DCI Tools for Java

The following is a guide for the DCI Tools library, created by maxant, for enabling DCI in Java using the Reenskaug Exection Model[1]. 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 1.1.0 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.

[1] DCI Execution Model http://heim.ifi.uio.no/~trygver/2010/DCIExecutionModel.pdf

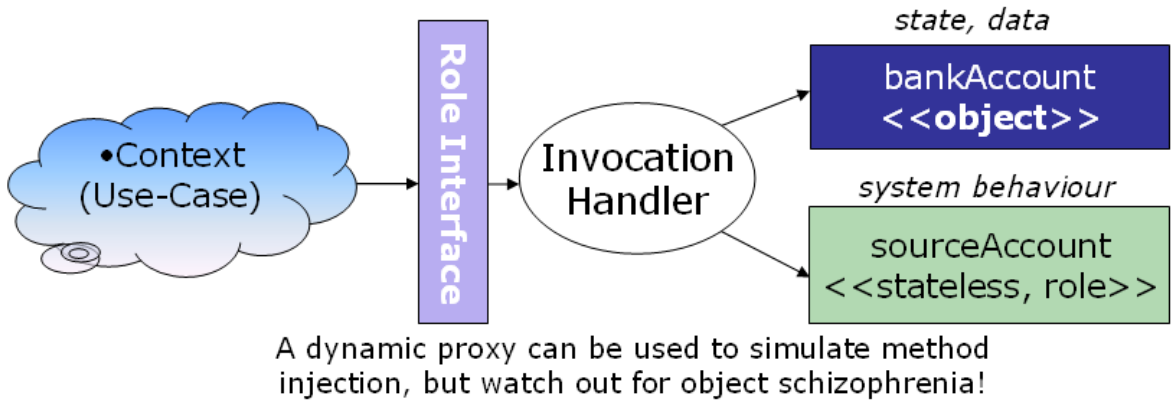


Figure 1: The dynamic proxy used to implement DCI in Java

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 is one class in the library which assists in creating the dynamic proxy, namely the `BehaviourInjector`.

