Java for Advanced Programmers

Design Principles

Bernd Kiefer
Jörg Steffen

January 10, 2023

Software Systems Degrade

- Most larger systems, even if designed well in the beginning, start to rot over time
- Predominant reason: unforeseen requirements require changes that do not fit well with the original design
- Requirements will always change, so what can we do to prevent our software from rotting?
- Properly manage the dependencies between classes and packages!

Design Principles

- Software design principles are general rules how to organize your software to support
  - Reuseability
  - Maintainability
  - Stability
- There are five fundamental principles which you should keep in mind when building software
- Design patterns, which we will talk about later, are blueprints that help you to adhere to the design principles

S.O.L.I.D

The five fundamental design principles:

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

S.O.L.I.D

The five fundamental design principles:

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

There a lot more, but those are the most important ones.

Single Responsibility Principle

A class should have one, and only one, reason to change.

Single Responsibility Principle

A class should have one, and only one, reason to change.

- What's a "reason to change"? This seems a bit vague

Single Responsibility Principle

A class should have one, and only one, reason to change.

- What's a "reason to change"? This seems a bit vague
- To me: Don't throw two responsibilities (functionalities) into a class that do not really belong together
- Why so:
  - Lower reusability: you can't use one independent of the other
  - Harder to test and maintain: if you can clearly separate functionality, each is easier to understand and fix if problems occur

Single Responsibility Principle

A class should have one, and only one, reason to change.

- What's a "reason to change"? This seems a bit vague
- To me: Don't throw two responsibilities (functionalities) into a class that do not really belong together
- Why so:
  - Lower reusability: you can't use one independent of the other
  - Harder to test and maintain: if you can clearly separate functionality, each is easier to understand and fix if problems occur
- This is similar to the Separation of Concern principle
- Another reformulation: Create classes that are
  - small enough to lower coupling (dependance)
  - large enough to maximize cohesion (things that will change together are in the same package / class)

Single Responsibility Principle: Example

A class that generates a tabular report from a data source and presents it as a table

```
class Report {
  public void generate(DataSource d) { ... }
  public String print() { ... }
}
```

Everything's fine here because the `print` method needs access to most internals of the `Report` class.

Single Responsibility Principle: Example

A class that generates a tabular report from a data source and presents it as a table

```
class Report {
  public void generate(DataSource d) { ... }
  public String print() { ... }
}
```

