A comparison of DCI and SOA, in Java.

DCI is a paradigm used in computer software to program systems of communicating objects. Its goals are to make code more readable by promoting system behaviour to first class status, by avoiding the fragmentation of this behaviour as typically seen in an object oriented solution. This allows the rapidly changing system behaviour code to be developed and maintained independently of the slower evolving domain model (data classes). This allows programmers to reason directly about system-level state and behaviour rather than having to create a map between their mental model, and that of the user, which leads to more easily maintainable code[1].

In DCI, the data model is used for just that, namely data structure. The behaviour is not partitioned along data class boundaries; rather it is self containing and as such has boundaries more natural to behaviour rather than data. In DCI, objects can dynamically take on roles determined by the context. A role is partly the system behaviour which relates to a particular use case or algorithm, but is also a way of viewing a particular object within such a use case.

These things make DCI very powerful, and make DCI a paradigm in itself, comparable to object oriented programming in its time, or service oriented programming. The author has architected, designed, programmed and maintained Service Oriented Architecture (SOA) solutions for nearly a decade and sees parallels between DCI and SOA. These parallels are discussed here.

This paper presumes a prerequisite knowledge of DCI, for example, see the Artima article http://www.artima.com/articles/dci_vision.html or the paper entitled "DCI in 1585 Words" at http://www.maxant.co.uk/whitepapers.jsp.

Services

SOA can be used to write software that fulfils most of the goals of DCI as shown in the introduction above. However, SOA does not necessarily concentrate on those goals, and the OASIS Reference Model[2] is completely unrelated. In fact, the definition of services has changed somewhat over the past decade. In 1997 when Sun released the EJB specifications for stateless session beans (SLSB), the first SOA solutions were built. Today, SOA very often relates to Web Services, distribution, metadata and interoperability. In this paper, it is the older original style of services which are considered.

Services define a contract which has to be fulfilled in order to use them. The first part is a definition of the data structures which a service method takes and returns when it is called. The second part of the contract is the service method signature, including the documentation about what such a method does. Optionally, a service might also define its location if it is remote, and ways to call it, but these are not necessary to create a local service. Local services can be written using frameworks such as Spring[3], or to specifications like EJB 3.1[4], or indeed by creating static methods in a stateless class.

Here is a random Spring service configured using annotations. Note how it is unrelated to a use-case, in this instance.

[1] Base upon the DCI definition in Wikipedia: http://en.wikipedia.org/wiki/Data,_Context,_and_Interaction
[2] SOA RM, OASIS: http://docs.oasis-open.org/soa-rm/v1.0/soa-rm.pdf
[3] Spring: http://www.springsource.org/
[4] EJB 3.1 Specifications: http://jcp.org/en/jsr/detail?id=318

```
@Service("designService")
@Transactional(
        propagation=Propagation.REQUIRED,
        isolation=Isolation.DEFAULT,
        rollbackFor=ApplicationException.class)
public class DesignService {
```

It contains service methods (not shown) for manipulating artistic designs in a system which lets artists upload and sell their art/designs online. Notice, it also contains annotations related to transactions, again, nothing to do with use-case code!

To use such a service, another piece of code (in this case, also a service) can ask the container in which it is running, to supply/inject the service:

```
public class OrchestrationService extends Mapper {

    @Autowired
    private DesignService designService;
```

In EJB, the annotations are different, but the idea is similar. At any time which code needs to use a service, it asks the container for an instance of the service class.

The use of the word service, in this paper, can be defined as a class which is stateless and has methods. The methods are called service-methods. These operate on dumb objects which are passed as parameters to the service-methods. These objects are called service-objects, but can also be known as transfer objects. In the interests of completeness, it is worth mentioning that services may also provide service-methods access to resources which are temporarily added to the service instance by the container in which the service is running.

Parallels

In DCI a role has a role-contract. The role-contract is made up of the methods which the role requires a data object[5] to have, in order that the data object play the role. These are typically the methods on the data object which role-methods need to call. Additionally, there are the role-methods (system behaviour) which the role enriches the data object with. Ignoring the role-methods in the first instance, allows one to consider the role-contract as a particular *view* on an object playing the role. As an example, a person has millions of attributes which define them. When that person plays a role, for example a clown, the number of attributes which are relevant, and even their organisation (structure) can change radically. While playing the role of a clown, the *view* of a persons attributes is different.