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

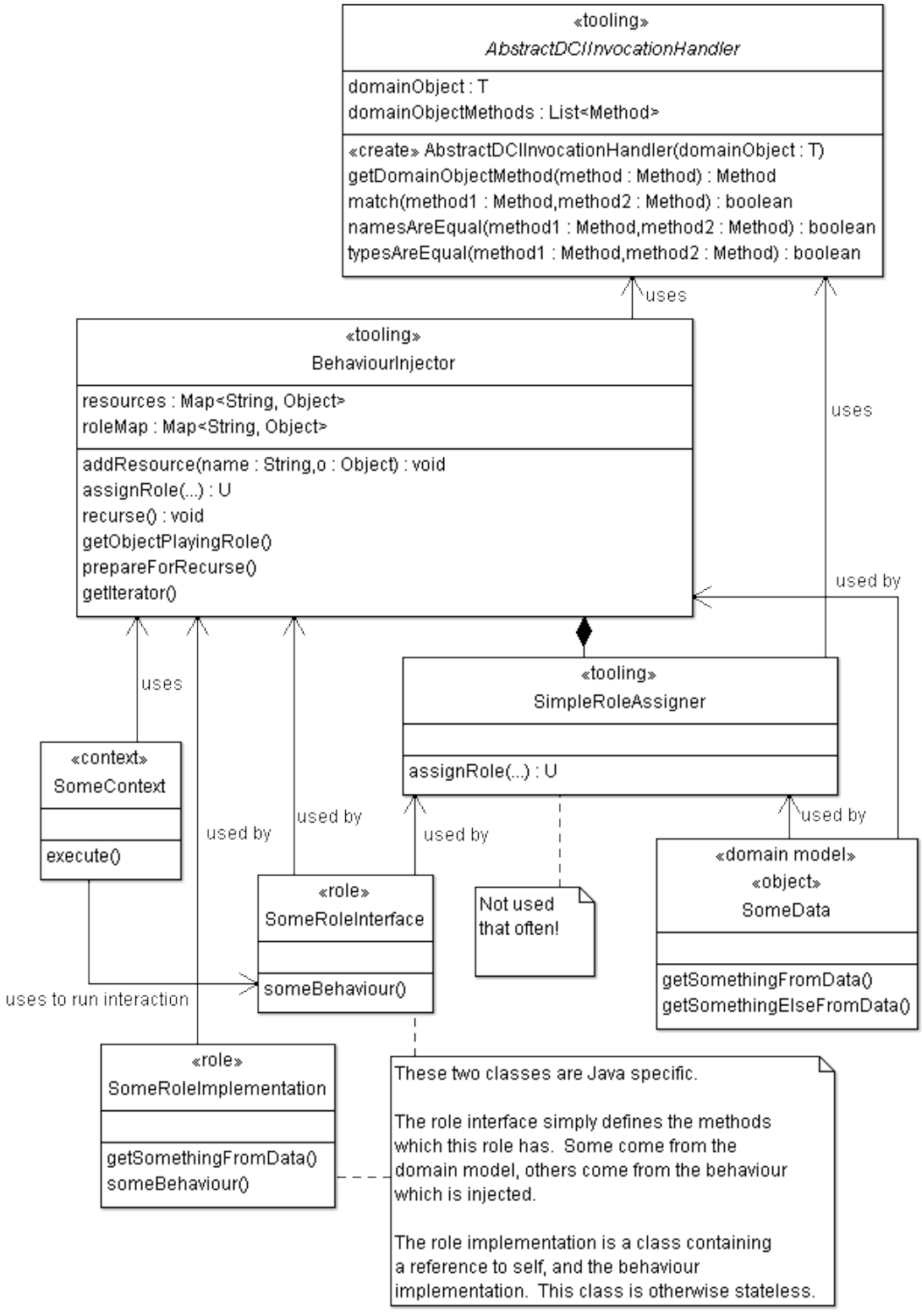


Figure 2: Class Diagram

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.

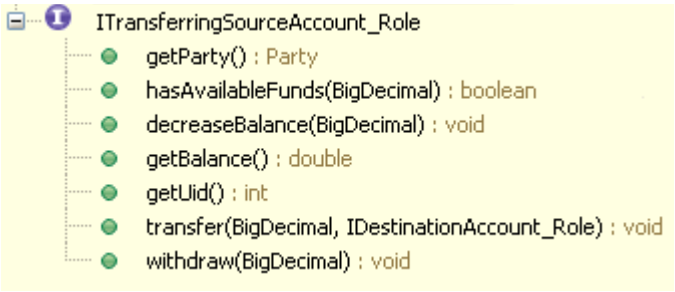
In order for the context to assemble objects and cast them into roles, it instantiates a new `BehaviourInjector`:

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

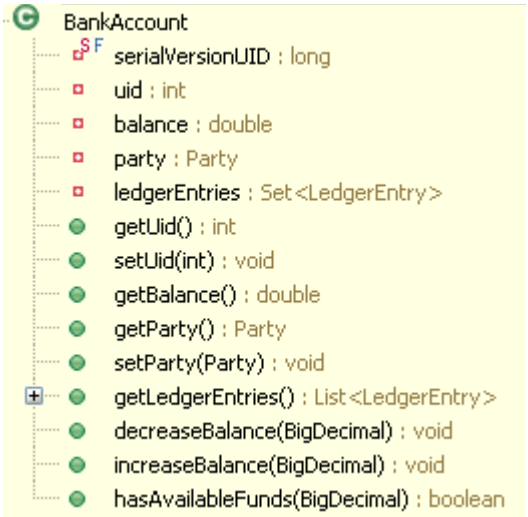
The behaviour injector is then used to cast objects to roles and inject behaviour:

```
// convert the domain object into a role, and inject the relevant role methods into it
ITransferringSourceAccount_Role source = behaviourInjector.assignRole(
        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:

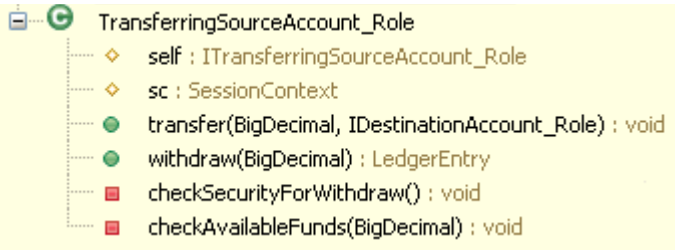


In this example, the `transfer(BigDecimal, IDestinationAccount_Role)` and `withdraw(BigDecimal)` methods are the two being injected into the data object. All the others already exist in that object, as well as others, not relevant to the role, for example, below is the domain model class:



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:

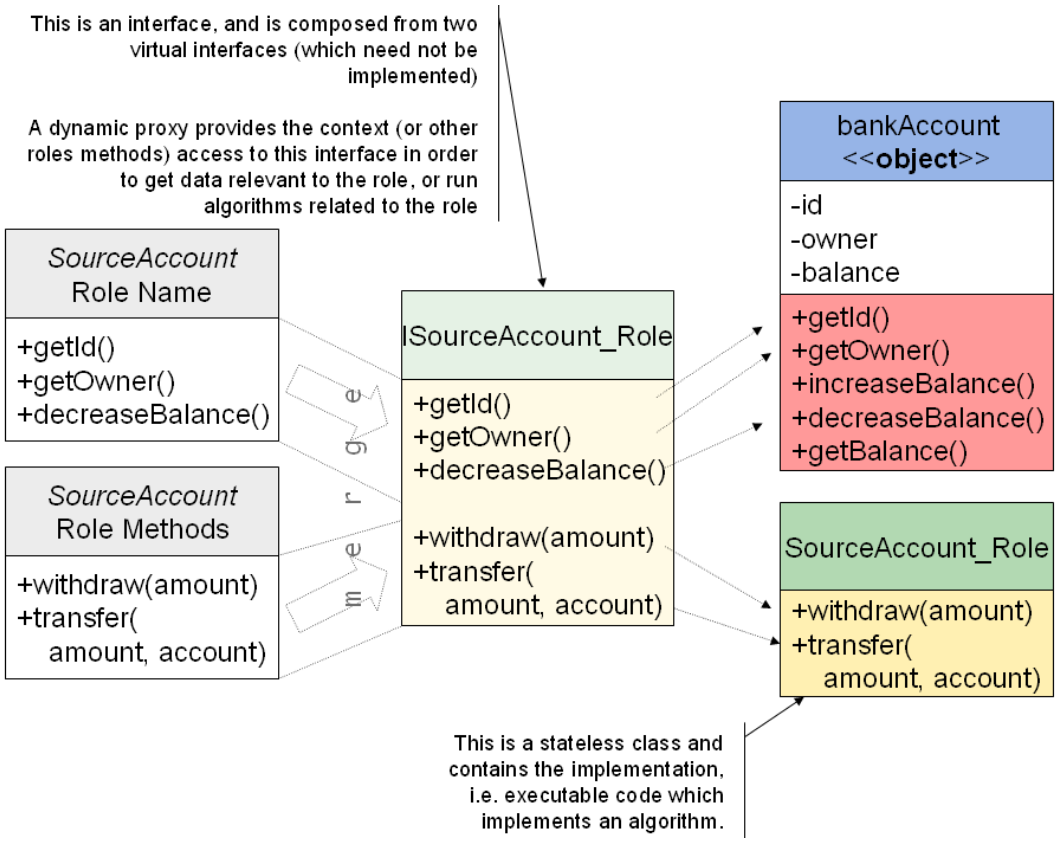
```
public class TransferringSourceAccount_Role extends AbstractLedgerEntryCreator {

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

Above, the role implementation of type `TransferringSourceAccount_Role` has a reference to the `ITransferringSourceAccount_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[2]:

> *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[3], 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.

