DCI Tools for Java

The following is a guide for the DCI Tools library, created by maxant, for enabling DCI in Java. It is not a guide to DCI and requires prerequisite knowledge of DCI. See http://www.maxant.co.uk/whitepapers.jsp for more information about DCI.

A download of the library is available from http://www.maxant.co.uk/tools.jsp.

This guide documents version 1.0.0 of the library.

Introduction

DCI (Data, Context, Interaction) is a paradigm used in software development. See http://en.wikipedia.org/wiki/Data,_Context,_and_Interaction for more details.

From a technical point of view, DCI injects behaviour into data objects so that they can assume a given role in order to interact with other objects in relevant roles. As such, system behaviour is implemented outside of the classes defining the data objects (domain model objects) which assume these roles. In languages which support it, this behaviour can be injected (or added) into existing objects, for example using traits. Java however, being statically typed, does not currently allow such things. In order to be able to implement DCI oriented solutions in Java, a dynamic proxy can be used instead, to simulate such method injection. It is not a *pure* DCI solution, but to all intents and purposes gives the programmer the impression that the data object has had behaviour added to it. This allows the programmer to read and review the code in a manner as intended by the inventors of the DCI paradigm.

This guide discusses the implementation provided by the library, and how it is intended to be used.

Implementation

In the ideal world, a data object would have methods added to it dynamically in the context, at the point where the context decides which role the object needs to play. The objects interface (i.e. the methods it exposes) is temporarily increased during the context of the interaction. Java cannot do this. Java also cannot *dynamically* join interfaces together to create a new interface, because you would need to assign such a dynamic conglomeration to a variable, and the compiler needs to know its type, hence no longer being dynamic. So the best that can be done is to define an interface which specifies all the methods which a role will expose. This interface is sometimes known as the methodless-role, or role-name.

In Java, this interface can be used as the façade which sits in front of a dynamic proxy. The dynamic proxy contains an invocation handler which then decides how to handle incoming calls, whether they should be handled by the actual data object, or by the class which provides the role implementation (system behaviour), called the methodful-role, or role-method class. This can be depicted using the following example, where a `BankAccount` object is the data object, and the `SourceAccount` is an instance of the class providing the system behaviour which the role exhibits.

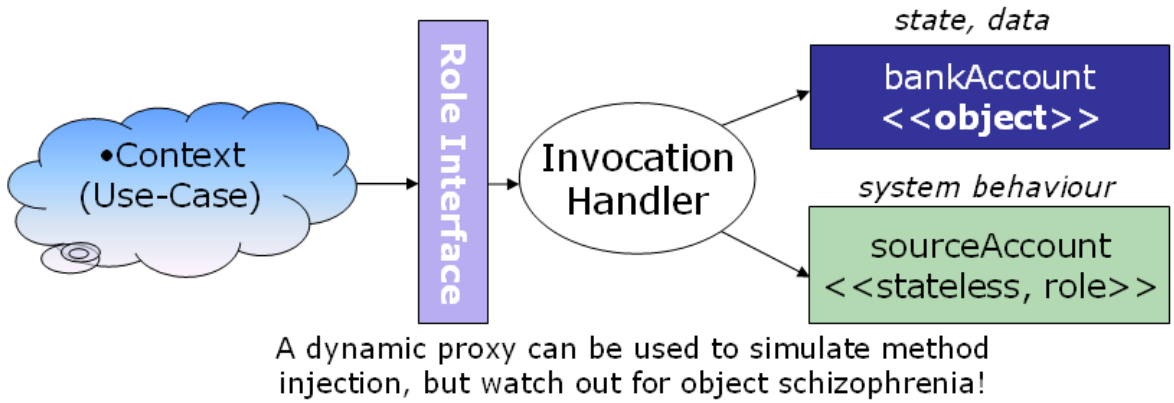


Figure 1: The dynamic proxy used to implement DCI in Java

In Figure 1, the context casts the data object into the required role and from then on uses the object in that role, by using the methods exposed in the role interface. It casts any other data objects into suitable roles, and lets the objects interact. During this interaction (which occurs inside the role methods of the role implementation class (e.g. `SourceAccount.transfer()`), objects only know each other in terms of their role.

