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.

The associated Eclipse plugin which provides a DCI outline for your Java Projects can be retrieved using the Eclipse Update site at http://dci.maxant.co.uk/updateSite/.

This guide documents version 1.2.0 of the library and version 1.0.0 of the plugin.

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.

DCI improves OOP by allowing behaviour to live outside of the data classes, so that it is not fragmented. It associates behaviour with data objects at the time when they need it, by injecting the behaviour into the data objects so that they can assume a role in order to interact with other objects in other 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 DCI 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

© 2010 Dr Ant Kutschera, www.maxant.co.uk

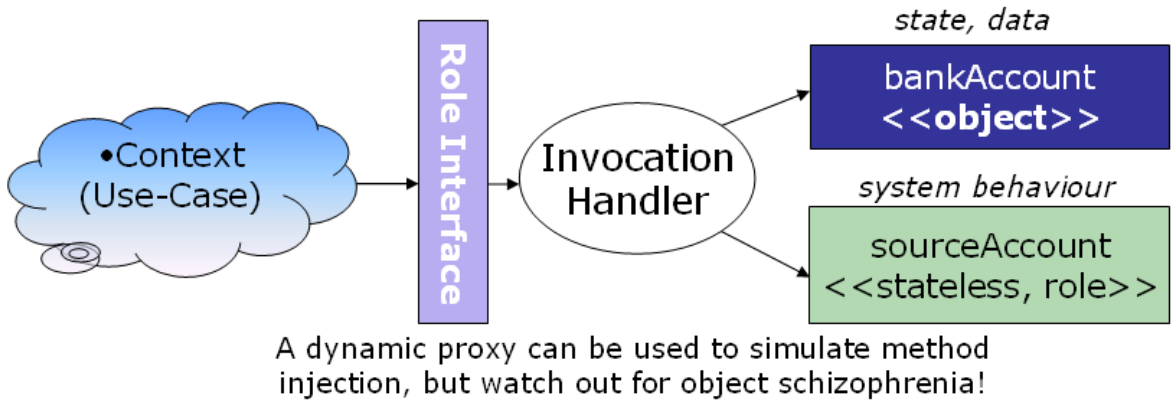


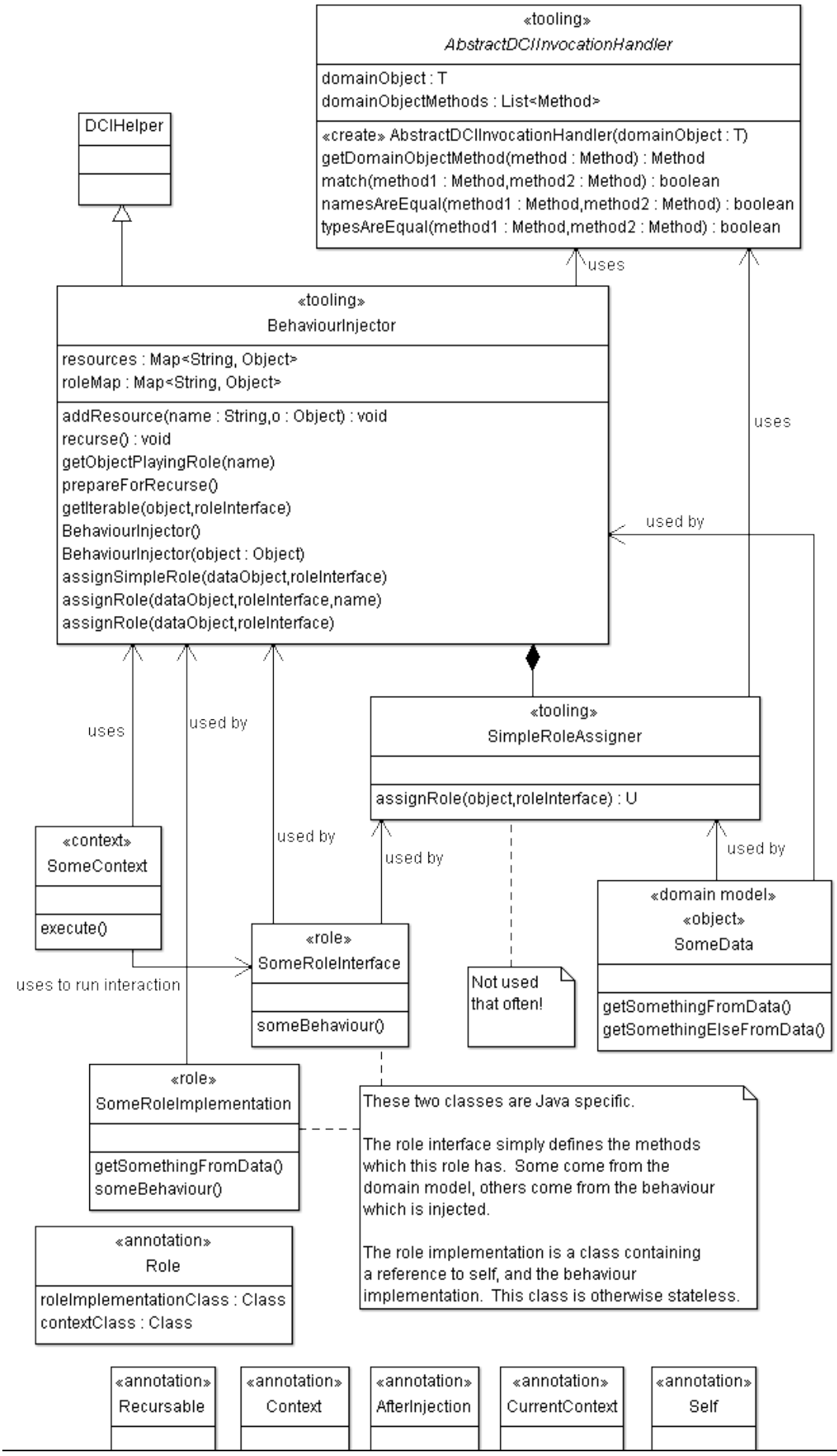
Figure 1: The dynamic proxy used to implement DCI in Java

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 and annotations involved in implementing DCI in Java using these tools.

© 2010 Dr Ant Kutschera, www.maxant.co.uk



© 2010 Dr Ant Kutschera, www.maxant.co.uk

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 can either instantiates a new `BehaviourInjector`, or it can subclass the `BehaviourInjector`. Normally, subclassing is preferred, but in order to allow for contexts needing to subclass other classes, it is possible to simply instantiate the `BehaviourInjector` and use it to do the role assignment. In such cases, it is important to pass the actual context as a parameter to the `BehaviourInjector` constructor. If no object is passed in the constructor, the `BehaviourInjector` assumes that subclassing is in use, and a reference to `this` is taken to be the context.

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

```
/** start the interaction */
public void startParty() {
    Clown clown = assignRole(adult, Clown.class);

    clown.applyMakeup();