Returning to SOA, before calling a service-method, the programmer must map data objects from their world, into the data structures which form part of the service-contract. This transformation does nothing more than create a new view on the existing data, and as such can be thought of as equivalent to what happens in DCI when assigning the role-contract (i.e. the role without the role-methods) to the data object. However there is a difference, in that DCI *narrows* the view of existing attributes, whereby SOA allows the signatures to be completely changed.

[5] A data object is simply an object in DCI. The word data is used here to explicitly indicate that the object contains no system behaviour, and only has simple logic to ensure the object's attributes are maintained in a consistent state.

Back in DCI, there are also the role-methods. What happens to these in a SOA implementation? They are simply the service-methods.

In DCI there are also Contexts. The responsibility of a context is to decide which data objects will be assigned which roles, and to start the interaction by calling a role-method on an object which has been cast into a role. In rare cases, a context might contain code related to an algorithm, or use-case which does not specifically fall under the responsibility of any of the roles contained in the context. Role-methods in DCI should be considered as the inner workings of the context, and can even be implemented inside the context class. If system behaviour in DCI is to be used over and over, it is the context and roles which are put into a library.

In the service world, explicit context objects do not exist, but there is definite code which is responsible for mapping data objects into service objects (assigning roles), and then calling the service (starting the interaction). In the service world, there is the additional step at the end, which is the mapping/merging of the response object or modified call parameters back into the data model. In SOA, a context does nothing apart from assigning roles and starting the interaction, because, if the service is to be used over and over, it must encapsulate all the system behaviour itself. In SOA, a service method is passed the equivalent of role-players, but they are dumb. They are similar to the role-players of DCI just they have no methods injected. Their interface is the role-contract. A service-method is not specifically related to individual role-players, but rather it is related to all the role-players in the context. Hence a service method can contain code which in DCI might have to belong in the context.

So to recap, a little dictionary:

DCI World	Service World
Object	Data Object
Role Method	Service Method
Role Interface	Service Contract
Role Contract, i.e. Role Interface without Role Methods	Service Object, sometimes named Transfer Object
Context	Code in a Service Client

So, the service world can be used to do similar things to the DCI world, but in itself, it is unsatisfying because service solutions do not tend to relate to DCI – the goals are very different. There is no explicit requirement for a service solution to create use-case centric code, or to encourage the programmer's mental model to be the same as the users model.

Let us now consider some lower level direct comparisons between the two paradigms. Note these comparisons relate to implementing DCI and SOA using Java. The DCI examples posted below are related to a Java DCI implementation as documented in the DCI Tools for Java library, found at http://www.maxant.co.uk/tools.jsp. Other DCI implementations exist in Java, for example Qi4J, or ObjectTeams, but are not considered here.

Comparisons of Contexts and Roles

Consider the Frontloader example which has been discussed on the Object-Composition forum[6]. In this example, a Project consists of Tasks. Each Task has itself a list of Tasks upon which it depends. These tasks are planned using a frontloading[7] algorithm, which plans tasks to start as early as possible, so long as it is after the start of the project, and

[6] Google Group Object-Composition: http://groups.google.com/group/object-composition
[7] Front loading: http://www.thefreedictionary.com/front-loading

after the last task upon which they depend has been completed. Figure 1 shows the relationship in the data objects.

In a DCI solution, the project can be assigned the role of a frontloader, and each task can be assigned the role of an activity. The frontloader role provides the behaviour which iterates or recurses over the object graph and the activity role provides the behaviour which determines a start date for an individual task based on its dependencies. These two roles are also shown in Figure 1. In the DCI solution for this example, there is no algorithmic code in the context related to frontloading.

In a SOA solution, the data model is mapped to a dumb data model (transfer objects). The service contains the algorithms and system behaviour in service methods and is passed the transfer objects in the service-method call. These are also shown in Figure 1.