There are two classes in the library which assist in creating the dynamic proxy. The first, the `BehaviourInjector`, creates a proxy as shown in Figure 1. The second, the `SimpleRoleAssigner`, is a simpler version, which leaves out the system behaviour injection (the `SourceAccount` in the example). It simply narrows the interface of the data object to that of a simple role which has no additional behaviour injected. It can be used in cases where the domain model is complex, and the interaction only needs to know about the simpler parts of the domain model.

The following class diagram shows all classes involved in implementing DCI in Java using these tools.

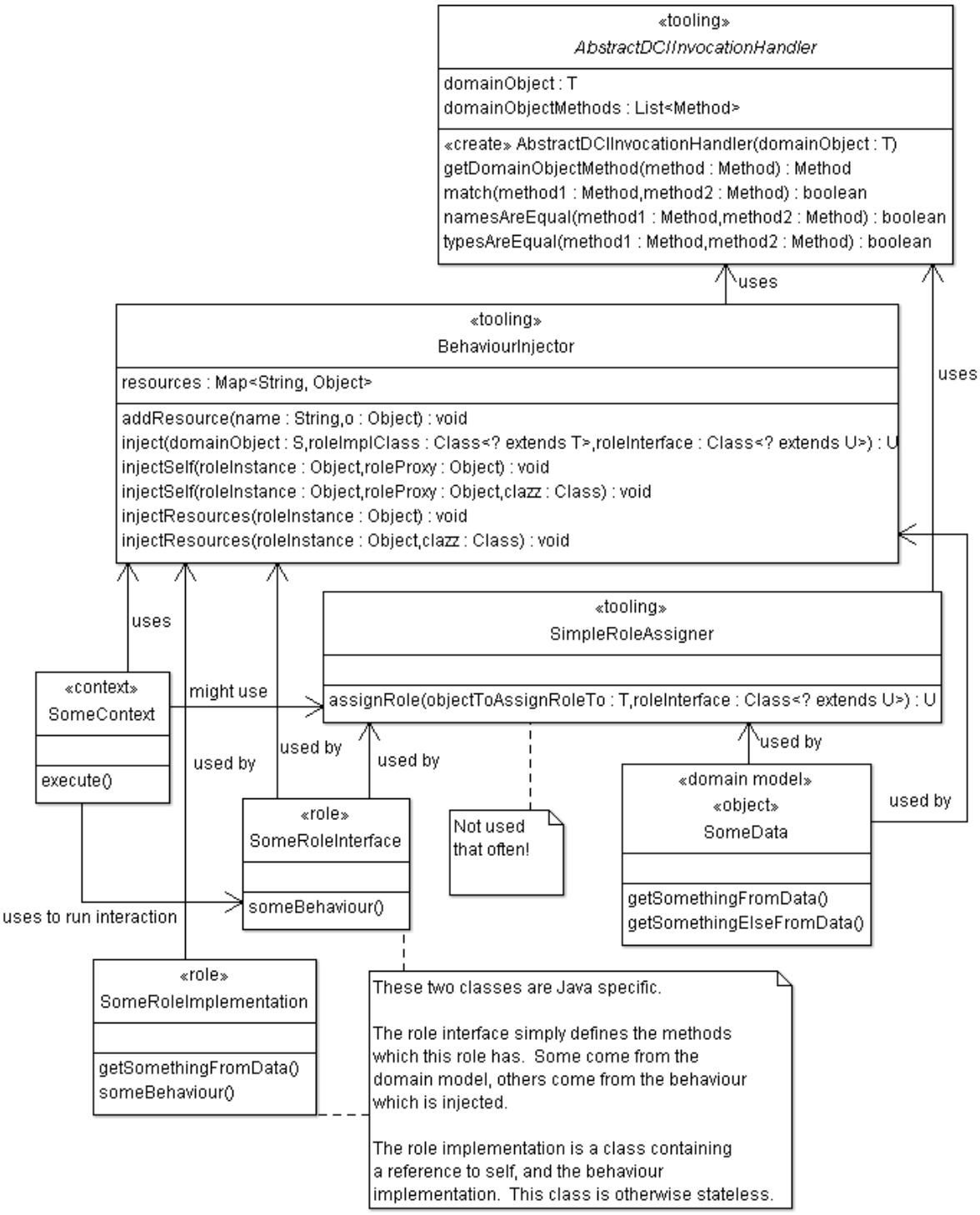


Figure 2: Class Diagram

The classes `SomeRoleImplementation`, `ISomeRoleInterface` and `SomeData` are the ones which are context specific, e.g. `SomeData` would be replaced by `BankAccount` in the example used above.