```

Above, the `adult` object is a data object which gets enriched with clown behaviour. The `Clown` class is the role interface, and as such specifies the methods which the role exposes. This interface is defined as follows:

```
@Role(contextClass = ClownContext.class,
      implementationClass = ClownImpl.class)
public interface Clown {

    /** put some makeup on (initialise) */
    public void applyMakeup(); // role

    /** makes kids laugh */
    public void makeKidsLaugh(IIterable<Kid> kids); // role

    public void setHairColour(Color c); // data

}
```

The above interface is marked with the `@Role` annotation which tells the `BehaviourInjector` where to find the implementation of the role methods. The annotation also requires that the programmer explicitly specify the context, because in DCI, roles belong to contexts and are not usable outside of contexts. The implementation class is a (stateless) class where the behaviour is actually implemented, and in the above case looks as follows:

© 2010 Dr Ant Kutschera, www.maxant.co.uk

```
/** role impl for the IClown role */
public class ClownImpl {

    @Self
    private Clown self;

    /** @see Clown#applyMakeup() */
    public void applyMakeup(){
        self.setHairColour(Color.GREEN);
    }

    /** @see Clown#makeKidsLaugh(IIterable) */
    public void makeKidsLaugh(IIterable<Kid> kids) {
        for (Kid kid : kids) {
            kid.laugh();
            System.out.println();
        }
    }
}
```

The implementation does not implement the interface! Rather it contains the two role methods which the role interface defines. The role implementation does not need to contain the methods otherwise found in the data object, such as `setHairColour(Color)`. The dynamic proxy handles locating and calling the correct method at runtime. There are no special annotations on the role implementation class, however it may use any of `@Self`, `@AfterInjection`, `@Resource`, or `@CurrentContext`. See later for details.