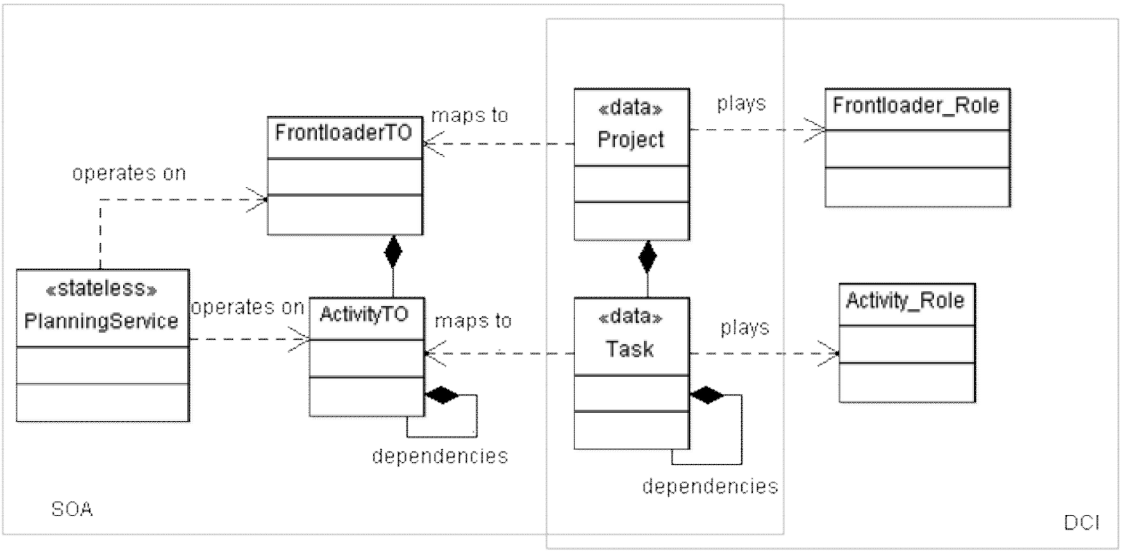


Figure 1: Class Diagram showing classes involved in both the DCI and SOA solutions to the frontloader example.

The following listing shows the code from the DCI context.

```
public static class FrontLoad_Context {

    private Project_Data project;
    private BehaviourInjector bi = new BehaviourInjector(this);

    public FrontLoad_Context(Project_Data project){
        this.project = project;
    }

    public void frontLoadFrom(Date projectStart){

        //casts the project into a front loader role, so it can plan itself.
        //to do so, it depends on activities.  the casting from task->activity is
        //done inside the role method, using the current context.
        IFrontLoader_Role fl = assignRole(project);

        fl.doFrontloadingFrom(projectStart);
    }
```

Listing 1: DCI Context Code for the Frontloader example

And the code below shows the analogous class in the service world, i.e. a class which calls a service.

```
public static class FrontLoad_Context {

    private Project_Data project;
    private Mapper_ForExample7Iteration mapper = new Mapper_ForExample7Iteration();

    public FrontLoad_Context(Project_Data project){
        this.project = project;
    }

    public void frontLoadFrom(Date projectStart){

        //casts the project into a front loader TO.
        //at the same time, it casts all tasks into activity TOs.
        //that is ok, because in the context of frontloading, we know that
        //a project consists of activities!
        IFrontLoaderTO frontLoaderTO = mapper.mapFromProject(project);

        FrontLoaderService.doFrontloadingFrom(projectStart, frontLoaderTO);

        //in SOA we also unmap / merge
        mapper.mergeResults(project, frontLoaderTO);
    }
```

Listing 2: SOA Service Calling Code for the Frontloader example

Comparing Listing 1 & Listing 2, one can see the similarities, but there are marked also differences.

The biggest difference, is that role-methods are called on objects (e.g. `objectPlayingRole.roleMethod()`), rather than the objects being passed to a method (e.g. `service.serviceMethod(service-object)`). Because the service-method is in a stateless class, it can be though of as just hanging in the air. The fact that it belongs to a class is more to do with Java, as well as a need to somehow call it, rather than anything else.

There is a good reason why DCI insists on the methods being injected into the objects, and that is that a goal of DCI is to keep the code object oriented. Having methods which just hang in the air, which are passed objects to operate on, is not object oriented. While OO can live with individual objects having behaviour, the service paradigm considers there to be higher powers, namely services, which do work on objects, rather than the objects doing work on themselves and their friends. See the end of this paper for more discussion on this topic.

One goal of DCI is to make the code read the same as a use-case. We shall assume that the DCI listing does this. As such, the mental model being used contains a Project and some Tasks, a Frontloader and Activities.

