Assessing the DCI Approach to Preserving Use Cases in Code: Qi4J and Beyond

Jozef Zaťko and Valentino Vranić
Institute of Informatics and Software Engineering
Faculty of Informatics and Information Technologies
Slovak University of Technology in Bratislava
Ilkovičova 2, Bratislava, Slovakia
zatko7071@gmail.com, vranic@stuba.sk

***Abstract*—DCI (Data, Context and Interaction) comes from role-based programming and separates the system state from its behavior making it possible to preserve use cases in code to the great extent. In its Java implementation, DCI relies on the Qi4J (renamed to Apache Zest at the time of finalizing this paper) framework for role injection. This paper provides an assessment of DCI via its Qi4J implementation and beyond based on an independent study of a small car dealer system development. Two most important conceptual findings are that roles can reduce inheritance and decrease maintainability and that generic roles can be played by objects of inappropriate classes. The findings specific to the Qi4J implementation include loss of the direct domain model access from the generic context roles, entities defining their casting rules, use of interfaces instead of classes as templates for objects, no access management of the data class attributes and methods, and no direct support of polymorphism.**

***Keywords*—DCI, use case, role, Java, Qi4J, Apache Zest**

I. Introduction

DCI (Data, Context and Interaction) is a relatively new approach to software development [1], [2], yet it is based on long time known role based programming. At the implementation level, DCI captures and embodies the programmer's intent in the form of the interaction of objects that take their part in this interaction through the corresponding roles subsequently departing from them. In effect, this makes use cases significantly more visible in code, as they are expressed in a procedural fashion in terms of the roles that bear expressive names.

The implementation setting that assumes objects changing their roles sounds as if dynamic programming languages are the condition sine qua non to comply to DCI. However, the use of DCI has been demonstrated in several static programming languages including Java [2]. In its Java implementation, DCI relies on the Qi4J framework (renamed to Apache Zest at the time of finalizing this paper) for role injection. This paper provides an assessment of DCI via its Qi4J implementation and beyond based on an independent study of a small car dealer system development.

Section II explains the fundamentals of DCI. Section III provides a fresh look at implementing DCI in Qi4J. Section IV reports the conceptual assessment findings of this implementation. Section V reports the implementation assessment findings. Section VI discusses the related work. Section VII concludes the paper.

II. DCI

In DCI, each use case is implemented by a single, so-called *context* class. The attributes of this class are roles. The use case behavior is executed by calling the methods of use case roles, which represents *interaction*. Roles are generic [3], implemented separately, and they are independent of the domain objects. The only purpose of the domain objects, the *data* part, is to represent the system state.

However, a DCI role has to reference domain data object that plays it in order to access to the attributes of the domain data object. The biggest challenge in DCI is to implement this connection without creating a direct dependency between the role and the domain object that plays it.

A. Data

The data part of the DCI architecture is represented by the classes of the domain model. The only allowed methods are attribute accesors (i.e., setters and getters) or methods that do not belong to any particular use case. Data represents the stable part of the system.

B. Context

Each context class represents one use case or habit (a habit is a kind of a rudimentary use case: a sequence of steps without direct business value [2]). The context should be close to the use case specification as much as possible in order to reduce the gap between the software specification and its implementation. The structure of each context class consists of the use case attributes including references to roles, initialization method (to set up use case attributes), execution method (the trigger method that represents the main flow of the use case), and methods representing alternative flows.

The same role can be played by different objects. The context usually starts with the same method of the same role, but it can follow different paths according to the state of the domain objects playing individual roles [2].

C. Interaction

The interaction part of the DCI is about roles and their interaction, hence the name. The roles come from the user's mental model. DCI recognizes two types of roles: methodful and methodless roles. The methodful roles encapsulate the

system behavior maintaining the real knowledge of what-the-system-does. They reference the data objects (domain objects) and work with their attributes.

The purpose of the methodless roles is to define which roles can by played by domain objects. Their implementation differs according to the programming language. For example, they can be implemented using Java or C#'s interfaces or C++'s abstract base classes [2].

A methodfull role is stateless: its state is defined by the object that plays it. The most challenging task in the DCI implementation is to make an object play a role. For this, the role behavior needs to be injected into the data object. However, no dependency between the role and the data class may be created.