The final part of setting up the context and roles in DCI is to mark the context with the `@Context` annotation:

```
/** the context in which a clown entertains some kids. */
@Context
public class ClownContext extends BehaviourInjector {
```

The Clown role can be played by any data object which contains the methods in the role interface that are not implemented in the role implementation; in this case the `setHairColour(Color)` method. The list of methods which a data object must have in order to play a role are known as the role contract and is implicit not explicitly specified (e.g. by an interface). An example is the `Human`:

© 2010 Dr Ant Kutschera, www.maxant.co.uk

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

The role implementation class shown earlier was stateless, although it did contain an attribute marked with the `@Self` annotation. This attribute is a special one which is a reference to the object currently playing the role. It is similar to *`this`*, except that in Java, `this` would refer to the instance of the role implementation, and so there would be no access to the data methods. The `BehaviourInjector` looks for such annotations after instantiating an instance of the role implementation during role injection, and sets such attributes to be the proxy.

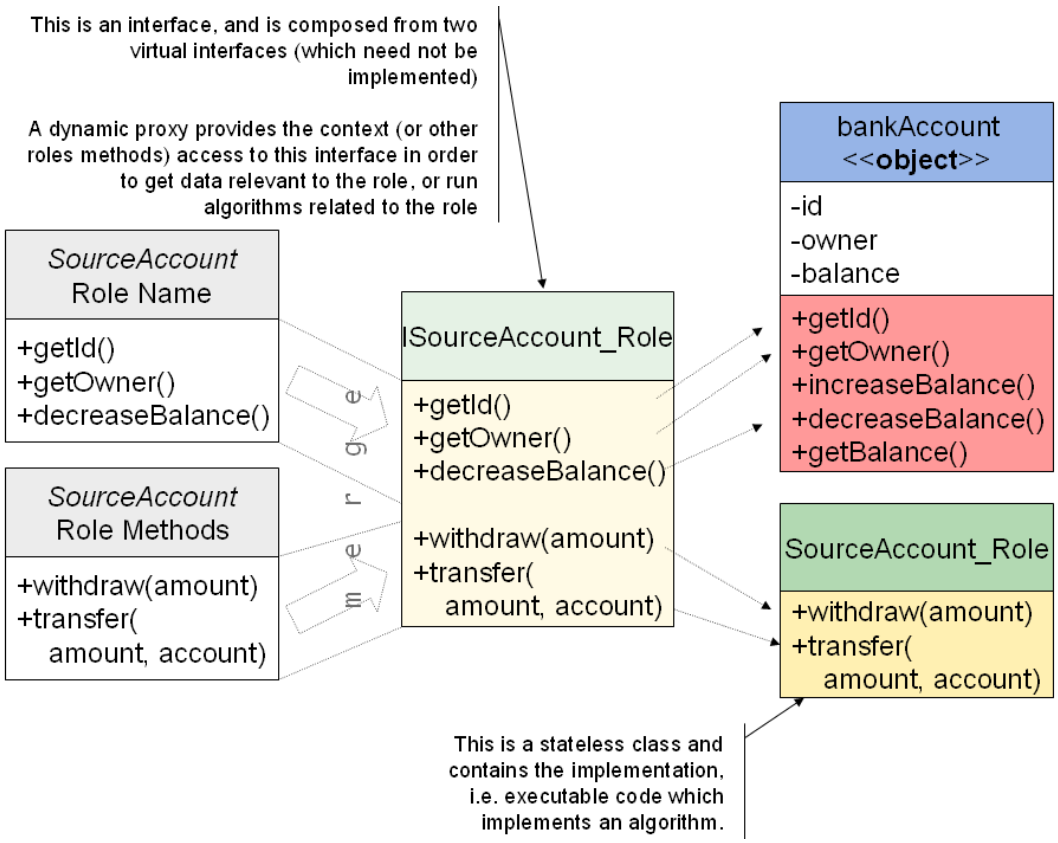
To assign roles, which effectively injects methods into data objects, the `BehaviourInjector` does the following things:

