Dependency Injection, CDI and Guice

Fritz Solms

November 17, 2014
Dependency Injection

- Software design pattern implementing
  - inversion of control.
- Client requiring resource (e.g. service) does not
  - determine which concrete resource to use,
  - create or lookup the resource.
- Instead
  - resource reference initialized by client environment.
The Problem

Assume an instance of an **OrderProcessor** requires some **InvoiceGenerator** to generate invoice for order

```java
public interface InvoiceGenerator {
    public Invoice generateInvoice(Order order);
}

public class OrderProcessor {
    public OrderConfirmation processOrder(Order order) {
        ...
        Invoice invoice = invoiceGenerator.generateInvoice(order);
        ...
    }
} 
```
The Problem: Potential List of Concerns

1. **Tight coupling:**
   - Replace InvoiceGenerator implementation class.

2. **Reusability** of OrderProcessor:
   - In environments where different class is used to generate invoices.

3. **Breaking single responsibility principle:**
   - In addition to its core responsibility
   - OrderProcessor creates/sources instances of its service providers.

4. **Dependency inversion:**
   - low level class should *never* have dependency on high-level class

5. **Architecture/framework lock-in:**
   - Tightly couple application code to infrastructural classes.
   - e.g. when service provider is entity manager, adapter, ...

6. **Dependent object initialization:**
   - May be complex and may change

7. **Difficult to unit test:**
   - Unit testing environment: use mock object for InvoiceGenerator
   - Testing IoC can inject mock objects.
The solution provided by dependency injection

- **Inversion of Control (IoC)** container which
  - Scans for resources (including classes) on which classes have dependencies,
  - Takes over life cycle management for resource
    - e.g. instantiates and initializes resources,
  - Container initializes reference to resources in client
    - prior to resource being used.
Elements of Dependency Injection

a) *Inversion of Control* (IoC) container which manages & provides dependencies
b) an interface (contract) for the resource,
c) implementation of a resource
d) the client class depending on the service;
e) an injector object responsible for injecting the resource into the client.
Dependency Injection Implementations

- Widely implemented across the *Java* eco-system
  - CDI (Context and Dependency Injection)
    - standard API spec
  - *SquareDagger, PicoContainer*
    - minimalist/light-weight frameworks
  - *Spring DI*
    - Very feature-rich DI framework implementation
    - meant to be used within the *Spring* framework
  - *Google Guice*
    - Framework independent feature rich DI framework
- *C++* DI frameworks:
  - *Walleroo* (released under Boost license)
- *JavaScript*
  - *AngularJS DI*
- *C#*
  - *Spring.Net*
Context and Dependency Injection (CDI)

- **Community-managed standard**
  - for Dependency Injection in Java

- It provides
  - A **CDI context**
    - enabling one to bind life cycle & interactions
    - to extensible CDI life cycle contexts.
  - **Dependency Injection**
    - type-safe dependency injection of components managed by CDI context
What Does CDI Provide?

1. A standard for context and dependency injection which supports:
   - Stand-alone & Java-EE embedded CDI containers
   - Type Safety:
     - CDI uses interfaces to resolve injections
   - Wide injection target:
     - any Java object whose life-cycle can be managed by a CDI container.
     - e.g. enterprise beans, persistence contexts, Web service references, . . .
   - Decorators:
     - Can decorate injected components.
   - Events:
     - Send and receive type-safe events with loose coupling.
   - Interceptors:
     - Can associate interceptors with components.
   - Support for Unified Expression Language:
     - facilitates injection into facelets, . . .
   - Service Provider Interface (SPI):
     - enables 3’rd party frameworks to integrate with CDI

Decoupling through CDI

CDI decouples clients and service providers in a couple of ways:

1. *Server implementation may vary*
   - and CDI injects an instance of that class which
     - currently implements the interface for the server, and
     - satisfies certain characteristics.

2. *Decouples client and service life cycles*
   - by making components contextual,
   - with automatic life cycle management

3. *Decoupling message producers & consumers*
   - through events mechanisms.