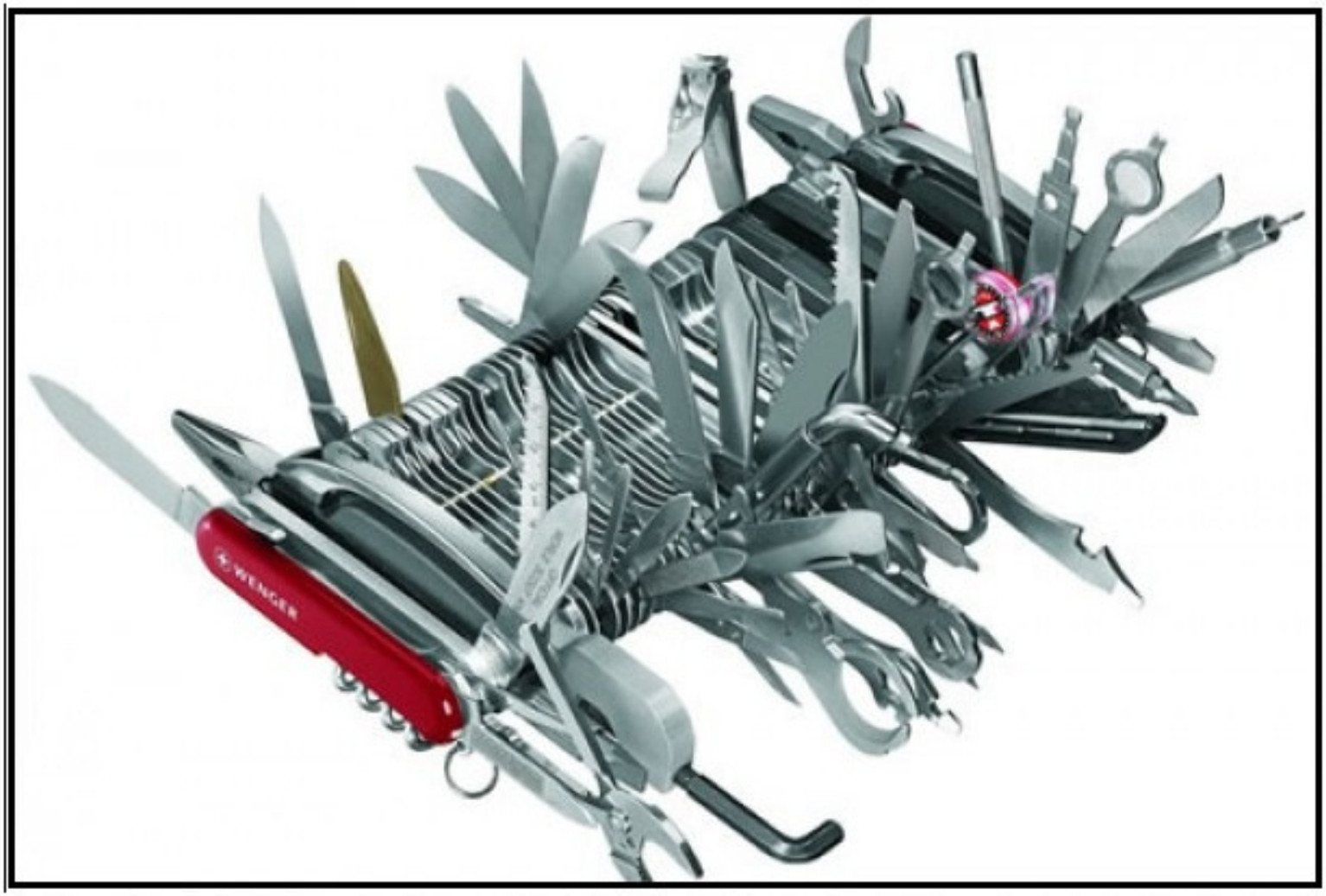
And now there's new requirements ...

```
class Report {
  public void generate(DataSource d) { ... }
  public String print() { ... }
  public String printHtml() { ... }
  public String printXml() { ... }
}
```

Single Responsibility Principle: Example

```
interface Formatter {
  public String tableHeader(...);
  public String tableRow(...);
}

class Report {
  public void generate(DataSource d) { ... }
  public String print(Formatter f) { ... }
}
```



SINGLE RESPONSIBILITY PRINCIPLE

Just Because You Can, Doesn't Mean You Should

Open-Closed Principle

A module should be open for extension but closed for modification.

- Write modules such that they can be extended (put to new uses) without requiring them to be modified
- How's that possible?
- We've already seen a quite complex example: `Visitor`
- The key to achieve this is Abstraction

Open-Closed Principle: Bad Example

```
class Shape { ... }

class Rectangle extends Shape {
  int x, y, width, height;
}

class Circle extends Shape {
  int x, y;
  float radius;
}

class GraphicEditor {
  void draw(Shape s) { // code changes for every new Shape
    if (s instanceof Rectangle) drawRectangle((Rectangle)s);
    else if (s instanceof Circle) drawCircle((Circle)s);
    // etc.
  }
}
```

Open-Closed Principle: Reworked

```
class Shape { abstract void draw(GraphicEnvironment g); }

class Rectangle extends Shape {
  int x, y, width, height;
  void draw(GraphicEnvironment g){ ... }
}

class Circle extends Shape {
  int x, y;
  float radius;
  void draw(GraphicEnvironment g){ ... }
}

class GraphicEditor {
  // add shape classes without changing this class
  void draw(Shape s) {
    s.draw(graphicEnvironment);
  }
}
```

Open Closed Principle

You don't need to rewire your MoBo to plug in "Mr Happy"

