1 Dependency Injection

Dependency injection is a software design pattern which implements an inversion of control. Instead of the client requiring the resource determining which resource to use through either looking up or instantiating the resource, the resource reference is initialized by the client environment.

1.1 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);
        ...
    }
}
```

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.

1.2 The solution provided by dependency injection

The solution provided by the dependency injection pattern is to introduce an Inversion of Control (IoC) container which manages and provides dependencies. In particular, the IoC scans for resources (including classes) one which a class can have dependencies, takes over life cycle management for these resources (e.g. instantiates and initializes resources) and initializes the references or pointers to the resources in client classes.

The elements of a dependency injection include

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.

1.3 Implementations

Dependency injection implementations exist for many programming languages and frameworks.

• 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:
  – Walledoo (released under Boost license)
2 Context and Dependency Injection (CDI)

2.1 Introduction to CDI

Context and Dependency Injection (CDI) is a community–managed standard for dependency injection in Java. It provides

- a CDI context enabling one to bind lifecycle and interactions to extensible CDI life cycle contexts, and
- type-safe dependency injection of components managed by CDI context.

2.1.1 What does CDI provide?

CDI provides

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

2.1.2 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 and consumers* through an events mechanisms.
4. *Decoupling of orthogonal concerns* through interceptors.

2.2 Understanding CDI
It helps to understand how CDI works under the hood.

2.2.1 What is a CDI bean?
A CDI bean is any injectable object. The following can be injected:

- 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, . . .

2.2.2 Specifying the state retention period With bean scope
CDI Beans are singletons in some scope. The state of a CDI bean is maintained within its scope. The CDI bean scopes are

- `@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.

2.2.3 Assigning Beans EL Names

Beans are readily injectable into and accessible from Java code. At times, need beans accessible also from EL expressions, e.g. from facelets. To this end we commonly annotate CDI beans as @Named. By default the class name is used, but we can assign different name via @Named("BeanName").

2.2.4 Injecting a CDI bean

In order to inject one CDI bean into another, we annotate the field or setter with with @Inject. For example, below we inject a persistence context into a stateless session bean:

```java
@Stateless
class MyStatelessSessionBean
{
  public List<Client> getOverdrawnClients()
  {
    Query query = persistenceContext.createNamedQuery("overdrawn");
    return query.getResultList();
  }
  @Inject
  private PersistenceContext persistenceContext;
}
```

2.2.5 Using qualifiers

CDI beans are singletons in some scope. Sometimes we require different implementations of a bean type. In such cases we need to define a qualifier. For example, we might want different types of message senders concurrently deployed (e.g. a SmsMessageSender and a JabberMessageSender). We then require a qualifier annotation which enables users to specifically request a Jabber message sender:

```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 static java.lang.annotation.RetentionPolicy.RUNTIME;
import javax.inject.Qualifier;
```
We can now qualify different bean implementation types using the qualification annotations. This enables us to

- to have different bean implementations concurrently deployed, and
- to specify which bean implementation should be injected.

Now, if we want to inject a qualified bean, we annotate the bean field or setter with both, @Inject and the qualification annotation:

Note that client still decoupled from implementation class.

### 2.2.6 A simple example

Below is a simple interface for a message sender. The @Named annotation is to make the CDI bean accessible from EL expressions and the @Dependent annotation is unnecessary as it would have been the default anyway.
public class EmailSender implements MessageSender {
    public boolean sendMessage(String message, String recipient) {
        // code for sending message over Jabber
    }
}

The messaging client has the CDI container inject a message sender.

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

private MessageSender messageSender;
private String messageText;
private String message;

Should we replace the deployed message sender to a Jabber or SMS message sender, we would not have to make any changes to our messaging client. Furthermore, our client can be unit tested in a mocking environment.

2.2.7 How are bean references obtained?

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

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

The method is not called directly by the bean/resource client, but from annotations and Expression Language Processors. Qualifiers are required if multiple instances of same the type of CDI bean is required.

3 Google Guice

Developed by Google and released under the 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.

Google Guice claims to be the first generic framework for dependency injection that relies heavily on annotations and generics for configuration. Offers all the usual benefits of dependency injection:

- Eliminates the need to create Java classes the conventional way. (new or factories etc).
- Simplifies code maintenance and unit testing etc.
3.1 Design philosophy

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.

3.2 Simple example

A simple example, is now used to illustrate the dependency injection functionality offered by the API. We define an interface that offers a single service that adds two integers and returns the result.

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

A class is created that implements the method and provides the actual service’s functionality.

```java
public class BasicArithmeticImpl implements BasicArithmetic {
  public Integer add(Integer number1, Integer number2) {
    return number1 + number2;
  }
}
```

A class that implements the Google Guice `Module` interface is now defined.

```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);
  }
}
```
This module is used to configure a Google Guice Injector. Inside the configure method you will notice that the concrete implementation of the BasicArithmetic interface is bound to the interface. This binding enables the injector to instantiate the appropriate implementation when need.

```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));
    }
}
```

This code illustrates how the Injector is constructed using the Guice utility class and the module defined above. The injector is then used to create and return an implementation of the BasicArithmetic interface, although the actual implementation is never exposed, and is implicitly bound to the interface, as it should be.

### 3.3 Google Guice API

There are four main interfaces and classes that represent the main concepts that constitute the API. These are the Binder, Injector, Module and Guice.

#### 3.3.1 Binder

As its name implies the Binding offers binding-related services. More specifically, it offers the service of binding concrete implementations to their corresponding interfaces.

In the spirit of fluent interface design, an intuitive method naming strategy has been adopted, that is both easy to read and interpret. The methods call themselves are ‘chained’, which further increases readability. For example, bind an interface to an implementation.

In the simple example we bound the BasicArithmetic interface to the BasicArithmeticImpl class. There are a few ways to achieve this binding. You can bind the actual implementation class such as

```java
binder.bind(BasicArithmetic.class).to(BasicArithmeticImpl.class);
```

You can create directly instantiate an instance that is bound to the interface and returned when needed.

