Dependency Injection with Guice

Technion – Institute of Technology
236700
Class decoupling

• Consider the following interface:

```java
public interface CreditCardProcessor {
    void process(PizzaOrder order, CreditCard cc);
}
```

• A highly coupled implementation:

```java
class BillingService {
    private CreditCardProcessor p = new VisaProcessor();
    public void chargeOrder(PizzaOrder o, CreditCard cc) {
        p.process(o, cc);
        ...
    }
}
```
Class decoupling (cont.)

What’s the problem? We used interfaces!

• **new always applies** dependency/coupling!
  • Although we use interfaces, the implementation is highly **coupled** with VisaProcessor
  • It is much harder to **test** using **real** dependencies
  • **How can we replace it with mocks?**

Solution

• Pass collaborators (dependencies) to the object
• This is called **Dependency Injection**
DI – Constructors

• We can pass the object at the construction of the object

```java
public class BillingService {
    private final CreditCardProcessor processor;
    public BillingService(CreditCardProcessor p) {
        processor = p;
    }
}
```

• Dependencies are injected upon object instantiation
  • Object is always in a correct state
• Class can be immutable
• We can implement default behavior using a default constructor
• We can use constructor injection to provide a default constructor in an inheriting class
  • We saw this when we discussed about using OOP in tests: we injected the actual object under test to the parent constructor
Factories

What if we actually want to create objects in our class?

- `new` applies dependency/coupling
- If we could override constructors like in Smalltalk/Squeak...
- We could fake it by passing a “constructor-interface”, called an Abstract Factory
- This is an example of favouring composition over inheritance
Abstract Factory Pattern

```java
Processor
  process(PizzaOrder, CreditCard)
  implements

VisaProcessor
  process(PizzaOrder, CreditCard)
  <<creates>>

ProcessorFactory
  create(): CreditCardProcessor
  implements

VisaProcessorFactory
  create(): CreditCardProcessor
  <<creates>>
```

Author: Gal Lalouche - Technion 2016 ©
Abstract Factory – Participants

To change the dependencies, simply send a new factory:

```java
class VisaProcessorFactory implements ProcessorFactory {
    public VisaProcessor create() {
        return new VisaProcessor();
    }
}

class PayPalProcessorFactory implements ProcessorFactory {
    public PayPalProcessor create() {
        return new PayPalProcessor();
    }
}

class MockProcessorFactory implements ProcessorFactory {
    public MockProcessor create() {
        return new MockProcessor();
    }
}
```
Abstract Factory – Client

Client:

class BillingService {
    private final ProcessorFactory pf;
    public BillingService(ProcessorFactory pf) {
        this.pf = pf;
    }
    public void chargeOrder(PizzaOrder o, CreditCard cc) {
        Processor p = pf.create();
        p.process(o, cc);
        ...
    }
}
Guice

A Dependency Injection framework
Dependency Injection – Drawbacks

What happens when we need to perform **deep injections** with no **defaults**?

• Consider the following simple design:
  • A is dependent on B, which is dependent on C, which is dependent on D...
  • In order to create an instance of A, we need to inject it with an instance of B, which needs to be injected with an instance of C which needs to be injected with an instance of D...
    • A a = new AImpl(new BImpl(new CImpl(new DImpl())));
  • **What happens if we have parallel dependencies?**
    • e.g., one for local database, another for cloud, one for development
    • It is hard to wire everything up without mistakes
      • We could just put everything in a single **Abstract Factory**...
      • ...but that breaks **encapsulation**
Dependency Injection Frameworks

- Completely separate object configuration from creation, allowing for cleaner code
- Can inject recursively
- Easy to manage parallel instantiations
Guice – Supplier

• Supplier code (using annotations):

```java
class MyBillingService implements BillingService {
    private final CreditCardProcessor processor;

    @Inject
    public MyBillingService(CreditCardProcessor p) {
        processor = p;
    }
    ...
}
```

• `@Inject` can be applied to:
  - Constructors
  - Methods
    - All methods annotated with `@Inject` will be invoked after the object’s instantiation
  - Fields (not recommended)
Guice – Complex Dependencies

- Dependencies can also be complex to create:

```java
class VisaCreditCardProcessor implements CreditCardProcessor {
    private final BankAccount account;
    private final TxApprover txApprover;

    @Inject
    public VisaCreditCardProcessor(BankAccount a, TxApprover t) {
        account = a;
        txApprover = t;
    }
    ...
}
```

- BankAccount may have dependencies of its own
- TxApprover may be an interface
Guice – Module

- Dependencies are described in a separate **Module** class:

```java
public class BillingModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(CreditCardProcessor.class).to(VisaCreditCardProcessor.class);
        bind(TxApprover.class).to(VisaIsraelTxApprover.class);
        bind(BillingService.class).to(MyBillingService.class);
    }
}
```

- Client code is now:

```java
Injector injector = Guice.createInjector(new BillingModule());
BillingService billingService = injector.getInstance(BillingService.class);
```

- **Guice will recursively** build the requested **binding** objects
- **In this case a** `MyBillingService` **object will be created with DatabaseTxLog and a VisaProcessor**
Factories with provider\(<T>\)s

How do we pass factories?