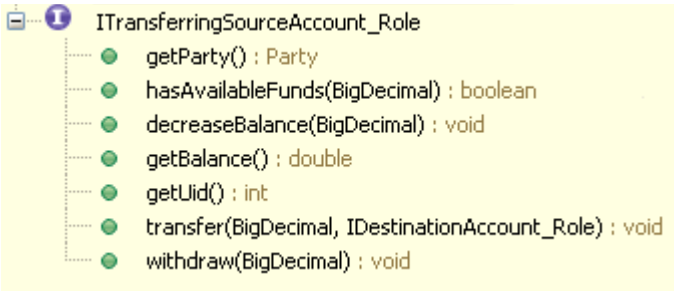
In order for the context to assemble objects and cast them into roles, it instantiates a new `BehaviourInjector`:

```
// prepare the injector
BehaviourInjector behaviourInjector = new BehaviourInjector();
```

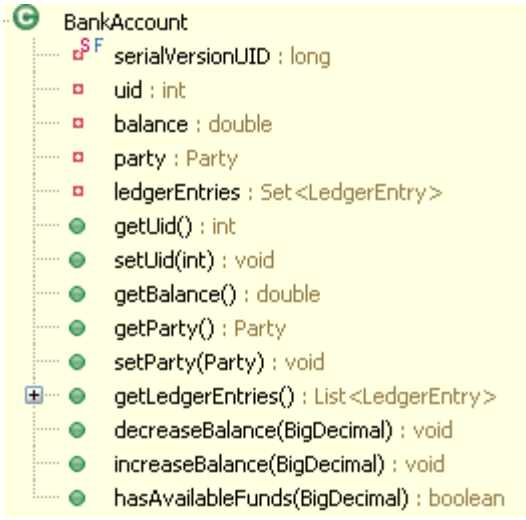
The behaviour injector is then used to cast objects to roles and inject behaviour:

```
// convert the domain object into a role, and inject the relevant role methods into it
ITransferringSourceAccount_Role source = behaviourInjector.inject(
        sourceAccount, //domain object
        TransferringSourceAccount_Role.class, //class providing all the impl
        ITransferringSourceAccount_Role.class); //the entire role impl
```

Above, `TransferringSourceAccount_Role` is the stateless class which contains the system behaviour that needs to be injected, that is, the implementation. The `sourceAccount` is the data object, being cast into the role. The `ITransferringSourceAccount_Role` is the interface which defines all methods which an object in the role exposes, for example:

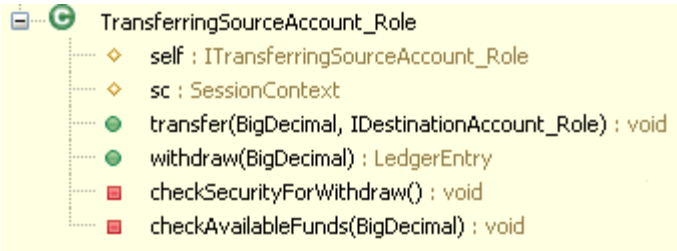


In this example, the `transfer(BigDecimal, IDestinationAccount_Role)` and `withdraw(BigDecimal)` methods are the two being injected into the data object. All the others already exist in that object, as well as others, not relevant to the role, for example, below is the domain model class:



This domain model class has many more methods, and even attributes, which are not visible in the role.

The class containing the system behaviour implementation looks like this:



This role implementation class is stateless, although it does contain two attributes:

- "self" – a reference to the proxy which gets injected by the `BehaviourInjector` (see below),
- "sc" – in this example a `SessionContext`, namely a resource, also injected by the `BehaviourInjector` (see below, under Rich Behaviour).

Functional decomposition has lead to the two check* sub-methods being added. These are simply called by the public (green circle) methods which the role exposes.

