The Influence of First-Class Relations on Coupling and Cohesion A Case Study

Rob van der Horst
University of Amsterdam
Master Software Engineering 2011
+31 6 28 45 92 55
rob@equate.nl

ABSTRACT
In object oriented programming languages associations between objects are often implemented through object field members. This paper discusses what the effects would be on coupling and cohesion if those associations were instead implemented with first-class constructs for relations or first-class relations.

First-class relations have positive effects on coupling and cohesion according to various papers [31, 27]. However, we found only one paper [32] in which the first-class relations have actually been used in real-world software. That paper however, did not report effects on coupling or cohesion. Therefore, we conducted an empirical research to find out if first-class relations indeed have effects on coupling and cohesion and which characteristics of first-class relations cause these effects. In this research we used an existing software program and replaced associations implemented through object field members with first-class relations.

From our findings we concluded that three characteristics of first-class relations indeed have a positive effect on coupling or cohesion. The characteristics that were found to have positive effects are (1) being one entity, (2) being the only relation construct and (3) support for relational constraints. Another important characteristic is that first-class relations have their own special notation. Although, we do consider this a vital characteristic to first-class relations, we could not find any results supporting this claim. In some cases a java object also sufficed as relation construct. We expected that roles would are also important for decoupling. However, this could not be established. Finally, we concluded that a first-class relation can not decouple a parent from its property if its role cannot be expressed in terms of its public methods. It does provide a dependency injection mechanism.

Keywords
Relations, Associations, Aspect-Oriented Programming

1. INTRODUCTION

1.1 Background and Context
The idea for this research arose from the marketplace business model. This business model revolves around the idea of a marketplace in which independent contractors can register for developing and testing small chunks of software. For decomposing the system-to-be into small chunks it is essential to apply the principle of information hiding [30]. A prerequisite for information hiding is that chunks are loosely coupled and have high cohesion [9]. Later on in the development process, these chunks of software will be assembled using connectors to compose the software system. This idea has been implemented in different fields of software engineering such as component-based development [7], service-oriented software engineering [7] and role-based development [23]. What separates our idea from those fields of software engineering is its granularity. We search for a more fine-grained decomposition. Fine-grained decomposition provides more ways to decompose the system into sub-systems. Consequently, during composition some components will end up together in the same subsystem while others end up in different subsystems. Intra-subsystem connectors differ from inter-subsystem connectors. The Java virtual machine for example provides intra-subsystem connectors for objects while SOAP can be used as inter-subsystem connector. The choice of connector must be made early on in development. If Java would support connectors that provided the same interface as the SOAP interface then the decision of dividing into subsystems could be made further downstream in the development process. The objects would be unaware of the type of connector or even be unaware that they would be connected to other objects. We believe that a first-class relation (FCR) can be such a connector.

1.2 Motivation
In January 2011, we stumbled upon a piece of Java code that had a circular dependency between two objects. Both objects maintained references to instances of each other. If an object instance was added or removed both references had to be updated. The objects had to know which method of the other object to use to prevent an infinite loop. In other words, the objects required knowledge of the internal implementation details of the other object. A first-class relation, if available to Java, could have easily implemented this relation and prevented the dependency on the internal details. It would do so by removing the cyclic dependency the two objects have with each other.

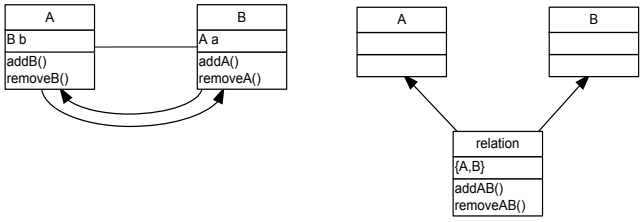

Figure 1 Basic idea of first-class relation.

Figure 1 shows that the coupling between the two objects has been replaced by coupling between the objects and the relation. The picture also shows that the reference to the other object is no longer required and thus the relation is no longer maintained

inside the objects themselves. The object can now focus on its core functions now that the burden of maintaining the relation has been removed. This leads to the expectation that cohesion or coherence should improve.

The relation construct provides the maintenance methods out-of-the-box. Maintenance methods are getters and setters that operate on the relation as a whole. This embodies the basic idea of the first-class relation.

Various sources have reported advantages of relations implemented as first-class relations as opposed to implemented through field members. A few of these are:

- higher cohesion/lower coupling [31, 27];
- guaranteed referential integrity under transactional control [28];
- relational consistency and integrity through support of operations and constraints on relationships as a whole [32, 28];
- the programs are easier to understand [26, 27];
- the programs are easier to modify [27];
- alignment with database is better [32];
- alignment with domain model is better [32];
- traceability improves [27].

Except for Rumbaugh [32] these sources did not provide real world empirical prove. They used small pieces of sample code or were logical argumentations. Therefore, we chose to conduct an empirical idiographic case study on one existing software program to validate these claims made about first-class relations.

1.3 Scope

This research considers only the effects of first-class relations on coupling and cohesion. It does not consider effects on other quality attributes such as maintainability. Therefore, the conclusions we drew were from the perspective of coupling and cohesion. Because we performed empirical research of an existing software program, only the characteristics that were relevant for the software program were investigated.

Although relations can exist between more than two objects [21, 32] we consider only binary relations.

The scope of our research is limited to:

- effects on coupling and cohesion metrics;
- characteristics of the first-class relation relevant to the software program;
- binary relations;
- static first-class relations (see paragraph 3.4 for more information);
- relations between object instances as opposed to relations involving object classes;
- relations that are associations i.e. not inheritance or implementation of interfaces.

1.4 Concepts of Coupling and Cohesion

The definition of coupling used by Chidamber and Kemerer [11] says “two objects are coupled if and only if at least one of them acts upon the other, *X* is said to act upon *Y* if the history of *Y* is affected by *X,* where history is defined as the chronologically ordered states that a thing traverses in time”. They continue to conclude that by that definition any action performed by X on Y or by Y on X constitutes coupling. These actions can be method calls or access to field members.

The definition of cohesion used by Chidamber and Kemerer [11] is based on the principle of similarity of methods within an object. This similarity is based on the intersection that methods have through field members. The degree of similarity is viewed as the object class cohesiveness. An object class is considered cohesive if it has different methods performing different operations on the same set of field members. One such set is considered as a function area by Hitz and Montazeri [19].

1.5 Characteristics of First-Class Relations

For a relation construct to be considered first-class it must exhibit a number of characteristics. These characteristics are:

1. it has its own special notation;
2. it is constructed as one entity;
3. no other construct defines a relation between the same participants in parallel with a first-class relation;
4. it represents the same concept in run-time;
5. operations can be performed on relationships;
6. constraints can be applied to relationships.

The first characteristic is that a first-class relation must have its own special notation. Rumbaugh [32] argues that a special notation aids in the visualization of relations in the program code and the communication with other stakeholders. Communication approves because without the special notation it is not possible to describe relationships without also describing the actual implementation of relations. He compares the special notation with the notation of relations in the modeling language Object Modeling Technique (OMT), the predecessor of UML. In OMT relations are represented as lines between object classes with the name of the relation written above the line.

The second characteristic has been derived from Jacksons definition for a relation [21]: *A relation is a structure that relates atoms. It consists of a set of tuples, each tuple being a sequence of atoms. You can think of a relation as a table, in which every entry represents an atom. The order of the columns matters, but not the order of the rows. Each row must have an entry in every column.* For object-oriented programs the term “atom” must be substituted by the term “object instance”. It is the structure i.e. the table, which makes the relation to be one entity. In OMT, that one entity is the line between the object classes.

The third characteristic is that no other relation constructs are possible between the same participants. Different relation constructs may be used but between the same participants only one construct may be used at a certain moment in time. If more than one construct is used at one time then it is not possible to guarantee relational integrity between participants.

The fourth characteristic is that the relation construct must remain in tact at run-time. Run-time support is required for a number of reasons. These reasons include dynamic lookup of participants, debugging and serialization. The run-time concept of a relation also aids in thinking in terms of relations and hence facilitates the communication among stakeholders of the software program.

The fifth characteristic is that it must be possible to apply operations to relationships. We recognize two types of operations: structure operations and behavior operations. Structure operations allow us to change and query the structure of a relationship. Behavior operations consist of the behavior of the participants in the relationship.

The sixth characteristic is that it must be possible to apply constraints to all participants in a relationship. Constraints are used by Rumbaugh in [33] to support propagation of operations. The operations act on the participants across relationships. Also, Rumbaugh and Jackson [21] both recognize the necessity for constraining the cardinality of the participants in a relationship.

1.6 Research Question

The absence of case studies on real world software in literature on first-class relations creates doubt about the proclaimed advantages. Although some studies have been done on small pieces of code these are not expected to be sufficient to uncover the intricacies typical of applying theories to real world software. This could very well be the reason that first-class relations up till now have not been adopted by mainstream programming languages.

In this paper we take a first step in uncovering these intricacies. We do so by focusing on coupling and cohesion because this most clearly relates to our requirement for decomposing the software program. Therefore, central in this paper is the following research question:

Which characteristics contribute to a successful implementation of first-class relations with respect to coupling and cohesion, and which characteristics contribute to failure?

In support of our main research question we have formulated a number of sub-questions.

1. *Which elements constitute a first-class relation?*

We found different elements of first-class relations in the papers but these have not always been implemented in the models that followed. For example, Balzer et al mentioned the existence of invariants in [3, 4] and Rumbaugh mentioned propagation of operations in [33].

2. *How should first-class relations be applied to object-oriented code?*

For traceability we describe how we will apply first-class relations to the software program. We have to identify the relations and their characteristics from the source code because we did not have access to the design of the software program.

3. *Which definitions of coupling and cohesion are appropriate for first-class relations?*

The definitions of coupling and cohesion we choose can have a profound impact on our results. The definitions have to be applicable to the programming language of the software program. Furthermore, earlier versions of coupling and cohesion have been questioned in papers that followed.

1.7 Related Work

The relation has been around as a first-class citizen in modeling languages such as UML [22] and entity relationship diagrams [10]. However, the relation never got the same support in mainstream object-oriented programming languages such as Java, C++ or Smalltalk. Back in 1987, James Rumbaugh suggested an implementation of relations as first-class citizens in the object-oriented language Data Structure Manager (DSM) [32]. After Rumbaugh's implementation of DSM other programming languages followed, such as RelJ [6]. RelJ resembled DSM in that it too supported relations as part of an object-oriented language. However, RelJ never got implemented.

It wasn't until 1995 that the work of Rumbaugh was followed up. Noble and Grundy published their paper on relationship in object-oriented development [27]. They implemented relationships with relationship objects. Using these relationship object Nobles and Grundy observed advantages such lower coupling, higher cohesion, smaller programs, easier to understand code and better alignment between design and program code. Noble reported the same advantages in 1997 in his paper on basic relationship patterns. These patterns described how objects could be used to model relationships.

More people joined the research of first-class relations from 2002. The research focused on implementations of dedicated relational languages (RelJ and Rumer), implementations of add-ons (RAL, CaesarJ and Noiai) or theory.

In the fields of add-ons, Hannemann and Kiczales published their paper in 2002 on their implementation of design patterns in AspectJ [18]. Although, this paper was not specifically aimed at relations their implementation did show how relations could be implemented using aspect-oriented programming (AOP). The paper of Noble and Pearce in 2006 [31] presented the Relationship Aspect Library (RAL). RAL supported two types of relations: static relations and dynamic relations. Static relations were implemented with AspectJ while dynamic relations were implemented with Java. In 2007, Østerbye presented his implementation of a library for association relationships in C# called NOIAI ("No object is an island") [29]. In 2008, Noble, Pearce and Nelson presented a three-level model of relationships [24, 25] in which relations form the third tier after the object tier and association tier. The relation tier adds roles and relationship constraints. They did not however, present an implementation of their model like Noble and Pearce did earlier with RAL.

In the field of relational languages, Bierman and Wren presented RelJ in 2005 [6]. RelJ was a language that supported a subset of the features of Java and added support for relationships. RelJ was never implemented however. In 2006, Aracic et al introduced the concepts of their language CaesarJ in [1]. CaesarJ combines object-oriented programming with aspect-oriented programming in one language. In 2007, Balzer et al presented a relational model [4]. This model proposed member interposition for object members specific to a relation. It also proposed relationship invariants to constraint the objects in a relation.

Our concept of the first-class relation is not confined to development time but extends into run-time as well. The first-class relation construct must be interchangeable with other technologies. In other areas of software engineering, technologies have been developed that show similarities with relation constructs. One such area is that of role-based development. Lee and Bae [23] proposed an implementation of a role model using Javassist to modify code at source level. The resulting code contained classes similar in functionality to that of relations. In the area of service oriented software engineering relations are perhaps best compared with the service composability design principle described by Thomas Erl [13]. Applying this design principle depends on the implementation of several design patterns. CORBA defines a relationship by the set of roles the entities have [28]. The Relationship Service implements the relationships. It also implements roles. CORBA objects represent the roles in a relationship.

1.8 Organization of this Paper

The next section describes the research method we used. Section 3 describes the explorative part of our research. Section 4 describes the application part of our research. Section 5 presents an overview of the results for each element of a first-class relation. Section 6 presents the analysis of the results and refers back to sections 4 and 5. The analysis is presented from a viewpoint of each of the characteristics of a first-class relation. Section 7 sums up the conclusions.

2. RESEARCH METHOD

We split our research into two phases: exploration and application. The objective of the exploration phase is to gain an understanding of the matter, seek answers to our sub questions, select a software program to refactor and set up our development environment. The exploration will be done iteratively: (1) learn - define a first-class relation based on work of others (2) apply - implement in an existing software program and (3) refine - analyze results and adapt model of first-class relation.

The objective of the application phase is to measure and analyze the effects of first-class relations on coupling and cohesion. We identify relations in the baseline code, match it to our first-class relation model, and then apply each element of the first-class relation. After the application of each element we validate the functions of the software program and measure the effects the application has on the participants in the relation and the rest of the software program. The effects are analyzed and explained. See Figure 2 for an overview of the application research method. Eventually, all results are combined to form our conclusion.

Due to timing constraints we limit the research to include only part of the relations found in the software program. To make a fair comparison to first-class relations we try to avoid code that is tightly coupled or has low cohesion due to bad programming. Therefore, we select relations that follow established programming practices. One such practice is the use of design patterns. To make the results more predictable we select only design patterns, which are known to have a certain effect on coupling and cohesion [8, 14, 15].

We also select a number of straightforward relations i.e. relations between an object and its property. We refer to those relations as property relations. Design patterns often introduce additional elements such as interfaces and abstract classes. These additional elements may influence the outcome of the refactoring because they are not considered part of the business domain and therefore may become redundant in the process of refactoring to first-class relations. With property relations we intentionally refrain from factoring out the parent itself into a relation. The property relations are taken from an arbitrary test case by following its call tree.

For each relation, we analyze its characteristics. These characteristics are translated to the elements of our model of a first-class relation. Paragraph 3.3 describes this model. The elements include cardinality, how client objects use the relation, which methods constitute the behavior of the relation and what type of behavior it concerns.