D. Execution Model

The DCI execution model consists roughly of four steps (the order of steps 2 and 3 is not defined):

1) The execution starts by instantiating one of the context classes (based upon the use case activated by the user)
2) The context object finds, creates, or retrieves objects from the data part that play the roles of the context
3) The context injects the neccessary role behavior into domain objects making them play the corresponding roles
4) The context invokes the role methods (the interaction starts)

III. DCI IN QI4J

Qi4J (newly called Appache Zest) is a framework for composite-oriented programming [4]. It's building blocks are *composites* that in turn consist of simple elements called *fragments*. Fragments are reusable building blocks of the Qi4j framework that bear the system behavior and state of composites (in mixins), validate and constrain their usage (in constraints), and handle cross-cutting concerns (in concerns) [5].

The study presented in this paper is based on Qi4J version 1.3.

A. Data Implementation

To be able to create the composite of the data object and role object in Qi4J (required for the execution of contexts), we are forced to use interfaces instead of classes. Java classes do not support multiple inheritance, but interfaces do. DCI defines that a role can be theoretically played by objects of multiple classes. To be able to keep this idea in source code, interfaces are the only option.

Up to Java 8, Java interfaces supported only method signatures in their bodies. Now they support method implementation (so-called default methods), but still only static attributes. For DCI, it is crucial to find the way to include the data attributes. A solution to this problem is provided by Qi4J in the org.qi4j.api.property.Property interface. Properties behave like method signatures. They are not inherited and can be used in the interface body. They provide set/get methods for a full access to the stored data. Properties are generic: they can store any data type. The only limitation is that we cannot set access modifiers for them: all interface elements are public in Java. Listing 1 demonstrates how properties are used to model the car data in our car dealer system. It also includes a reference to another data interface (ContractData).

```
public interface CarData {
    @Optional Property<ContractData> contract();
    @Optional Property<String> manufacturerName();
    @Optional Property<String> modelName();
    @Optional Property<String> color();
    @Optional Property<Boolean> isNew();
    @Optional Property<Double> price();
    @Optional Property<Integer> year();
    ...
}
```

Listing 1. The use of Qi4J properties in a data interface.

The @Optional annotation at properties tells Qi4J not to require their initialization in the process of entity creation (explained in Section III-D).

The purpose of Java interfaces is just to declare the implemented methods. The actual implementation is in the classes that implement these interfaces. The best solution is to implement these interfaces in a nested abstract class, i.e., in a mixin. This solution is illustrated in Listing 2.

```
public interface CarData {
    ...
    public String getName();
    abstract class Mixin implements CarData {
        /* Returns appended car name (manufacturer + model) */
        public String getName() {
            StringBuffer sb = new StringBuffer("");
            sb.append(this.manufacturerName().get());
            sb.append(" ");
            sb.append(this.modelName().get());
            return sb.toString().trim();
        }
    }
}
```

Listing 2. The use of a nested mixin class to define method in the Qi4J data interface.

The @Mixins annotation tells Qi4J that the interface includes a mixin and will be a part of a composition (of the data object and its role). Listing 3 shows the usage of the @Mixins annotation.

```
@Mixins(CarData.Mixin.class)
public interface CarData {
    ...
}
```

Listing 3. The usage of the @Mixins annotation in the Qi4J data interface.

B. Context Implementation

The context structure can be fully captured in Qi4J. The only limitation is the use of interfaces. A context interface declares public context methods. Usually, two public methods—one to initialize and one to execute the context—are sufficient. These methods have to be implemented in a class. For this, a nested mixin class is used. Listing 4 shows the basic structure of a context interface in Qi4J.

```
@Mixins(InsureContext.Mixin.class)
public interface InsureContext extends TransientComposite {
    public void initContext(...);
    public void executeContext();
    abstract class Mixin implements InsureContext {
```

```
        public void initContext(...) { }
        public void executeContext() { }
    }
}
```

Listing 4. The basic structure of the context in the Qi4J.