1. Check that the context class specified in the `@Role` annotation is marked with the `@Context` annotation, and is the same class as the context is,
2. Creates a new instance (using Java Reflection) of the class providing the role implementation,
3. Creates an invocation handler which knows about the data object, and the newly created instance of the role implementation,
4. Creates a Java Dynamic Proxy which exposes the given role interface and calls the invocation handler anytime a call to the interface is made,
5. 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,
6. Recurses through the role implementation class and its super-classes and injects any resources (see later, under Rich Behaviour),
7. Recurses through the role implementation class and its super-classes and injects the current context into attributes marked with the `@ch.maxant.dci.util.CurrentContext` annotation,
8. Call any methods in the Role implementation class marked with the `@ch.maxant.dci.util.AfterInject` annotation.

© 2010 Dr Ant Kutschera, www.maxant.co.uk

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

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

© 2010 Dr Ant Kutschera, www.maxant.co.uk

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.

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.

[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.

[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.

© 2010 Dr Ant Kutschera, www.maxant.co.uk

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
SourceAccount_Role source = behaviourInjector.assignRole(
      bankAccount,
      SourceAccount_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.

© 2010 Dr Ant Kutschera, www.maxant.co.uk

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 to set a system property. The system property tells 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(this);

    try {

        // inject the role into the domain model object
        bi.assignRole(account, 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:

© 2010 Dr Ant Kutschera, www.maxant.co.uk

```
/** an example of a role interface. */
@Role(
    contextClass = BehaviourInjectorTest1.class,
    implementationClass = Test_Role.class
)
public static interface ITest_Role {
    String doSomething();

    int getUid();
}

/**
 * an example of a role interface, with methods which DONT exist in the domain.
 */
@Role(
    contextClass = BehaviourInjectorTest1.class,
    implementationClass = Test_Role.class
)
public 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:

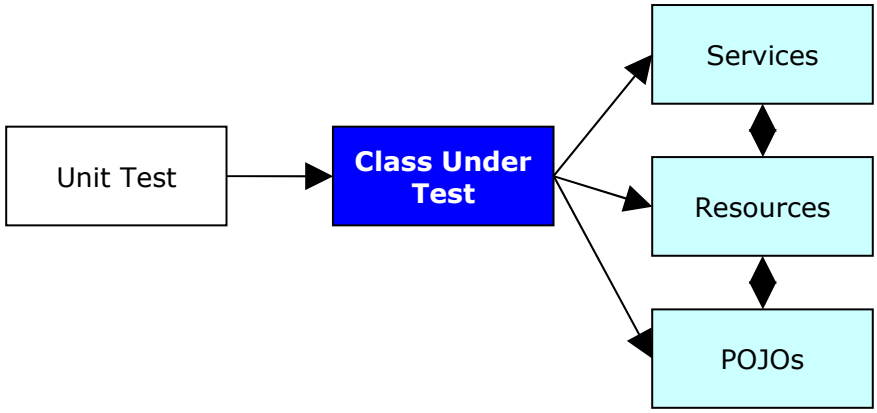


Figure 3: A Class under Test

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

© 2010 Dr Ant Kutschera, www.maxant.co.uk

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 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.

[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.

© 2010 Dr Ant Kutschera, www.maxant.co.uk

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:

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

    @Self
    private IFrontLoader_Role self;

    @CurrentContext
    private IContext currentContext;

    public void doFrontloadingFrom(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,
                                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:

© 2010 Dr Ant Kutschera, www.maxant.co.uk

```
/** 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:

© 2010 Dr Ant Kutschera, www.maxant.co.uk

```
public void frontloadFrom(Date projectStart) {

    //select an activity which has not been planned
    while(true){
        boolean plannedAll = true;
        IIterable<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<IActivity_Role> getActivities() {
    IIterable<IActivity_Role> iter =
        currentContext.getIterable(
                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:

© 2010 Dr Ant Kutschera, www.maxant.co.uk