The first-class relation is then implemented, one element at a time. Every element is then switched on and off individually. There are some dependencies however between the elements. For example cardinality is part of the structure of the first-class relation and cannot be switch on and off independent of structure. The implementation follows the guidelines described in paragraph 3.4.

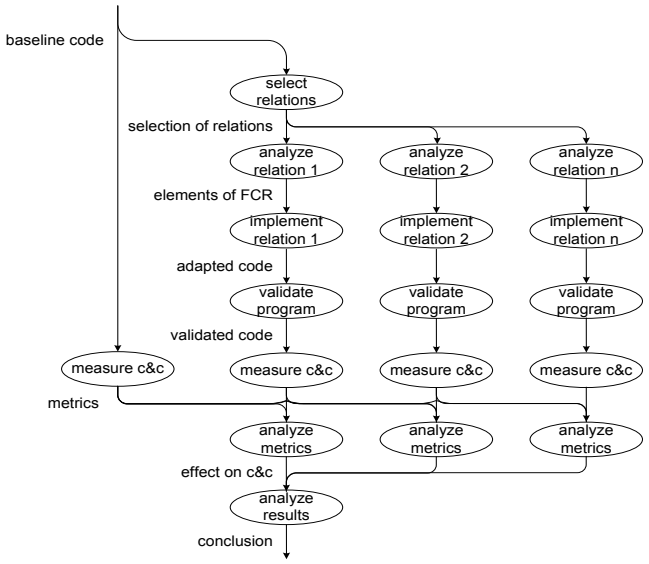

Figure 2 Research Method in Application Phase

We validate the resulting program code with regression tests. To validate the implementation of the first-class relations we follow the results presented by Kiczales and Hannemann in [18] and let experts in the field of first-class relations review the code. After each application step, the resulting program code is saved to a source code version system.

After each finished application step we record the metrics. We start with recording the baseline. Then for each relation, every time an element is switched on or off, the resulting values of the metrics are recorded in a database. For coupling we record the values between two related objects but also values in relation with all objects in the software program. In addition to the metrics themselves, we record which objects were changed and what roles they played in the relation. For the design patterns, the naming of the roles are taken from Gamma et al [14]. For the relations in the test case, the role names are restricted to client, parent and property. The application steps are matched with the revisions in the source code version system.

All information recorded during the application phase is coded. The set of codes is: name of the relation, name of the participants, coupling or cohesion and element of first-class relation. The coding helps us with identifying patterns during the analysis.

The validity of the measurements is ascertained in two ways: (1) the measurements are performed with tools that have proven themselves in other studies and (2) the internal code of the metric tools has been analyzed during the exploration phase and compared to their definitions.

The metrics are not used for quantitative analysis. Instead, we use the metrics to support us in understanding the effects of first-class relations and to ascertain that we do not miss unexpected effects. We compare the resulting metrics to the baseline and to the previous step. We do this for all objects that were changed and for the clients.

Finally, during analysis we look at the elements of first-class relations and how they translate to the characteristics of first-class relations. To support us with the analysis, we use our notes and where necessary do the measurements again with tracing enabled for the metric tools so we can examine intermediate results such as collections of objects.

3. EXPLORATION PHASE RESEARCH

In this section we describe the explorative part of our research. As part of the exploration we present our development environment and answer the sub-questions. The answers are given in the form of the metrics used for coupling and cohesion, a model of first-class relations and a procedure for identifying and applying first-class relations to existing source code.

We use the term object class to indicate the type of an object. We use the term object instance to indicate an instance of an object. Throughout this paper we use the term object if it is irrelevant to the discussion whether it concerns an object class or object instance. We use the term relation class to indicate the type of relation. We use the terms relationship to indicate an instance of a relation. A relationship is the set of tuples of object instances in a relation that are linked together, or group of interacting object instances [4]. See Nelson et al [25] for a more precise description. Throughout this document we use the term relation if it is irrelevant whether it concerns a relation class or relationship.

3.1 Definition of Coupling and Cohesion

Of the advantages mentioned, coupling and cohesion are directly related to decomposing systems. How these quality attributes change is indicative for the value of first-class relation with respect to decomposition. This paragraph describes which metrics are appropriate for our research.

3.1.1 Coupling

Coupling metrics appropriate for object-oriented systems can be divided into three categories [12]: (1) inheritance coupling, (2) component coupling and (3) interaction coupling. The coupling metrics we consider were introduced by Chidamber and Kemerer in [11], or derived from those metrics. The metrics included CBO (coupling between object classes), WMC (weighted methods per class) and RFC (response set for a class).

Inheritance coupling pertains to class inheritance, interfaces and abstract classes. These constructs exist only during design-time. Two object classes are inheritance coupled if one is a direct or indirect sub-class of the other. During run-time super-classes and interfaces are no longer recognizable as a separate construct. For this type of coupling, Java already provides first-class constructs. This type of coupling can influence decomposition but we could not find any metrics for inheritance coupling.

Component coupling pertains to relations between objects that may exist at some point in the lifetime of an object. Component coupling is typically implemented in Java code using field members and method parameters. This type of coupling also exists during run-time.

Of the component coupling metrics, CBO, Fan-In (afferent coupling) and Fan-Out (efferent-coupling) are very useful. They pertain to static relations between objects. In Java, a static relation is created using object field members. Fan-In of an object is the number of static references from other objects to this object. Fan-Out of an object is the number of static references to other objects. CBO is the number of static references between two objects or the union of Fan-In and Fan-Out.

Interaction coupling pertains to interactions between objects. Interaction coupling is implemented in Java through method calls and attribute access. Two interaction coupling metrics are MPC (message passing coupling) and RFC.

Table 1 Appropriate Coupling Metrics

Metric	Formula	Category
CBO	The size of the intersection of the set of object classes referenced by this class with the set of object classes that reference this class.	Component
Fan-In	The number of classes that reference this class.	Component
Fan-Out	The number of classes referenced by this class.	Component
MPC	The number of calls made to methods in other classes.	Interaction

Table 2 Appropriate Cohesion Metrics

Metric	Formula	Category
ILCOM	Number of connected components	Class
TCC	Number of pairs of directly connected methods divided by number of pairs of methods	Class
LCC	As TCC but includes indirectly connection methods as well.	Class

Both metrics are useful to our research but only one is required because they are closely related. Message related coupling pertains to the messages sent between classes i.e. the methods called. The metrics tool JHawk [17] defines MPC as the number of methods from another class that are called. JHawk defines RFC as the number of methods in a class plus the value of MPC.

Table 1 shows the coupling metrics appropriate for our research.

3.1.2 Cohesion

Cohesion metrics appropriate for object-oriented systems can be divided into three categories [12]: (1) method cohesion, (2) class cohesion and (3) inheritance cohesion.

Method cohesion describes the binding of the elements defined within the same method. Individual methods are not considered in our research. Objects are the smallest grained elements we consider.

Class cohesion describes the binding of the elements defined within the same object. Inheritance cohesion is the same as class cohesion except that it also takes inheritance into account. Since inheritance is a measure for reuse the functionality in super-classes should be taken into account as if it were implemented without inheritance.

Class and inheritance cohesion metrics are roughly divided into two categories: those measuring lack of cohesion in methods (LCOM) and those measuring cohesion itself. The former has many variants. LCOM was introduced by Chidamber and Kemerer [11] but was considered to be counter-intuitive by Hitz and Montazeri [19] and others. In response Hitz and Montazeri introduced an improved version of LCOM (ILCOM) using graph theory. ILCOM measures the number of disjoint function areas in an object. They referred to it as the number of connected components of a graph.

Metrics to measure class cohesion itself are tight class cohesion (TCC) and loose class cohesion (LCC). These metrics were introduced by Bieman and Kang in [5]. They are similar to ILCOM. TCC is the ratio between pairs of connected methods

and all pairs of methods in a class. LCC measures both directly connected methods and indirectly connected methods while TCC measures only directly connected methods. Valid values for TCC lie between 0 and 1. Table 2 shows the cohesion metrics appropriate for our research.

3.2 Development Environment

The software program we chose for our research is JUnit 4.9, which is maintained by Kent Beck and Erich Gamma. JUnit is a unit-testing framework for Java programs. It has a graphical and a text user interface for viewing results. It is integrated with several programming environments such as Eclipse.

JUnit is a relatively small program making it easy to comprehend. It also contains design patterns for which the effects on coupling and cohesion are known. JUnit comes included with a suite of test cases, which will help us doing the regression tests.

JUnit is split into two parts: one supporting version 3.8 clients and another supporting version 4 clients and providing classes for backward compatibility. Figure 24 gives an overview of the classes we encountered during our study. It distinguishes version 3.8 classes from version 4 classes. The relations that have been refactored are shown with dotted lines. Notice that the figure does not intent to be a complete overview of JUnit.

We chose RAL [31] for the implementation of the first-class relations. The number of programming languages with support for relations we known of is limited to RelJ, Rumer, Noiai, CaesarJ and RAL. Neither RelJ nor Rumer has an implementation and consequently no software program has been developed. Noiai is an add-on for C# and cannot be used for JUnit. CaesarJ seems a viable option. It is a Java-based collaboration-oriented programming language. It extends AspectJ. However, the implementation of CaesarJ is complex compared to that of RAL. This will make it difficult to extend if required. RAL is also Java-based and uses AspectJ. RAL does have some issues such as missing support for polymorphic pair types, missing support for n-ary relations and limited support for multiple instantiations of relationships [25]. However, none of those issues will hinder us in using it.

For detection of design patterns we chose the Design Pattern Detection tool from CSSE Laboratory. It's an easy to use stand-alone tool that operates against a directory of Java sources. And as a bonus it has been tested with JUnit 3.7. The tool has been documented in [35] and is available for download together with the results for JUnit 3.7 from the web site of CSSE Laboratory [34].

We did not find a tool that provided all metrics we required. Therefore, we chose two tools: JHawk 5 Professional Edition by Virtual Machinery for coupling and VizzMaintenance 2.0 of ARiSA AB for cohesion. JHawk comes as a stand-alone tool as well as an eclipse-plugin. It is available for download from the company's web site [36]. VizzMaintenance comes as an eclipse-plugin. It is available for download from the company's web site [2].

Both metrics tools perform their measurements on the Java source code. They do not take aspects of AspectJ source code into account. JHawk allows one to select the objects that must be measured during analysis. This makes it possible to exclude or include the object classes in the JDK at will. VizzMaintenance does not support LCC and for TCC it counts only the public methods.

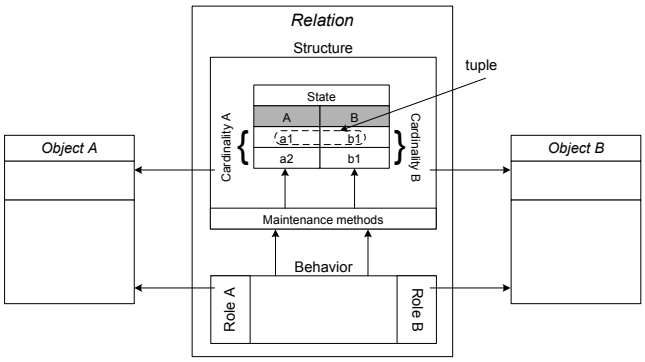

Figure 3 Theoretical Model of a Relation

3.3 Model of First-Class Relation

This paragraph presents our model of a first-class relation (Figure 3). It identifies the elements of the model and thus answers sub-question 1. The main two elements we distinguish are structure and behavior.

Structure itself is divided into three sub-elements: (1) state, (2) maintenance methods and (3) cardinality. Structure is the container for the objects instances that participate in the relationship. The object instances that are part of the relationship at a given time define the state of the relationship. Access to the structure is provided by maintenance methods. Cardinality limits the number of object instances on each side of the relationship.

The second element is behavior. There are two types of behavior: active and reactive. Reactive behavior is triggered by object instances that take part in the relation causing other object instances in the relationships to react. This kind of behavior is typically implemented with the Observer design pattern. Active behavior is initiated by a third object; a client object.

Roles are also part of behavior. Roles describe the public interface of the objects that is used by the relation when an object participates in a relationship.

3.3.1 Structure

State is the collection of tuples of object instances that make up the relationship. An object in a relation is referred to as a participant, Balzer et al [3, 4]. The binary relation has two participants that are each other's partner. Together two partner instances are referred to as a tuple. The set of tuples can be viewed as a table [21, 32]. State changes when tuples are added or deleted. State can also be queried.

Maintenance methods provide access to state. State is changed when maintenance methods add or delete tuples. State is queried when maintenance methods get tuples, participant instances or the size of state. Typical maintenance methods are add(), remove(), size() and get(). Additional maintenance methods exist to facilitate use of lists of participants. Maintenance methods are used by clients of the relation and by the behavior methods of the relation. The name of maintenance methods was taken from [18] where Kiczales and Hannemann describe the implementation of design patterns with aspects.

Cardinality constrains the number of tuples in a relationship. An n-to-m relation has a maximum of n*m tuples. Cardinality also constrains the number of object instances in a relationship. For one participant n distinct instances can be in the relationship and for its partner m distinct instances can be in the relationship.

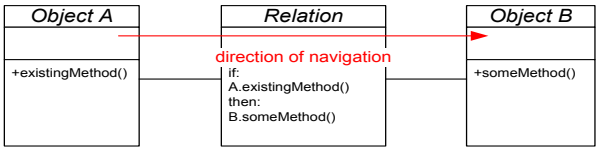

Figure 4 Reactive Behavior

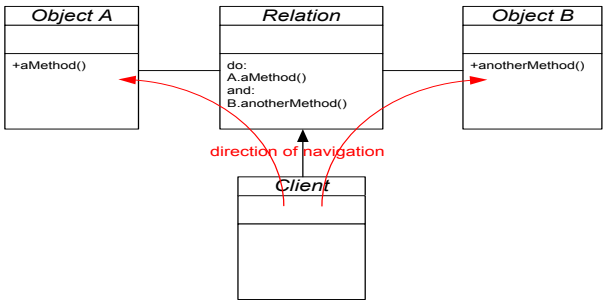

Figure 5 Active Behavior

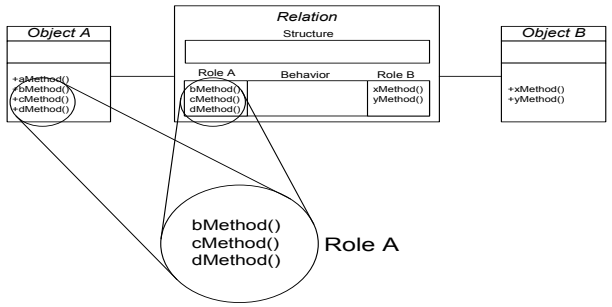

Figure 6 Roles Dictate Which Objects can Participate

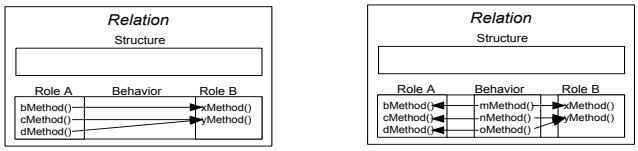

Figure 7 Reactive Behavior (l) versus Active Behavior (r)

3.3.2 Behavior

Balzer et al in [3] mention that relations contain the behavior that participants have in common. We define behavior as the interaction between the participants of a relation. Interaction is initiated through behavior methods. The behavior methods use only the public methods of the participants. This may conflict with the idea of Pearce and Noble in [31] that objects can behave differently when they are participating in a relation. However, we use only the public methods of the participants to honor the principle of information hiding.

