPrinter.java
│   │   ├── IntegerPrinter.java
│   │   ├── NewLinePrinter.java
│   │   └── NewLineStringPrinter.java
│   ├── StandardFizzBuzz.java…
```

```
…│   ├── strategies
│   │   ├── adapters
│   │   │   ├── FizzBuzzOutputStrategyToFizzBuzzExceptionSafeOutputStrategyAdapter.java
│   │   │   └── LoopContextStateRetrievalToSingleStepOutputGenerationAdapter.java
│   │   ├── BuzzStrategy.java
│   │   ├── comparators
│   │   │   ├── doublecomparator
│   │   │   │   ├── FirstIsLargerThanSecondDoubleComparator.java
│   │   │   │   └── FirstIsSmallerThanSecondDoubleComparator.java
│   │   │   └── integercomparator
│   │   │       ├── IntegerForEqualityComparator.java
│   │   │       ├── ThreeWayIntegerComparator.java
│   │   │       └── ThreeWayIntegerComparisonResult.java
│   │   ├── constants
│   │   │   ├── BuzzStrategyConstants.java
│   │   │   ├── FizzStrategyConstants.java
│   │   │   └── NoFizzNoBuzzStrategyConstants.java
│   │   ├── converters
│   │   │   └── primitivetypesconverters
│   │   │       ├── DoubleToIntConverter.java
│   │   │       └── IntToDoubleConverter.java
│   │   ├── EnterpriseGradeFizzBuzzSolutionStrategy.java
│   │   ├── FizzStrategy.java
│   │   ├── NoFizzNoBuzzStrategy.java
│   │   ├── SingleStepOutputGenerationStrategy.java
│   │   ├── SingleStepPayload.java
│   │   └── SystemOutFizzBuzzOutputStrategy.java
│   ├── stringreturners
│   │   ├── BuzzStringReturner.java
│   │   ├── FizzStringReturner.java
│   │   ├── IntegerIntegerStringReturner.java
│   │   └── NewLineStringReturner.java
│   └── visitors
│       ├── FizzBuzzOutputGenerationContext.java
│       └── FizzBuzzOutputGenerationContextVisitor.java…
```

```
…└── interfaces
    ├── factories
    │   ├── FizzBuzzOutputStrategyFactory.java
    │   ├── FizzBuzzSolutionStrategyFactory.java
    │   ├── IntegerPrinterFactory.java
    │   ├── IntegerStringReturnerFactory.java
    │   ├── IsEvenlyDivisibleStrategyFactory.java
    │   ├── OutputGenerationContextVisitorFactory.java
    │   ├── StringPrinterFactory.java
    │   └── StringStringReturnerFactory.java
    ├── FizzBuzz.java
    ├── loop
    │   ├── LoopContextStateManipulation.java
    │   ├── LoopContextStateRetrieval.java
    │   └── LoopPayloadExecution.java
    ├── parameters
    │   └── FizzBuzzUpperLimitParameter.java
    ├── printers
    │   ├── DataPrinter.java
    │   ├── IntegerPrinter.java
    │   └── StringPrinter.java
    ├── strategies
    │   ├── FizzBuzzExceptionSafeOutputStrategy.java
    │   ├── FizzBuzzOutputStrategy.java
    │   ├── FizzBuzzSolutionStrategy.java
    │   ├── IsEvenlyDivisibleStrategy.java
    │   ├── OutputGenerationStrategy.java
    │   └── SingleStepOutputGenerationParameter.java
    ├── stringreturners
    │   ├── IntegerStringReturner.java
    │   └── StringStringReturner.java
    └── visitors
        ├── OutputGenerationContext.java
        └── OutputGenerationContextVisitor.java…
```

25 directories, 89 files

Some of the FizzBuzz strategies



Eclipse

The Eclipse framework is huge with the ability to plugin almost anywhere. Eclipse plugins tend to have several files of classes that extend from something without adding any code, or empty classes. You need IDE support to navigate around such code if you are need in a project.

```
package org.eclipse.papyrus.moka.fuml. /* ... */;

public class ArrivalSignal { /* Yes, really an empty class */
}
```

References

- Joshua Bloch. *Effective Java*. 3rd ed. Addison-Wesley, 2017. ISBN: 9780134685991.
- Barbara Liskov. “Keynote Address - Data Abstraction and Hierarchy”. In: *SIGPLAN Not.* 23.5 (Jan. 1987), pp. 17–34. ISSN: 0362-1340. DOI: 10.1145/62139.62141.
- Barbara Liskov and Jeannette Wing. “A Behavioral Notion of Subtyping”. In: *ACM Trans. Program. Lang. Syst.* 16.6 (Nov. 1994), pp. 1811–1841. ISSN: 0164-0925. DOI: 10.1145/197320.197383.
- Robert C. Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 2003. ISBN: ISBN 978-0135974445.
- Robert C. Martin. *Clean Code: A Handbook of Agile Software Craftsmanship*. 1st ed. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2008. ISBN: 9780132350884.

References

- Robert C. Martin. *The Single Responsibility Principle*. 2014. URL: https://blog.cleancoder.com/uncle-bob/2014/05/08/SingleReponsibilityPrinciple.html.
- Robert C. Martin. *OO Design Quality Metrics*. 1995. URL: https://www.cin.ufpe.br/~alt/mestrado/oodmetrc.pdf.
- Robert C. Martin. *The Principles of OOD*. 1996. URL: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.
- Bertrand Meyer. *Object-Oriented Software Construction*. Prentice Hall, 1988. ISBN: 0-13-629049-3.