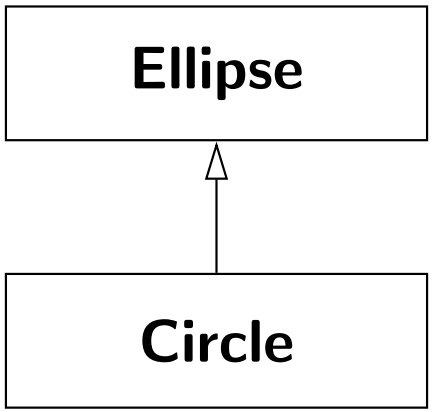
Liskov Substitution Principle

If S is subtype of T, the program behaves the same if objects of type T are replaced by objects of type S

- This is also called strong behavioural subtyping (Barbara Liskov, 1987)
- Similar to Design by Contract
- Consequence: If I know what objects of class T do, i can rely on the this for objects of type S
- May seem counterintuitive: How can i have a more specific class that does not behave differently from its superclass?
- The key is: you may only add functionality, but not change it

Breaking LSP: Ellipse vs. Circle

A circle is a degenerate form of an ellipse, we are tempted to use inheritance because `Circle` is-a `Ellipse`



`Ellipse` has two Foci, which have to be identical if it is a `Circle`, which has to be enforced by the implementation

Breaking LSP: Ellipse vs. Circle

```
class Ellipse {
  private Point focusA, focusB;
  public void setFoci(Point a, Point b);
  public double getCircumference();
  public double getArea();
  public Point getFocusA();
  public Point getFocusB();
  ...
}

class Circle extends Ellipse {
  public void setFoci(Point a, Point b) {
    if (! a.equals(b)) throw new IllegalArgumentException();
    super.setFoci(a, b);
  }
}
```

Breaking LSP: A Client Using Ellipse

Write client code that uses the contracts given by `Ellipse`:

```
void f(Ellipse e) {
  e.setFoci(new Point(1, 0), new Point(0, 1));
  assert(e.getFocusA().equals(new Point(1, 0));
  assert(e.getFocusB().equals(new Point(0, 1));
  ...
}
```

Breaking LSP: A Client Using Ellipse

Write client code that uses the contracts given by `Ellipse`:

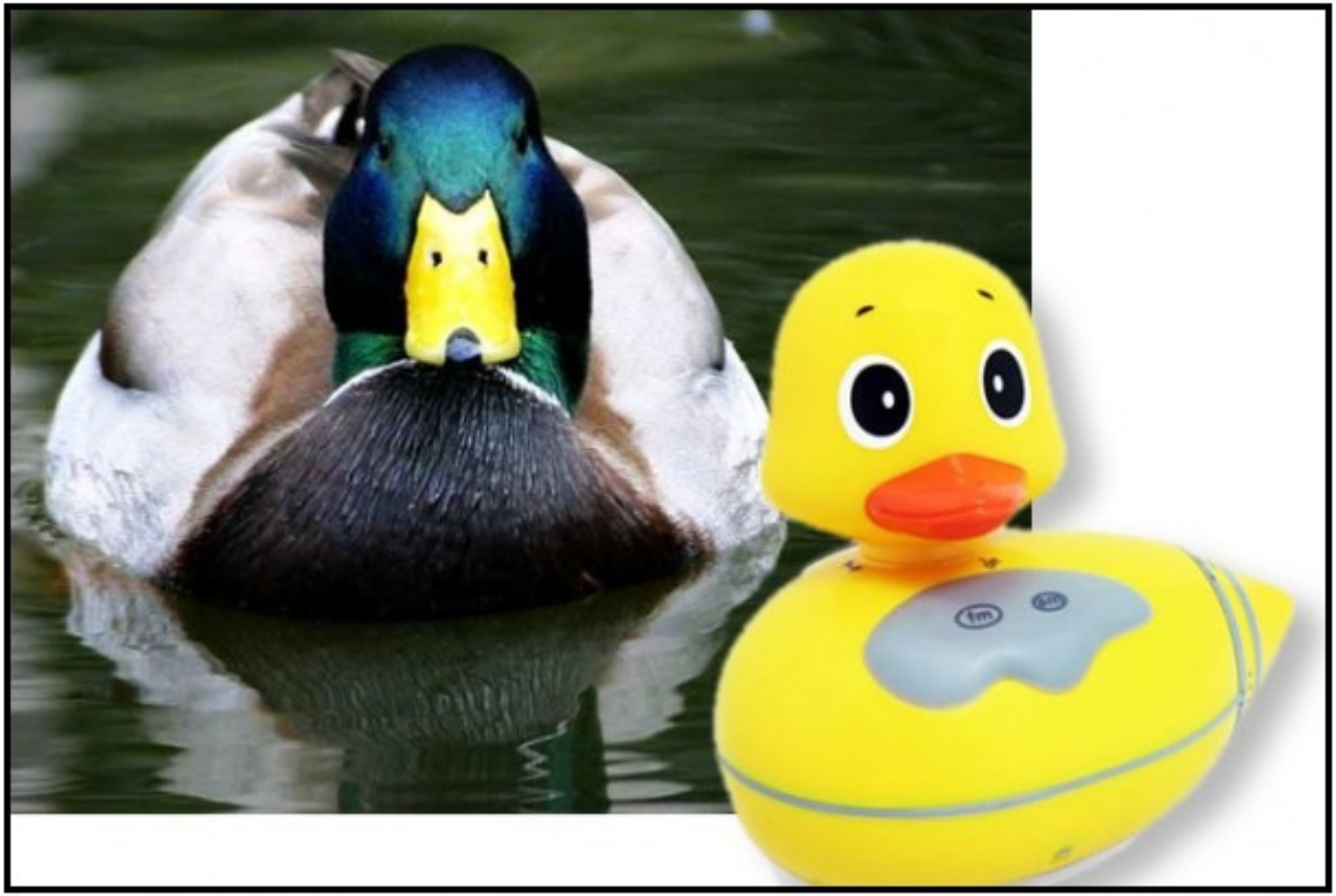
```
void f(Ellipse e) {
  e.setFoci(new Point(1, 0), new Point(0, 1));
  assert(e.getFocusA().equals(new Point(1, 0));
  assert(e.getFocusB().equals(new Point(0, 1));
  ...
}
```