```
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,
            IFrontLoader_Role.class);

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

    recurse();
}

@Recursable
public void recurse(){
    @SuppressWarnings("unchecked")
    IIterable<IActivity_Role> currentActivities =
                (IIterable<IActivity_Role>)
                bi.getObjectPlayingRole(ACTIVITIES);

    frontLoader.frontloadFrom(projectStart,
                    currentActivities); //start the interaction
}

public static class FrontLoader_Role {

    @CurrentContext
    private IContext currentContext;

    public void frontloadFrom(Date projectStart, IIterable<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.getIterable(
                    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.

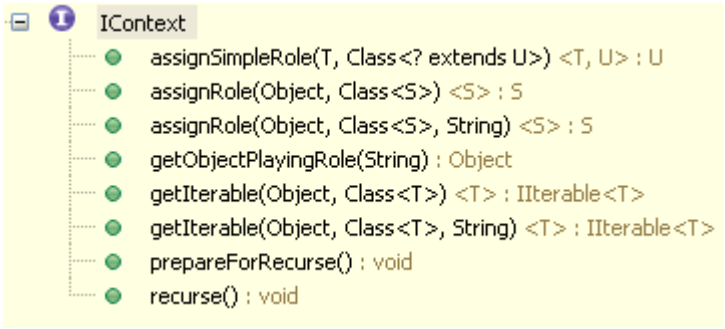
© 2010 Dr Ant Kutschera, www.maxant.co.uk

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 IContext currentContext;
```

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



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

```
IIterable<IActivity_Role> iter =
    currentContext.getIterable(
            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 "`getIterable()`" method, one needs to have told the `BehaviourInjector` which role implementation belongs to the role interface passed to the method (see the code slightly above).

Thirdly, 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 `Iterable`, 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 `Iterable` from the `BehaviourInjector`, and to start the interaction using that `Iterable`:

© 2010 Dr Ant Kutschera, www.maxant.co.uk

```
@Recursable
public void recurse(){
    @SuppressWarnings("unchecked")
    IIterable<IActivity_Role> currentActivities =
                (IIterable<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 `Iterable` 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 is the context. The `BehaviourInjector` uses this object to locate the `@Recursable` method. In cases where no object is passed, the `BehaviourInjector` assumes it has been subclassed and so will look in the subclass for the `@Recursable` method.

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 every time 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.getIterable(
        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.

© 2010 Dr Ant Kutschera, www.maxant.co.uk

Thread Safety

The `BehaviourInjector` is not thread safe. When the potential exists that it could be used by several threads at the same time, such as in Spring or EJB Services, the `BehaviourInjector` should be instantiated per thread. If the context is instantiated on each call, then the context can extend the `BehaviourInjector`. If the same context is reused per thread, then it may not extend the `BehaviourInjector`. In such cases, the `BehaviourInjector` should be instantiated inside the method being called.

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 available in the sources JAR file which can be downloaded from http://www.maxant.co.uk/tools.jsp.

OSGi

The JAR file is also an OSGi Bundle!

Code Re-use

DCI dictates that roles are the inner workings of contexts. In order to make contexts flexibly reusable, that is, usable with as many data object types as possible, the context should not receive or handle objects of a given type. The context should receive data objects and immediately cast them into roles. All object use should be with the objects in given roles.

If the context knows about certain data types, the implication is that any one wanting to use the context for their data objects may need to make their data object implement interfaces which you define, or they may need to map their objects into objects of the type which the context knows.

In order to be as flexible as possible, and in order to help future generations of programmers use your contexts and their roles, use the convention that contexts only take objects of type `Object` rather than any specific data type.

Sometimes, data objects need no additional behaviour, but need to be used by other roles. If you are taking the advice above, and the context only knows data objects as type `Object`, you cannot use the object outside of a role. At other times, it may be interesting to simply "narrow" the interface of a data object, so that it is more reviewable.

In such cases, you can cast objects into simple roles by using the `T BehaviourInjector.assignSimpleRole(Object, Class<T>)` method, which takes a data object as its first parameter, and an interface class as its second parameter (the simple role interface). It returns a proxy which wraps only the data object, with the interface which was provided, but no additional behaviour. To use this method, the role interface passed into the method must also have the `@Role` annotation as per a normal role definition. However, it is allowed to not specify a role implementation, unlike in normal role definition. Here is an example:

```
// inject the role into the domain model object
ITest_Role role = bi.assignSimpleRole(account, ITest_Role.class);
```

Where the role interface is:

© 2010 Dr Ant Kutschera, www.maxant.co.uk

```
/** an example of a role interface. */
@Role(contextClass=SimpleRoleAssignerTest.class)
public static interface ITest_Role {
    int getUid();
}
```

Notice the `@Role` annotation has no role implementation class in this case.

Roles in DCI can be considered as the inner workings of the context, and as such are not usable outside of the context to which they belong. So what do you do if a role method contains some behaviour that you would like to use elsewhere? What you do, is create a context which knows only those roles which you would like to use elsewhere (e.g. it may even only contain the single role). To use the roles in more than one place, you use this new context as a sub-context. So, for example, a role method in one role that needs to re-use some behaviour simply passes itself and perhaps some of its friends to the new context and starts an interaction.

Alternatively, if it makes sense, a context may contain several "execute" methods. Normally in DCI, a context is the implementation of a use case, or some system behaviour. To run the use case or behaviour, it has a method which is called, which executes the use case / behaviour. However, if several roles are useable within several use cases / contexts, then instead of creating sub-contexts to encapsulate the re-usable parts, it is possible to have a single context with several methods, each of which runs a single use-case or algorithm. As an example, consider a SourceAccount role which has behaviour for withdrawing money. This behaviour might be also usable within the context of paying bills or transferring money. The solution hence might be to create a single context (say a Banking Context) which contains one method for withdrawing money from a source account, one method for paying bills from the source account and a third method for transferring money from the source account into a destination account. The bills and destination account would be other roles within the banking context, but the source account would be a role which is assigned to a data object in all three methods.

The solution using sub-contexts would be to the source account role in a context called say "WithdrawContext", which does the withdrawal. In a bill paying context, or transferring context, the sub-context could be called in order to do the withdrawal part of each use case.

Performance

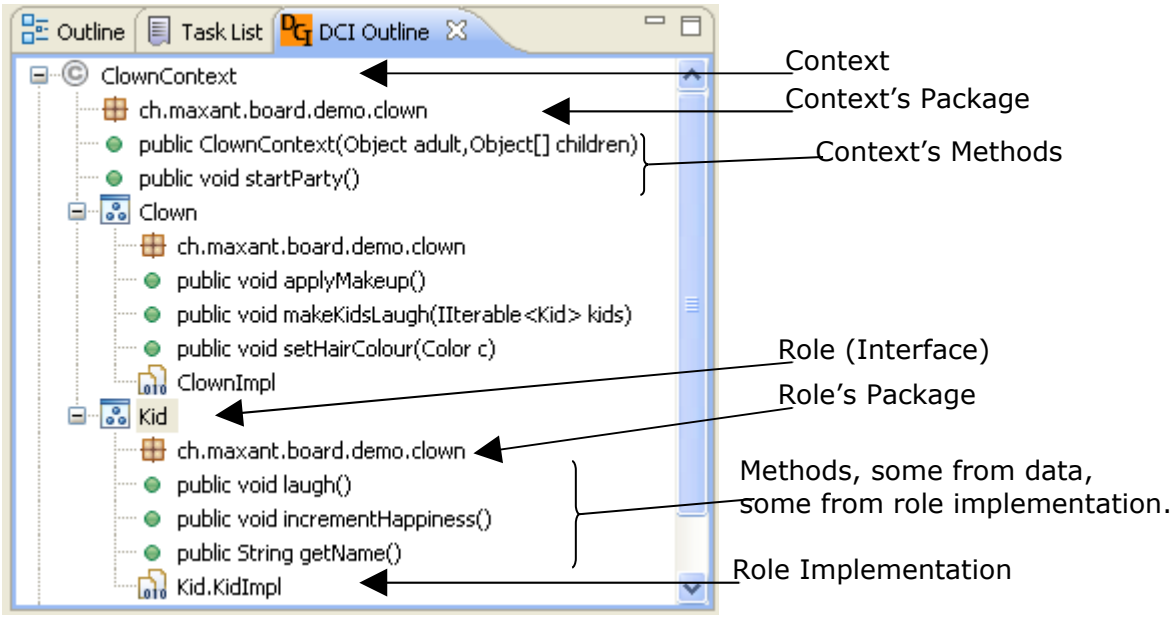