4. *Decoupling of orthogonal concerns*
   - through interceptors.
Understanding CDI

- It helps to understand how CDI works under the hood.
What is a CDI Bean?

- CDI Bean = *injectable object*

- Any concrete Java class which
  
  - Must have appropriate constructor
    
    - Default constructor
    
    - Constructor annotated with `@Inject`
  
  - is not a static inner class

- Examples
  
  - JSF managed beans,
  
  - Local & remote enterprise beans,
  
  - Persistence contexts, JSF managed beans,
  
  - JNDI resources (e.g. queues and topics, connection pools, . . .)
  
  - Web service references, . . .
Specifying State Retention Period With Bean Scope

- **CDI Beans are Singleton**
  - within some scope
  - state maintained within that scope.

- **CDI bean scopes:**
  - `@RequestScoped`
    - State maintained single user interaction (e.g. single HTTP request)
  - `@SessionScoped`
    - State maintained across interactions within user session.
  - `@ApplicationScoped`
    - Shared instance (state) across all interaction with (web) application
  - `@Dependent`
    - The injected bean shares the life cycle of the context it is injected into. This is the default scope.
    - This is the *default scope*.
  - `@ConversationScoped`
    - Multiple cycles of a JSF request life cycle.
    - Allows for program determined conversation start and end.
  - `@Singleton`
    - State shared among all clients.
Conversation Scope

```java
import javax.enterprise.context.Conversation;
import javax.enterprise.context.ConversationScoped;

@ConversationScoped
public class MyConversationBean implements Serializable {
    private Conversation conversation;

    public String startConversationalProcess() {
        if (!FacesContext.getCurrentInstance().isPostback() && conversation.isTransient()) {
            conversation.begin();
        }
        return "firstPageName";
    }

    public String endConversationalProcess() {
        if (!conversation.isTransient()) {
            conversation.end();
            return "firstPageAfterConversation";
        }
        @Inject
        private Conversation conversation;
    }
```
Beans injected into and accessible from Java code.
- At times, need beans accessible also from EL expressions.
- e.g. from facelets.

Annotate with @Named
- Will use bean class name by default.
- Can assign different name via @Named("BeanName").
Injecting a CDI Bean

- Annotate field or setter

```java
@Stateless
Class MyStatelessSessionBean
{
  
  public list<Client> getOverdrawnClients()
  {
    Query query = persistenceContext.createNamedQuery("overdrawn");
    return query.getResultList();
  }

  @Inject
  private PersistenceContext persistenceContext;
}
Qualifiers

- CDI beans are singletons in some scope.
- Sometimes we require different implementations of a bean type.
  - Then need to define a qualifier.
- e.g. different message sender types concurrently deployed:
  - SmsMessageSender, JabberMessageSender, ...
- Define annotation for qualifier:

```java
import static java.lang.annotation.ElementType.FIELD;
import static java.lang.annotation.ElementType.METHOD;
import static java.lang.annotation.ElementType.PARAMETER;
import static java.lang.annotation.ElementType.TYPE;
import java.lang.annotation.RetentionPolicy;
import java.lang.annotation.Retention;
import java.lang.annotation.Target;
import javax.injectQualifier;

@Qualifier
@Retention(RUNTIME)
@Target({TYPE, METHOD, FIELD, PARAMETER})
public @interface Jabber {}
```
Qualifying a Specific Bean Implementation Class

- Use different qualification annotation
  - to annotate different bean types.
- Facilitates
  - Having different bean implementations concurrently deployed.
  - Being able to specify which bean implementation should be injected.