In the service solution, if it also reads like the use case, then the mental model contains different objects, namely it too has a Project consisting of Tasks, but it then has a Frontloader Service and an Activity Service. Some would argue, that this is a technical model, rather than one shared by end users and analysts. So consider Listing 3, where the class and variable names have been modified.

```
public static class FrontLoad_Context {

    private Project_Data project;
    private BehaviourInjector_ForExample6Iteration bi = new BehaviourInjector_ForExamp.

    public FrontLoad_Context(Project_Data project){
        this.project = project;
    }

    public void frontLoadFrom(Date projectStart){

        //casts the project into a front loader.
        //at the same time, it casts all tasks into activities.
        //that is ok, because in the context of frontloading, we know that
        //a project consists of activities!
        IFrontLoader_Role frontLoader = bi.assignRoles(project);

        frontLoader_Role.doFrontloadingFrom(projectStart, frontLoader);

        //unassign roles
        bi.unassignRoles(project, frontLoader);
    }

}
```

Listing 3: SOA Service Calling Code, but with modified class and variable names

Changing the names of classes and variables could be considered a parlour trick, but the resulting code (Listing 3) is now almost identical to the DCI code (Listing 1). We have achieved the goal of making the service solution use the same mental model as the DCI solution. We have also used the same objects for system state and roles for system behaviour as DCI, but note how the semantics are different. In DCI, the method `doFrontloadingFrom()` takes a parameter, and is called on the object playing the role of the front loader. In the service solution, the method takes two parameters, and is called as a static method rather than a method on an object. The two solutions differ because of the semantics, and as such, it is not recommended that a service solution be modified to use an identical mental model as the DCI solution. The semantics in the service solution are now wrong – no analyst would model their world like this, with the front loader being passed the front loader data.

One could argue that a programmer would get used to reading the service solution's `frontLoader_Role.doFrontloadingFrom(projectStart, frontLoader)` as "the object playing the role of the front loader, does frontloading based on a start date", rather than "the object playing the role of the front loader, does frontloading based on a start date and itself", but to do this, the programmer is translating what he sees to what he says, and this translation, or mapping is an unnecessary one, which just complicates the entire situation.

So, for a SOA solution to gain the benefits of DCI, the mental model used by programmers, analysts, users and other stake holders needs to be different than the mental model used in DCI.

Let us now consider the role-methods / service-methods, and compare them in both paradigms. Here is the DCI code for the front loader role, which a project plays:

```
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;
            }
        }
    private boolean hasUnplannedPredecessors(Task_Data task) {
        //does this task have any predecessors who are unplanned?
        //if so, it cannot be planned
        boolean hasUnplannedPredecessor = false;
        for(Task_Data p : task.getDependencies()){
            if(!p.hasBeenPlanned()){
                hasUnplannedPredecessor = true;
                break;
            }
        }
        return hasUnplannedPredecessor;
    }
}
```

1

2

3

Listing 4: DCI Role-Method for Iterating over Tasks

The DCI code in Listing 4 has to keep assigning Task data objects the Activity role in order to get them do to anything useful (annotations 1 & 2 in the listing). This role-assignment is done on the fly. The role-assignment requires a reference to the "current context", in order to do the role assignment, because it is the responsibility of the context to do role assignment. It could be argued that the role-assignment does not need to be done by the current context, but could be done by a language construct or framework (or whichever mechanism is in use), directly. If it *is* the current context which does the role assignment, it is not a real problem, because the context and roles are two parts of a "whole" in DCI; they belong together and are inseparable because a role makes no sense without a context. As such, there is no real cyclic dependency, especially, because the "current context" is (in this example) a reference to an interface

– an object playing the role of the current context, rather than the context object itself. It has been decoupled.

Note also the "self" object, which is a reference to the object currently playing the role (the interface in the role-contract).

Annotation 3 in the listing shows that the role-method is also operating on data objects which have not been cast into roles. The objects could be assigned roles, but in the interest of saving code, have not been. To the author's knowledge, DCI does not dictate that assignment is always necessary.

Now the service solution:

```
public static void doFrontloadingFrom(Date projectStart, IFrontLoaderTO frontLoader) {
    //select an activity which has not been planned
    while(true){
        boolean plannedAll = true;
1       for(IActivityTO activity : frontLoader.getActivities()){
            if(!activity.hasBeenPlanned()){
                if(!hasUnplannedPredecessors(activity)){
2                   ActivityService.frontloadFrom(projectStart, activity);
                }else{
                    //gotta wait until predecessors are planned :-(
                    plannedAll = false;
                }
            }
        }
        if(plannedAll){
            break;
        }
    }
}

private static boolean hasUnplannedPredecessors(IActivityTO activity) {
    //does this task have any predecessors who are unplanned?
    //if so, it cannot be planned
    boolean hasUnplannedPredecessor = false;
    for(IActivityTO p : activity.getPredecessors()){
3       if(!p.hasBeenPlanned()){
            hasUnplannedPredecessor = true;
            break;
        }
    }
    return hasUnplannedPredecessor;
}
```

Listing 5: SOA Service-Method for Iterating over Tasks

In Listing 5, the service-method receives the transfer object which contains all its child objects and their children – the entire object graph was mapped before the interaction started. As such, at annotations 1 & 3 in the listing, no new casting is required, and so there is no need to have a reference to the current context or a mapper.