To "inject" the methods, the `BehaviourInjector` does the following things:

1. Creates a new instance (using Java Reflection) of the class providing the role implementation,
2. Creates an invocation handler which knows about the data object, and the newly created instance of the role implementation,
3. Creates a Java Dynamic Proxy which exposes the given role interface and calls the invocation handler anytime a call to the interface is made,
4. Recurses through the role implementation class and its super-classes and injects any attributes marked with the `@ch.maxant.dci.util.Self` annotation with a reference to the newly created proxy. As such, the "self" attribute in any role implementation must have the same type as the role interface class. For example:

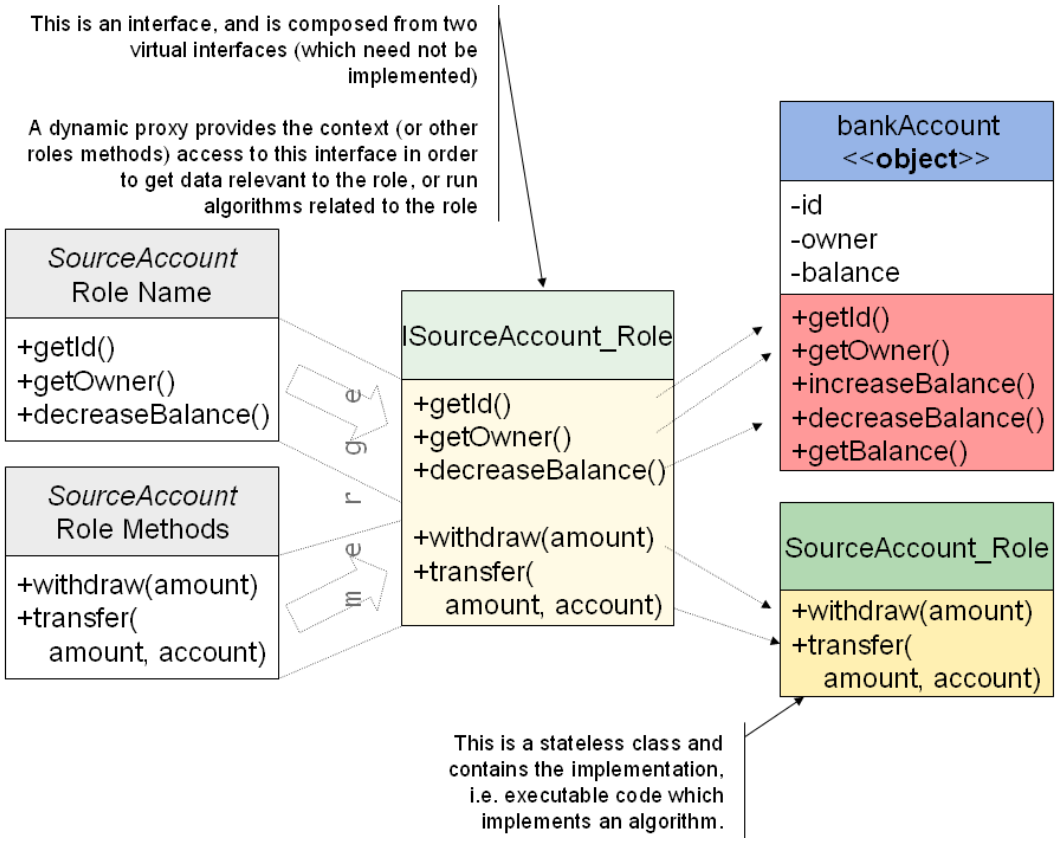
```
public class TransferringSourceAccount_Role extends AbstractLedgerEntryCreator {

    /** injected by the {@link BehaviourInjector}. */
    @Self
    protected ITransferringSourceAccount_Role self;
```

Above, the role implementation of type `TransferringSourceAccount_Role` has a reference to the **`I`**`TransferringSourceAccount_Role` which is the interface.

That means, the object returned by the `BehaviourInjector` is a proxy exposing the role interface, and it passes the method calls which it receives to either the role implementation, or the data object. The proxy itself does nothing else. It always checks the role implementation first, and only passes the method call to the data object if the role implementation does not contain the method being called.