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

[3] 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[4] entity manager used to persisting new data objects. Such an entity manager can be created or looked up by the context[5], 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.

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

[5] 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.assignRole(
      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 Unit Testing and Mock Objects applied to DCI), 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(null); //can pass null, as no recursion is done here

    try{
        //inject the role into the domain model object
        bi.assignRole(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[6] or ObjectTeams[7]. 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.

Unit Testing and Mock Objects applied to DCI

Consider a class under test:

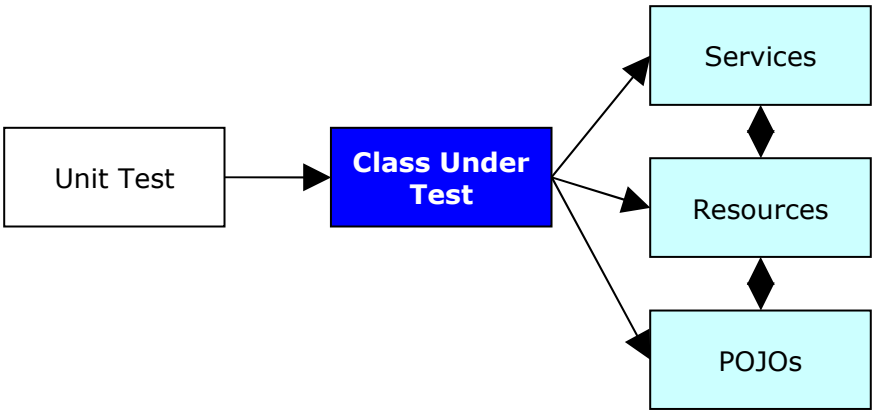


Figure 3: A Class under Test

In Figure 3, a class which is being tested is shown, together with possible dependencies. It can have dependencies on things like services, resources or POJOs. Indeed, the dependencies can have dependencies between themselves. Depending upon the test, it could require some of these dependencies to be mocked, in order to test specific scenarios.

Now consider unit testing of the three basic building blocks of DCI, namely the data, the context and the roles. If the class under test is a data object, it is unlikely that any mocking is required because data objects should be very simple, and the resulting unit tests will be simple and only test accessor and/or basic methods in the class. In fact, the nature of DCI makes unit testing data classes almost unnecessary, because they contain no complex behaviour.

Unit testing of roles is possible in conjunction with data. The roles must be applied to data in order to be tested. However, this is something which would automatically be part of testing the contexts, because the contexts are responsible for mapping suitable objects into roles. Only seldom, when role methods are very complex will it be necessary to unit test roles outside of the context.

This leaves unit testing of contexts, which gets interesting. If an interaction uses several objects playing different roles, it is quite possible that some of these roles may require mocking in order to get good test coverage. DCI makes it very easy to mock such roles, by simply casting to a specialised role which knows about mocking. (see http://www.maxant.co.uk/whitepapers.jsp for details about The Specialised Role Design). In such cases, the role which is cast is actually a sub-class of the actual role, and using polymorphism implements mocking behaviour, as required. The easiest way to allow for the mocking of roles is to build the context so that is uses a sub-method to cast each

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

role. That way, during unit testing, the context itself can be sub-classed and the relevant casting method can be overridden, to cast the data object into a mocking role rather than a supporting role. After all, a mock role implementation is nothing more than a specialised role implementation. The context itself does not need to know that the role has been specialised for test purposes, it only cares about the role's interface.

Iteration, Recursion and other Complex Cases

There are cases where a role-method needs access to the context in order to either cast a new object into a role, or to re-start the context. Examples of such cases are iteration, where a role method uses a method from the data object to get a list of child data objects, and wants each one to do some useful work, which requires the child data object to take on a role. Another example is recursion, where the role-method needs to recurse through a graph of child objects.