The purpose of the initialization method is to set up the context roles. Each context works with roles and their methods, and not directly with the domain objects. Qi4J composites are used to bind the objects to roles (explained in Section III-D). It is important that the injection of the role behavior into the data object is not performed in a context class: Qi4J does that job. The required attributes of the initialization method are methodfull roles. If they are set up as context object attributes, their methods can be called to execute the context via the context execute method. The context execution method covers the steps of the use case. Listing 5 presents examples of the implementation of context methods.

```
@Mixins(InsureContext.Mixin.class)
public interface InsureContext extends TransientComposite {
    /* The context methods being used */
    public void initContext(InsuranceContractRole insurance, InsurerRole insurer,
        InsurableRole insurable);
    public void executeContext();
    abstract class Mixin implements InsureContext {
        /* The roles being used */
        InsuranceContractRole insurance;
        InsurerRole insurer;
        InsurableRole insurable;
        /* Context initialization */
        public void initContext(InsuranceContractRole insurance, InsurerRole insurer,
            InsurableRole insurable) {
            this.insurance = insurance;
            this.insurer = insurer;
            this.insurable = insurable;
        }
        /* Context execution */
        public void executeContext() {
            /* This is the way how to keep the use case algorithm
               readable */
            this.insurer.prepareInsuranceContract(insurance);
            this.insurance.setInsurer(insurer);
            this.insurable.insure(insurance);
            this.insurer.confirmInsuranceContract(insurance);
        }
    }
}
```

Listing 5. Context methods.

Each implemented context in the example extends the TransientComposite interface. The extension of the TransientComposite interface allows for the role and data object binding. Note that the context is generic. It only depends on roles, and these can be played by objects of different classes.

C. Role Implementation

The real challenge starts with the implementation of roles. To retain the underlying DCI idea in code, it is necessary to find the way how to inject the role behavior into data objects without creating a dependency. As a role has to be playable by objects of different classes, interfaces have to be used again. Each role interface extends the TransientComposite interface because the role is actually a composite. The role interface includes signatures of the role methods. These methods are implementated in a nested mixin class. Listing 6 shows an example of this.

```
@Mixins(InsurerRole.Mixin.class)
public interface InsurerRole extends TransientComposite {
    /* Role methods -- behavior */
    public void prepareInsuranceContract(InsuranceContractRole insurance);
    public void confirmInsuranceContract(InsuranceContractRole insurance);
    abstract class Mixin implements InsurerRole {
        public void prepareInsuranceContract(InsuranceContractRole insurance) {
            ...
        }
        public void confirmInsuranceContract(InsuranceContractRole insurance) {
            ...
        }
    }
}
```

Listing 6. The basic structure of a role.

The implementation of the role methods requires to obtain the reference to the object playing the current role. Without the knowledge contained in the data objects, the role algorithm cannot work. A reference to the data object playing the current role is provided by the @This annotation. We annotate the object of the new nested interface defined in the role body. This interface has its own properties. Qi4J maps each property of the data object to the property of the interface annotated with @This. This is done in runtime and it corresponds to the underlying DCI idea. The runtime role to objects binding guarantees that each change in the role property will cause changes in the object playing the current role. Listing 7 presents the structure of a role interface in Qi4j.

```
@Mixins(InsurerRole.Mixin.class)
public interface InsurerRole extends TransientComposite {
    /* Role methods -- behavior */
    public void prepareInsuranceContract(InsuranceContractRole insurance);
    public void confirmInsuranceContract(InsuranceContractRole insurance);
    /* An interface represents the data of the objects
       playing the current role */
    interface InsurerData {
        @Optional Property<String> firstName();
        @Optional Property<String> familyName();
        @Optional Property<List<InsuranceContractRole>> contracts();
    }
    abstract class Mixin implements InsurerRole {
        @This
        InsurerData data;
        public void prepareInsuranceContract(InsuranceContractRole insurance) {
            insurance.setApproveFlag(Boolean.FALSE);
            insurance.setCompleteFlag(Boolean.FALSE);
            insurance.setInsurer(data.firstName().get(), data.firstName().get());
            insurance.setCreateDate(new Date());
            data.contracts().get().add(insurance);
        }
        public void confirmInsuranceContract(InsuranceContractRole insurance) {
            insurance.setCompleteFlag(Boolean.TRUE);
            insurance.setSignDate(new Date());
        }
    }
}
```

Listing 7. A complete structure of a role.

D. Qi4J Entities

An entity is the basic element of a Qi4J application. Entities represent the persistent data of the system. We can implement them as composites, the building blocks of the Qi4J application. Qi4J supports the whole lifecycle of the entity via the UnitOfWork library.