Annotation 2 shows that a finer-grained service (the `ActivityService`) is being used in order to plan the individual activity (set start date equal to latest end date of dependencies). In SOA it is more likely that the method doing this part of the algorithm would be part of the `FrontLoaderService`, rather than a lower level service, because all code relating to the specific business-case (planning) is contained within the service. Because DCI is about adding behaviour specific to roles to those objects who play the role, the planning of individual tasks fits more naturally inside the role played by the individual tasks (i.e. the Activity role). But the role is still specific to the context, and as such, is contained in the same namespace as the main role-method and context. This is

exactly what a SOA solution would strive to do. The code shown near annotation 2 is probably more related to creating and calling a sub-context in DCI, than it is to calling a different role-method in DCI.

Let us now consider the DCI role-methods for planning individual tasks.

```
public void frontloadFrom(Date projectStart) {
    // set my earlyStart to the maximum of projectStart and
    // the earlyFinish of all predecessors
    Date earliest = projectStart;
    for(IActivity_Role predecessor : getPredecessorsList()){
        if(predecessor.getEnd().after(earliest)){
            earliest = predecessor.getEnd();
        }
    }
    self.setStart(earliest);
}

public List<IActivity_Role> getPredecessorsList(){
    //not so nice: activity role cant get the list of predecessors in the
1   //right role, it needs to cast each object into the role!  it does that,
    //with the help of the context
    List<IActivity_Role> predecessors = new ArrayList<IActivity_Role>();

    for(Task_Data t : self.getDependencies()){
        predecessors.add(currentContext.assignRole(t, IActivity_Role.class));
    }
    return predecessors;
}
```

Listing 6: DCI Role-Method for Planning an individual Task

Listing 6 shows two methods for planning individual tasks in DCI. The upper method is fine and almost identical to the method shown in the SOA solution (Listing 7, below), but relies on a helper method (annotation 1) which must again do casting of child data-objects into roles.

```
public static void frontloadFrom(Date projectStart, IActivityTO activity) {
    // set my earlyStart to the maximum of projectStart and
    // the earlyFinish of all predecessors
    Date earliest = projectStart;
    for(IActivityTO predecessor : activity.getPredecessors()){
        if(predecessor.getEnd().after(earliest)){
            earliest = predecessor.getEnd();
        }
    }
    activity.setStart(earliest);
}
```

Listing 7: SOA Service-Method for Planning an individual Task

Listing 7 shows that in the SOA solution, no re-assignment of roles is necessary, because of the upfront mapping which took place.

Finally, in the interests of full disclosure, it is important to consider what happens in the context, when in the DCI solution, roles are assigned, or in the SOA solution, objects are mapped.

```
private IFrontLoader_Role assignRole(Project_Data project) {
    IFrontLoader_Role fl = bi.assignRole(
            project,
            FrontLoader_Role.class,
            IFrontLoader_Role.class);
    return fl;
}
```

Listing 8: DCI Role Assignment

Below is the analogous code for the service solution.

```
public IFrontLoaderTO mapFromProject(Project_Data project) {

    //map.
    //it is IMPERATIVE, that the roles know nothing about the data objects.  data objects
    //are in a different world than these service objects.
    Map<String, Task_Data> task_index = new HashMap<String, Task_Data>();
    Map<String, IActivityTO> activity_index = new HashMap<String, IActivityTO>();
    FrontLoaderRole fl = new FrontLoaderRole();
    for(Task_Data t : project.getTasks()){
        IActivityTO a = new ActivityRole(t.getId(), t.getEstimatedMinutes());
        fl.getActivities().add(a);

        activity_index.put(t.getId(), a);
        task_index.put(t.getId(), t);
    }

    //now map dependencies
    for(IActivityTO a : fl.getActivities()){
        Task_Data t = task_index.get(((ActivityRole)a).getId());
        for(Task_Data t2 : t.getDependencies()){
            a.getPredecessors().add(activity_index.get(t2.getId()));
        }
    }

    return fl;
}
```

Listing 9: SOA Mapping

Comparing Listing 8 and Listing 9, shows that casting of roles in DCI is a simple affair. The same objective in the SOA solution is comparatively unsatisfactory, although it must be said that mapping of structures with interdependencies is particularly difficult. In SOA, a normal mapping involves simple mappings from one graph to another, with no need for indexes as shown in Listing 9 to track dependencies and set them up in the target graph.