The following diagram shows these concepts visually, albeit with a slightly different example:



From section 3.2 of The DCI Execution Model[1]:

> *RoleMethods are not associated with a particular class so RoleMethods cannot access the instance variable in the receiver object. Instead, RoleMethods address Data objects indirectly through the Role name…*

The `BehaviourInjector` puts the RoleName interface and RoleMethod interface together, and calls it the RoleInterface. This partly makes the code somewhat simpler to write and read, but more importantly, the use of a dynamic proxy requires it to be so, because the context needs to refer to it as a single type in order to be able to call methods on it.

Rich Behaviour

DCI states that system behaviour should be built into role methods. Sometimes, system behaviour is more complex than just modifying system state. Sometimes it needs access to resources in order to fulfil the requirements. In such cases, and where the designer/architect chooses to have such complex behaviour inside a role method[2], resources could be passed into role methods as parameters, but this makes the reading, reviewing and understanding of the code harder, because technical issues such as resources are mixed up with the roles and domain model.

[1] The DCI Exectuion Model, Trygve Reenskaug, 2010.
http://heim.ifi.uio.no/~trygver/2010/DCIExecutionModel.pdf

[2] Rich behaviour could also be inside the context, although this is discouraged because the context has the job of gathering data objects, casting them to their roles and running interactions. Rich behaviour could also be outside of DCI, for example in an MVC controller, a service, or an application/process layer on a server. It is the job of the architect to ensure that the use-case remains reviewable, as required by DCI.

The solution in such cases is to add resources to the `BehaviourInjector` before doing the injection, so that it can inject those resources into the role methods during method injection. As an example, consider a JPA[3] entity manager used to persisting new data objects. Such an entity manager can be created or looked up by the context[4], and passed to the behaviour injector:

```
// and add some resources
behaviourInjector.addResource("em", entityManager);
behaviourInjector.addResource("sc", sessionContext);
```

Inside the role implementation class, the following code can be added, which the `BehaviourInjector` spots, and uses to inject these resources:

```
/** injected by the {@link BehaviourInjector}. */
@Resource(name="sc")
protected SessionContext sc;

/** injected by the {@link BehaviourInjector}. */
@Resource(name="em")
protected EntityManager em;
```

Here, the name "em" used when adding the `EntityManager` to the `BehaviourInjector`, is used as a key. When examining a role implementation, the `BehaviourInjector` identifies any attributes in the class or super-classes with the `javax.annotation.Resource` annotation. Such attributes may be private, protected, package-scoped or public. It then searches for a key with the same name as the field name. If no key is found, it checks whether the annotation has the "name" attribute set, and if so, it searches for a key with that name. As soon as it finds the key, it sets the attribute to be equal to the reference it was passed during the addition of the resource. If the setting cannot be done, for example because the type is wrong, an exception is thrown, behaviour injection is aborted, and control returns to the context.

[3] Java Persistence Architecture – an Object-Relation-Mapping (ORM) tool which is part of the Java Enterprise Edition specifications.

[4] If the context is managed by a container, such as Spring or an EJB container, the resources can be injected by the container into the context, allowing them to be fully managed outside of the application code.

Object Schizophrenia

The `BehaviourInjector` returns an instance of the role interface rather than the data object, so you end up with object schizophrenia, whereby the object is not the data object, but also not the object implementing the role methods – it doesn't know who it is. This is not necessarily as criminal as sometimes suggested, so long as the problem is carefully managed.

The role interface defines not only the methods it wishes to add to the data object, but also methods which the role method implementation needs from the domain model (e.g. accessors for the ID), or which other role method implementations need, when operating on objects in the given role. As such, the role interface defines methods for accessing the identity of the domain model object (e.g. `getUid()`), so its identity is clear.

The role implementation never contains state, and because it is the data object to which we conceptually add behaviour, it is always the data object which is of interest. It is the identity of the data object which is relevant.

As such, the `BehaviourInjector` is intelligent enough that if the `equals` method is called on the proxy, it calls the `equals` method of the data object.