The DCI tools presented here are heavily reliant upon reflection and dynamic proxies which are well known performance trouble spots. One of the unit tests assigns a role and then calls a data method and a role method on the role, while measuring performance. On a dual core Intel T2400 1.83 GHz Processor with 2GB RAM running Windows XP, an average role injection takes around 0.15 milliseconds, compared quite a lot less than 0.01 milliseconds for a method call (through the proxy). While less than a database call or remote EJB call, it is still expensive to assign roles. However, bear in mind that many modern frameworks use such techniques and as such these DCI tools may not perform any worse.

Eclipse Plugin

Using the standard Eclipse Update mechanisms, you can install a plugin from the update site at http://dci.maxant.co.uk/updateSite/. This plugin lets you view all contexts and their roles, within the selected Java project. Once installed, the plugin lets you open the DCI Outline view, from the Windows -> Show View -> Other… menu, found under the "Data, Context & Interaction" category.

© 2010 Dr Ant Kutschera, www.maxant.co.uk

The view looks like this:



The view shows all contexts which it finds in the selected Java project. For each context, it shows the package, followed by a list of all roles it finds. Any class with the `@Role` annotation must specify a context class, and this information is used to build the model.

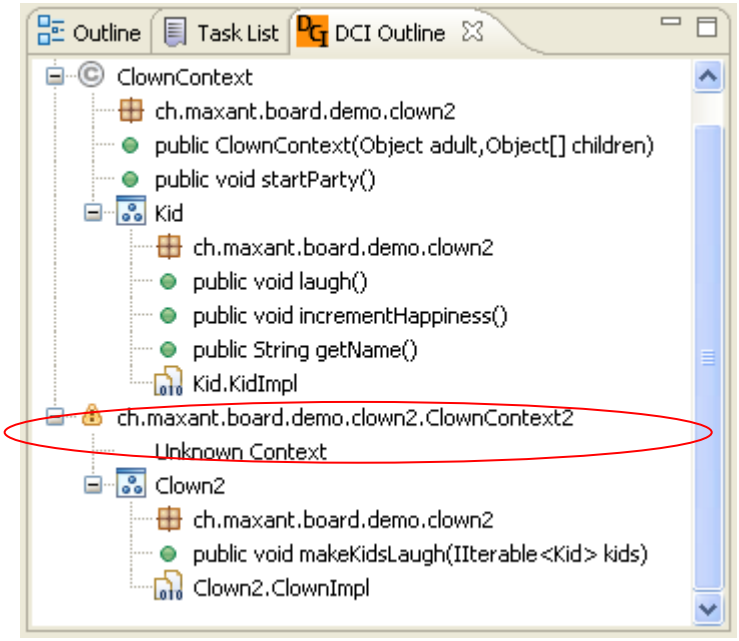
For each role interface, the package is listed, followed by a list of methods which the role interface contains. At the bottom of a role branch, is the role implementation class.

The idea of this plugin is to give the programmer a view of what contexts are available and what roles they contain. It puts contexts, roles and their methods into the mind of the programmer letting them think in terms of DCI, exactly the same way which a standard class browser lets the programmer think in terms of classes.

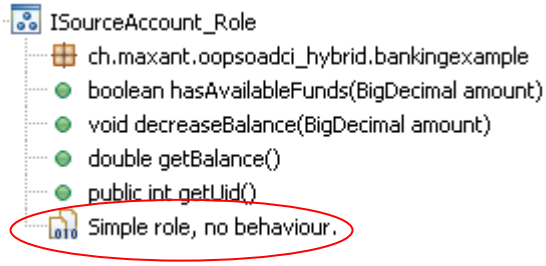
Double clicking on a role interface, role implementation or context will open it in the editor.

The plugin has limited validation capabilities and can determine for example, if a context specified in a role annotation cannot be found within the project, and highlights this so that the programmer can see there is a problem. In such cases, the view looks like this:

© 2010 Dr Ant Kutschera, www.maxant.co.uk



Finally, for cases where only a simple role exists, with no additional behaviour, then the view displays the role like this:



More Examples

More examples are available online at http://www.maxant.co.uk/tools.jsp. Below is the complete "Clown" example used above. The blog at http://blog.maxant.co.uk/pebble also contains some examples.

Tester method which calls the context:

© 2010 Dr Ant Kutschera, www.maxant.co.uk