This negative point in the SOA solution can be turned into a positive point. In DCI, a role-contract requires any data object playing the role to have a list of methods. The role-contract specifies the exact signatures of those methods (even in dynamic languages where no explicitly “interface” exists, there is still an implicit requirement for objects to have the correct data methods in order to play a role). In cases where a context and role are to be used by many data objects, all of the data objects must contain the same methods, even if the designer of the domain model would prefer slightly different names in individual data classes. Or consider a case where a German company want’s to use a context & role from a library written by an English company. In order to do this, the German company is required to have English naming in their data model, which may go against company policy. Of course a mapper / wrapper / other mechanism can be used to solve this problem, but in such cases, DCI no longer fairs better than the SOA solution. The SOA solution always requires a mapping, so such problems are naturally solved.

Returning to Listing 6 shows how DCI requires on the fly role assignment. The SOA solution does not require this because it is able to "assign roles" up front, to all objects in the graph. DCI cannot do this for the following reasons. If the `Frontloader_Role` which the project plays is to have a method `getActivities()` which returns a list of activities (i.e. tasks already cast into their role), and not do this on the fly, it needs a place to hold the list of activities which are assigned during such an up front assignment. But there is no place for this list to go, because roles in DCI are stateless.

Nonetheless, a more elegant solution can be created. The `BehaviourInjector` from the DCI Tools for Java library, has a method for getting an `IIterable_Role`. This object is a Java `java.lang.Iterable`[8] playing a special generic role from the library, which does the role assignment automatically. The code in annotation 3 from Listing 6 reduces to the following single line:

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

More complex cases

While probably affecting less than 1 ‰ of production code, attempting recursion in DCI and SOA makes an interesting comparison, partly because it is a demonstration of more complex code.

```
public void doFrontloadingFrom(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
1       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(
2               activity.getDependencies().iterator(),
                IActivity_Role.class,
                FrontLoad_Context.ACTIVITIES);

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

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

Listing 10: DCI Role-Method for Frontloading Recursively

Listing 10 shows how recursion has been implemented in the DCI Tools for Java library, based on suggestions from the DCI Execution Model[9]. Annotation 1 shows how the current context is suspended and its role-map is frozen. A new role-map is pushed onto the stack. Annotation 2 shows re-assignment of the role "Activities", which sets the object playing the role of "Activities" (an IIterable_Role of generic type IActivity_Role) to be the next level in the hierarchy to be recursed. Annotation 3 restarts the context with the new objects playing the relevant roles. Once that restart finishes, the new role-map is popped off the stack and the suspended role-map once again becomes active.

[8] An Iterable is a class which allows "for loops" to iterate over lists by typing code like this: `for(IActivity_Role activity : anIterable){}`, which reads as "for each activity in anIterable".

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

```
public static void frontloadFrom(Date projectStart, List<IActivityTO> currentActivities) {

    for(IActivityTO activity : currentActivities){

        if(activity.hasBeenPlanned()) continue;

    1   frontloadFrom(projectStart, activity.getPredecessors());

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

Listing 11: Service-Method for Frontloading Recursively

Listing 11 shows how in a SOA solution, recursion is just like normal it would be in procedural programming.

Other Noteworthy Points

SOA Flexibility

It is feasible that a frontloading algorithm might be made available, either via a library, or as an online web service, and that it be published as a service by a company interested in attracting business partners.

As an example of the flexibility of SOA services, now consider a company which is creating software for publishing menus. The user is able to select a set of courses and the application provides the user with a list of ingredients and timings for when to start each step in each recipe, and the recipe itself. The system then orders the ingredients and arranges their delivery. The data model which they might create could contain Recipes which are made up of Recipe Steps and interdependencies, very similar to the Project/Task data model used in the examples above.

In order to do the planning of such recipes it is feasible that they, not being planning experts, would seek the advice of a business partner and find the one mentioned above who publishes a planning service.

SOA and its mappings are flexible enough to allow the recipes from the one company to be planned by the planning service from the other company, even though the data models are fairly different. Consider the following listings, and notice how while the data models use different language, the planning service is fully integratable into the recipe solution.