```
package za.co.solms.training.cdi.MessageSender;

@Jabber
public class JabberSender implements MessageSender {
    public boolean sendMessage(String message, String recipient) {
        // code for sending message over Jabber
    }
}
```
Injecting Qualified CDI Bean

- In order to inject a qualified CDI bean, we annotate it with both
  - @Inject
  - the qualification annotation

```java
@Named
@RequestScoped
class QualifiedMessagingClient {
  public String processOrder() {
    // some fancy code
    messageSender.sendMessage(recipient, message);
    return "orderConfirmation";
  }

  @Inject @Jabber
  private MessageSender messageSender;

  private String messageText;
  private String message;
}
```

- Note that client still decoupled from implementation class.
A Simple Example: MessageSender Bean

- Interface for the server

```java
package za.co.solms.training.cdi.MessageSender;

@Named // So that it can be accessed from EL expressions
@Dependent // could omit this as this is default
public interface MessageSender {
    public boolean sendMessage(String message, String recipient);
}
```

- An Implementation for the server

```java
public class EmailSender implements MessageSender {
    public boolean sendMessage(String message, String recipient) {
        // code for sending message over Jabber
    }
}
```
Understanding CDI

A Simple Example: MessageSender

- A client having a message sender dependency injected.

```java
@Named
@RequestScoped
class MessagingClient {
  public String processOrder()
  {
    // some fancy code
    messageSender.sendMessage(recipient, message);

    return "orderConfirmation";
  }

  @Inject
  private MessageSender messageSender;

  private String messageText;
  private String message;
}
```

- We can undeploy the email sender and deploy a Jabber or SMS message sender
  - Client will use new sender without any changes.
  - Can unit test client in mocking environment.
How are Bean References Obtained?

- CDI’s BeanProvider
  - implements Singleton per scope.
  - provides bean (resource) references

```java
T getReference(Bean<T> type, Qualifier... qualifiers);
```

- Not called directly
  - from Annotations and Expression Language Processor
- Qualifiers required if multiple instances of same type of CDI bean required.
Google Guice

- Developed by Google and released under Apache License.
- Open source dependency injection framework used to
  - bind concrete implementation classes to interfaces, and
  - inject implementations into other classes via constructors, methods or fields.

- Claim
  - First generic framework for dependency injection that relies heavily on annotations and generics for configuration.

- Offers all the usual benefits of dependency injection:
  - Eliminates need to create Java classes the conventional way. (new or factories etc).
  - Simplifies code maintenance
  - Simpler unit testing
  - Simpler reuse
  - ...
The Google Guice API designs defined a set of values that governed and influenced their decisions, and the resultant product. These include:

- **Main aim:**
  - to increase application development speed and reduced complexity.

- **Minimalist framework**
  - Not intended to be a “one size fits all”.
  - Each feature to be justified by at least 3 use cases.

- **Instead:**
  - Well defined mechanism for extending framework.
Minimalist example: Simple Interface & Class

```java
public interface BasicArithmetic {

    /** Adds two integer numbers and returns the result. */
    public Integer add(Integer number1, Integer number2);
}

Implementing class:

```java
public class BasicArithmeticImpl implements BasicArithmetic {

    public Integer add(Integer number1, Integer number2) {
        return number1 + number2;
    }
}
```
Minimalist example: Class Implementing Guice Module

- Used to configure Guice *Injector*.
- Configure method:
  - Concrete implementations bound to interfaces
  - Binding enables injector to instantiate/provide appropriate implementations

```java
import com.google.inject.Binder;
import com.google.inject.Module;

public class BasicArithmeticModule implements Module {

    public void configure(Binder binder) {
        binder.bind(BasicArithmetic.class).to(BasicArithmeticImpl.class);
    }
}
```
Simple example: Client class

- Create *Injector* for module.
- Obtain implementations for interfaces from injector.

```java
import com.google.inject.Guice;
import com.google.inject.Injector;

public class Client {
    public static void main(String[] args) {
        Injector injector = Guice.createInjector(new BasicArithmeticModule());
        BasicArithmetic basicArithmetic = injector.getInstance(BasicArithmetic.class);
        System.out.println(basicArithmetic.add(2, 2));
    }
}
```
Google Guice API

- 4 main interfaces and classes
  1. Binder
  2. Injector
  3. Module
  4. Guice
Binder

- **Binding**
  - Service of binding implementations to interfaces.
  - Fluent interface design: intuitive method naming
    - e.g. **bind** an interface **to** an implementation
  - Methods calls 'chained'
    - increases readability
Binder

- Binding implementation to interface
  
  ```java
  1 binder.bind(BasicArithmetic.class).to(BasicArithmeticImpl.class);
  ```

- Directly instantiate instance bound to interface and returned when needed.
  