Consider a data object with a method for getting hold of child objects. This data object method will return a collection of objects whose type will also be a data class. When casting the parent data object into a role, the role might require that method for getting hold of the child data objects. Alternatively, it might require a role-method which returns a collection of the child objects already playing a role. This is likely to be more useful than simply returning a collection of data objects. But how would the role-method cast the child objects into a role? Think of an example: a Project (data object) contains a list of Tasks (data object). That project can play the role of a FrontLoader so that it can plan itself[8]. In this roles role-method which does the planning, it may need to cast the tasks (data objects) into the Activity role, which can plan an individual task based upon its predecessors last end date.

To solve these recasting Problems, Reenkaug's Execution Model considers giving role-methods access to the current context.

In order to recast, there are a number of option:

1. Cast child data objects into a role on the fly, as shown in the following code,

[8] Front loading is a planning technique whereby a task is planned to start as early as possible, with its start date no earlier than any predecessors end date, and no earlier than the project start date.

```
/** role impl for front loader role, taken on by the project */
public static class FrontLoader_Role {

    @Self
    private IFrontLoader_Role self;

    @CurrentContext
    private Context currentContext;

    public void frontloadFrom(Date projectStart) {

        //select an activity which has not been planned
        while(true){
            boolean plannedAll = true;
            IActivity_Role activityNeedingPlanning = null;
            for(Task_Data task : self.getTasks()){
                if(!task.hasBeenPlanned()){
                    if(!hasUnplannedPredecessors(task)){
                        activityNeedingPlanning = currentContext.assignRole(
                                task,
                                Activity_Role.class,
                                IActivity_Role.class);
                        activityNeedingPlanning.frontloadFrom(projectStart);
                    }else{
                        //gotta wait until predecessors are planned :-(
                        plannedAll = false;
                    }
                }
            }
            if(plannedAll){
                break;
            }
```

Cast the task and plan it

2. Create a sub-context (unrelated to the current context, and use that to do individual task planning,

```
/** role impl for front loader role, taken on by the project */
public static class FrontLoader_Role {

    @Self
    private IFrontLoader_Role self;

    public void frontloadFrom(Date projectStart) {

        //select an activity which has not been planned
        while(true){
            boolean plannedAll = true;
            for(Task_Data task : self.getTasks()){
                if(!task.hasBeenPlanned()){
                    if(!hasUnplannedPredecessors(task)){

                        //plan the task using a sub-context!
                        new Activity_Context(task).plan(projectStart);

                    }else{
                        //gotta wait until predecessors are planned :-(
                        plannedAll = false;
                    }
                }
            }
            if(plannedAll){
                break;
            }
        }
    }
```

Create new sub-context for planning tasks, which casts the task into the activity role.

3. Magically get hold of an iterator of the correct generic type which returns data objects already cast into the required role,

```
public void frontloadFrom(Date projectStart) {

    //select an activity which has not been planned
    while(true){
        boolean plannedAll = true;
        IIterable_Role<IActivity_Role> activities = getActivities();
        for(IActivity_Role activity : activities){
            if(!activity.hasBeenPlanned()){
                if(!hasUnplannedPredecessors(activity)){
                    activity.frontloadFrom(projectStart);
                }else{
                    //gotta wait until predecessors are planned :-(
                    plannedAll = false;
                }
            }
        }
        if(plannedAll){
            break;
        }
    }
}

/** casts the list self.tasks into a list of activities */
private IIterable_Role<IActivity_Role> getActivities() {
    IIterable_Role<IActivity_Role> iter =
        currentContext.getIterator(
                self.getTasks().iterator(),
                IActivity_Role.class);
    return iter;
}
```

Cast the iterator of the data object collection into a generic role which can pass back objects playing roles.