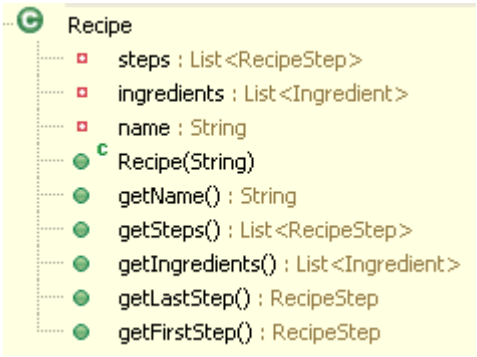
Equally, a third company, wanting to offer parties (e.g. bachelor parties) with a number of events in each party, could use the planning service too for planning the day and the event timings.

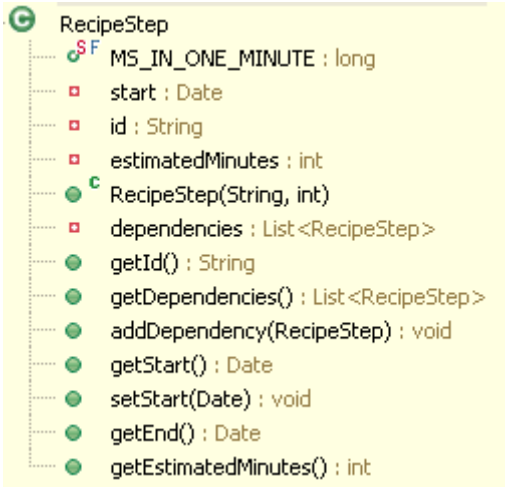
```
/** tester method */
public static void main(String[] args) {

    Recipe breakfast = RecipeBook.getBreakfast();

    KitchenService.assembleMeal(breakfast, new Date());
    System.out.println("Done: " + RecipeBook.outputRecipe(breakfast));
}
```

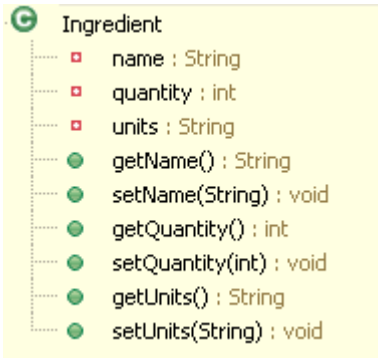
Listing 12: Tester Method, to start the Application

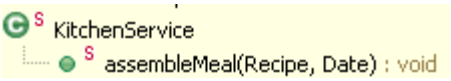
```
public static void assembleMeal(Recipe breakfast, Date projectStart){

    // ... check ingredients in stock

    //map from kitchen world, into project planning world
    Mapper mapper = new Mapper();
    IFrontLoaderTO frontLoaderTO = mapper.mapFrom(breakfast);

    //do planning
    ProjectPlanningService.frontloadFrom(projectStart, frontLoaderTO.getActivities());

    //merge results so we can display them
    mapper.mapBack(breakfast, frontLoaderTO);

    // ... create order

    // ... restock, etc
}
```

Listing 13: The Kitchen Service Implementation

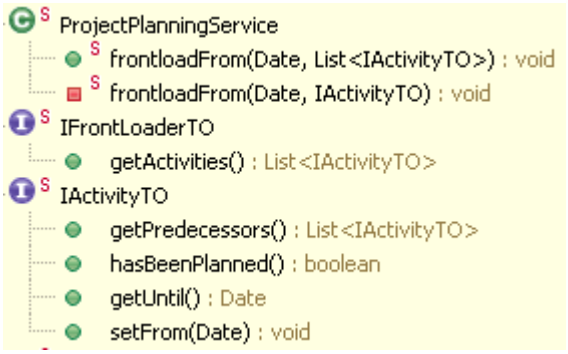








Listing 14: The Kitchen Service and its Service Objects



Listing 15: The Planning Service and its Service Objects

Notice how the two data models are quite different and none of the methods are the same. Note that the `PlanningService` uses the recursive code exactly as shown in Listing 11.

Suspension of Polymorphism

The DCI Execution Model talks about suspending polymorphism for role-methods. In SOA it is unusual for a service to be sub-classed. If different functionality is required, it is normal to use configuration to make a different implementation of the service available to the container. As such, both paradigms make static analysis of algorithms possible, however to do so in a SOA environment might require static analysis of a configuration (file).