Obviously, this breaks when a `Circle` is passed to `f`

We are changing the contract in the subclass: This makes the abstraction worthless

LSP and Contracts

- A contract consists of
  - preconditions that must hold before a method call, otherwise the result is undefined (an exception is thrown, etc.)
  - guarantees that will be fulfilled when the method has been completed, called postconditions
- A derived class is substitutable for its base class if:
  1. Its preconditions are no stronger than the base class method.
  2. Its postconditions are no weaker than the base class method.
- When this holds, LSP is not violated



LISKOV SUBSTITUTION PRINCIPLE

If It Looks Like A Duck, Quacks Like A Duck, But Needs Batteries - You Probably Have The Wrong Abstraction

Interface Segregation Principle

Many client specific interfaces are better
than one general purpose interface

Clients should not be forced to depend upon interfaces they don't use.

- Don't put too much functionality into an interface:
  you can implement as many as you want
- Instead of using one large interface, use many small interfaces
- But: put functionality together belonging to the same responsibility
  (cohesion!)

Interface Segregation Principle: Example

```
interface IWorker {
  public void work();
  public void eat();
}

class Worker implements IWorker {
  public void work() { /* ... working */ }
  public void eat() { /* ... eating lunch */ }
}

class SuperWorker implements IWorker {
  public void work() { /* ... working faster */ }
  public void eat() { /* ... eating lunch */ }
}

class Manager {
  IWorker worker;
  public void setWorker(IWorker w) { worker=w; }
  public void manage() { worker.work(); }
}
```

Now add `Robot` that does `recharge` instead of `eat`

Interface Segregation Principle: Example

```
interface IWorker { public void work(); }
interface NeedsFood { public void eat(); }
interface NeedsRecharge { public void recharge() ; }

// ...

class SuperWorker implements IWorker, NeedsFood {
  public void work() { /* ... working faster */ }
  public void eat() { /* ... eating lunch */ }
}

class Robot implements IWorker, NeedsRecharge {
  public void work() { /* ... working */ }
  public void recharge() { /* ... mmmm */ }

class Manager {
  IWorker worker;
  public void setWorker(IWorker w) { worker = w; }
  public void manage() { worker.work(); }
}
```

Interface Segregation Principle

- ISP needs more effort in the design phase and may produce code of higher complexity
- But it's well invested time
- The code is more independent, and the design is more flexible
- What if the harm's already done, or you are using an existing library?