```java
binder.bind(BasicArithmetic.class).to(new BasicArithmeticImpl());
```

And finally you can specify a Provider that is essentially a custom factory containing logic used for instance creation.

```java
binder.bind(BasicArithmetic.class).to(new BasicArithmeticProvider());
```

#### 3.3.2 Injector

Injectors are responsible for constructing and maintaining the concrete instances that associated with the interfaces. A set of default bindings are maintained by the injectors.
Along with configuration information, these bindings are used to produce the concrete classes that are associated with the interfaces.

In the simple example, the injector was used to produce a concrete implementation of the `BasicArithmetic` interface.

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

It is possible to retrieve a map of all bindings that are associated with the Injector, using the `Injector.getBindings()`. For example:

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

### 3.3.3 Module

Implementations of the `Module` interface maintain the set of bindings between interface and classes or providers. Injectors use these module to retrieve bindings information for interfaces. There is an `AbstractModule` which provides some boiler plate code for concrete Module implementations.

### 3.3.4 Guice

`Guice` is a utility class which is a factory of injectors. It creates and maintains injectors for different modules. The class is supplied with an instance of a module when requested to construct an `Injector`.

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

### 3.4 Useful annotations

Google Guice fully embraces Java annotations for configuration. As such, a set of annotations has been provided that enables one to define and control object creation and injection behaviour.

#### 3.4.1 Inject annotation

There are three methods one can use to inject instances of an interface into a class. One can do so via the constructor, a method, or even directly into a field.

The `@Inject` annotation is used to identify where the injection should take place. For example, if we wanted to inject an instance of our `BasicArithmetic` instance into the client class we would place the annotation directly above the appropriate constructor. The implementation would need to be bound to the interface in a module class.

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

The `ImplementedBy` annotation explicitly identifies the class that must bound to the interface. Typically this would be used to identify a default implementation for an interface, although one must consider the disadvantages of doing so. An interface should ideally be fully independent of its realisations. Let see how this annotation would be used in our example.

```java
@ImplementedBy(BasicArithmeticImpl.class)
public interface BasicArithmetic {
    public Integer add(Integer number1, Integer number2);
}
```

3.4.3 ProvidedBy annotation

Object creation can be customised by implementing the `Provider` interface. Providers are essentially factory classes, where one can specify the logic needed to produce classes that implement a particular interface. A provider could dynamically instantiate different implementations and initialize them differently depending on the state of the environment. This provides can be identified using the `ProvidedBy` annotation on the interface.

```java
@ProvidedBy(BasicArithmeticProvider.class)
public interface BasicArithmetic {
    ...
}
```

3.4.4 Singleton

By default, a new object instance is constructed when the `getInstance()` utility method is called on the `Injector` class. If the method is called multiple times, multiple instances will be produced. If needed, the `Singleton` annotation can be used to restrict this behaviour, and ensure that the same instance is return each time the `getInstance()` method is invoked.

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

3.4.5 JSR 300 integration

From version 3, you can now choose whether to use the non-proprietary, specification-compliant Java dependency injection annotations, rather than the Google Guice annotations.

---

1. JSR-330 places additional constraints on injection points. Fields must be non-final, and methods must be non-abstract with no type parameters.
3.5 Writing a custom provider

Google Guice provides a facility that enables one to control how concrete instances are created by the Injector. To create such a factory you need to implement the Provider interface. The example that follows illustrates how such a provider is defined.

```java
public interface Person {
  public String getName();
}
```

First we define a interface that presents a Person.

```java
public class MockPersonProvider implements Provider<Person> {
  @Override
  public Person get() {
    return new MockeryOfAPerson();
  }
}
```

During unit testing, we may want to return a Mock test object representing the person, as opposed to the actual concrete implementation. To do so seamlessly, we change the provider to one that produces a Mock object.

```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);
  }
}
```
3.6 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.

3.6.1 Binding annotation example

In this example we show how the Google Guice is capable of selecting which concrete implementation to inject based on a specified annotation.

First we define an interface.

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

We then define two annotations that will be used to differentiate between implementation instances.

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

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

Likewise, we create two discrete implementations that realise the `President` interface.

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

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

A module is defined the specifies which implementation should be associated with which annotation. Both implementations are associated with the same interface.

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

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

And finally we create an example test client class that illustrated how the inject behaviour is defined using an annotation. In this case the `Good` annotation has been specified with the interface, which implies that the implementation that has been bound to that interface and annotation combination will be injected.
3.6.2 Named annotation

Google Guice provides what it deems to be a more convenient means of binding concrete instances to interfaces using an approach which is similar to the one covered in the previous example. This approach involves using a common annotation called Named, which is associated with a name (key), and fulfills a similar role to an explicitly defined annotation. In the example that follows, we have modified our example code to use the Named annotation for binding in the module, and injection in the client.

```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);
    }
}
```

```java
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);
    }
}
```

While this 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 also lose type 'awareness', and IDE support that comes along with it. Referential integrity between the names, now has to be manually maintained.