We differentiate between two types of behavior just like Kiczales and Hannemann do in [18]. These types are reactive and active. They used reactive behavior for the Observer pattern, and active behavior for the other patterns.

Behavior is reactive (Figure 4) when it is triggered by one participant and then performs an action on its partner (to keep it in sync). Reactive behavior cannot be accessed by clients. Reactive behavior is useful for attaching functionality to an existing software program.

Behavior is active (Figure 5) if a third object initiates the interaction between the two participants. The third object, the client, operates directly on the behavior in the relation. It does not have to be aware of the object instances in the relation though.

3.3.3 Roles

Roles define the behavior that is expected of the participants. Nelson in [24] uses roles to add state and behavior to participants in the context of a relation. He does so by adding fields and methods to the participant. Again we do not want to break the principle of information hiding and therefore use only the public methods participants already have. An object may only participate in a relation if has the methods defined by the role.

A participant has formalized a role if the methods required for the role are part of a separate construct. In Java, this construct is the interface. If a participant has not formalized its roles then it is said to be oblivious of what it is used for.

Figure 6 shows that role A is defined by methods bMethod(), cMethod() and dMethod(). If object A implements these methods it can play role A and thus participate in the relation.

Figure 7 shows that behavior of the relation is defined by the interaction between the methods in the roles. With reactive behavior, any time a method in role A is executed, methods in role B get executed. With active behavior, the methods of roles A and B are executed together, part of the behavior methods, which in turn are triggered by a client.

3.3.4 Clients

Clients are objects that operate on a relation. Clients can change or query the state of a relationship. Clients can also execute the behavior methods of a relationship if it concerns active behavior. Relationships can have more than one client

3.4 Implementation in RAL and AspectJ

RAL is a library of relation aspects and relation objects. The relation aspects support static relationships while the relation objects support dynamic relationships. Relationships are static if they always exist during run time. Relationships are dynamic if they can be created and destroyed during run-time.

For the implementation of first-class relations we used static relations only because aspects can be implemented invisible to the metric tools. Invisibility is important because we want to prevent that aspects are regarded as objects.

The static relations of RAL provide structure, including state, maintenance methods and cardinality. They do not however, provide behavior. To implement behavior we had to extend the static relations with aspect code. To guide us with this we used the description of Kiczales and Hannemann in [18].

The static relations are used to create relations between two objects[1]. In RAL, static relations are available only with many-to-many cardinality. RAL provides two aspects for this: `StaticRel` and `SimpleStaticRel`. The former differs from the latter in that it provides support for pairs of objects where the pair can be defined as an object. Behavior is added by extending or adapting `SimpleStaticRel`.

[1] According to [31] unary relations are also supported with the `SimpleStaticReflexiveRel` aspect, but this aspect was not present in the library.

```
aspect Attends extends
   SimpleStaticRel<Student, Course> {
   ...
   void grade(Student s, Course c) {...}
}
```

Below is illustrated how access to the Attends relation from the object source code may look.

```
Attends.aspectOf().grade(aStudent,aCourse);
```

3.4.1 Structure

SimpleStaticRel is an abstract aspect, which forms the basis of structure. We adapt SimpleStaticRel to add support for one-to-one relations (SimpleStaticOneToOneRel) and one-to-many relations (SimpleStaticOneToManyRel). The add() function ascertains that the cardinality does not exceed one and it provides methods for returning a single object rather than a Set of objects. Note that RAL does support dynamic relations with one-to-one and one-to-many cardinality.

SimpleStaticRel and SimpleStaticOneToManyRel can contain only one relationship while SimpleStaticOneToOneRel can contain multiple relationships. Concrete static relations are defined by extending an abstract relation and adding behavior.

State is implemented in RAL using java.util.HashSet. This implementation does not guarantee preservation of the order of the entries in the relation. We added support for this too using java.util.LinkedHashSet. Note however, that this conflicts with the definition of a relation by Jackson [21].

RAL provides basic maintenance methods out-of-the-box. We have added more sophisticated methods for working with collections and making the roles visible e.g. addObserver().

3.4.2 Behavior

We added support for behavior the same way as Pearce and Noble do in [31]. They add methods to the static relation aspects. RAL however, does not explicitly support behavior.

We use pointcut/advice pairs to implement reactive behavior. Pointcut/advice pairs allow us to react to events that occur in one of the participants. Reactive behavior supports propagation of operations as mentioned by Rumbaugh in [33].

Active behavior is added as regular methods to an aspect. Active behavior can operate on the relation as a whole. The participants are kept in sync with each other because they are manipulated in the same operation.

3.4.3 Roles

In RAL, the name of the role is attached to the static relation through the generic type parameters. In the first example the role of Person has not been formalized through a Java interface.

```
aspect Attends extends  SimpleStaticRel<Person,Course>
```

In the next example, Person implements the role of Student using the interface Student. The Attends relation uses the methods in the interface Student to access to Person object (or other objects that implement Student).

```
aspect Attends extends SimpleStaticRel<Student,
Course> {
   ...
   void grade(Student s, Course c) {
      String id = s.getStudentId();
      ...
   }
}
```

```
// interface Student formalizes role
interface Student {
   public String getStudentId();
   public void addKnowledge(Knowledge k);
}

// class Person declares that it can play role Student
class Person implements Student {...}
```

3.4.4 Participant Methods

Following Østerbye, we use participant methods to make code more readable and provide a convenient way of accessing relationships. They also prevent that first-class relation constructs are regarded as objects by the metrics tools. Østerbye already recognizes the convenience to access relationships through its roles rather than directly [29] and therefore adds support for this to Noiai.

AspectJ provides a mechanism for attaching methods or code in general to objects, called inter-type declarations (ITD). In our case we will attach methods of the first-class relation to participants or their role. We use participant methods for our maintenance methods but also for our behavior methods.

The code example below shows the relation ResultObserver. First, a relationship is accessed without using a participant method and then with the use of a participant method.

```
ResultObserver.aspectOf().add(fResult,this);

fResult.addObserver(this);
```

Without the participant method the metric would count aspectOf() as a method call.

3.5 Identifying Relations in Existing Code

In this paragraph we describe how we identify relations in existing code and how they are mapped to first-class relations. We identify relations in two ways. First, we use the Design Pattern Detection tool to identify the design patterns and thereby the relations involved. Then we identify relations manually from a test case.

The relations we identify in the test case are either of the type aggregation or composition. Other types exist too as Guéhéneuc and Albin-Amiot showed in [16]. They defined that a binary relationship in its most general form exists when an object instance can send messages to another object instance. They called such a relationship an association. They defined aggregation and composition relations as a more restricted association. Aggregation and composition relations can only be established through field members.

It concerns an aggregation relation when the property is passed on to the parent through its constructor. It concerns a composition relation when the property is created in the constructor. Either way, the parent cannot exist without the property.

To replace the relation between a field member and its parent with a first-class relation we remove the getter, setter and size methods as well as constructors. First-class relations use the maintenance methods instead.

We must derive cardinality from the code as well because we do not have access to the design of the software program. The lower bound of the field member is at least one if the field member is created by the constructor. If not, it is at least zero.

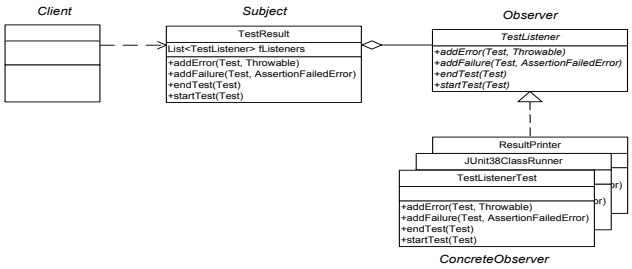

Figure 8 Instance of Observer Pattern in JUnit

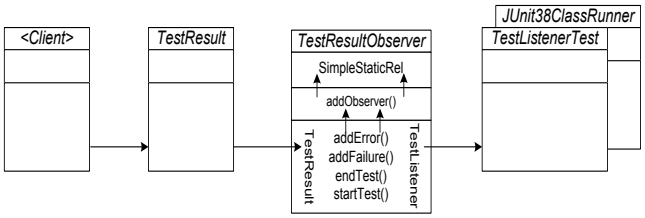

Figure 9 Observer Pattern Implemented as First-Class Relation

The upper bound is at most "many" if a Collection class or derived class was used. If not, it is at most one.

The interfaces that Java objects implement indicate the roles they can play. One of the interfaces may be used for the relation.

The choice between reactive behavior and active behavior depends on the client. If the client uses one participant as argument when invoking the method of the other participant we will use active behavior. If not, the relation is a candidate for reactive behavior.

All refactoring that takes place will pertain to the first-class relations only. We will not move other parts of code to other objects, or remove code that is not used in the regression tests.

4. APPLICATION PHASE RESEARCH

In this section we describe the application phase of our research. During this phase we select the relations that will be refactored. For each relation we identify which elements it is made up of, refactor it to a first-class relation, validate the code, measure the coupling and cohesion and then analyze the results.

The application phase is divided into two sub-phases. In the first sub-phase we refactor relations that were taken from design patterns. In second sub-phase we refactor relations that were taken from the test case. Each sub-phase is described in a separate paragraph. Both paragraphs start with the step in which the relations are selected and then discuss each relation per sub-paragraph.

We analyze all relations, mapping them to the elements of a first-class relation. These elements are structure, behavior, roles and participant methods. Per element we describe how the code is refactored and what the effects are on coupling and cohesion. To validate the adapted code we run the test suites that accompany the source code of JUnit. Log statements are added to the adapted code and that of the first-class relations to ascertain that the adapted code is actually used.

The test suites we use for testing are:

- junit.samples.AllTests.java
- junit.samples.SimpleTest.java
- junit.samples.money.MoneyTest.java
- junit.tests.AllTests.java
- junit.tests.extensions.AllTests.java
- junit.tests.framework.AllTests.java
- junit.tests.runner.AllTest.java
- org.junit.samples.ListTest.java

4.1 Design Patterns

Many sources report the advantages of design patterns with respect to coupling and cohesion. Many of them however, refer to Gamma et al [14]. Gamma et al mention that the following design patterns are good for lower coupling: Abstract Factory, Bridge, Chain of Responsibility, Command, Façade, Mediator and Observer. For cohesion much less information is available for specific design patterns.

Although, many sources report that increased cohesion is an advantage of design patterns, we found that the Composite pattern is bad for cohesion [15]. For the other design patterns, we did not find any reports that showed what the effects were on cohesion that could be attributed to one specific design pattern.

Of the aforementioned design patterns, the Design Pattern Detection tool found one instance of the Observer, one instance of the Composite pattern and 23 instances of the Command pattern. Of those 23 instances 18 instances were not deemed usable because they consisted of (anonymous) inner classes. The effect of inner classes had not been considered during the exploration phase nor had they been described by the metric documentation.

From the remaining 5 instances one was considered to be an instance of the Adapter pattern, based on the names of the classes. The names revealed the intent of the pattern. An instance of the Command pattern can also be an implementation of the Adapter pattern since the difference between both patterns is mostly intent [35]. Of the remaining 4 instances one was selected arbitrarily.

4.1.1 Observer Pattern

The Observer pattern "defines a dependency between objects so that when one object changes state, all its dependents are notified and updated automatically" [14]. The pattern minimizes the coupling between the Subject and the Observer [14]. The pattern decreases coupling by using an interface that all Observers must implement. Removing this interface should lead to lower coupling between Subject and Observer.

Figure 8 shows the UML class diagram of the implementation in JUnit of the Observer pattern without the first-class relation. The Observer pattern implements the relation between `TestResult` and interface `TestListener`.

Figure 9 shows the implementation of the Observer pattern with the first-class relation. `TestResult` no longer depends on the `TestListeners` and vice versa. Note that the first-class relation is unidirectional. See Figure 25 for the source code.

Structure

In the start situation, the structure consists of the `List` objects that hold the `TestListeners`. Access to the structure consists of methods to add and remove listeners. The `TestListeners` can be listening to multiple `TestResults` and so the cardinality is many-to-many.

To apply structure, we use the `SimpleStaticRel` aspect. This aspect supports many-to-many cardinality and provides the

maintenance methods by default. The getters, setters and List objects are therefore removed from TestResult.

After refactoring structure, the MPC (message passing coupling) value drops in TestResult. The drop is caused by the removal of the calls to java.util.Collection and related classes. MPC does not drop as much as possible because other relations exist that also use java.util.Collection. Part of the code removed from TestResult is shown below.

```
protected List<TestListener> fListeners;
public synchronized void
        addListener(TestListener listener) {
                fListeners.add(listener);
}
```

Support for structure does not influence component coupling (CBO, Fan-In or Fan-Out) or cohesion because we cannot completely remove the references to TestListener. Only MPC drops but that is not due the lower coupling between TestResult and the TestListeners but because we removed method calls to utility classes.

Reactive behavior

In the start situation, the Observer pattern has behavior that can be characterized as reactive because the client needs access only to the Subject to trigger the notify methods. TestResult is the Subject and it has four notify methods to which TestListeners must subscribe. The TestListeners do not query state of TestResult while Gamma et al [14] do describe the pattern that way. Instead the four notify methods pass arguments from TestResult to TestListeners.

To implement the behavior with RAL/AspectJ we moved the code in the notify methods to the first-class relation. Before the move, the notify method addError() in TestResult looked like this:

```
public synchronized void
addError(Test test, Throwable t) {
   fErrors.add(new TestFailure(test, t));
   for (TestListener each : cloneListeners())
      each.addError(test, t);
}
```

Afterwards, with the loop over the TestListeners removed, the method looks as follows:

```
public synchronized void
addError(Test test, Throwable t) {
   fErrors.add(new TestFailure(test, t));
}
```

The code below shows the implementation of the addError() notify method in the first-class relation. We used pointcuts/advice to attach the behavior to the notify method of TestResult.

```
protected pointcut subjectAddError(TestResult s, Test
test, Throwable t) :
(call (void TestResult.addError(Test, Throwable))  )
&& target(s) && args(test, t);

after(TestResult s, Test test, Throwable t)
returning: subjectAddError(s, test, t) {
   @SuppressWarnings("unchecked")
   Iterator iter= from(s).iterator();
   while (iter.hasNext()) {
      ((TestListener) (iter.next())).addError(test,t);
   }
}
```

Effectively, the first-class relation facilitates the transport of the arguments Test and Throwable from TestResult to the TestListeners through the method addError(). The JVM is the vehicle for transport but instead another vehicle could also be used. This would make it possible to decouple two (sub)systems.

After refactoring, the coupling between TestResult and TestListener was removed completely. For TestResult, the overall values for Fan-Out and CBO dropped by one. For TestListener, the overall values for Fan-In and CBO dropped by one.