Interface Segregation Principle

- ISP needs more effort in the design phase and may produce code of higher complexity
- But it's well invested time
- The code is more independent, and the design is more flexible
- What if the harm's already done, or you are using an existing library?
- Use the Adapter pattern!

Special Adapter Example (for ISP)

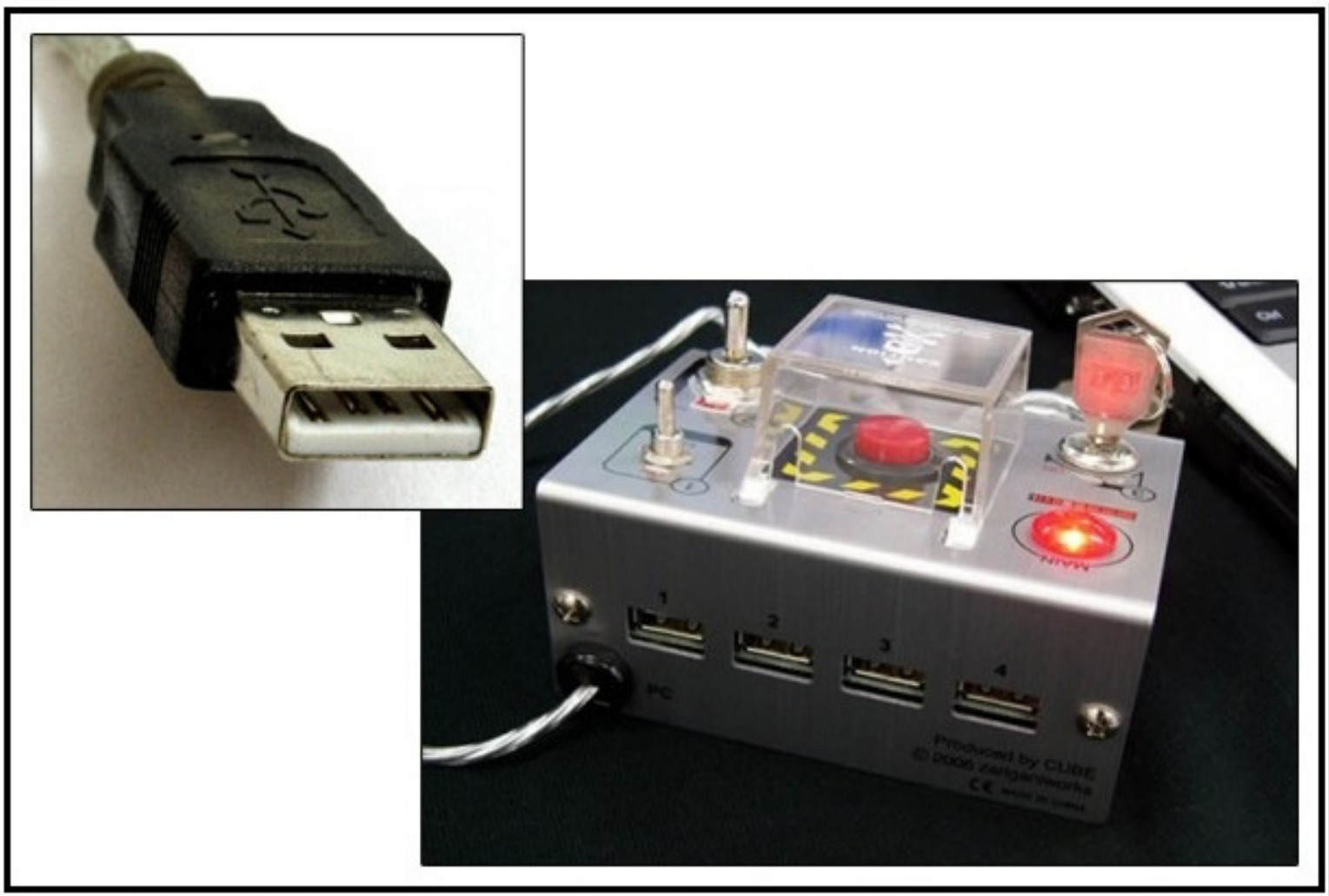
```
interface MonsterLegacyWorker {
  public void work();
  public void eat();
  public void recharge();
}

public class SuperWorker { /* ... */ }

// Legacy codes ends here ------------------------------

interface Worker { public void doWork(); }

public class WorkerAdapter implements Worker {
  private MonsterLegacyWorker worker;
  public WorkerAdapter(MonsterLegacyWorker w) { worker = w; }
  public void doWork() { worker.work() }
}
```



INTERFACE SEGREGATION PRINCIPLE

You Want Me To Plug This In, Where?