```
@Test
public void testExecute() {

    //create an adult (the clown) and some children,
    //and let the party start...

    Human adult = new Human();
    List<Human> children = new ArrayList<Human>();
    Human child = new Human();
    child.setName("Johnny");
    children.add(child);
    child = new Human();
    child.setName("Jane");
    children.add(child);

    ClownContext cc = new ClownContext(adult, children.toArray());
    cc.startParty();

    //check the party went well...
    assertEquals(0, adult.getHappiness());
    for (Human c : children) {
        assertEquals(1, c.getHappiness());
    }
}
```

The context class:

```
package ch.maxant.board.demo.clown;

import java.util.ArrayList;
import java.util.List;

import ch.maxant.dci.util.BehaviourInjector;
import ch.maxant.dci.util.Context;
import ch.maxant.dci.util.IIterable;

/** the context in which a clown entertains some kids. */
@Context
public class ClownContext extends BehaviourInjector {

    /** the object who will play the clown role */
    private Object adult;

    /** the objects who will play the kid role */
    private List<Object> children;

    /**
     * Creates a new Context object. takes "objects" rather than specific types to allow it to be reusable.
     */
    public ClownContext(Object adult, Object[] children) {
        this.adult = adult;

        this.children = new ArrayList<Object>();

        for (Object child : children) {
            this.children.add(child);
        }
    }
```


© 2010 Dr Ant Kutschera, www.maxant.co.uk

```
    /** start the interaction */
    public void startParty() {
        Clown clown = assignRole(adult, Clown.class);

        clown.applyMakeup();

        IIterable<Kid> kids = getIterable(children.iterator(), Kid.class);

        clown.makeKidsLaugh(kids);
    }

}
```

The Clown role (interface):

```
package ch.maxant.board.demo.clown;


import java.awt.Color;

@Role(contextClass = ClownContext.class,
      implementationClass = ClownImpl.class)
public interface Clown {

    /** put some makeup on (initialise) */
    public void applyMakeup(); // role

    /** makes kids laugh */
    public void makeKidsLaugh(IIterable<Kid> kids); // role

    public void setHairColour(Color c); // data

}
```

The Clown role implementation:

© 2010 Dr Ant Kutschera, www.maxant.co.uk

```
package ch.maxant.board.demo.clown;

import java.awt.Color;

/** role impl for the IClown role */
public class ClownImpl {

    @Self
    private Clown self;

    /** @see Clown#applyMakeup() */
    public void applyMakeup(){
        self.setHairColour(Color.GREEN);
    }

    /** @see Clown#makeKidsLaugh(IIterable) */
    public void makeKidsLaugh(IIterable<Kid> kids) {
        for (Kid kid : kids) {
            kid.laugh();
            System.out.println();
        }
    }
}
```

The Kid role (interface) and implementation (as an inner class, for no particular reason):

```
package ch.maxant.board.demo.clown;

import ch.maxant.dci.util.Role;

@Role(contextClass = ClownContext.class,
      implementationClass = Kid.KidImpl.class
)
public interface Kid {

    public void laugh(); // role method
    public void incrementHappiness(); // data method
    public String getName(); // data method


    /** role impl for the Kid role */
    static class KidImpl {

        @Self
        private Kid self;

        public void laugh() {
            System.out.println("hehehe says " + self.getName());
            self.incrementHappiness();
        }

    }
}
```


© 2010 Dr Ant Kutschera, www.maxant.co.uk