4. Use recursion which casts a list's iterator into the generic `Iterable` role used above, but then restarts the context on the child objects playing this `Iterable` role,

```
public void frontLoadFrom(Date projectStart){

    this.projectStart = projectStart;

    //casts the project into a front loader role, so it can plan itself.
    frontLoader = bi.assignRole(
            project,
            FrontLoader_Role.class,
            IFrontLoader_Role.class);

    //add the iterator to the context with the name ACTIVITIES
    bi.getIterator(
                project.getTasks().iterator(),
                IActivity_Role.class,
                ACTIVITIES);

    //help the injector know which implementation to choose (saves code later on)
    bi.setupRoleInterfaceToRoleImplMapping(IActivity_Role.class, Activity_Role.class);

    recurse();
}

@Recursable
public void recurse(){
    @SuppressWarnings("unchecked")
    IIterable_Role<IActivity_Role> currentActivities =
                (IIterable_Role<IActivity_Role>) bi.getObjectPlayingRole(ACTIVITIES);
    frontLoader.frontloadFrom(projectStart, currentActivities); //start the interaction
}


/** role impl for front loader, applicable to the project */
public static class FrontLoader_Role {

    @CurrentContext
    private Context currentContext;

    public void frontloadFrom(Date projectStart, IIterable_Role<IActivity_Role> currentActivities) {

        for(IActivity_Role activity : currentActivities){

            if(activity.hasBeenPlanned()) continue;

            //plan the predecessors first! do it recursively by
            //using the context to re-cast
            currentContext.prepareForRecurse();

            //tell the context that the kids are now playing the role of the iterator
            //ie recast the role in the role-map
            currentContext.getIterator(
                    activity.getDependencies().iterator(),
                    IActivity_Role.class,
                    FrontLoad_Context.ACTIVITIES);

            //step into context with new role-map
            currentContext.recurse();

            //finally we can plan this task, because all predecessors are now planned
            activity.frontloadFrom(projectStart);
        }
    }
```

This is the context. Assign the top level list of tasks in the project to play the role of "activities".

Start the recursion

Get hold of the list of tasks currently requiring processing, and start the interaction

Prepare for recursion by suspending the current role-map and pushing a new role-map onto the stack

Put the child objects into the role which the parents were playing.

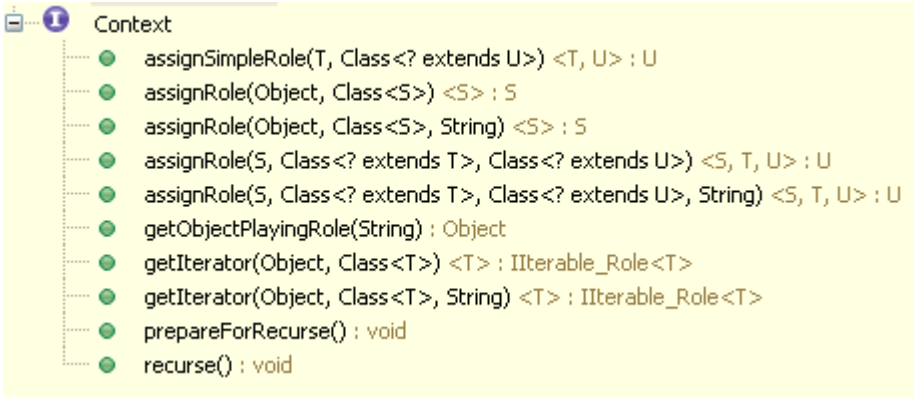
Recurse into the context.

In all these cases, there are a number of useful features supplied by the `BehaviourInjector`.

Firstly, if access to the current context is needed from within a role-method, then the role implementation class may contain a field like this:

```
@CurrentContext
private Context currentContext;
```

During method injection, the `BehaviourInjector` spots these annotated fields and injects itself (playing the role of "`Context`"). The `Context` interface allows role-methods to do things like recasting or recursion:



Secondly, in order to save code, the `BehaviourInjector` can be told which role implementation class belongs to which role interface class:

```
//additional info that will be useful to the context
bi.setupRoleInterfaceToRoleImplMapping(IActivity_Role.class, Activity_Role.class);
```

This means that the role implementation class parameter does not need to be passed in calls like this one:

```
activityNeedingPlanning = currentContext.assignRole(
        task,
        Activity_Role.class,
        IActivity_Role.class);
```

By telling the `BehaviourInjector` this information once, shortly after construction, it never needs to be told again, which role implementation class to use for a given role interface.