• Naïve solution: simply inject an AbstractFactory

```java
class MyBillingService implements BillingService {
    @Inject
    public MyBillingService(CreditCardProcessorFactory pf) {
        this.factory = pf;
    }
}
```

• We have to create an explicit VisaProcessorFactory class to bind to

• We bind both to VisaProcessor and VisaProcessorFactory; that’s not very DRY...
Factories with provider<T>s (cont.)

- Using providers achieves just that!

  ```java
  class MyBillingService implements BillingService {
      @Inject
      public MyBillingService(Provider<CreditCardProcessor> pf) {
          this.factory = pf;
      }
  }
  ```

- We don’t have to bind the provider explicitly, since we already bound VisaProcessor

- This makes the Guice injector instance in essence a FactoryFactory, but it is okay in this case because we don’t need any extra code to support it!
Outcome

• What we’ve achieved?
  • Both client and supplier are **decoupled** from **implementation** details
  • All dependency logic is executed at runtime and is contained in **Modules**
    • A module encapsulates all creation logic in our program
    • Testing & Mocking is easy – since we’re essentially using constructor injection, we can ignore Guice in our tests if we want

• Side-note:
  • Other DI frameworks (e.g. Spring) use XML files to define dependencies
  • This enables changing the code’s behavior without recompiling
  • But it also has its disadvantages (e.g. it’s less type safe)
Dependency Injection – Is it really that important?

• If we want to write real units tests, then we have to isolate our units
• If we want to isolate our units, then we have to use mocks
• If we want to use mocks, we have to inject them somehow

But we aren’t doing it (just) for the tests!

• If we use dependency injection, our code is less coupled, more modular, robust, reusable, cohesive, cleaner and DRYer.

Remember, **Testable Code** $\Rightarrow$ **Good Code**!
Don’t over-design

- A little coupling is okay
- You should use injection for service objects, and `new` for data objects
  - generally speaking, it’s okay to use `new ArrayList<>()`
- If we replaced every field with a factory our code would blow up
- Know when to use constructor dependency injection, and when you need a factory
- Guice can make debugging much harder...
- Remember, the creational logic needs to be decided somewhere
  - Be careful of creating `FactoryFactory`!
Appendix

Guice advanced binding

- Can bind concrete types by:
  - Type
  - Name
  - Annotation

- Can bind to:
  - A class (new instance every time)
  - An object (a specific instance)
  - Generic classes
Annotation binding – annotations

Binding to annotated class:

```java
public class PersistentLogger {
    @Inject
    public PersistentLogger(@Persistent TransactionLog l) {
        this.l = l;
    }
}
```

- **Persistent** TransactionLogs should bind to SqlTxLog
- **Un-persistent** TransactionLogs should bind to StringTxLog

```java
public class BillingModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(TransactionLog.class).to(StringTxLog.class);
        bind(TransactionLog.class)
            .annotatedWith(Persistent.class)
            .to(SqlTxLog.class);
    }
}
```
Binding constants using @Named

- We can use an annotation “shortcut” using @Named

```java
import javax.inject.Inject;

public class SqlTxLog {
    @Inject
    public SqlTxLog(@Named("JDBC URL") String connectionURL,
                     @Named("login timeout seconds") Integer timeout)
    {
        ... 
    }
}
```

- Bind to specific constant object (instantiated only once)

```java
import javax.inject.Inject;

public class SqlTxLogModule extends AbstractModule {
    @Override
    protected void configure() {
        bind(String.class)
            .annotatedWith(Names.named("JDBC URL"))
            .toInstance("jdbc:mysql://localhost/pizza");
        bind(Integer.class)
            .annotatedWith(Names.named("login timeout seconds"))
            .toInstance(10);
    }
}
```
More on Binding to Instance

There are two ways to bind an instance:

• **Using `toInstance` passing an instance**
  ```java
  bind(SqlConnection.class)
      .toInstance(new SqlConnection());  // eager
  ```

• **Using `.to(SomeClass.class).in(Singleton.class)`**
  ```java
  bind(BillingService.class)
      .to(MyBillingService.class).in(Singleton.class);  // lazy
  ```

• A single instance is created by Guice
• Dependencies are injected according to the Module
• We can also use `@Singleton` on the *bound* class
Creational logic with @Provides

If we have some non-trivial creational logic we can write it in a @Provides method in the Module

• If this method receives any arguments they will be injected
• We can also annotate the return type

```java
public class MyPaymentModule extends AbstractModule {
    @Override
    protected void configure() { ... }

    @Provides
    @Persistent // will be provided to dependencies annotated with @Persistent
    TransactionLog provideTxLog(@Named("JDBC URL") String conURL) {
        DatabaseTxLog transactionLog = new DatabaseTxLog();
        transactionLog.setJdbcUrl(conURL);
        transactionLog.setThreadPoolSize(30);
        return transactionLog;
    }
}
```
Binding Generics

Binding generics is complicated by **type erasure**

- Binding classes with generic dependencies:

```java
class A {
    @Inject public A(List<String> list) {...}
}
...
bind(new TypeLiteral<List<String>>(){}).to(new TypeLiteral<ArrayList<String>>(){});
...
injector.getInstance(A.class); // A will be injected an ArrayList
```

- Binding generic classes:

```java
class B<T> {
    @Inject public B(List<T> list) {...}
}
...
bind(new TypeLiteral<List<Integer>>(){}).to(new TypeLiteral<ArrayList<Integer>>(){});
...
injector.getInstance(new Key<B<Integer>>(){}); // B will be injected an ArrayList
```