# P2: Exercise Session 9

Claudio Corrodi

May 6, 2016

# Dependency Problem

- Our code should not depend on classes, only on interfaces

- But at some point we need to specify the concrete type…

```
public class FileVisitor {
  private final FileSystem fileSystem;
  public FileVisitor() {
    fileSystem = new WindowsFileSystem();
  }
  //...
}
```

# Dependency Problem

- Our code should not depend on classes, only on interfaces

- But at some point we need to specify the concrete type…

```
public class FileVisitor {
  private final FileSystem fileSystem;
  public FileVisitor() {
    fileSystem = new WindowsFileSystem();
  }
  //...
}
```

Client / dependant

# Dependency Problem

- Our code should not depend on classes, only on interfaces

- But at some point we need to specify the concrete type…

```
public class FileVisitor {
  private final FileSystem fileSystem;
  public FileVisitor() {
    fileSystem = new WindowsFileSystem();
  }
  //...
}
```

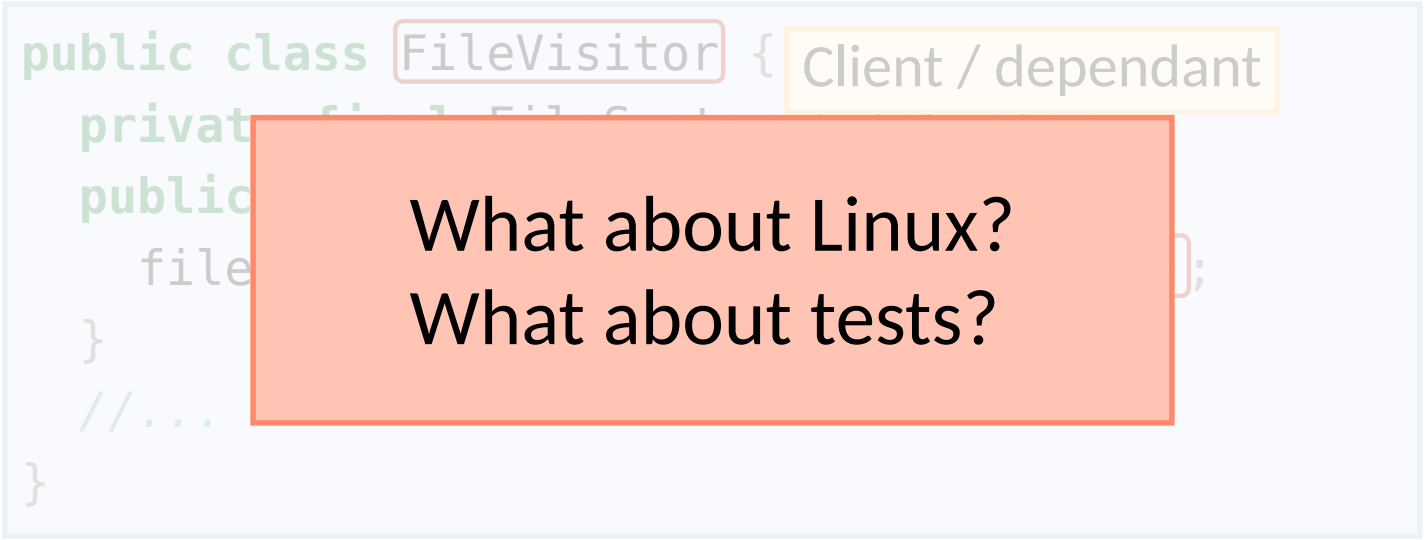
Client / dependant

Service / dependency

# Dependency Problem

- Our code should not depend on classes, only on interfaces

- But at some point we need to specify the concrete type…



# Constructor Injection

One solution: Pass the service as an argument to the constructor.

```
public class FileVisitor {
  private final FileSystem fileSystem;
  public FileVisitor(FileSystem injectedSystem) {
    fileSystem = injectedSystem;
  }
  //...
}
```

# Constructor Injection

One solution: Pass the service as an argument to the constructor.

```
public class FileVisitor {
  private final FileSystem fileSystem;
  public FileVisitor(FileSystem injectedSystem) {
    fileSystem = injectedSystem;
  }
  //...
}
```

Easy to implement, safe (`fileSystem` can not be **`null`**)

ugly signature (possibly lots of parameters)