  ```java
  1 binder.bind(BasicArithmetic.class).to(new BasicArithmeticImpl());
  ```

- Can specify Provider
  - Factory creating implementation instances
    
    ```java
    1 binder.bind(BasicArithmetic.class).to(new BasicArithmeticProvider());
    ```
Injector

- Responsible for
  - constructing instances for interfaces, and
  - maintaining instances
- Set of default bindings maintained by injectors.
**Injector**

- Simple example: explicitly retrieve object currently deployed to realize contract:

  ```java
  BasicArithmetic basicArithmetic = injector.getInstance(BasicArithmetic.class);
  ```

- Can query map of all bindings associated with Injector:

  ```java
  Map<Key, Binding> allBindings = injector.getBindings();
  ```
Module

- Maintains set of bindings of interface onto classes or providers.
- *Injectors* retrieve bindings info from modules
- *AbstractModule* provides some boiler plate code for module implementations.
Utility class
- Factory of injectors.
- i.e. creates injector for specified module.
- Maintains injectors for modules.

The class is supplied with an instance of a module when requested to construct an `Injector`.

```java
1 Injector injector = Guice.createInjector(new AddModule());
```
Annotations

- Fully embraces Java annotations
- Set of annotations to
  - define & control object creation
  - configure injector behaviour.
Inject annotation

- 3 ways to inject instances into class
  1. on constructor
  2. on getter
  3. on field

- Example:

```java
public class Client {
  @Inject //Constructor injection.
  public Client(BasicArithmetic basicArithmetic) {...}
}
```

- Implementation would have to be bound in a module.
ImplementedBy annotation

- Used to specify class that must bound to interface.
  - typically to specify default binding
  - Use with care
    - interface should have no knowledge of implementations

- Example:

```java
@ImplementedBy(BasicArithmeticImpl.class)
public interface BasicArithmetic {

    /** Adds two integer numbers and returns the result. */
    public Integer add(Integer number1, Integer number2);
}
```
Object creation can be customised
- by implementing the *Provider* interface.

**Providers**
- Factory classes
- Localize logic needed to create and initialize implementations.
- Could
- dynamically instantiate different implementations
- initialize implementations to suit environment

**Can annotate interface with *ProvidedBy***
- to specify provider to be used to instantiate appropriate implementation.

```java
1 @ProvidedBy(BasicArithmeticProvider.class)
2 public interface BasicArithmetic
3 {
4   ...
5 }
```
Singleton annotation

- By default:
  - New instance constructed on `getInstance()`
  - $\Rightarrow$ potentially multiple instances.

- Often
  - injected resources should be singleton service providers.
  - Annotate as `Singleton`:
    - ensures same instance returned every time.

```java
@Singleton
public interface BasicArithmetic {
    public Integer add(Integer number1, Integer number2);
}
```
JSR 300 integration

- Should use non-proprietary, JSR-330-compliant Java dependency injection annotations

<table>
<thead>
<tr>
<th>JSR-330 (javax.inject)</th>
<th>Guice (com.google.inject)</th>
<th>Interchangeability</th>
</tr>
</thead>
<tbody>
<tr>
<td>@Inject</td>
<td>@Inject</td>
<td>Interchangeable (With T&amp;Cs ¹)</td>
</tr>
<tr>
<td>@Named</td>
<td>@Named</td>
<td>Interchangeable</td>
</tr>
<tr>
<td>@Qualifier</td>
<td>@BindingAnnotation</td>
<td>Interchangeable</td>
</tr>
<tr>
<td>@Scope</td>
<td>@ScopeAnnotation</td>
<td>Interchangeable</td>
</tr>
<tr>
<td>@Singleton</td>
<td>@Singleton</td>
<td>Interchangeable</td>
</tr>
<tr>
<td>@Provider</td>
<td>@Provider</td>
<td>Guice provider extends JSR provider.</td>
</tr>
</tbody>
</table>