Cohesion for TestResult increased but it remained the same for TestListener. The increase was caused by the removal of the fields and methods associated with TestListener. Effectively, one disjoint function area had been removed.

With the support of behavior in first-class relations complete decoupling between two objects can be achieved. Both objects become independent of the other and can be developed independently. The relation can later on be imposed onto the two objects.

Roles

In the start situation, the role of TestResult is defined by the notify methods: addError(), addFailure(), startTest() and endTest(). The role of TestListeners is defined by the same methods.

To implement roles with RAL/AspectJ we created a new interface SubjectRole, which declared the four notify methods and added it to TestResult using the open class mechanism of AspectJ:

```
declare parents: TestResult implements SubjectRole;

public interface SubjectRole {
   public void addError(Test test, Throwable t);
   public void addFailure(Test t,
                          AssertionFailedError a);
   public void startTest(Test t);
   public void endTest(Test t);
}
```

After refactoring the roles, we noticed no changes to coupling or cohesion in any of the objects. The reason was that a role was already present in the start situation: TestListener itself is the role. Adding an explicit Observer role to TestListener would perform the same function, a second time.

Roles are formally defined by interfaces that potential participants of the relation have to implement to participate. It is a communication aid for the first-class relation to potential participants. As such it can serve as a contract.

Participant methods

Participant methods do not exist in the start situation. They are added to the first-class relation for readability. The participant method addObserver() was added to TestResult.

After implementation we noticed that the participant method prevented that MPC of the clients increased by one. The participant method had no effect on cohesion or component coupling. Influence of the participant method occurred in the clients only. Table 3 shows the effect on MPC for all clients.

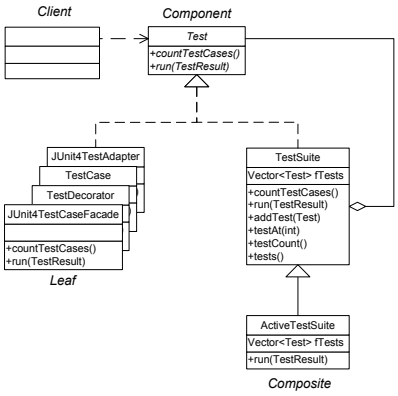

Figure 10 Instance of Composite Pattern in JUnit

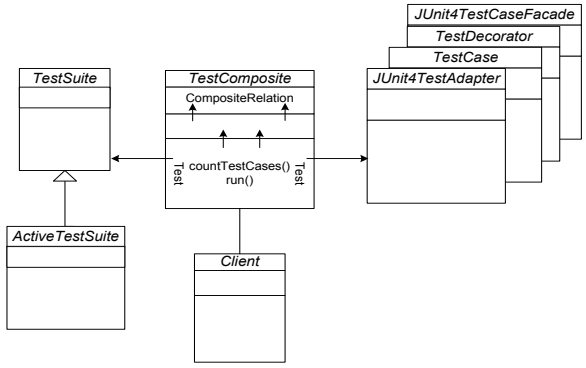

Figure 11 Composite Pattern Implemented as First-Class Relation

Table 3 Effects of Participant Methods on MPC of Clients

Object	Without	With
`TestListenerTest`	6	5
`TestRunner`	25	24
`JUnit38ClassRunner`	24	23
`ForwardCompatibilityTest`	22	21
`InitializationErrorForwardCompatibilityTest`	17	16

The lack of supporting participant methods or a similar mechanism (unless perhaps the Java reflection classes are used) is the reason for not using a POJO[2] as first-class relation construct. POJO's would unnecessarily increase MPC. However, further on in the application phase we see that abstract super classes do support a mechanism for participant methods. Concrete classes however, do not.

4.1.2 Composite Pattern

The Composite pattern "composes objects into tree structures to represent part-whole hierarchies" [14]. The pattern increases cohesion according to Grand in [15]. The reason is that it includes specialized methods in general purpose classes. Therefore, we do expect to see improvement in cohesion.

Figure 10 shows the UML class diagram of the instance of the Composite pattern in JUnit. It implements the relation between `TestSuite` and `Test`. `TestSuite` is the parent and `Test` is its property. `TestSuite` is a `Test` itself. `Test` can be in more than one `TestSuite` and `TestSuite` can contain more than one `Test`. Instances of both `TestSuite` and `ActiveTestSuite` can play the role of Composite. This instance of the Composite pattern has four Leaf classes.

Figure 11 shows the implementation of the Composite pattern with the first-class relation. The Leaf role (`Test` interface) has been removed from `TestSuite` and has been placed in the first-class relation. The client now acts on the first-class relation and not on `Test`. The code of the resulting first-class relation is shown in Figure 26.

Structure

In the start situation, the structure consists of the `Vector` object that holds the `Test` objects. Access to the `Test` objects is given by an add method, a size method and an index-based query method. No remove method exists. `Test` objects can be added to multiple `TestSuites`. Therefore the cardinality is many-to-many.

To apply structure we created a new relation aspect, `CompositeRelation`. The reason was that the clients required that order of the tuples were preserved. RAL's `SimpleStaticRel` does not guarantee order because it uses `java.util.HashSet` for the tuples. We used `java.util.LinkedHashSet` instead.

It does not mean that order must be preserved by definition. Indeed, in [21] Jackson defines relations without order. We decided to support order to minimize impact on JUnit.

We also added extended maintenance methods to support use of `Vectors` and for querying by index, both to minimize impact on the client code:

```
public void addAllTests(TestSuite suite, Vector<Test>
_Tests) {
    Iterator<Test> iter = _Tests.iterator();

    Test test;
    while (iter.hasNext()) {
        test = (Test) iter.next();
        TestComposite.aspectOf().add(suite,test);
    }
}
```

After the refactoring, the MPC in `TestSuite` dropped by 2 because the use of methods of the `Vector` class dropped. CBO and Fan-Out didn't change even though the `Vector` of `Tests` was removed from `TestSuite`. The reason was that the `Test` interface was still used by the behavior methods as well as many static utility methods.

We notice that support for structure does not remove the necessity of the partner objects. Also, all relations with the same partner object must be implemented as first-class relation. Otherwise, the necessity for the partner objects remains.

Active Behavior

In the start situation the Composite pattern has behavior that can be characterized as active because the client requires access to both participants. It does so using an interface that both participants have in common, the Component.

In this instance the behavior consisted of the methods `run()` and `countTestCases()`. Both are enforced by the `Test` interface, the Component.

[2] POJO is an acronym for Plain Old Java Object. A POJO is an ordinary java object.

To apply behavior we moved the behavior in the Composite to the first-class relation. We did not move the behavior in the Leaf to the first-class relation. Kiczales and Hannemann in [18] do also move the Leaf behavior methods to the aspect. We believe that is not correct because the behavior in the Leafs is part of their core functionality. Therefore, the first-class relation now has methods `run()` and `countTestCases()` that were previously implemented by `TestSuite`. These methods iterate over the Components in the relation.

```
public int TestSuite.countTestCases() {
   int testCases= 0;

   @SuppressWarnings("unchecked")
   Enumeration enu = Collections.enumeration(
      TestComposite.aspectOf().from(this));

   while (enu.hasMoreElements()) {
      testCases += ((Test) (enu.nextElement())).
        countTestCases();
   }
   return testCases;
}
```

After refactoring, ILCOM remains the same for `TestSuite` but TCC (tight class cohesion) drops almost to zero. The reason that cohesion does not improve is the static utility methods. After we removed them from `TestSuite` we saw that TCC doubled to that of its original value. The utility methods did not use any fields and were therefore regarded as separate function areas. Without the utility methods all that remains of `Testsuite` is a class with only `name` as property, a getter method and a setter method. See Figure 27 for the source code.

Without the utility methods CBO and Fan-Out values drop to zero for `TestSuite`, as did MPC. With the utility methods there was no change in the coupling.

Coupling in the Leafs did not drop. It was expected to drop on account of Fan-In. That it did not drop was because JHawk did not take the implemented `Test` interface into account.

As was expected from [15] we did see an improvement in cohesion. However, the utility methods did pose an obstacle to achieving this. They even gave the impression that cohesion decreased because the ratio of the number of connected methods over the number of disconnected methods decreased after the first-class relation had been implemented.

Roles

In the start situation, the Composite pattern adds the role of Leaf to a container object to make it behave like a Leaf. For this Composite instance the Leaf role was implemented by the `Test` interface. `Test` was added to `TestSuite` and `ActiveTestSuite` and its operation methods `countTestCases()` and `run()` were implemented. These methods looped over the `Tests` and called the methods of the same name in the Leaf objects. The `Tests` were stored in a `Vector` object.

To implement the role for `TestSuite` in the first-class relation we removed the `Test` interface and had it added back again by the first-class relation using the open class mechanism of AspectJ. We did not have to create a dedicated interface for this role since `Test` was already present.

```
declare parents: TestSuite implements Test;
```

After the role had been added we saw no effect on coupling or cohesion. The role of `Test` was imposed on `Testsuite` so that it could behave as Leaf. Notice also that the role could have been imposed on any object.

Although roles have no effect on coupling or cohesion they are useful as is shown by their presence in the Composite pattern. In the Composite pattern the role of `Test` formally ties Composite and Leaf together.

Participant Methods

We added participant methods for the regular maintenance methods but also for the extended maintenance methods (with `Vector` support). These participant methods were added to `CompositeRelation` using parameterized types to preserve the generic nature of the maintenance methods:

```
public void Fwd.addAllComponents(Vector<Component>
_Components) {
   Fwd<Component> composite = (Fwd<Component>) this;
   Iterator<Component> iter = _Components.iterator();

   Bwd<Composite> component;
   while (iter.hasNext()) {
      component = (Bwd<Composite>) iter.next();
      composite.fwd.add((Component)component);
      component.bwd.add((Composite)composite);
   }
}
```

A client using a participant method is shown below:

```
suite.addAllComponents(TestSuite.createTests(TestListen
erTest.class,AssertTest.class,TestImplementorTest.cla
ss,NoArgTestCaseTest.class,ComparisonCompactorTest.cla
ss,ComparisonFailureTest.class,DoublePrecisionAssertTe
st.class,FloatAssertTest.class));
```

The participant methods prevented that MPC of the clients increased. The same result was noticed with the Observer pattern. Although participant methods do nothing for coupling and cohesion for the participants they do indeed make the code in the clients more readable. The relations on the other hand are no longer visible.

4.1.3 Command Pattern

The Command pattern "encapsulates a request as an object, thereby letting you parameterize clients with different requests, queue or log requests, and support undoable operations" [14]. The pattern reduces coupling between the Receiver and the Invoker [14]. Reduction is achieved by introducing the (Concrete)Command object. Therefore, we do not expect coupling to reduce by introducing a first-class relation [8].

Figure 12 shows the UML class diagram of the implementation of the Command pattern in JUnit. The Command pattern introduces the `FilterRequest` object to create a relation between `Filter` and sub-classes of the abstract class `Request`. Thus `FilterRequest` stores relations between one of its siblings and a `Filter` object.

T he execute method of the pattern is `getRunner()`. It is implemented in the concrete sub-class `FilterRequest` and is used by clients. The execute method triggers the action method `apply()` in `Filter`. Clients use the constructor of `FilterRequest` or its `filterWith()` method to create new relationships.

Figure 13 shows the implementation of the Command pattern with the first-class relation. The first-class relation replaced the `FilterRequest` object and `Request` was made a concrete object class. Its `filterWith()` method was removed and instead the clients use the maintenance methods of the first-class relation.

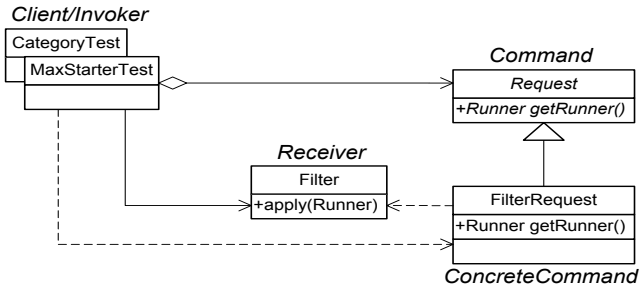

Figure 12 Instance of Command Pattern in JUnit

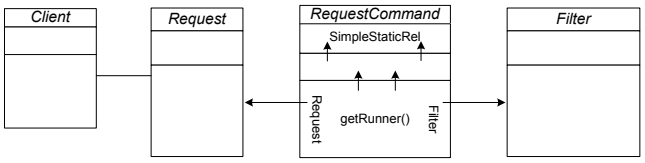

Figure 13 Command Pattern Implemented as First-Class Relation

The `getRunner()` method constitutes the behavior of the first-class relation. The clients access its behavior directly. The code of the resulting first-class relation is shown in Figure 28.

Structure

In the start situation, structure is contained in `FilterRequest`. It consists of two field members, one of type `Request` and the other of type `Filter` making cardinality one-to-one. Other than the constructor, `FilterRequest` does not provide any methods to access the field members.

To apply structure we used our `SimpleStaticOneToOneRel` aspect. It supplies a method to add new one-to-one relations. This makes the constructor of the in `FilterRequest` redundant as well as both field members. What remains in `FilterRequest` is the `getRunner()` method.

However, `getRunner()` no longer knows on which instances of `Request` and `Filter` it must act. Although it can use the maintenance methods of the first-class relation, it still requires one of the participant instances to get to the other. In other words, the bond between `FilterRequest` and its participants has been lost. To resolve this we added `Request` as parameter to the method call of `getRunner()` and the method is made static:

```
public static Runner Request.getRunner(Request req) {
   Filter filter = RequestCommand.aspectOf().
      getFrom(req);
   try {
      Runner runner= req.getRunner();
      filter.apply(runner);
      return runner;
   } catch (NoTestsRemainException e) {
        return new ErrorReportingRunner(Filter.class,
                new Exception(String.format(
                "No tests found matching %s from %s",
                filter.describe(), toString())));
   }
}
```

After structure was added we saw no change in coupling or cohesion. Even in `FilterRequest` there was no change in cohesion.

We notice that where it concerns structure, support of relations does not necessarily require a special construct; specialized relation objects can also achieve lower coupling and higher cohesion. The introduction of the `FilterRequest` object is such a specialized relation object, which forms the relation between `Filter` and `Request`.

Active Behavior

In the start situation, `FilterRequest` contains a static `getRunner()` method to implement the behavior. Clients trigger behavior through both participants making it active. Structure is already implemented in the first-class relation.

To apply behavior to the first-class relation we move the `getRunner()` method to the first-class relation. This makes the `FilterRequest` object completely redundant.

After the implementation of behavior we still see no change in coupling and cohesion. This confirms the idea that specialized relations objects already pose advantages over relations implemented through field members.

Roles

In the start situation the role of `Request` is defined by its methods `getRunner()` and `toString()`. The role of `Filter` is defined by its methods `apply(Object)` and `describe()`. To implement roles with RAL/AspectJ we created new interfaces `ReceiverRole` and `CommandRole`, which declared the methods and were added to `Filter` and `Request` using the open class mechanism of AspectJ:

```
declare parents: Filter implements ReceiverRole;
declare parents: Request implements CommandRole;
```

After applying the roles we saw no effect on coupling or cohesion.