Each Qi4J entity is implemented as an interface that extends the org.qi4j.api.entity.EntityComposite interface. The purpose of an entity is to guarantee the composability of

the data objects and roles. An entity behaves like the corresponding data object and like the role it plays as well. This is achieved by inheritance: each entity has to extend the corresponding data interface and each interface of the roles it is going to play. Listing 8 shows this.

```
public interface SellerEntity extends EntityComposite,
    //Data
    SellerData,
    //Roles
    ApproverRole,
    ContractorRole,
    InsurerRole
    { }
}
```

Listing 8. An entity.

Because of inheritance, each entity object behaves like a data object. Listing 9 shows how to work with an entity.

```
...
UnitOfWork uow = assembler.unitOfWorkFactory().newUnitOfWork();
...
SellerData seller;
seller = uow.newEntity(SellerEntity.class, "seller");
seller.title().set("");
seller.firstName().set("Chris");
seller.familyName().set("Froome");
seller.contact().set("froome@sky.com");
...
SellerEntity seller = uow.get(SellerEntity.class, "seller");
...
```

Listing 9. Working with an entity.

E. Qi4J Specifics to Support the DCI Approach and Context Execution

Qi4J supports a multilayered architecture with each layer consisting of several modules. Each module has its own set of entities. The interfaces specific to Qi4J (such as those for entities, composites, or services) need to be registered in the assembler object. The presented model example consist only of one module. The SingletonAssembler class offers methods to perform that registration. Listing 10 demonstrates this.

```
public class EntityContextsRolesAssembler extends SingletonAssembler {
    public void assemble(ModuleAssembly module) throws AssemblyException {
        // All the Qi4J entities being used
        module.addEntities(
            ...
            CarEntity.class,
            CarEquipmentEntity.class,
            ContractEntity.class,
            EmployeeEntity.class,
            InsuranceEntity.class,
            SaleEntity.class,
            ...
            );
        // All used DCI contexts and roles
        module.addTransients(
            ...
            InsureContext.class,
            ...
            InsuranceContractRole.class,
            InsurableRole.class,
            InsurerRole.class,
            ...
            );
        // Recommended services
        module.addServices(MemoryEntityStoreService.class,
            UuidIdentityGeneratorService.class);
    }
}
```

Listing 10. Implementation of SignletonAssembler.

A Qi4J application starts by instantiating SingletonAssembler, UnitOfWork, and TransientBuilderFactory. The management of the instances of context classes—called transients—is provided by TransientBuilderFactory. Listing 11 demonstrates this instantiation.

```
...
SingletonAssembler assembler = new EntityContextsRolesAssembler();
UnitOfWork uow = assembler.unitOfWorkFactory().newUnitOfWork();
TransientBuilderFactory tbf = assembler.module().transientBuilderFactory();
...
```

Listing 11. Instantiation of SignletonAssembler, UnitOfWork, and TransientBuilderFactory.

Listing 12 shows a triggering of the context.

```
...
CarEntity car = uow.get(CarEntity.class, "car");
CarInsuranceEntity insurance = uow.get(CarInsuranceEntity.class, "contract");
SellerEntity seller = uow.get(SellerEntity.class, "seller");
InsureContext context = tbf.newTransient(InsureContext.class);
context.initContext(insurance, seller, car);
context.executeContext();
...
```

Listing 12. InsureContext triggering.

IV. Conceptual Assessment

The DCI approach and its implementation bring many changes into object-oriented programming that affect the development process in different ways. Here we attempt to assess the most significant influences.

A. Roles Can Reduce Inheritance and Decrease Maintainability Effort

In common object-oriented programming the classes that share the same behavior are often part of the same inheritance hierarchy. These classes are extensions of a superclass: they share or extend its state and behavior. Class inheritance has many advantages—allows for polymorphism, supports reusability, and reduces source code duplication—and also disadvantages—higher maintainability and refactoring effort, close coupling between the classes in the inheritance hierarchy, and worse readability of inherited source code.

The DCI architecture is able to reduce the degree of inheritance use, decrease the maintainability effort, and increase the code quality. A high degree of inheritance use requires a high maintainability effort [6].