# Constructor Injection: The `main` method

```
public class Main {
  public static void main(String[] args) {
    //construct service
    FileSystem fileSystem = new WindowsFileSystem();
    //construct Client by injecting service
    FileVisitor visitor = new FileVisitor(fileSystem);
    //run the main logic
    visitor.visit("C:\\Windows");
  }
}
```

# Setter Injection

```
public class FileVisitor {
  private FileSystem fileSystem;

  public FileVisitor() {
  }

  public void setFileSystem(FileSystem injectedFileSystem) {
    fileSystem = injectedFileSystem;
  }

  //...
}
```

# Setter Injection

```
public class FileVisitor {
  private FileSystem fileSystem;

  public FileVisitor() {
  }

  public void setFileSystem(FileSystem injectedFileSystem) {
    fileSystem = injectedFileSystem;
  }

  //...
}
```

Easy to implement, clean constructors

Can't be sure client sets a `FileSystem` before using it!

# Setter Injection: The `main` method

```
public class Main {
  public static void main(String[] args) {
    //construct service
    FileSystem fileSystem = new WindowsFileSystem();
    //construct client
    FileVisitor visitor = new FileVisitor();
    //inject service
    visitor.setFileSystem(fileSystem);
    //run the main logic
    visitor.visit("C:\\Windows");
  }
}
```

# Setter Injection: The `main` method

```
public class Main {
  public static void main(String[] args) {
    //construct service
    FileSystem fileSystem = new WindowsFileSystem();
    //construct client
    FileVisitor visitor = new FileVisitor();
    //inject service
    visitor.setFileSystem(fileSystem);
    //run the main logic
    visitor.visit("C:\\Windows");
  }
}
```

Careful! Do not forget to set the filesystem before using the visitor!

# Service Locator

```
public Game() {
        this.board = new Board();
        this.scanner = new ConsoleScanner();
        this.logger = new ConsoleWriter();
}
```

Same problem: Game should not need to know about `ConsoleScanner` and `ConsoleWriter`, just about the interfaces `Scanner` and `Writer`

# Service Locator

Idea: Central *service locator* creates objects upon request, client becomes oblivious of concrete types.

```
public Game() {
    this.board = ServiceLocator.instance().getBoard();
    this.scanner = ServiceLocator.instance().getScanner();
    this.logger = ServiceLocator.instance().getLogger();
}
```

```
public abstract class ServiceLocator {
  private static ServiceLocator instance;
  protected ServiceLocator() {}
  public static ServiceLocator instance() {
    if (instance == null) {
      instance = defaultServiceLocator();
    }
    return instance;
  }
  public static ServiceLocator defaultServiceLocator() {
    return new DefaultServiceLocator();
  }
  public static void
        setServiceLocator (ServiceLocator serviceLocator) {
    instance = serviceLocator;
  }
  public abstract Logger getLogger();
  public abstract Scanner getScanner();
  //...
}
```

```
public abstract class ServiceLocator {
  private static ServiceLocator instance;
  protected ServiceLocator() {}
  public static ServiceLocator instance() {
    if (instance == null) {
      instance = defaultServiceLocator();
    }
    return instance;
  }
  public static ServiceLocator defaultServiceLocator() {
    return new DefaultServiceLocator();
  }
  public static void
        setServiceLocator (ServiceLocator serviceLocator) {
    instance = serviceLocator;
  }
  public abstract Logger getLogger();
  public abstract Scanner getScanner();
  //...
}
```

Only one instance of `ServiceLocator`

```
public abstract class ServiceLocator {
  private static ServiceLocator instance;
  protected ServiceLocator() {}
  public static ServiceLocator instance() {
    if (instance == null) {
      instance = defaultServiceLocator();
    }
    return instance;
  }
  public static ServiceLocator defaultServiceLocator() {
    return new DefaultServiceLocator();
  }
  public static void
       setServiceLocator (ServiceLocator serviceLocator) {
    instance = serviceLocator;
  }
  public abstract Logger getLogger();
  public abstract Scanner getScanner();
  //...
}
```

