Dependency Injection with Guice

Technion – Institute of Technology
236700
Complex Dependency Injection

- Deep dependencies (with no default)
  - A depends on B, which depends on C, which depends on D...
  - A a = new AImpl(new BImpl(new CImpl(new DImpl())));
  - What if each class has two or more dependencies?

- Consistent dependencies
  - Pass VisaProcessor to all CreditCardProcessor dependencies

- Parallel dependencies
  - Several dependencies that need to have matching values
  - One for Visa, one for Paypal, one for testing
  - What about composing these
    - Combine configuration for Visa processing, local database, distributed request handling
Dependency Injection Frameworks

- Completely separate object configuration from creation, allowing for cleaner code
- Understand the dependency graph on its own
  - We just have to tell it what dependencies to bind to
- Provide abstractions for dependencies injection
  - And abstractions are easy to compose
- Solve all of the problems in the previous slide...
- But come at a cost of added complexity
  - of course
Dependency Injection Libraries – Java World

- **Guice**
  - [https://github.com/google/guice](https://github.com/google/guice)
- **Dagger**
  - [http://square.github.io/dagger/](http://square.github.io/dagger/)
- **Spring**
- **Pico**

Author: Gal Lalouche - Technion 2017 ©
Injected class

- Configuring injected classes (using annotations):

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

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

- `@Inject` can be applied to:
  - (private) Constructors
  - (private) Methods (best to avoid)
    - Methods annotated with `@Inject` will be invoked after the object’s instantiation
  - (private) Fields (best to avoid)
AbstractModule and configure()

- Dependencies are described in a separate Module class:
  ```java
  public class VisaModule extends AbstractModule {
    @Override
    protected void configure() {
      bind(CreditCardProcessor.class).to(VisaCreditCardProcessor.class);
    }
  }
  ```

- `bind(x.class).to(y.class)` tells Guice to create and pass an instance of Y whenever it needs an X

- Client code:
  ```java
  Injector injector = Guice.createInjector(new VisaModule());
  BillingService billingService = injector.getInstance(BillingService.class);
  ```

- We did not bind `BillingService`; Guice already has all the information it needs to create it!
  - If Guice can't create a class, we will get a runtime exception when we call getInstance
Complex Dependencies

• Dependencies can also be **complex** to create:

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

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

• BankAccount **may have dependencies of its own**
• TxApprover **may be an interface**
• Guice handles all **recursive** dependencies itself
Advanced binding

- Modules can be:
  - Combined when creating an injector
  - Composed
  - Overridden

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

- Can bind to:
  - A class (new instance every time)
  - An object (a specific instance)
  - Generic classes
Composing modules

There are several ways to compose modules

• **Simplest way:** `createInjector` accepts `varargs`

```java
Guice.createInjector(new FooModule(), new BarModule());
```

• Installing modules (composition over inheritance):

```java
install(new BarModule()); // invoke in the configure method
```

• **Complicated ways:** Modules utility class

  • `Modules.combine` composes modules

```java
Module foobar = Modules.combine(new FooModule(), new BarModule());
```

  • Both of the above throw an exception on `duplicate` bindings when attempting to create an injector

  • `use Modules.override` to avoid that

```java
Modules.override(new FooModule()).with(new BarModule());
```

• Composing modules gives us a lot of power, but it can be a debugging nightmare
Annotation bindings

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);
    }
}
```
@Named bindings

- A slightly easier way of binding multiple types to different values

```java
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
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 instances

There are two ways to bind an instance:

• **Using** `.toInstance`  
  ```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  
  • This will create an instance **per binding**  
  • We can also use `@Singleton` on the **bound** class  
    • Ensures a **single** instance will be created, even if there are multiple bindings  
  • There are other interesting **scopes**
@Provides

If we have some non-trivial creational logic, we can write it in a method in the module, and annotate it with @Provides

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

- We can also extract this logic to a class to
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>>(){});
...
// A will be injected an ArrayList
injector.getInstance(A.class);
```

- Binding generic classes:

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

How do we pass factories?

• Naive solution: simply define and inject an Abstract Factory

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

• Cons?
  • We have to create two classes (factory interface and implementation)
    • If we used `Supplier<T>`, we need ugly `TypeLiterals`
  • We bind both `VisaProcessor` and `VisaProcessorFactory`
    • that’s not very `DRY`...
**Provider<T>s**

- We can use Guice **Providers to inject factories**

  ```java
  class MyBillingService implements BillingService {
      @Inject
      public MyBillingService(Provider<CreditCardProcessor> pf) {
          this.factory = pf;
      }
  }
  // module:
  bind(CreditCardProcessor.class).to(VisaProcessor.class);
  ```

- We don’t have to bind the provider **explicitly**, since we already provided a binding to `VisaProcessor`

- Easy to replace with lambdas in tests

- If your factory needs additional parameters in order to create, then you have to inject it explicitly

- If your factory takes a mixture of injected and non-injected arguments, use **AssistedInjected** (requires an additional maven dependency)
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**
  • Modules **abstract** the creation logic of our program
  • Modules are **composable**
  • Not **perfectly typesafe** (but we fail fast when we mess up)
• Code flow can be **very** hard to understand
  • Especially when you start combining modules, overriding, etc.
  • "Spooky action at a distance"
• 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 not as type safe)