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.

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.

Classical Top-Down Design

1. Create a high-level description of the system

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.

Classical Top-Down Design

1. Create a high-level description of the system
2. Refine the design into modules and high-level classes

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.

Classical Top-Down Design

1. Create a high-level description of the system
2. Refine the design into modules and high-level classes
3. Create low-level implementations to provide the needed functionalities

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.

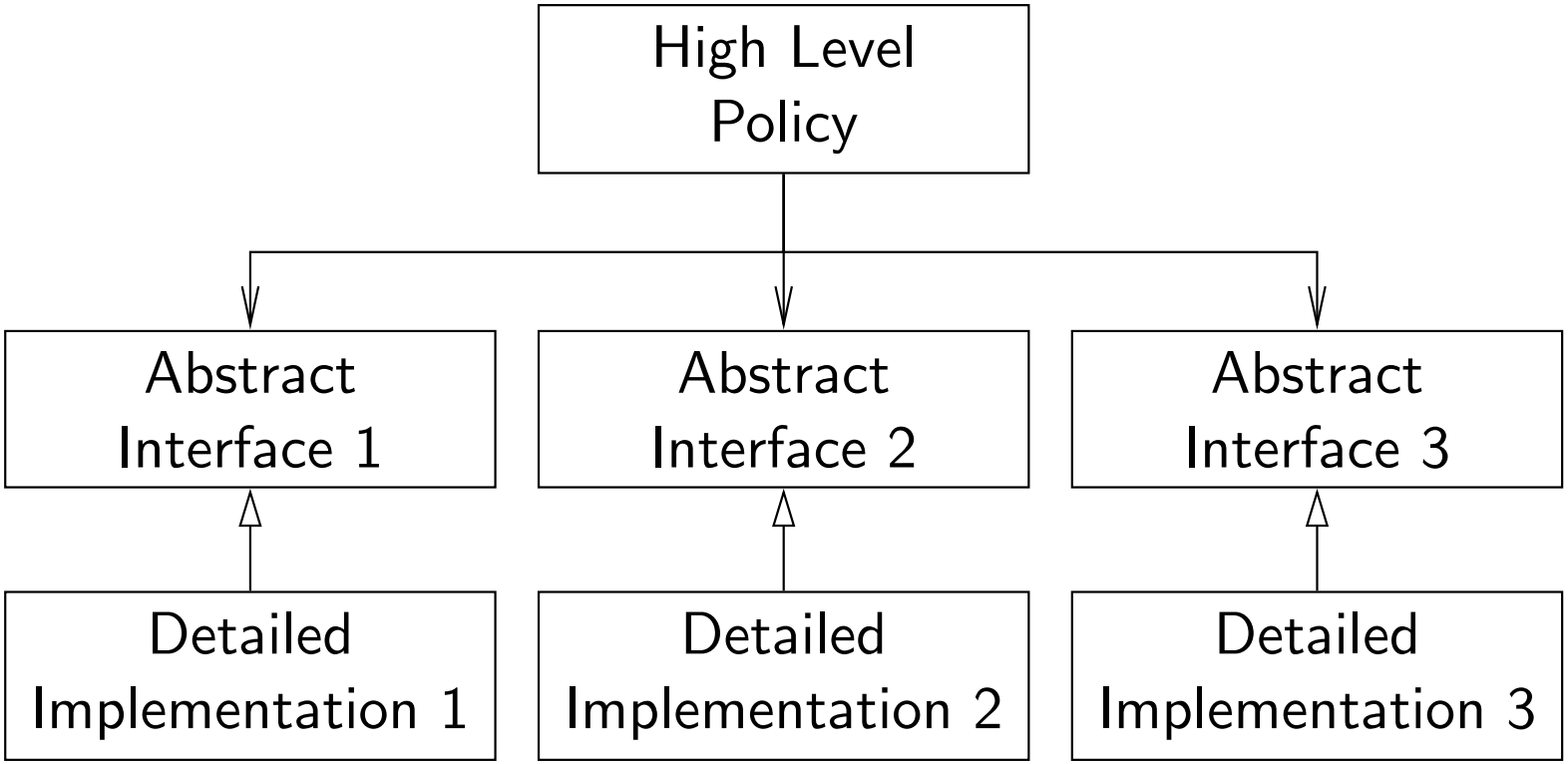
Classical Top-Down Design

1. Create a high-level description of the system
2. Refine the design into modules and high-level classes
3. Create low-level implementations to provide the needed functionalities

Consequences

- The high-level classes depend on the implementations
- the low-level classes are very hard to replace

Dependency Inversion Principle



Dependency Inversion Principle: Bad Example

```
public class PowerSwitch {
  public LightBulb bulb;
  public boolean on;

  public attach(LightBulb b) { bulb = b; }

  public void press() {
    if (on) {
      bulb.turnOff(); on = false;
    } else {
      bulb.turnOn(); on = true;
    }
  }
}
```

Dependency Inversion Principle Applied

```
public interface Switchable {
  public void turnOn();
  public void turnOff();
}

public interface Switch {
  public void attach(Switchable s);
  public void press();
}

public class LightBulb implements Switchable {
  public void turnOn() { /* Let there be light! */ }
  public void turnOff() { /* Into darkness */ }
}

public class PowerSwitch implements Switch {
  public Switchable client;