Set the instance if needed

```
public abstract class ServiceLocator {
  private static ServiceLocator instance;
  protected ServiceLocator() {}
  public static ServiceLocator instance() {
    if (instance == null) {
      instance = defaultServiceLocator();
    }
    return instance;
  }
  public static ServiceLocator defaultServiceLocator() {
    return new DefaultServiceLocator();
  }
  public static void
        setServiceLocator (ServiceLocator serviceLocator) {
    instance = serviceLocator;
  }
  public abstract Logger getLogger();
  public abstract Scanner getScanner();
  //...
}
```

Get services (implemented later)

# Service Locator Implementations

```
public class DefaultServiceLocator extends ServiceLocator {
        @Override
        public Scanner getScanner() {
                return new ConsoleScanner(...);
        }
        //...
}
```

```
public class TestServiceLocator extends ServiceLocator {
        @Override
        public Scanner getScanner() {
                return new ScriptedScanner(...);
        }
        //...
}
```

# Service Locator: Using the test locator

```
@Test
public void someTest() {
    // configure service locator
    TestServiceLocator locator = new TestServiceLocator();

    // set up 'simulated' input
    locator.setScriptedInput(UP, DOWN, RIGHT, RIGHT, ...);

    // set locator instance
    ServiceLocator.setLocator(locator);

    // Create and use game as usual
    Game game = new Game();

    // ...
}
```

# Coding Issues

```
@Test
public void aTest() {
  try{
    game.run(program);
  }catch(RenderException e){
    assertTrue(false);
  }
}
```

# Coding Issues

```
@Test
public void aTest() throws RenderException {
  game.run(program);
}
```

Test fails if exception is thrown.

# Coding Issues

```
catch(Exception e){
    System.out.println("Could not load level!");
    if(e.getClass().equals(java.io.FileNotFoundException.class)){
        System.out.println("File not found!");
    }
    if(e.getClass().equals(IOException.class)){
        System.out.println("Could not read from the file " + path);
    }
    if(e.getClass().equals(NumberFormatException.class)){
        System.out.println("Either height or width could not be parsed!");
    }
    if(e.getClass().equals(InvalidSizeException.class)){
        System.out.println("Either height or width was 0!");
    }
    if(e.getClass().equals(MultiplePlayerException.class)){
        System.out.println("There is more than one Player!");
    }
    // ...
}
```

# Coding Issues

```
catch (FileNotFoundException e) {
    System.out.println("File not found!");
}
catch (IOException e) {
    System.out.println("Could not read from the file " + path);
}
catch (NumberFormatException e) {
    System.out.println("Either height or width could not be parsed!");
}
catch (InvalidSizeException e) {
    System.out.println("Either height or width was 0!");
}
catch (MultiplePlayerException e) {
    System.out.println("There is more than one Player!");
}
```

# Coding Issues

```
catch (FileNotFoundException|IOException|NumberFormatException|
       InvalidSizeException|MultiplePlayerException e) {
    System.out.println(e.getMessage());
}
```

# Require `toString()` implementation

Problem: `toString()` is implemented in class `Object`, how can we make sure that we don't forget to implement it?

# Require `toString()` implementation

Problem: `toString()` is implemented in class `Object`, how can we make sure that we don't forget to implement it?

```
interface Entity {
  public String toString();
}
class Player implements Entity { ... }
```

# Require `toString()` implementation

Problem: `toString()` is implemented in class `Object`, how can we make sure that we don't forget to implement it?

```
interface Entity {
  public String toString();
}
class Player implements Entity { ... }
```

This doesn't work! We still get the default implmentation from `Object` and are not required to implement `toString()`...

# Require `toString()` implementation

Problem: `toString()` is implemented in class `Object`, how can we make sure that we don't forget to implement it?

```
abstract class Entity {
  public abstract String toString();
}
class Player extends Entity { ... }
```

This works, we are forced to (re-)implement `toString()`.

Is it needed? I prefer writing tests…

# Exercise 9

- Use service locator and dependency injection in your Sokoban game
  - Introduce interfaces if needed!
  - E.g. depend on `IGame` instead of `Game`
- Compare the two
  - Which one is better suited for your needs?
  - Which one do you prefer?
  - …
- Use git tags again, branching encouraged

# Remaining schedule

- Next week
  - Lecture: Design patterns
  - Exercise hour: **Exam preparation session**
- 3 more exercises (including this one)
  - You need to pass 9 out of 11 for the Testat!