If a superclass includes the common behavior of its subclasses, it will reduce source code duplication. But DCI allows to put that behavior into a role. A role can be played by objects of different classes: the classes from the same inheritance hierarchy. This reduces the degree of inheritance use without source code duplication.

An inheritance hierarchy is often formed to exploit polymorphism. Often, objects of different classes can participate interchangeably in the same use case due to their structural similarity. In common object-oriented programming, those classes would have to be in the same inheritance hierarchy. The solution provided by DCI is to create a common role. The role will work with the common attributes and each object playing the current role will behave according to the values of its own attributes. That approach brings additional reduction of the degree of inheritance use.

B. Generic Roles Can Be Played by Objects of Inappropriate Classes

The concept of generic roles [7] along with simplicity [2], makes DCI be good at responding to change. Here we provided some evidence that DCI supports code maintainability, flexibility, and readability. However, the agility of DCI brings out one conceptual issue. Theoretically, a role can be played by an object of an inappropriate class. In dynamic programming languages, there is no mechanism to prohibit a data object from playing inappropriate roles. A developer can easily make the object play a role and cause wrong role behavior. Wrong identification of roles or their granularity can enhance that issue.

V. Implementation Assessment

In this section, benefits and consequences of implementing DCI in Qi4J are going to be discussed.

A. Loss of Direct Access to the Domain Model from Context and Generic Roles

In the end, the purpose of each software product is to create, read, update, or delete data and connections among them. In common object-oriented programming, use case methods have access to objects that store the data of a software system. Data objects are sent to methods as parameters. This establishes dependency between use cases and data object classes.

The system behavior is captured in the form of roles and their methods. But in the presented Qi4J example a role does not have access to data objects. Each role has only an indirect access to the object that plays it. However, the role usually needs to work with attributes of objects playing other roles. This forces the developer to create roles whose job is only to provide access to the attributes of the underlying object. Listing 13 shows this.

```
@Mixins(InsuranceContractRole.Mixin.class)
public interface InsuranceContractRole extends TransientComposite {
   ...
   /* The interface represents the data of the objects
      playing current role */
   public interface InsuranceData {
      /* The attributes from the data object */
      @Optional Property<Date> createDate();
      @Optional Property<Date> signDate();
      @Optional Property<InsurerRole> seller();
      @Optional Property<Boolean> isApproved();
      ...
   }
   abstract class Mixin implements InsuranceContractRole {
      @This
      InsuranceData data;
      public void setCreateDate(Date d) { data.createDate().set(d); }
      public void setSignDate(Date d) { data.createDate().set(d); }
      public void setInsurer(InsurerRole i) { data.seller().set(i); }
      public void setApproveFlag(Boolean b) { data.isApproved().set(b); }
      ...
```

Listing 13. An example of the role providing access to the attributes of the object that plays it.

No direct access to data (or domain) objects is not a disadvantage. Access via roles and their operations reduces dependency between the system state and behavior. This complies to DCI.

B. Entities Define Their Casting Rules

Java, as a statically typed language, needs the concept of methodless roles: identifiers. In Qi4J, which roles can be played by which data objects—or, in other words, which data object can be cast into which role—is determined directly by the entities. These casting rules are defined by the entity's extend relationships. Each entity extends one data interface and several roles. By this, the casting rules are kept in one place to support readability of source code and support capturing the intent of the programmer. Listing 14 demonstrates this.

```
public interface SellerEntity extends EntityComposite,
   //Data
   SellerData,
   //Roles
   ApproverRole,
   ContractorRole,
   InsurerRole
   { }
```

Listing 14. An example of an entity defining its casting rules.

C. Interfaces Instead of Classes as Templates for Objects

Usually, objects are instances of classes. The presented DCI implementation in Qi4J does not use classes as templates for objects. Classes are replaced with interfaces and this brings in many limitations. Class attributes cannot be used. Each method has to be defined in the nested class and its signature has to be duplicated in the interface body. Qi4J uses interfaces for other purposes. Java interfaces have usually only a couple of lines, but in Qi4J the number of lines per interface is significantly higher. This is another consequence we have to deal with.

In Qi4J, objects are instances of composite interfaces (e.g., EntityComposite and TransientCompisite) and have to be managed via the framework, namely its UnitOfWork and TransientBuilderFactory interfaces. This can be considered as another framework trade-off.