  public boolean on;

  public attach(Switchable c) { client = c; }
  public void press() {
    if (on) {client.turnOff(); on = false; }
    else {client.turnOn(); on = true; }
  }
}
```

Dependency Inversion Principle: Example II

What's wrong here?

```
public Result readComplicatedSyntax(String filename) {
  FileReader f = new FileReader(filename);
  while (f.canRead()) {
    char c = f.read();
    // Now follows a lot of complicated code
    // ...

    // even more complicated code
    // ...
  }
  return result;
}
```

Dependency Inversion Principle: Object Creation

- There is no way to instantiate objects of abstract classes (by definition)
- As a consequence, you always depend on a concrete class when creating objects (calling a constructor)

Dependency Inversion Principle: Object Creation

- There is no way to instantiate objects of abstract classes (by definition)
- As a consequence, you always depend on a concrete class when creating objects (calling a constructor)
- The elegant solution: The Factory pattern
- The Factory creates one concrete implementation of a common interface, based on arguments

Dependency Inversion Principle: Object Creation

- There is no way to instantiate objects of abstract classes (by definition)
- As a consequence, you always depend on a concrete class when creating objects (calling a constructor)
- The elegant solution: The Factory pattern
- The Factory creates one concrete implementation of a common interface, based on arguments
- There's an even more extreme version: the Abstract Factory pattern which uses (several) Factories to build objects

Factory: Simple Example

```
public interface Worker { /* ... */ }

public class WorkerFactory {
  public static Worker getWorker(int workload) {
    if (workload > 100) return new SuperWorker();
    else return new OrdinaryWorker();
  }
}
```