Resources and Cross Cutting Concerns

Services typically live inside containers. Containers are technical entities which allow the configuration and management of resources such as databases, email servers, or other external systems, to be separated from the deployable code. They also allow cross-cutting concerns such as transactions and security to be handled outside of the business code which service-methods implement. Scalability, reliability, concurrency and things such as resource pooling (such as threads, databases, or the service instances themselves) are all handled by the container. DCI currently mentions nothing about such things. However, if a Context were to be deployed as a "service" in the technical sense, it would immediately have access to all of these advantages. Role-methods starting sub-contexts would be similar to service-methods calling lower level services, which allows the sub-context / sub-service-method to reconsider the cross cutting concerns (e.g. start an inner transaction or re-check security for different roles) as well as address different resources.

Higher Powers

Think about writing a graphical game in which players have tanks, and the tanks move or shoot on each turn. When a tank shoots at another tank, the attributes might be direction, power, etc. One could imagine a method on a tank object called "shoot". It might even be passed another tank object which is the target. But what happens next? In real life, the shell may, or may not hit the target. Who decides whether it hits, and if it does, how much damage is inflicted on the tank? Fate, perhaps? Or the skill of the tank crew… However, a good way to model this in software is to let the gaming engine decide. If MVC were used in this game, it would be the controller which would receive an event about the user's decision to shoot. The controller would make a call to something in order that the shooting be calculated and the result applied to the model, so that the

view can be updated and the game can continue. If it were the author of this paper writing the game, the logic for the shooting would not be implemented in the Tank class, rather in the gaming engine, which can be considered to be a service. The gaming engine would receive the tanks, and user's choices, and it would use more complex algorithms to determine the result.

Consider something completely different from a tank game, namely a word processor. When the user wants to insert an image from a file, the user selects the file and clicks the OK button to get the system to read the file, place it in the current paragraph and display it on the screen. The author has quizzed a number of people about how they model such things in their minds and has always been told that it is either the user themselves, or "the system" or the word processor (a higher power, a service) which does these tasks. It has never been stated that the document or a paragraph or any other entity which is part of a typical document data model does such things.

Things such as game engines and word processors have state, but this state is the "M" in MVC; it is the data model which SOA or DCI enrich with functionality. Such state does not need to be modelled together with the higher-power behaviour which operates on it, as is suggested by object orientation, i.e. in the same class. Rather the higher-power behaviour can and should be split out into another class. Whatever the situation, in almost all problems solved by software, one can find a higher power responsible for things related to use-cases or algorithms.

User's Mental Model

DCI aims to make the user's mental model the programmer's mental model, so that there is no need to map between the two. In reality, there is never one user, and never one programmer, rather there are many of both. This means that the real aim is for everyone to work with the same mental model. In order for everyone to use that same mental model, there will be debate and discussion, when it is perfectly acceptable to encourage users (and analysts and other stake holders) to think of the higher powers in software which might actually contain the behaviour which needs implementing.

Layers

In SOA solutions it is typical to create a service hierarchy, in which services are assigned to specific layers. These layers have different responsibilities, such as an "Application Layer", a "Business Layer" and an "Integration or Persistence Layer". The application layer is responsible for containing use-case related code, which is comparable to the steps which a user might take while operating an application. The business layer is in charge of containing business logic, and algorithmic details of use cases, which the user is not necessarily directly concerned about. The lower level layers are normally very fine grained and allow the details of integrating other systems or persisting data to be hidden away.

For DCI, it has been suggested that it is currently unclear where business rules might be implemented[10], and it is not currently defined whether DCI needs to address this. It has also been suggested[11], that DCI would have the most use in the upper application layer, where use cases are normally modelled in SOA solutions.

Summary

DCI has powerful goals and characteristics. The service world could benefit from these goals and characteristics by starting with a model which is closer to the common mental model shared by all stakeholders. Too often, SOA solutions are too technical, and the

[10] Artima Article: http://www.artima.com/articles/dci_vision.html
[11] Google Group Object-Composition: http://groups.google.com/group/object-composition

mental model of the programmer differs from that of the user or analyst. That said, services alone do already splits system behaviour and data.

This paper has shown that a service solution and a DCI solution are very similar, although the semantics and mental models differ somewhat.

The source code used in this document is available from http://www.maxant.co.uk/tools.jsp.

About the Author

Dr Ant Kutschera has been working in IT since 2000, implementing software in the enterprise, in all aspects of the software life cycle, including but not limited to requirements gathering, architecture, design, programming, testing, delivery, support and maintenance. He currently works for various clients as an independent consultant, specialising in software architecture, Java EE, Rich Clients and SOA. He can be contacted through:

whitepapers@maxant.co.uk