Some test code shows the limitations of equality checks:

```
BankAccount bankAccount = ...get account from domain model
ISourceAccount_Role source = behaviourInjector.inject(
      bankAccount,
      SourceAccount_Role.class,
      ISourceAccount_Role.class);

//Prints false. Object schizophrenia!!
System.out.println(source == bankAccount);

//Prints true. No object schizophrenia, because the proxy is smart!
System.out.println(source.equals(bankAccount));

//Prints false. Object schizophrenia again! Because
//the proxy isn't *that* smart.
System.out.println(bankAccount.equals(source));
```

Listing 1: Test code showing the limitations of object equality

So, in most cases, object schizophrenia can be dealt with. The special case which needs attention is when a role method looks up a data object from the domain model and needs to test its equality to "self". In such cases, the "self" variable needs to *always* be on the left hand side of the equality check. Similar to the code shown in Listing 1, if the data object from the domain model is on the left side (`bankAccount.equals(self)`), then object schizophrenia causes the equality check to fail. This is because the `equals` method from the data object compares the data object to the instance of the proxy, to which it is clearly not equal. When the data object is on the right side (`self.equals(bankAccount)`), the invocation handler in the proxy passes the call to the `equals` method to the data object, so equality can be legally tested, because a data object is testing its equality to another data object.

Testing

The dynamic proxy used in this library can be dangerous, because there are no checks at compile time, as to whether the role implementation and the data object really contain all the methods defined in the role interface.

While compiler time checks are very useful, this library has been kept more dynamic, because it makes the code easier to read, and makes roles potentially more reusable (any data object from the domain model exposing the required methods can have behaviour added to it).

As such, a mechanism is needed, in order to ensure that all methods in the interface can indeed be found in either in the role implementation or the domain model object.

The solution is either to pass a boolean to the inject method of the `BehaviourInjector` / `SimpleRoleAssigner`, or to set a system property. Both the Boolean and the system property tell the `BehaviourInjector` to check that all methods are implemented, during the injection process. This is *not* done by default, for performance reasons. Example code is included in the JAR library (see below under Source Code), as well as below, where a JUnit tests shows how to set the system property. It is encouraged that unit tests be created to enforce these checks. The test case shown below tests individual role mappings, however a more suitable way to ensure all methods in role interfaces are implemented, is to simply write unit tests for every context, which not only ensures that you will not get runtime errors because of missing method implementations, but also helps to test every use-case at the same time!

```
@Test
public void testNegative1() {

    //force checks to ensure all interface methods exist somewhere
    System.setProperty(DCIHelper.SYSTEM_PROPERTY_CHECK_METHODS_DURING_CAST, "true");

    BankAccount account = new BankAccount();
    account.setUid(1);
    account.increaseBalance(new BigDecimal(1000.0));

    BehaviourInjector bi = new BehaviourInjector();

    try{
        //inject the role into the domain model object
        bi.inject(account, Test_Role.class, ITest_Role_Error.class);
        fail("should throw an UnsupportedOperationException, because ITest_Role_Error " +
                "contains methods that are not implemented anywhere!");
    }catch(UnsupportedOperationException e){
        //yay, we expected this, because we have set the sys prop
    }
}
```

In this test, a role interface with more methods than exist in the role implementation and data objects has been defined. The injection fails, because the system property has been set to ensure the checks are performed. The faulty role interface, with a method called `doIexist()` which is not in the data object, nor in the role implementation, is shown below:

```
/** an example of a role interface, with methods which DONT exist in the domain */
static interface ITest_Role_Error extends ITest_Role {
    boolean doIExist();
}
```

Finally, take care to watch out for auto-boxing. Java Reflection, which is used by the dynamic proxy does *not* consider an `int` to be the same as an `Integer`. So when it searches for methods in the role implementation and data object, it may fail, if you define an `int` in one place, and an `Integer` in another.

Other Options

While the library presented here is not purist DCI, it suffices until the time when the Java language is extended to offer dynamic injection of methods into objects. Alternative solutions for Java do exist, such as Qi4J[5] or ObjectTeams[6]. These solutions are much more complex than those offered here, but also bring extra benefits which may be of interest. The designer/architect is encouraged to evaluate the alternatives when choosing the Java solution they will use.

Source Code

This library is open source and made available free of charge under the Lesser GNU General Public Licence. As such, the source code is included in the JAR file which can be downloaded from http://www.maxant.co.uk/tools.jsp. The JAR file is also an OSGi Bundle!

[5] Qi4J: http://www.qi4j.org/
[6] Object Teams http://www.objectteams.org/