Roles are implemented as interfaces and added to participants using the open-class mechanism of AspectJ. JHawk however does not see that these interfaces have been added because they are added during compilation and JHawk measures the source code, not the byte code.

Participant Methods

In the start situation the first-class relation has no participant methods. It is expected that MPC values will drop when participant methods are added to the first-class relation because in the start situation the relation is implemented as an object and participant methods were not used.

Participant methods were implemented for the maintenance methods and for behavior method `getRunner()`:

```
// Participant structure methods
public void Filter.addRequest(Request request) {
   RequestCommand.aspectOf().add(this,request);
}

public void Request.addFilter(Filter filter) {
   RequestCommand.aspectOf().add(filter,this);
}
// Participant behavior methods
public Runner Request.getRunner() {
   return RequestCommand.aspectOf().getRunner(this);
}
```

```
public Runner Filter.getRunner() {
   return RequestCommand.aspectOf().getRunner(
      RequestCommand.aspectOf().getTo(this));
}
```

Compared to the start situation, with structure and behavior already implemented in the first-class relation, the MPC values in the clients dropped as expected. However, compared to the baseline situation, where the relation was implemented through a specialized object, the MPC values did not drop. For `CategoryTest` the MPC actually increased by one. The reason for this increase is that the specialized relation object was implemented as a sibling of one of the participants. Using the abstract super-class, the client cannot differentiate between the participant class and the specialized object class. In other words, abstract super-classes can simulate participant methods.

4.2 Test case

For the test case we confined ourselves to the property relations because these relations are static. Other types of relations, such as relations with local method scope are dynamic of nature. We limited the scope of our research to static relations.

For the test case we selected `junit.samples.AllTests`. The choice of test case was arbitrary since all test cases we used for regression tests had the same signature: creation of a test suite, add tests and run the test suite. The test case identified the relations depicted in Table 4.

The table shows that most relations are in fact aggregation associations. Aggregation associations occur when the cardinality of the property is always one but property can outlive the parent. The table also shows that two relations exist between `TestResult` and `TestFailure`, one for errors and one for failures.

Refactoring the relations from the test case happened in the same manner as those from the design patterns. All relations had the same structure in the start situation and implementing the structure in the first-class relation required the same steps. In the next paragraphs we show only the high lights of the measurements and analysis for these relations.

Structure
In the start situation, the structure for all one-to-one relations is implemented as field member (property) of an object (parent). Either the clients create the parent and pass the property to the parent through the constructor or they use an access method to add the relation after creation of the parent. For the one-to-many relations, the property is implemented with a collection class. Furthermore, all relations are unidirectional which implies that the property is unaware of the relation.

To apply structure we used the `SimpleStaticOneToOneRel` for the one-to-one relations and `SimpleStaticOneToManyRel` for the one-to-many relations. The property is removed from the parent as well as the associated access methods for clients. To access the property, the parent now uses the maintenance methods.

Active behavior and Roles
For all relations in the test case that have behavior, the behavior is active; the client has access to both participants. Also all relations that have behavior are aggregation associations.

For aggregation associations, we can implement only the role of the property because the role of the parent cannot be expressed in terms of public methods. Therefore, the behavior methods in the first-class relation are a copy of the methods of the property.

Table 4 Cardinality of Relations in Test case

Parent	Property	Cardinality	Remark
`JUnit4TestCaseFacade`	`Description`	1-1	
`TestResult`	`TestFailure`	1-n	Errors
`TestResult`	`TestFailure`	1-n	Failures
`TestFailure`	`Test`	1-1	
`TestFailure`	`Throwable`	1-1	
`ResultPrinter`	`PrintStream`	1-1	
`TestRunner`	`ResultPrinter`	1-1	

We added the roles as interfaces to the parent using the open class mechanism of AspectJ.

Participant methods
We applied participant methods as we did for the relations in the design patterns. We used participant methods both for maintenance methods and for behavior methods.

4.2.1 JUnit4TestCaseFacade versus Description

Figure 14 shows the start situation of the relation between `JUnit4TestCaseFacade` and `Description`. It concerns an aggregation relation; the instance of `Description` is only passed in as argument to the constructor of `JUnit4TestCaseFacade` and therefore cardinality will always be one. `JUnit4TestCaseFacade` also has an access method `getDescription()` that returns the instance of `Description`.

After applying structure we see coupling between both participants and lack of cohesion in `JUnit4TestCaseFacade` drop to zero. The application of structure replaces the instance of `Description` with a maintenance method call:

```
DescribableDescription.aspectOf().getTo(this);
```

Adding behavior does not further decrease coupling and lack of cohesion since both are already zero. The behavior consists of the `toString()` method of `JUnit4TestCaseFacade`. It returns the string value of `Description`. Behavior was implemented by placing this method in the first-class relation.

Adding roles to the first-class relation has no effect on coupling and cohesion. `Description` is the only participant that has a role. It is defined by its `toString()` method. The role was added using an interface:

```
public interface DescriptionRole {
   public String toString();
}
```

The participant methods made MPC drop by one. Adding the participant methods replaces the direct calls to the maintenance methods. Calls to the participant methods look the same as calls to the access methods:

```
getDescription();
```

The source code of the resulting first-class relation is shown in Figure 29.

4.2.2 ResultPrinter versus PrintStream

Figure 15 shows the start situation of the relation between `ResultPrinter` and `java.io.PrintStream`. It concerns an aggregation relation; `java.io.PrintStream` is only passed in as argument to the constructor of `ResultPrinter` and therefore

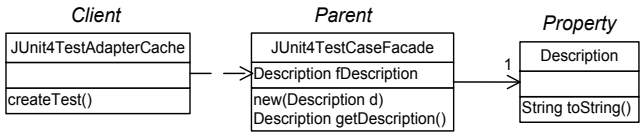

Figure 14 Relation between JUnit4TestCaseFacade and Description

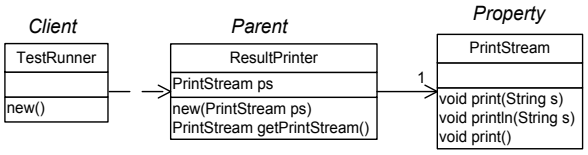

Figure 15 Relation between ResultPrinter and PrintStream

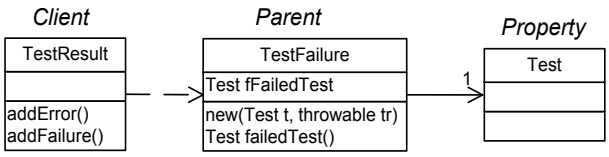

Figure 16 Relation between TestFailure and Test

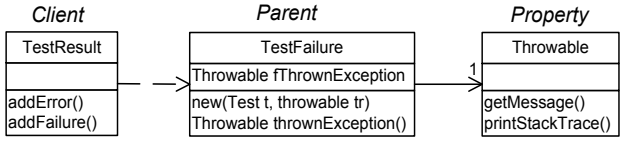

Figure 17 Relation between TestFailure and Throwable

cardinality will always be one. The methods that define the role of `java.io.PrintStream` are:

```
public void print(String text);
public void println(String line);
public void println();
```

Lack of cohesion and the coupling metrics drop by one for `ResultPrinter` after the implementation of structure. The method `getWriter()` and the constructor of `ResultPrinter` are removed. Instead the maintenance methods are used, a participant constructor method is created and a participant getter method. The participant methods prevent that MPC increases in the clients.

Adding behavior does not change coupling and cohesion. Implementation of the structure has replaced all references in `ResultPrinter` to `PrintStream` with references to the first-class relation. Using participant methods has removed even those references. What remains are the methods of `PrintStream`. `ResultPrinter` requires those to function, which means that it remains aware of the relation and the first-class relation acts like the dependency injection mechanism.

Implementation of the roles has no effect on coupling or cohesion either. `PrintStream` is the only participant that has a role.

The source code of the resulting first-class relation is shown Figure 30.

4.2.3 TestFailure versus Throwable and Test

Figure 16 and Figure 17 show the start situation of the relations of `TestFailure` with `Test` and `Throwable`. It concerns two aggregation relations; `Test` and `Throwable` are only passed in as argument to the constructor of `TestFailure`. After creation the relation is never mutated. Therefore cardinality with `Test` and `Throwable` will always be one.

```
public TestFailure(Test failedTest, Throwable
      thrownException) {
   fFailedTest= failedTest;
   fThrownException= thrownException;
}
```

Strictly speaking the properties can have value null. However, the code in the remainder of the object, does not anticipate on that i.e. nowhere is checked for value of null. `TestFailure` contains methods for accessing the `Test` and `Throwable` fields.

`Test` does not have a role in the relation with `TestFailure` i.e. `TestFailure` is used as container only. `Throwable` does play a role. Its role is defined by the following methods:

```
public void printStackTrace(PrintWriter p);
public String getMessage();
```

Lack of cohesion and the coupling metrics drop by 2 for both participants after implementing structure. Coupling and cohesion did not change for any object after implementing behavior.

After implementation of the roles no change to coupling and cohesion is measured. The resulting implementation of the first-class relation between `TestFailure` and `Throwable` is shown in Figure 31 and the relation between `TestFailure` and `Test` in Figure 32. Notice that the latter implements the constructor for `TestFailure`.

4.2.4 TestResult versus TestFailure

Figure 18 and Figure 19 show the start situation of the two relations that `TestResult` has with `TestFailure`. One is for errors and the other for failures. Both relations are used in the Observer pattern in which `TestResult` plays the role of Subject.

For both relations, `TestResult` can have zero or more instances of `TestFailure`. `TestResult` stores errors and failures in `Lists` and provides access methods to them. The relations do not contain any behavior.

Both relations must be implemented to cause the coupling and lack of cohesion to drop. Implementing only one of these two relations as a first-class relation causes the ILCOM to drop by one for `TestResult` but coupling remains the same. Only after implementing the second relation the coupling decreases.

MPC drops in `TestResult` because it no longer uses the `List` object to store `TestFailure` but only after the first-class relations have replaced both relations.

The source code of the resulting first-class relations is shown in Figure 33 and Figure 34.

4.2.5 TestRunner versus ResultPrinter

Figure 20 shows the start situation for the relation between `TestRunner` and `ResultPrinter`. It concerns an aggregation relation; `ResultPrinter` is only passed in as argument of the constructor of `TestRunner`. `TestRunner` also provides a default `ResultPrinter`. Furthermore, the relation can be mutated using the `setPrinter()` method which replaces the current

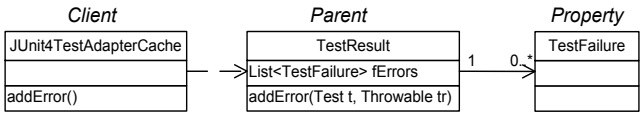

Figure 18 Error Relation between TestResult and TestFailure

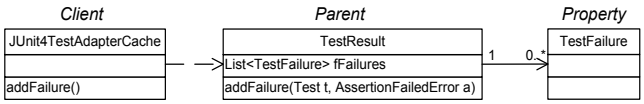

Figure 19 Failure Relation between TestResult and TestFailure

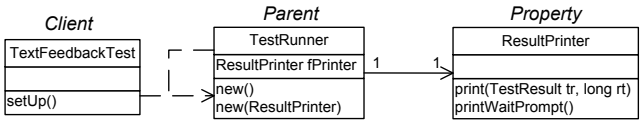

Figure 20 Relation between TestRunner and ResultPrinter

`ResultPrinter`. The role of ResultPrinter is defined by the following methods:

```
void print(TestResult result, long runTime);
void printWaitPrompt();
```

Structure for this relation requires an adapted version of `add()`. It must replace the existing tuple instead throwing an exception. We implemented this by creating a new version of abstract `SimpleStaticOnetoOneRel` aspect.

We noticed that the RAL/AspectJ implementation of first-class relations can not support roles in all situations. To superimpose the role on `ResultPrinter` we had to adjust the visibility of the two methods involved. We changed the visibility from "`package`" to "`public`". Without it AspectJ throws a "reducing visibility" exception.

The implementation of structure in the first-class relation decreases the value of ILCOM, CBO and Fan-Out of `TestRunner` and the value of CBO and Fan-In of `ResultPrinter`. MPC depends on a participant method to stay the same.

The implementation of behavior and roles does not change coupling or cohesion. The source code of the resulting first-class relation is shown in Figure 35.

5. RESULTS

This section presents an overview of the results of the application phase of our research. Table 5 in Appendix F shows the relations we refactored and the characteristics of the first-class relations that were created. In total we created 10 first-class relations involving 13 participants and several clients.

Seven of the first-class relations had behavior, one of which was reactive. We measured the effects of structure, cardinality, participant methods, behavior and roles on coupling and cohesion.

The subsequent paragraphs present overviews of the effects we measured. For clarity, the objects for which we did not find measurable effects were left out.

5.1 Structure

We did not implement the sub-elements of structure (maintenance methods, state and cardinality) individually. Instead they were implemented together in one step.

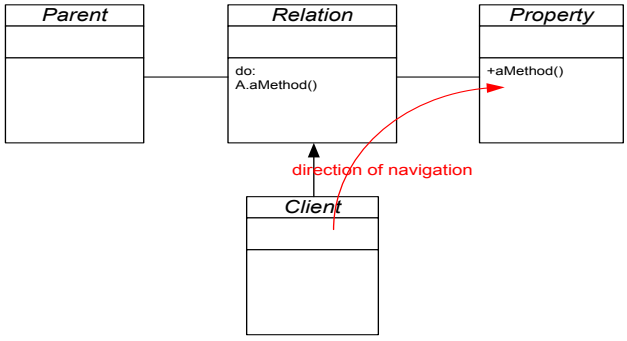

Figure 21 Behavior in Test Case Relation Depended on Property only

Occasionally, the source code in the client objects in JUnit required us to add to the functionality provided by the RAL aspects. Additional functionality was required for (1) cardinality other than many-to-many and (2) preservation of the order of the tuples in a relationship.

To support cardinality other than many-to-many we implemented two new relation aspects, one for one-to-many and one for one-to-one cardinality. To preserve cardinality in the add() method of one-to-one relations we had to support two mechanisms: (1) blocking the operation and (2) replacing the tuple.

To support preservation of the order of the tuples in the many-to-many relationships we replaced the `java.util.HashSet` with the `java.util.LinkedHashSet` in the predefined RAL aspects. This support was required in one occurrence only. For the other occurrences we used the RAL aspects.

The effect of the changes mentioned above was evident in the component coupling and cohesion of the participants of the relation, but not in any of the clients. Improvement of MPC in the parents, on the other hand, was not caused by our changes but was mostly due to removing the getters and setters needed to access the collection objects that were used to store the properties.

In the property relations, the component coupling metrics showed that parent and property became completely decoupled. However, because the role of the parent could not be extracted, it was now coupled to the first-class relation instead. Prerequisite for decoupling between parent and property was that all relations between the same participants were implemented as first-class relation.

An overview is presented in Table 6 in Appendix F.

