Software-Architecture — Dependency Injection, Aspect Oriented Programming and Google Guice

Prof. Dr. Axel Böttcher

25. Oktober 2011
Objectives

- Understand the Concept of Dependency Injection and AOP.
- Rate the different methods for DI/AOP
- Understand the concept of binding
- Build Injectors using Guice
Arthur C. Clarke’s third law

Any sufficiently advanced technology is indistinguishable from magic.

Clarke ist the author of „2001: A Space Odyssey“ (German title: „2001: Odyssee im Weltraum“)
Wertpapier-class revisited

Blob-class (Wertpapier does everything):
Wertpapier-class revisited

Blob-class (Wertpapier does everything):

Concerns separated (access to server delegated to concrete class):
Dependency Inverted:
DI by hand (immediate instanciation)

```java
public class Foo {
    private Bar bar;
    public Foo(){
        bar = new Bar();
    }
}
```
Constructor/Setter Injection

Constructor Injection:

```java
1 class Foo {
2    private final Bar bar;
3    public Foo(Bar bar) {
4        this.bar = bar;
5    }
6 }
```
2. Using a Factory (class or method)

```java
class Foo {
    private final Bar bar = BarFactory.createBar();
    public Foo() {
        ...
    }
}
```

2. Service Locator as alternative for line 2:

   Bar bar = new ServiceLocator().get("Bar");

These two solutions require discrimination between test and production code.
class Foo {
  private final Bar bar;
  @Inject
  public Foo(Bar bar) {
    this.bar = bar;
  }
}

A whole object hierarchy is created with
Guice.createInjector(new
AppGuiceModule()).getInstance(Foo.class);
The power of Guice with large Object Graphs

```java
class Module extends AbstractModule{
    public void configure() {
        bind(IStockInfoProvider.class).
            to(StockInfoProvider.class);
        //... // more bindings
    }
}

class TestModule extends AbstractModule{
    public void configure() {
        bind(IStockInfoProvider.class).
            to(MockStockInfoProvider.class);
        //... // more bindings
    }
}
```

Instance of Portfolio and all depending objects correctly "wired".
Guice’s Types of injection

1. Constructor Injection:

```java
class Foo {
    private final Bar bar;
    @Inject
    public Foo(Bar bar) {
        this.bar = bar;
    }
}
```
Guice’s Types of injection

2. Setter Injection (Called immediately after instanciation):

```java
public class Foo {
    private Bar bar;
    @Inject
    public void setBar(Bar bar) {
        this.bar = bar;
    }
}
```

2. Field Injection: @Inject private Bar bar;

Use of this injection method is discouraged (final not possible, ...).

2. Requested injection:

```java
injector.injectMembers(object);
```

Typical: object=this
Implementing Modules

1. One Module (a class inheriting from `com.google.inject.AbstractModule`) must be implemented.
2. Override the method `protected void configure()`.
3. Provide your bindings there:
   - `bind(Interface.class). to(ConcreteClass.class);`
   - `bind(Interface.class). toInstance(concreteInstance);`
   - `bind(Interface.class). annotatedWith(Annotation.class). to...`
4. Instantiate objects using the Module: `Guice.createInjector(new AppGuiceModule()).getInstance(Foo.class);`
5. Discriminate between test and production by the module (using different modules or discrimination within module).
Annotation-based bindings

Hilfsklassen von Guice:

1. Annotation `Named`

2. Klasse mit statischer Methode `Names.named`

```java
1 bind(Integer.class).annotatedWith(Names.named("port"))
2 .toInstance(5);
```

```java
class IpConnection {
1   @Inject
2   public IpConnection(@Named("port") int port) {
3   }
4 }
5 }
6 }
```
Providers can be used as a Foo-factory in any class:

```java
@Inject Provider<Foo> fooProvider;
```

A module can be configured to provide providers:

```java
@Provides
public Foo[] provideFoos() {
  return new Foo[] { new foo(), new foo(), };
}
```
The Singleton Pattern
Kent Beck: How do you provide global variables in a language without global variables? Don’t. Your programs will thank you for taking the time to think about design instead.