¹ JSR-330 places additional constraints on injection points.
- Fields must be non-final
- Methods must be non-abstract with no type parameters.
Writing a custom provider: Interface

```java
public interface Person {
  public String getName();
}
```
Writing a custom provider: Provider

```java
public class MockPersonProvider implements Provider<Person>
{
    @Override
    public Person get()
    {
        return new MockeryOfAPerson();
    }
}
```
Writing a custom provider: Client

```java
import com.google.inject.Guice;
import com.google.inject.Injector;
import com.google.inject.Module;

public class Client {
    public static void main(String[] args) {
        Injector injector = Guice.createInjector(new Module()
            @Override
            public void configure(Binder binder) {
                binder.bind(Person.class).toProvider(MockPersonProvider.class);
            }
        );
        Person person = injector.getInstance(Person.class);
    }
}
```

- **Unit testing:**
  - Use provider of mock objects
    - test component only
- **Integration testing & Production deployment:**
  - Use provider of production objects
Examples

- In this section we take a look at a few examples. These examples have been selected to illustrate various concepts and usage scenarios within Google Guice.
Binding annotation example

- Guice can select concrete implementation to inject based on annotation.

- First we define an interface.

```java
public interface President{}
```

- Then define two annotations differentiating between implementation instances.

```java
import java.lang.annotation.*;
import com.google.inject.BindingAnnotation;

@Retention(RetentionPolicy.RUNTIME)
@BindingAnnotation
@Target(ElementType.LOCAL_VARIABLE)
public @interface Good{}
```
Dependency Injection, CDI and Guice

Google Guice

Examples

# Binding annotation example

```java
import java.lang.annotation.*;
import com.google.inject.BindingAnnotation;

@Retention(RetentionPolicy.RUNTIME)
@BindingAnnotation
@Target(ElementType.LOCAL_VARIABLE)
public @interface Bad{}
```

- Create two discrete implementations that realise the *President* interface.

```java
public class BelovedRuler implements President{}
```

```java
public class Dictator implements President{}
```

- Define module specifying implementations associated with annotations.
- Both implementations associated with same interface.
Binding annotation example

```java
1 import com.google.inject.*;
2
3 public class PresidentModule implements Module
4 {
5
6   public void configure(Binder binder)
7   {
8     binder.bind(President.class).annotatedWith(Good.class).to(BelovedRuler.class);
9     binder.bind(President.class).annotatedWith(Bad.class).to(Dictator.class);
10   }
11 }
```

- Test client illustrating how inject behaviour defined using annotation.
- *Good* annotation specified with interface,
Google Guice provides a “convenient” means of binding concrete instances to interfaces using an approach that is similar to the one covered in the previous example.

Involves using a common annotation called *Named*, which is associated with a name (key), and fulfills a similar role to an explicitly defined annotation.

We modified our example code to use the *Named* annotation for binding in the module, and injection in the client.
Named annotation example

```java
import com.google.inject.*;

public class PresidentModule implements Module {
    public void configure(Binder binder) {
        binder.bind(President.class).annotatedWith(Names.named("Good")).to(BelovedRuler.class);
        binder.bind(President.class).annotatedWith(Names.named("Bad")).to(Dictator.class);
    }
}
```
Named annotation example

```java
import com.google.inject.Guice;
import com.google.inject.Injector;
import com.google.inject.Module;

public class Client {

    public static void main(String[] args) {
        PresidentModule presidentModule = ;
        Injector injector = Guice.createInjector(new Module[]{presidentModule});

        @Named("Good") President president = (President) injector.getInstance(President.class);
    }
}
```
Approach may initially appear to be more convenient than binding annotation approach,

This convenience comes at the price of not having concepts explicitly defined and represented as concrete annotations.

You lose type ’awareness’, and IDE support that comes along with it.

Referential integrity between the names, now has to be manually maintained.