5.2 Behavior

We encountered different types of behavior in JUnit. The behavior was either active or reactive. The behavior in the relations from the design patterns depended on the roles of both participants. The behavior in the relations in the test case on the other hand, depended on the role of the property only. These relations had active behavior only. See Figure 21. Either parent or client could initiate the behavior.

To support behavior we added methods to the relation aspects. For reactive behavior we had to use pointcuts/advices to attach the relation methods to the methods of the triggering participant.

For the property relations, the relational behavior methods of the first-class relation used only the methods of the property and the entire parent object was passed in as argument. These relations

were triggered either by the parent object or by a client. The property had cardinality that was always one. In other words, the lifetime of the relation was equal to the lifetime of the parent object.

Only the relations in the Observer pattern and the Composite pattern showed improvement in coupling and cohesion due to support of behavior. These relations contained behavior in which both participants played a role. Coupling between participants in the Observer pattern dropped to zero. This also happened in Composite pattern after the static utility methods had been removed from `TestSuite`.

The property relations showed no effects on coupling and cohesion under the influence of support for behavior. Coupling and cohesion had both already dropped due to the implementation of structure.

An overview is presented in Table 7 in Appendix F.

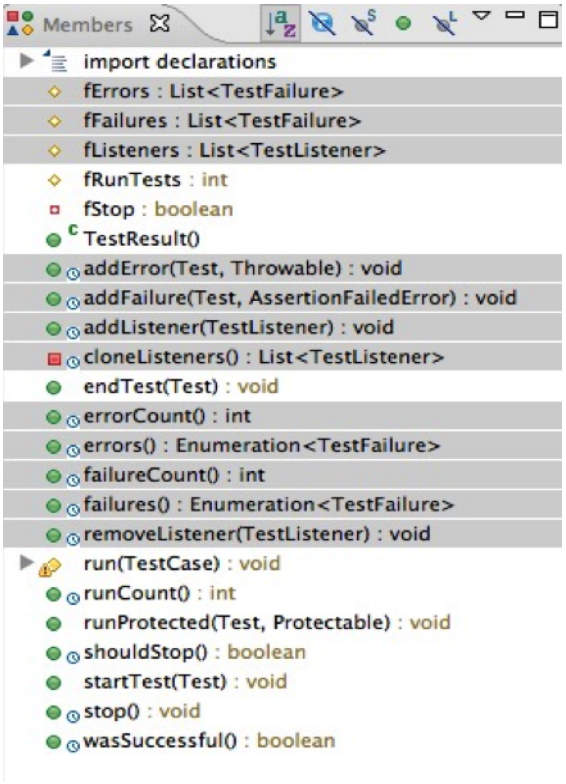

Figure 22 Boilerplate Code in TestResult

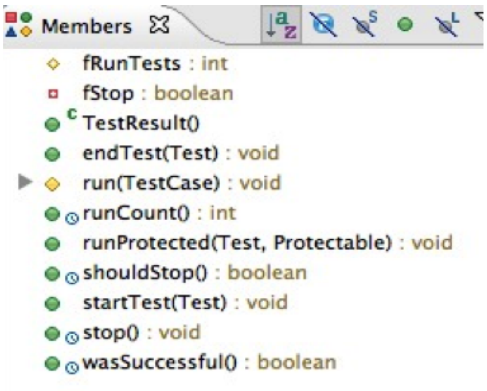

Figure 23 Boilerplate Code Removed from TestResult

5.3 Roles

Roles were used to formalize the interaction between the relation and its participants. By doing so the number of potential methods required of the participant is restricted.

To support roles we created java interfaces in which the methods were listed that were required of the participant. Using the open class mechanism of AspectJ we added them to the participant. Furthermore, role names were specified as parameterized type to the generic relation aspects. We did not measure any effects on coupling or cohesion due to roles.

5.4 Participant Methods

The purpose of the participant methods was to make our RAL/AspectJ implementation of first-class relations appear as real first-class relation constructs to the metrics tools. As a consequence, the first-class relation was no longer visible as such in the client code. Now, to a client it appeared as though relations had been implemented using the orthodox way of field members. Participant methods were used to hide both maintenance methods and behavior methods. On some occasions we also used them to hide constructors.

The effects of participant methods were visible especially in the client objects or in participants that also acted as client. The effect was on MPC only. The `FilterRequest` object in JUnit showed that participant methods can be implemented in specialized relation objects too. The way to do this is to make a specialized relation object as a sibling of one of the participants by giving it the same abstract super-class. As a consequence, it prevents MPC to drop.

Participant methods do attribute to the ease of use of first-class relations. This cannot however be expressed through coupling or cohesion.

An overview of the effect of participant methods is presented in Table 8 in Appendix F.

5.5 Other Effects

In addition to the effects on coupling and cohesion we encountered other effects as well some of which are worth mentioning here:

- The amount of boilerplate code was reduced. The getters and setters one must implement to maintain relations are no longer necessary. Figure 22 shows the `TestResult` object before applying two first-class relations and Figure 23 shows the same object afterwards.
- Interaction between two objects is formalized. The use of roles in first-class relations is the opposite of how interfaces are used today: showing what an object can do, instead of what it must be able to do. First-class relations demand that an object can play a certain role. Even though, this role has not been explicitly identified by the object it self, we can examine its public interface and determine whether or not it can play that role. This allows us to super impose relations onto objects.
- The relations were all unidirectional before refactoring. During refactoring directionality needed not to be considered because first-class relations support access from both sides by default.

6. ANALYSIS

In this section we analyze our observations. Using these observations we map the elements of first-class relations onto

the characteristics of first-class relations to understand the effects the latter have on coupling and cohesion.

6.1 Special Notation

We identified one element that supported the characteristic of special notation. This element was the participant method. We used the participant method to ascertain that the metric tools would not count the first-class relation construct as object. In the Command pattern implementation we noticed that a specialized relation object can simulate participant methods if they have been made sibling to one of the participants.

Special notation probably is important, but not for achieving lower coupling or higher cohesion. We did see the MPC drop in client objects and parent objects due to participant methods. However, one could argue that it is the metric tools that drive the need for the special notation and not coupling or cohesion metrics themselves. To our metric tools, without the participant methods, first-class relations have the same appearance as POJO's.

We do expect that special notation is especially useful to promote using first-class relations. The implementation with RAL/AspectJ however is a bit cumbersome to our judgment. To illustrate our idea of a special notation we present a code snippet of the Observer relation in a hypothetical object oriented programming language.

```
role Subject {
   public void doSomething();
   public State getState();
}
role Observer {
   public void doSomethingToo(State state);
}
// Reactive relation
relation ResultObserver[Subject, Observer] {
   Subject.cardinality = 1;
   Observer.cardinality = many;

   if Subject.doSomething() {
      Observer.doSomethingToo(Subject.getState());
   }
}
class LocalParent() {
   // Use property relation to make result available
   // globally
   property TestResult result = new TestResult();
   ...
   // TestResult used further downstream
   result.doSomething();
}
class RemoteClient() {
   // TestResult remotely initialized
   TestResult result = TestResult.getInstance();

   // ResultPrinter locally initialized
   ResultPrinter printer = new ResultPrinter();
   try {
      ResultObserver resultObserver = new
        ResultObserver[TestResult, ResultPrinter];
      resultObserver.add(result, printer);
   }
   catch(RoleNotSupportedException e){
      // role(s) not supported by participant(s)
      ...
   }
}
```

This code snippet shows the definition of the `ResultObserver` relation with its two associated roles. The `LocalParent` class creates the instances of the `TestResult` object. The `property` keyword places the instance in a global space, which is accessible to remote clients. The instance of `TestResult` is unaware of what is happening in other parts of the software or in remote clients. The `RemoteClient` class retrieves the one instance of `TestResult` making it available to the rest of the remote client. The `ResultObserver` relation is created if `TestResult` supports the `Subject` role and `ResultPrinter` supports the `Observer` role. If not, an exception is thrown.

6.2 One Entity

State is the only element of first-class relations that supports the characteristic of one entity. State was not implemented individually but rather as part of structure together with the maintenance methods and cardinality.

The effect of structure was not directly evident in the Observer pattern and the Composite pattern. We noticed that the structure itself had influence on MPC only, but not on component coupling, and not on cohesion.

The property relations did show improvement in cohesion and coupling from the implementation of structure. The inability to extract the role from the parent made the parent become dependent on the first-class relation instead. The metrics did not show this because JHawk doesn't recognize aspects as objects. Support for structure did not succeed in removing this dependency. Instead it provided a dependency injection mechanism.

Since aggregation associations profit directly from first-class relations we expect composition association to do so too but lack evidence to support this. For less restricted types of associations it is a prerequisite that first-class relations are implemented as one entity.

6.3 The Only Relation Construct

The third characteristic of a first-class relation construct is that it does not allow other relation constructs between the same objects.

The importance of this characteristic was evident in the relations between `TestResult` and `TestFailure`. After implementation of only one of the two relations we saw no increase in cohesion or decrease in coupling. We had to implement both relations as a first-class construct to achieve this.

The relation between `Testsuite` and `Test` in the Composite pattern again confirmed the importance of this characteristic. `TestSuite` provides many static utility methods for `Test` that obstruct decoupling and cohesiveness. Although one can also argue that static utility methods should not be counted by the metrics since they are references to object classes and not references to object instances.

6.4 Same Run-Time Concept

The fourth characteristic is that of run-time concept. A first-class relation must be recognizable as the same entity during run-time and development.

We found no evidence to support this claim. Aspects remain distinct entities during run-time but we could not test this because there was no element in the first-class relation model that we could switch on and off.

6.5 Support of Operations

The fifth characteristic was the support for operations on relations. These are operations that are performed on the relation as a whole i.e. all relationships formed by the relation.

We found no evidence to support this claim. Our one-to-one implementation of the first-class relation can hold multiple relationships and could possibly support this characteristic. However, we did not come across any situation that required this type of operation.

We do believe that once people start working with first-class relations these operations will be used. We expect it to have the same or better results on coupling and cohesion for two reasons: (1) the same relational behavior methods can be used and (2) there is no need to specify the participant instances on which to operate because it will operate on all participant instances in all relationships.

6.6 Support of Constraints

The last characteristic we identified was support for constraints. Constraints keep participants in sync with each other.

Support for constraints was implemented by behavior. We distinguished two types of behavior: reactive and active. Reactive behavior can be compared to an if-then statement. If participant A executes then participant B must execute too. Active behavior can be compared to a block statement. It ensures that behavior of participant A and behavior of participant B are executed together.

Behavior also provides the means to implement transactional control and reactive behavior in particular provides support for propagation of operations as mentioned by Rumbaugh [33].

6.7 Other Observations

In this paragraph we discuss observations that were not related to the characteristics of first-class relations.

6.7.1 Roles

Roles can be declared in RAL using parameterized interfaces or object classes of generic aspects. We switched roles on and off like we did with the other elements. Roles helped us to formalize the relations. It clarified what was expected of an object to participate in a relation. To honor information hiding we always used only the public interface of the participants. However, we hardly ever needed the entire public interface of the participants to support a relation. And only that part of the public interface that is needed is part of the definition of the role.

We envision a system in which relations and objects are orchestrated through their associated roles. In such a system the roles of an object matter, not the class of an object. Although, our metrics do not differentiate between object roles and object classes, we do see advantages. A role limits the strength of the coupling between two objects to the methods defined by their roles.

Roles support the same mechanism as dependency injection and more. Using dependency injection only the interfaces of property objects are tied to the parent object. Using a relation for a parent and its properties, the interface also is the only entity that remains tied to the parent. Relation on the other hand also offers control over state.

6.7.2 Cardinality Constraints and Other Invariants

Different types of invariants are recognized in literature. Balzer et al in [4] differentiate between value-based and structural invariants and between intra-relationship and inter-relationship invariants. An intra-relationship invariant is also called cardinality.

Cardinality is the only invariant we encountered. Other invariants could easily be implemented as property of a first-class relation and enforced using post-conditions on the relation methods.

6.7.3 Utility Objects

Utility objects should not be taken into account by the metrics. In all our implementations we see the effect of maintenance methods on coupling. The effect however, is often related to the use of Collection objects or derived objects.

6.7.4 Reporting

Reporting functions are important in understanding the behavior of a software system. First-class relations provide a centralized repository of relationships. This repository can be queried independent of the software system itself. It can present the current situation of a system and keep a history of relationships. This gives stakeholders guidance for future activities on the software system, such as software maintenance and scaling.

6.7.5 Bidirectional Relation Example

The example of the bidirectional relation that we presented in our motivation was bidirectional only to allow maintenance from both participants. It was unrelated to the behavior of the relation. A first-class relation makes the choice for directionality irrelevant. Maintenance occurs directly on the relation not through the participants.

6.7.6 Inter-Application Communication

In this paragraph we discuss how first-class relations can be implemented to support inter-application communication. Based on behavior we can distinguish four types of static relations:

1. reactive relation: an object depends on another object and waits for it to be triggered by that other object but the triggering object is unaware of that dependency;
2. active relation: a third object depends on the cooperation of two objects;
3. property relation: a parent depends on its property and actively engages that property;
4. siblings relation: two objects share the same parent but have no shared behavior.

A reactive relation is imposed onto two objects that can exist completely without the each other. The reactive relation can be compared to the synchro-servo system in mechanics. The synchro device senses the position of an object. The servo device actuates another object to move its position according to the change in position of the former object. Thus position can be compared to state in the Observer pattern. Synchro and servo devices are typically used when both objects are at different locations and are typically connected through electrical wiring, which is used to transport the change in position. Thus synchro and servo devices are the roles played by the two objects. One object must have an (output) interface that allows sensing its position; the other must have an (input) interface that allows actuating by position. Notice that the position of the first object does not have to be the same type of position of the second object. In this way the synchro-servo system and the reactive

relation differ from the Observer pattern. In the Observer pattern both Subject and Observer must share the knowledge of the same type of state.

A reactive relation is suitable for asynchronous inter-application communication. Instances of Observer patterns are good candidates for this type of communication. Gamma et al in [14] describes that communication of the state of the Subject is implemented using a callback function. A callback function however, requires synchronous communication. Passing in argument instead, as implemented in JUnit, keeps the communication asynchronous. In fact, any two objects that are independent of each other can be connected reactively.

An active relation is created if a third object, the client, wants to actively engage two other objects. These two objects are unaware of each other and the relation that has been imposed.

An active relation is suitable for synchronous communication across multiple components, not just two as discussed in this paper. It coordinates the method calls and can maintain state in between calls. The client of an active relation can itself be triggered through a reactive relation with another object. The active relation can be compared to the process manager, as described by Hohpe and Woolf in [20].

A property relation is a type of active relation in which only the property plays a role. Both parent and client can initiate the relation behavior. The parent cannot exist without the property. The parent is composed of its properties and its properties are added during the creation of the parent.

Our impression is that a property relation is best implemented for intra-application communication. When a first-class relation is used to implement a property relation, the parent becomes dependent on the first-class relation instead of the property. Although, a first-class relation still adds value by providing structure, the parent is now tightly coupled to the first-class relation.

Sibling relation is a relation between objects with the same parent. For example, `Test` and `Throwable` both had `TestFailure` as parent. This resulted in the implementation of two separate relations.