But sometimes you might really need them:

- Pools for Connections (Network, Databases, ...)
- Device Drivers
Classical Implementation with Lazy creation

Danger: Not thread-safe!

class SingletonClass {
    private static SingletonClass theInstance;

    private SingletonClass(){}

    public static SingletonClass getInstance(){
        if(theInstance == null)
            theInstance = new SingletonClass();
        return theInstance;
    }
}
Classical Implemetation with Eager creation

Eager: eifrig, arbeitsfreudig

```java
class SingletonClass {
    private static final SingletonClass theInstance =
        new SingletonClass();

    private SingletonClass(){}

    public static SingletonClass getInstance(){
        return theInstance;
    }
}
```
Enum-based implementation

```java
enum SingletonClass {
    INSTANCE;

    // methods etc.
}
```
public class ChocolateBoiler {
    private boolean empty = true;
    private boolean boiled = false;
    private static ChocolateBoiler theInstance = new ChocolateBoiler();

    private ChocolateBoiler() {}

    public static ChocolateBoiler getInstance() {
        return theInstance;
    }

    public void fill() {
        if (isEmpty()) {
            empty = false;
            boiled = false;
        }
    }

    public void drain() {
        if (!isEmpty() && isBoiled()) {
            empty = true;
        }
    }

    public void boil() {

Why does this test fail?

```java
import static org.junit.Assert.*;
import org.junit.*;

public class ChocolateBoilerTest {

    ChocolateBoiler boiler = ChocolateBoiler.getInstance();

    @Test
    public void testBoil() {
        boiler.fill();
        boiler.boil();
        assertTrue("content of Boiler should be boiled",
                   boiler.isBoiled());
    }

    @Test
    public void testFill() {
        boiler.boil();
        assertFalse("Boiling w/o filling must result in boiled=",
                    boiler.isBoiled());
    }

```
bind(IChocolateBoiler.class).to(ChocolateBoiler.class).in(Singleton.class);
or
bind(IChocolateBoiler.class).to(ChocolateBoiler.class).asEagerSingleton();
or
bind(IChocolateBoiler.class).toInstance(new ChocolateBoiler());
bind(IChocolateBoiler.class).to(ChocolateBoiler.class).in(Singleton.class);

or

bind(IChocolateBoiler.class).to(ChocolateBoiler.class).asEagerSingleton();

or

bind(IChocolateBoiler.class).toInstance(new ChocolateBoiler());

Class ChocolateBoiler is then no longer implemented as a Singleton!
Often you have requirements that repeat over subsystems:

- Logging
- Security
- Caching
- Transaction handling
- Performance monitoring
Often you have requirements that repeat over subsystems:

- Logging
- Security
- Caching
- Transaction handling
- Performance monitoring

These are called cross-cutting concerns
Anatomy of a classic business logic class

```java
public class SomeBusinessClass {

    public void someOperation() {
        ensureAuthorization();
        startTransaction();
        logStartOfOperation();
        DO_THE_OPERATION();
        logEndOfOperation();
        endTransaction();
    }
}
```
Idea for implementation

Use annotations to do some magic work

```java
public class SomeBusinessClass {

    @Secured
    @Transactional
    @Log

    public void someOperation() {
        DO_THE_OPERATION();
    }

}
```
Guice offers the possibility to implement method interceptors\(^1\).

**Definition of the Interceptor Pattern:** An interceptor performs additional logic by intercepting method invocations on an object.

Such a method-interceptor is bound (e.g. to an annotation) and called by guice before the method is called:

```java
bindInterceptor(
    Matchers.any(),
    Matchers.annotatedWith(Secured.class),
    new SecuringInterceptor()
);
```

For Matchers and their implementations please consider the package `com.google.inject.matchers`.

\(^1\)interception: das Abfangen, die Überwachung
Javadoc of the `MethodInterceptor`-interface

Intercepts calls on an interface on its way to the target. These are nested „on top“ of the target.
The user should implement the `invoke(MethodInvocation)` method to modify the original behavior. E.g. the following class implements a tracing interceptor (traces all the calls on the intercepted method(s)):

```java
1 class TracingInterceptor implements org.aopalliance.intercept.MethodInterceptor {
2     org.aopalliance.intercept.MethodInterceptor {
3         Object invoke(MethodInvocation i) throws Throwable {
4             System.out.println("method \"+i.getMethod()+" \"called on \"
5                 i.getThis()+" \"with \"args \""+i.getArguments());
6             Object ret=i.proceed();
7             System.out.println("method \"+i.getMethod()+" \"returns \""+
8                 return ret;
9         }
10    }
```