Thirdly, the `BehaviourInjector`, has the ability to create an `Iterable` of a given type:

```
IIterable_Role<IActivity_Role> iter =
    currentContext.getIterator(
            self.getTasks().iterator(),
            IActivity_Role.class);
```

The tasks which "self" is returning in the above code are of a data type, rather than a role type. The code needs these to be of a role type, so the current context can be used to get an `Iterable` (which can be used in a for loop for example) which when queried returns objects already in the given role.

In order to use the "`getIterator()`" method, one needs to have told the `BehaviourInjector` which role implementation belongs to the role interface passed to the method (see the slightly above).

Fourthly, the BehaviourInjector provides recursion over the context by allowing the programmer to mark a given method in the context as the method to call recursively. This method typically also restarts the interaction, based on objects playing roles which are needed for the current recursion iteration.

In the figure above (point 4, recursion) arrows show the program flow. The context casts a collection of top level child objects into the role of an iterator, and provides a specific name to them.

The context then calls a special method in the context, passing no parameters. This method is marked with the @Recursable annotation. This methods job is to extract the current iterator from the behaviour injector, and to start the interaction using that iterator:

```
@Recursable
public void recurse(){
    @SuppressWarnings("unchecked")
    IIterable_Role<IActivity_Role> currentActivities =
                (IIterable_Role<IActivity_Role>) bi.getObjectPlayingRole(ACTIVITIES);
    frontLoader.frontloadFrom(projectStart, currentActivities); //start the interaction
}
```

Notice how the method above is annotated (it is a method in the context). And notice how the method gets the iterator from the `BehaviourInjector` (i.e. the object which plays the current context) in order to recurse.

Note the `BehaviourInjector` takes a parameter in its constructor. That parameter is an object, which can be null except in cases where this @Recursable annotation is used. Then, it needs to be the object which contains this method:

```
public static class FrontLoad_Context {

    private static final String ACTIVITIES = "activities";
    private Project_Data project;
    private BehaviourInjector bi = new BehaviourInjector(this);
    private Date projectStart;
    private IFrontLoader_Role frontLoader;
```

To perform the recursion, the role-method has three steps which it undertakes. The first is to suspend the current role-map. The role-map is a mapping inside the current context (`BehaviourInjector`), which maps a name to a distinct object playing a role. This role-map is filled anytime a behaviour injection happens. Suspending the current role-map causes a new role-map to be placed on the stack (well, a stack inside the `BehaviourInjector` at least). Before recursing down a level, the current context is used to cast a new object into the relevant roles. Below, the object playing the role of "Activities" is set as the dependencies of the current activity.

```
//plan the predecessors first! do it recursively by
//using the context to re-cast
currentContext.prepareForRecurse();

//tell the context that the kids are now playing the role of the iterator
//ie recast the role in the role-map
currentContext.getIterator(
        activity.getDependencies().iterator(),
        IActivity_Role.class,
        FrontLoad_Context.ACTIVITIES);

//step into context with new role-map
currentContext.recurse();
```

The final step after this re-casting, is to cause the context to restart, and is done by calling the recurse() method on the current context. The BehaviourInjector (playing the role of the current context) then goes of, and looks at the object it was given during construction and searches for the method annotated with @Recursable. It calls that method, which as previously said, starts the interaction again, albeit this time using a different object in the role of "Activities", compared to the last time the interaction was started.

One question which may have arisen is why the current context role is played by the `BehaviourInjector` and the context does not simply inherit such behaviour from a super class. The reason is that it is important to allow contexts to inherit from anything which the application developer needs them to inherit from. The behaviour which every context needs to provide is hence encapsulated in the `BehaviourInjector`.

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!