Applying first-class relations can dissolve sibling relations. For example, a consequence of the two first-class relations of `TestFailure` with `Test` and `Throwable` was that the latter two are no longer visibly related as siblings. A way to resurface this sibling relation is to use the product of the two first-class relations.

6.7.7 Decomposition

Figure 36 shows that both Observer pattern and Composite pattern when refactored into first-class relation are locations where the system can be decomposed. This was expected of the Observer pattern since it binds together two object classes that are essentially independent. We did not expect it from the Composite pattern however. On hindsight this is not surprising. The Composite pattern appoints an object to perform the role of container. That object is then equipped with behavior to iterate over the Leafs. This behavior however was not its core functionality.

7. CONCLUSIONS

We conducted an empirical idiographic case study on JUnit to find out how the characteristics of first-class relation constructs attribute to coupling and cohesion of object-oriented code. We identified six characteristics. During an exploration phase we defined a model for the first-class relation constituted of its elements. The exploration phase was followed by the application phase in which we refactored the code to first-class relations. During the analysis of this phase we translated the elements of first-class relations to the characteristics of first-class relations.

We identified three major elements: structure, behavior and roles. Structure consisted of state, maintenance methods and cardinality. We distinguished active and reactive behavior. In the application phase these elements were applied one by one to parts of the source code of JUnit. We used design patterns and a test case for selecting these parts.

We noticed that the effectiveness of behavior depended on the extent to which participants in a relation could be made unaware of their partner. A participant that cannot be made unaware of its partner does not profit from support of behavior by the first-class relation. This occurred in relations between parent and their property hence we call this a property relation.

When both participants could be made unaware, then support for behavior in the first-class relation construct was also required to gain full profit on coupling and cohesion.

Support for roles was neutral to decoupling and cohesiveness. Although, they were part of the definition of behavior and restricted the public methods that were required from the participants we did not measure any effects on coupling and cohesion.

Three of the six characteristics of first-class relations support the claim for decoupling and cohesiveness: (1) being one entity, (2) being the only relation construct and (3) support for relational constraints. The other characteristics could not be measured or were neutral to coupling and cohesion. This also means that we did not find any characteristics that conflicted with decoupling and cohesiveness. Support for one entity is provided by state, which is a sub-element of structure.

Structure provides a mechanism similar to dependency injection if only the role of the property is implemented. Although structure externalizes maintenance of tuples of related object instances, it cannot make the parent unaware of the relation with its property if the role of the parent cannot be expressed in terms of its public methods. As a consequence, the relational behavior cannot incorporate the behavior of the parent but only that of the property.

Being the only relation construct is a prerequisite for decoupling and cohesiveness. After one out of two relations between the same participants had been refactored, coupling and cohesion had remained the same. Coupling and cohesion did improve after the second relation also had been refactored.

Relational constraints are supported by the behavior element because it manages the behavior of the participants. The relations from the Observer pattern and Composite pattern required support of behavior to fully profit in terms of decoupling and cohesiveness. The property relation however did not.

To promote using first-class relations we expect that they must have their own special notation. In our opinion this is an important characteristic for any first-class construct. However, we could not find any results supporting our opinion.

7.1 Threats to Validity

We cannot say with certainty to which extent the implementation of the metrics by JHawk and VizzMaintenance adhere to their definitions. Also, the metric definitions themselves are not always clear on what they measure. Cohesion metrics for example, as defined by Chidamber and Kemerer [11] are not clear on whether or not to include fields from super-classes or constructor. We resolved this issue by looking at the difference between the start situation and the new situation only, and not at the absolute values of the metrics.

Parts of our implementation that deviated from other sources may have been beneficial for decoupling and cohesiveness. The description of relational behavior given by Hannemann and Kiczales [18] was not specifically aimed at relations. Our decision to use this description may have led to a wrong implementation. Pearce and Noble do however add behavior the same way to RAL relations in [31]. Our decision to support preservation of the order of tuples deviates from the definition of a relation by Jackson in [21] as well as the implementation of RAL. Pearce and Noble do suggest the support for sorted tuples in [31]. To ascertain the validity of these decisions and of our code quality in general we had an expert in first-class relations review our code.

The logic that was moved to first-class relation constructs may have hidden it from the metric tools but it may still be required to be considered for coupling and cohesion. Further research is required for new metrics that do take first-class relations into account.

Our newly gained insight about behavior depends largely on the presence in JUnit of instances of the design patterns Observer and Composite and how these had been implemented. Gamma et al [14] for example describe using a callback function to retrieve state in the Observer pattern but JUnit uses argument passing instead. Further research involving other design patterns or other implementations of those design patterns can help with refining our knowledge of first-class relations.

One can argue that the extent of the functionality that has been placed in the relational behavior methods exceeds what is appropriate for a relation. And that in doing so, first-class relations start to resemble objects. However, one can just as well argue that we did not go far enough. Balzer et al in [4] for example go one step further and state that all behavior should be placed in the relation, leaving objects with only getters and setters thus degrading objects to entities. We believe that we have shown that for two cooperating objects, which are otherwise unaware of each other, their interaction should not be placed in one of the objects. We believe also that we have shown that if one object depends on the other, behavior should be placed in the depending object.

7.2 Future Work

First-class relations require more empirical case studies. Most studies have been theoretical dissertations. These concentrated mostly on well-known design patterns such as the Observer pattern. However, the bulk of relations do not stem from design patterns but are ordinary property relations. As a consequence the latter have been neglected so far.

The full potential of first-class relations has not been uncovered by our study. This may be attributed to the fact that with orthodox ways of programming relations remain in the background. Developing a new software program using first-class relation could help us understanding the full potential of first-class relations.

8. ACKNOWLEDGMENTS

Our thanks go to Stephanie Balzer for her feedback on our code and her help for pointing us in the right direction for this paper. Furthermore, our thanks go to our supervisor, Jurgen Vinju, who with his diligence kept pointing us in the right direction.

9. REFERENCES

[1] Aracic, I., Gasiunas, V., Mezini, M., Ostermann, K.: An Overview of CaesarJ, Transactions on Aspect-Oriented Software Development I. LNCS, Vol. 3880, (2006) 135-173.

[2] Arisa, VizzMaintenance, http://www.arisa.se/products.php (2010).

[3] Balzer, S., Gross, T.R.: Verifying Multi-Object Invariants with Relationships, ECOOP (2011).

[4] Balzer, S., Gross, T.R., Eugster, P.: A Relational Model of Object Collaborations and Its Use in Reasoning About Relationships, ECOOP (2007).

[5] Bieman, J.M., Kang, B.K.: Cohesion and reuse in an object-oriented system. Proceedings of the 1995 Symposium on Software Reusability (1995).

[6] Bierman, G.M., Wren, A.: First-class relationships in an object-oriented language. In: ECOOP. Volume 3586 of LNCS, Springer (2005) 262-286.

[7] Breivold, H.P., Larsson, M., Component-Based and Service-Oriented Software Engineering: Key Concepts and Principles, 33rd EUROMICRO Conference on Software Engineering and Advanced Applications (2007)

[8] Cacho, N., Sant'Anna, C., Figueiredo, E., Garcia, A., Batista, T., Lucena, C.: Composing Design Patterns: A Scalability Study of AOP. In: AOSD (2006).

[9] Cade, M., Roberts, S.: Sun Certified Enterprise Architect for J2EE Technology Study Guide, 2nd Edition (2010).

[10] Chen, P.P.: The entity-relationship model: toward a unified view of data. ACM Transactions on Database Systems I:1 (March 1976).

[11] Chidamber, S. R., Kemerer, C. K.: A Metrics Suite for Object Oriented Design. *IEEE Transactions on Software Engineering*, Vol. 20 (1994).

[12] Eder, J., Kappel, G., Schrefl, M.: Coupling and Cohesion in Object-Oriented Systems.Technical Report, University of Linz, Institut für Informationssysteme (1995).

[13] Erl, T.: *SOA Principles of Service Design*, The Prentice Hall Service-Oriented Computing Series from Thomas Erl (2007).

[14] Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995).

[15] Grand, M.: Patterns in Java, Volume 1, A Catalog of Reusable Design Patterns Illustrated with UML (1999).

[16] Guéhéneuc,Y.G., Albin-Amiot, H.: Recovering binary class relationships: Putting icing on the uml cake. In 19nd Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM, (2004).

[17] Hall, K. (Virtual Machinery): JHawk 5 Documentation-Metrics Guide, version 1.0, (2010).

[18] Hannemann, J., Kiczales, G.: Design pattern implementation in Java and AspectJ. In OOPSLA, ACM (2002).

[19] Hitz, M., Montazeri, B.: Measuring Coupling and Cohesion in Object-Oriented Systems. Proc. Int. Symposium on Applied Corporate Computing, (1995) 25-27.

[20] Hohpe, G., Woolf, B.: Enterprise Integration Patterns: Designing, Building and Deploying Messaging Solutions (2003).

[21] Jackson, D.: Software Abstractions: Logic, Language and Analysis, The MIT Press (2006).

[22] Jacobson, I., Booch, G., Rumbaugh, J.E.: The Unified Software Development Process. Addison-Wesley, Reading (1999)

[23] Lee, J.S., Bae, D.H.: An enhanced role model for alleviating the role-binding anomaly. Software: Practice and Experience, 32(14): 1317-1344 (2002).

[24] Nelson, S.: First-Class Relationships for Object-Orientated Programming Languages (2008).

[25] Nelson, S., Pearce, D.J., Noble, J.: Implementing First Class Relationships in Java, RAOOL (2008).

[26] Noble, J.: Basic relationship patterns. In EuroPLOP Proceedings (1997).

[27] Noble J., Grundy, J.: Explicit relationships in object-oriented development. In Proceedings of Technology of Object-Oriented Languages and Systems (TOOLS) (1995).

[28] OMG: Relationship Service Specification, Joint Object Services Submission, OMG 94-5-5, (1994).

[29] Østerbye, K.: Design of a class library for association relationships. In LCSD (2007).

[30] Parnas, D.: On the Criteria To Be Used in Decomposing Systems into Modules, Communications of the ACM, Vol. 15, No. 12 (December 1972).

[31] Pearce, D.J., Noble, J.: Relationship aspects. In: AOSD, ACM (2006) 75-86.

[32] Rumbaugh, J.: Relations as semantic constructs in an object-oriented language. In: OOPSLA, ACM (1987) 466-481.

[33] Rumbaugh, J.: Controlling propagation of operations using attributes on relations. In OOPSLA '88: Conference proceedings on Object-oriented programming systems, languages and applications (1988).

[34] Tsantalis, N.: Design Pattern Detection, http://java.uom.gr/~nikos/pattern-detection.html (2010).

[35] Tsantalis, N., Chatzigeorgiou, A., Stephanides, G., Halkidis, S. T.: "Design Pattern Detection Using Similarity Scoring", IEEE Transactions on Software Engineering, vol. 32, no. 11 (2006).

[36] Virtual Machinery: JHawk 5, http://www.virtualmachinery.com/jhawkprod.htm (2010).

Appendix A UML Class Diagram of JUnit

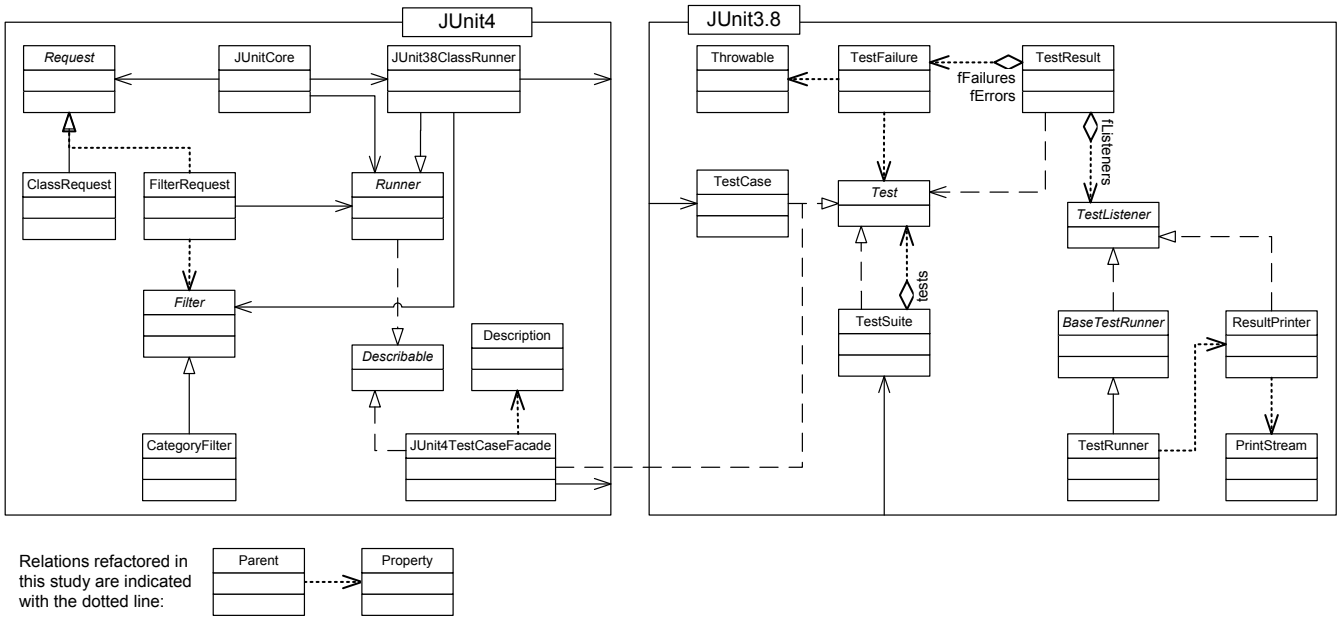

Figure 24 UML-like Class Diagram of JUnit Showing Objects Relevant to Our Research

Appendix B Observer Pattern

```
public aspect ResultObserver extends SimpleStaticRel<TestResult, TestListener> {

	// Participant methods
	public void TestResult.addObserver(TestListener o) {
		ResultObserver.aspectOf().add(this, o);
	}

	// Behavior reactive
	protected pointcut subjectAdds(TestResult s, Test test, Throwable t) :
		(call (void TestResult.addError(Test, Throwable))  ) && target(s) && args(test, t);

	after(TestResult s, Test test, Throwable t) returning: subjectAddError(s, test, t) {
		@SuppressWarnings("unchecked")
		Iterator iter= from(s).iterator();
		while (iter.hasNext()) {
			((TestListener) (iter.next())).addError(test,t);
		}
	}

	protected pointcut subjectAddFailure(TestResult s, Test test, AssertionFailedError e) :
		(call (void TestResult.addFailure(Test, AssertionFailedError))  ) && target(s) && args(test, e);

	after(TestResult s, Test test, AssertionFailedError e) returning: subjectAddFailure(s, test, e) {
		@SuppressWarnings("unchecked")
		Iterator iter= from(s).iterator();
		while (iter.hasNext()) {
			((TestListener) (iter.next())).addFailure(test,e);
		}
	}

	protected pointcut subjectStartTest(TestResult s, Test test) :
		(call (void TestResult.startTest(Test))  ) && target(s) && args(test);

	after(TestResult s, Test test) returning: subjectStartTest(s, test) {
		@SuppressWarnings("unchecked")
		Iterator iter= from(s).iterator();
		while (iter.hasNext()) {
			((TestListener) (iter.next())).startTest(test);
		}
	}

	protected pointcut subjectEndTest(TestResult s, Test test) :
		(call (void TestResult.endTest(Test))  ) && target(s) && args(test);

	after(TestResult s, Test test) returning: subjectEndTest(s, test) {
		@SuppressWarnings("unchecked")
		Iterator iter= from(s).iterator();
		while (iter.hasNext()) {
			((TestListener) (iter.next())).endTest(test);
		}
	}
}
```