Factory Example: DIP Again

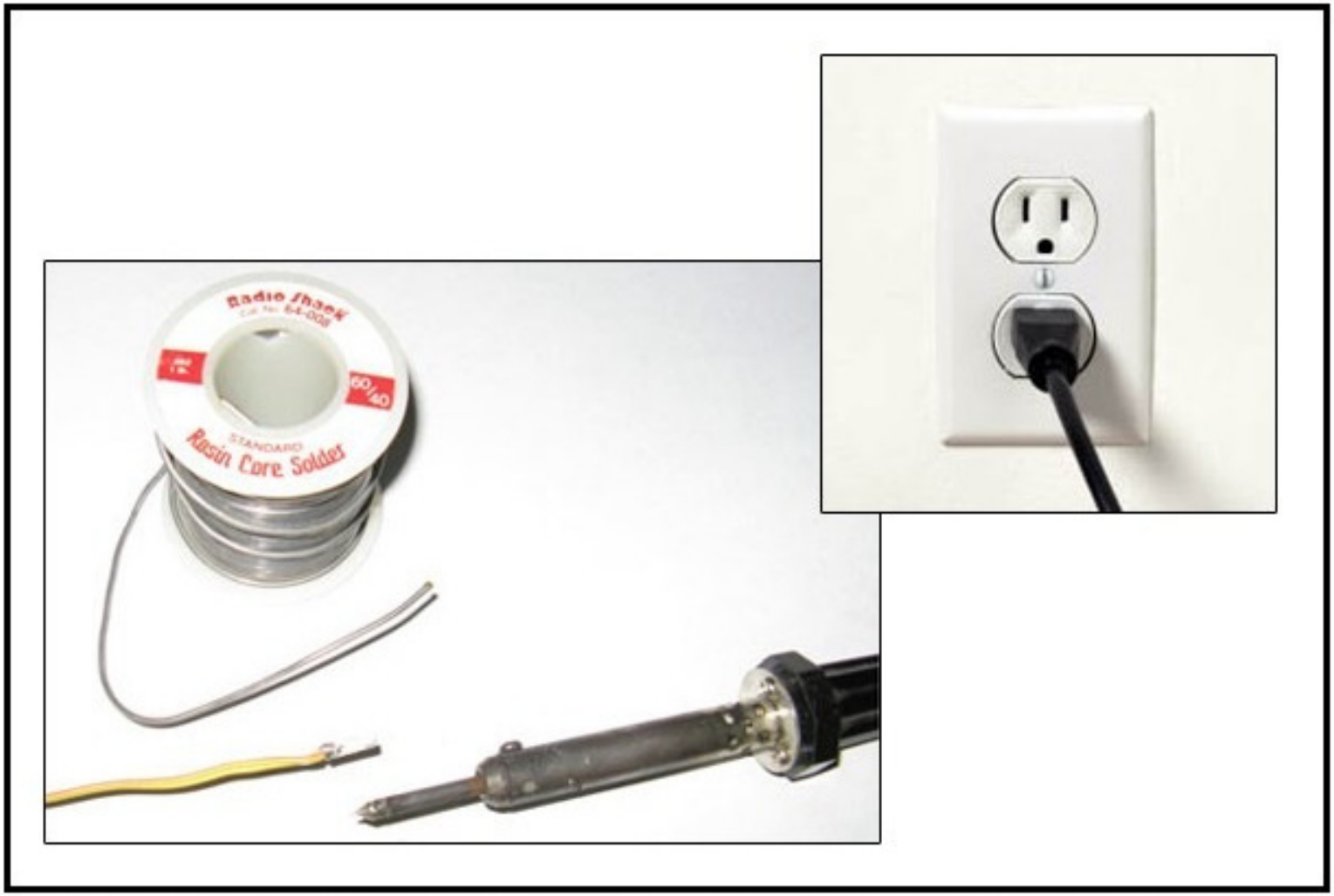
An example that uses the Prototype pattern:

```
public interface Copyable<T> { T copy(T object); }
public interface Worker extends Copyable<Worker> { /* ... */ }

public class WorkerFactory {
  HashMap<String, Worker> workerPrototypes = new HashMap<>();

  public void register(String type, Worker prototype) {
    workerPrototypes.put(type, prototype);
  }

  public Worker getWorker(String type) {
    Worker w = workerPrototypes.get(type);
    if (w != null) w = w.copy();
    return w;
  }
}
```



DEPENDENCY INVERSION PRINCIPLE

Would You Solder A Lamp Directly To The Electrical Wiring In A Wall?

Acknowledgement, Literature

For these slides, I borrowed massively from

- Robert C. Martin
  `www.objectmentor.com/resources/articles/Principles_and_Patterns.pdf`
- `http://www.oodesign.com`
- `https://en.wikipedia.org/wiki/SOLID_(object-oriented_design)`
- Pictures from `https://lostechies.com/derickbailey/2009/02/11/solid-development-principles-in-motivational-pictures/` and `http://www.abhishekshukla.com/net-2/solid-design-principles-open-closed-principle-ocp/`
- ... and tons more on the web