D. No Access Management of Data Class Attributes and Methods

Section III-A introduced the org.qi4j.api.property.Property interface. Its purpose is to enable storing the data attributes in a data interface. Each property has a public identifier and offers both set and get methods. That concept does not allow to modify the access to the attribute. This is not necessarily a negative consequence, but it should be mentioned.

E. No Direct Support of Polymorphism

According to DCI, a context algorithm consists of calls to role methods. A role method works with attributes of the data object playing the current role. Section III-C described that process. The whole use case behavior is implemented in roles and in their methods. Roles are generic and their methods are generic, too. That concept disables the standard usage of polymorphism, an important feature of common object-oriented programming.

However, there is a way to mimic polymorphism both at the level of the context and role. Each context can explicitly choose the right methods of the right role object. Polymorphism chooses the methods implicitly according to the type of the object. In DCI, the developer has that control.

Another way how to mimic polymorphism is to compare attributes of the @This object. If a domain object contains no attribute, Qi4J does not map it and it remains with the null value. Listing 15 illustrates this.

```
@Mixins(InsurableRole.Mixin.class)
public interface InsurableRole extends TransientComposite {
   public void insure(InsuranceContractRole insurance);
   /* The interface represents the data of the objects
      playing current role */
   interface Data {
      @Optional Property<Double> price();
      @Optional Property<Boolean> isNew();
   }
   abstract class Mixin implements InsurableRole {
      @This
      Data data;
      public void insure(InsuranceContractRole insurance) {
         if(data.isNew()!=null) {
            insurance.setAnnualPayment(data.price().get());
         }
         else {
            insurance.setAnnualPayment(new Double(0.0));
         }
      }
   }
}
```

Listing 15. The example of mimicking polymorphism by mapping the attributes.

VI. Related Work

Öberg presented how DCI can be implemented in Java focusing on how roles can be implemented and how objects can play them [8]. Öberg's role implementation in Qi4J is the basis for the DCI implementation presented in this paper. However, Öberg's DCI implementation does not consider the use of the data part to store the system state. In Öberg's implementation the data were stored in roles, but this does not match the idea of stateless roles in DCI [2].

Öberg also focuses on entities and their references rather than data interfaces. This leads to the use of the Association and ManyAssociations references instead of the Property reference.

Öberg described some implementation issues of common object-oriented programming, too. He's criticizing code duplication and large number of methods in classes.

VII. Conclusions

This paper reports an assessment of DCI via its Qi4J implementation and beyond based on an independent study of a small car dealer system development. Two most important conceptual findings are that roles can reduce inheritance and decrease maintainability and that generic roles can be played by objects of inappropriate classes. The findings specific to the Qi4J implementation include loss of the direct domain model access from the generic context roles, entities defining their casting rules, use of interfaces instead of classes as templates for objects, no access management of the data class attributes and methods, and no direct support of polymorphism.

Acknowledgment

The work reported here was supported by the Scientific Grant Agency of Slovak Republic (VEGA) under the grant No. VG 1/1221/12. This contribution/publication is also a partial result of the Research & Development Operational Programme for the project Research of Methods for Acquisition, Analysis and Personalized Conveying of Information and Knowledge, ITMS 26240220039, co-funded by the ERDF.

References

[1] B. Eckel, *Thinking in Java*, 4th ed. Prentice Hall, 2006.

[2] J. O. Coplien and G. Bjørnvig, *Lean Architecture: for Agile Software Development*. Wiley, 2010.

[3] E. Gamma, R. Johnson, J. Vlissides, and R. Helm, *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1994.

[4] The Apache Software Foundation, "What is Apache Zest?" http://zest.apache.org/.

[5] "Qi4j introduces composite oriented programming," http://www.infoq.com/news/2007/11/qi4j-intro.

[6] S. K. Dubey and A. Rana, "Assessment of maintainability metrics for object-oriented software system," *ACM SIGSOFT Software Engineering Notes*, vol. 36, pp. 1–7, 9 2011.

[7] T. Reenskaug and J. O. Coplien, "The DCI architecture: A new vision of object-oriented programming," http://www.artima.com/articles/dci˙vision.html, 3 2009.

[8] R. Öberg, "DCI in practice," https://vimeo.com/8235651.