Figure 25 Observer Pattern Implemented as First-Class Relation

Appendix C Composite Pattern

```
public privileged aspect TestComposite extends CompositeRelation<TestSuite, Test> {

	// Add Test interface so that others know TestSuite will now behave like Test
	declare parents: TestSuite implements Test;

	/*
	 * Behavior active
	 */
	public void TestSuite.run(final TestResult result) {

		@SuppressWarnings("unchecked")
		Enumeration enu = Collections.enumeration(TestComposite.aspectOf().from(this));
		while (enu.hasMoreElements()) {
			if (result.shouldStop())
				break;
			this.runTest(((Test) (enu.nextElement())), result);
		}
	}

	private void TestSuite.runTest(Test test, TestResult result) {
		test.run(result);
	}

	public void ActiveTestSuite.run(TestResult result) {
		this.setActiveTestDeathCount(0);
		super.run(result);
		this.waitUntilFinished();
	}

	private void ActiveTestSuite.runTest(final Test test, final TestResult result) {
		Thread t= new Thread() {
			@Override
			public void run() {
				try {
					test.run(result);
				} finally {
					ActiveTestSuite.this.runFinished();
				}
			}
		};
		t.start();
	}

	public int TestSuite.countTestCases() {
		int testCases= 0;

		@SuppressWarnings("unchecked")
		Enumeration enu = Collections.enumeration(TestComposite.aspectOf().from(this));
		while (enu.hasMoreElements()) {
			testCases += ((Test) (enu.nextElement())).countTestCases();
		}

		return testCases;
	}
}
```

Figure 26 Composite Pattern Implemented as First-Class Relation

```
public class TestSuite {

	private String fName;

	public TestSuite() {
	}

	public TestSuite(String name) {
		setName(name);
	}

	public String getName() {
		return fName;
	}

	public void setName(String name) {
		fName= name;
	}

	@Override
	public String toString() {
		if (getName() != null)
			return getName();
		return super.toString();
	 }
}
```

Figure 27 TestSuite Object after Implementation of First-Class Relation

Appendix D Command Pattern

```
public privileged aspect RequestCommand extends SimpleStaticOneToOneRel3<Filter, Request> {

	declare parents: org.junit.runner.manipulation.Filter implements ReceiverRole;
	declare parents: Request implements CommandRole;

	// Structure: participant method
	public void Filter.addRequest(Request request) {
		RequestCommand.aspectOf().add(this,request);
	}

	public void Request.addFilter(Filter filter) {
		RequestCommand.aspectOf().add(filter,this);
	}


	// Behavior active
	public Runner getRunner(Request req) {
		Filter filter = RequestCommand.aspectOf().getFrom(req);
		try {
			Runner runner= req.getRunner();
			filter.apply(runner);
			return runner;
		} catch (NoTestsRemainException e) {
			return new ErrorReportingRunner(Filter.class, new Exception(String
				.format("No tests found matching %s from %s", filter
					.describe(), req.toString())));
		}
	}

	// Participant behavior methods
	public Runner Request.getRunner() {
		return RequestCommand.aspectOf().getRunner(this);
	}

	public Runner Filter.getRunner() {
		return RequestCommand.aspectOf().getRunner(RequestCommand.aspectOf().getTo(this));
	}
}
```

Figure 28 Command Pattern Implemented as First-Class Relation

Appendix E Test Case Relations

```
public aspect DescribableDescription extends SimpleStaticOneToOneRel<Describable, Description> {

	declare parents: JUnit4TestCaseFacade implements Describable;

	// Participant maintenance methods
	public Description Describable.getDescription() {
		return DescribableDescription.aspectOf().getTo(this);
	}
	public JUnit4TestCaseFacade.new(Description d) {
		DescribableDescription.aspectOf().add(this,d);
	}

	// Behavior active.
	public String toString(Describable d) {
		return DescribableDescription.aspectOf().getTo(d).toString();
	}

	// Participant behavior method
	public String Describable.toString() {
		return DescribableDescription.aspectOf().toString(this);
	}
}
```

Figure 29 Relation between JUnit4TestCaseFacade and Description Implemented as First-Class Relation

```
public aspect ResultPrinterWriter extends SimpleStaticOneToOneRel<ResultPrinter, PrintWriterRole> {


	declare parents: java.io.PrintStream implements PrintWriterRole;

	// Participant constructor
	public ResultPrinter.new(PrintStream ps) {
		ResultPrinterWriter.aspectOf().add(this,ps);
	}

	// Participant maintenance method
	public PrintStream ResultPrinter.get() {
		return (PrintStream)ResultPrinterWriter.aspectOf().getTo(this);
	}

	// Behavior active
	public void println(ResultPrinter rp) {
		((PrintStream)ResultPrinterWriter.aspectOf().getTo(rp)).println();
	}
	public void println(ResultPrinter rp, String string) {
		((PrintStream)ResultPrinterWriter.aspectOf().getTo(rp)).println(string);
	}
	public void print(ResultPrinter rp, String string) {
		((PrintStream)ResultPrinterWriter.aspectOf().getTo(rp)).print(string);
	}

	// Participant behavior methods
	public void ResultPrinter.println() {
		ResultPrinterWriter.aspectOf().println(this);
	}
	public void ResultPrinter.println(String string) {
		ResultPrinterWriter.aspectOf().println(this, string);
	}
	public void ResultPrinter.print(String string) {
		ResultPrinterWriter.aspectOf().print(this, string);
	}
}
```

Figure 30 Relation between ResultPrinter and java.io.PrintStream Implemented as First-Class Relation

```
public aspect TestFailureThrowable extends SimpleStaticOneToOneRel<TestFailure, Throwable> {

	// Relation operations, notice that other partner object already has these methods
	public String exceptionMessage(TestFailure tf) {

		Throwable t = TestFailureThrowable.aspectOf().getTo(tf);
		return t.getMessage();
	}

	public boolean isFailure(TestFailure tf) {
		return TestFailureThrowable.aspectOf().getTo(tf) instanceof AssertionFailedError;
	}

	public void printStackTrace(TestFailure tf, PrintWriter writer) {

		Throwable t = TestFailureThrowable.aspectOf().getTo(tf);
		t.printStackTrace(writer);
	}

	// Participant behavior methods
	public String TestFailure.exceptionMessage() {

		return TestFailureThrowable.aspectOf().exceptionMessage(this);
	}

	public boolean TestFailure.isFailure() {

		return TestFailureThrowable.aspectOf().isFailure(this);
	}

	public void TestFailure.printStackTrace(PrintWriter writer) {

		TestFailureThrowable.aspectOf().printStackTrace(this, writer);
	}
}
```

Figure 31 Relation between TestFailure and Throwable Implemented as First-Class Relation

```
public aspect TestFailureTest extends SimpleStaticOneToOneRel3<TestFailure, Test> {

	// Participant maintenance methods
	public Test TestFailure.getTest() {
		return TestFailureTest.aspectOf().get(this);
	}

	public TestFailure.new(Test failedTest, Throwable thrownException) {
		TestFailureTest.aspectOf().add(this, failedTest);
		TestFailureThrowable.aspectOf().add(this,thrownException);
	}
}
```

Figure 32 Relation between TestFailure and Test Implemented as First-Class Relation

```
public aspect TestResultErrors extends SimpleStaticOneToManyRel<TestResult, TestFailure> {

	// Participant maintenance methods
	public synchronized void TestResult.addError(Test test, Throwable t) {
		TestResultErrors.aspectOf().add(this,new TestFailure(test, t));
	}

	public synchronized int TestResult.errorCount() {
		return TestResultErrors.aspectOf().size(this);
	}

	public synchronized Enumeration<TestFailure> TestResult.errors() {
		return Collections.enumeration(TestResultErrors.aspectOf().from(this));
	}
}
```

Figure 33 Relation between TestResult and TestFailure (failures) Implemented as First-Class Relation

```
public aspect TestResultFailures extends SimpleStaticOneToManyRel2<TestResult, TestFailure> {

	// Participant maintenance methods
	public synchronized void TestResult.addFailure(Test test, AssertionFailedError e) {
		TestResultFailures.aspectOf().add(this, new TestFailure(test, e));
	}

	public synchronized int TestResult.failureCount() {
		return TestResultFailures.aspectOf().size(this);
	}

	public synchronized Enumeration<TestFailure> TestResult.failures() {
		return Collections.enumeration(TestResultFailures.aspectOf().from(this));
	}
}
```

Figure 34 Relation between TestResult and TestFailure (errors) Implemented as First-Class Relation

```
public privileged aspect TestRunnerResultPrinter extends SimpleStaticOneToOneRel2<TestRunner, ResultPrinterRole> {

	declare parents: ResultPrinter implements ResultPrinterRole;

	// Participant constructor
	public TestRunner.new(ResultPrinter printer) {
		TestRunnerResultPrinter.aspectOf().add(this,printer);
	}

	// Participant maintenance methods
	public ResultPrinter TestRunner.get() {
		return (ResultPrinter)TestRunnerResultPrinter.aspectOf().getTo(this);
	}

	public void TestRunner.add(ResultPrinter printer) {
		TestRunnerResultPrinter.aspectOf().add(this,printer);
	}

	// Active behavior operations
	public void print(TestRunner tr, TestResult tres, long rt) {

		ResultPrinterRole rp = TestRunnerResultPrinter.aspectOf().getTo(tr);
		rp.print(tres, rt);
	}

	public void printWaitPrompt(TestRunner tr) {

		ResultPrinterRole rp = TestRunnerResultPrinter.aspectOf().getTo(tr);
		rp.printWaitPrompt();
	}

	// Participant behavior methods
	public void TestRunner.print(TestResult tres, long rt) {

		TestRunnerResultPrinter.aspectOf().print(this, tres, rt);
	}

	public void TestRunner.printWaitPrompt() {

		TestRunnerResultPrinter.aspectOf().printWaitPrompt(this);
	}
}
```

Figure 35 Relation between TestRunner and ResultPrinter Implemented as First-Class Relation

Appendix F Overview of Effects of First-Class Relations

Table 5 Relations Implemented

	Before applying first-class relation			The first-class relations and their characteristics		
#	Parent	Property	Cardinality	Name	Behavior	Structure
1	TestResult	TestListener	n-m	ResultObserver	reactive	SimpleStaticRel
2	TestSuite	Test	n-m	TestComposite	active	CompositeRelation[3]
3	Filter	Request	1-1	RequestCommand	active	SimpleStaticOneToOneRel
4	JUnit4TestCaseFacade	Description	1-1	DescribableDescription	active	SimpleStaticOneToOneRel
5	ResultPrinter	PrintStream	1-1	ResultPrinterWriter	active	SimpleStaticOneToOneRel
6	TestFailure	Throwable	1-1	TestFailureThrowable	active	SimpleStaticOneToOneRel
7	TestFailure	Test	1-1	TestFailureTest	-	SimpleStaticOneToOneRel
8	TestResult	TestFailure	1-n	TestResultErrors	-	SimpleStaticOneToManyRel
9	TestResult	TestFailure	1-n	TestResultFailures	-	SimpleStaticOneToManyRel
10	TestRunner	ResultPrinter	1-1	TestRunnerResultPrinter	active	SimpleStaticOneToOneRel2[4]

Table 6 Effects of Structure

Object	FCR	Role	MPC	CBO	FanIn	FanOut	ILCOM	TCC (%)
TestResult	ResultObserver	Parent	-2	-	-	-	-	-
TestSuite	TestComposite	Parent	-2	-	-	-	-	-
JUnit4TestCaseFacade	DescribableDescription	Parent	+1	-1	-	-1	-1	-
Description	DescribableDescription	Property	-	-1	-1	-	-	-
ResultPrinter	ResultPrinterWriter	Parent	+1	-1	-	-1	-1	-
PrintStream	ResultPrinterWriter	Property	-	-1	-1	-	-	-
TestFailure	TestFailureThrowable	Parent	-1	-1	-	-1	-1	-
Throwable	TestFailureThrowable	Property	-	-1	-1	-	-	-
TestFailure	TestFailureTest	Parent	-1	-1	-	-1	-1	-100
Test	TestFailureTest	Property	-	-1	-1	-	-	-
TestResult	TestResultErrors	Parent	-1	-	-	-	-1	-4
TestResult	TestResultFailures	Parent	-1	-1	-	-1	-1	-4
TestFailure	TestResultFailures	Property	-	-1	-1	-	-	-
TestRunner	TestRunnerResultPrinter	Parent	+1	-1	-	-1	-1	-100
ResultPrinter	TestRunnerResultPrinter	Property	-	-1	-1	-	-	-

[3] The main difference between CompositeRelation and SimpleStaticRel is that the former preserves the order of the tuples.

[4] SimpleStaticOneToOneRel2 replaces key-value pairs instead of throwing an AssertionError when trying to add the same key a second time.

Table 7 Effects of Behavior

Object	FCR	Role	MPC	CBO	FanIn	FanOut	ILCOM	TCC (%)
TestResult	ResultObserver	Parent	-	-1	-	-1	-1	+19
TestListener	ResultObserver	Property	-	-1	-1	-	-	-
TestSuite5	TestComposite	Parent	-	-1	-	-1	-	+50

Table 8 Effects of Participant Methods

Object	FCR	Role	MPC	CBO	FanIn	FanOut	ILCOM	TCC (%)
All clients	ResultObserver	Client	-1	-	-	-	-	-
All clients	TestComposite	Client	-1	-	-	-	-	-
CategoryTest	RequestCommand	Client	-1	-	-	-	-	-
MaxStarterTest	RequestCommand	Client	+1	-	-	-	-	-
All clients	DescribableDescription	Client	-1	-	-	-	-	-
JUnit4TestCaseFacade	DescribableDescription	Parent	-1	-	-	-	-	-
All clients	ResultPrinterWriter	Client	-1	-	-	-	-	-
ResultPrinter	ResultPrinterWriter	Parent	-1	-	-	-	-	-
All clients	TestFailureTest	Client	-1	-	-	-	-	-
TestFailure	TestFailureTest	Parent	-1	-	-	-	-	-
All clients	TestResultErrors	Client	-1	-	-	-	-	-
TestResult	TestResultErrors	Parent	-1	-	-	-	-	-
TestResult	TestResultFailures	Parent	-1	-	-	-	-	-
All clients	TestRunnerResultPrinter	Client	-1	-	-	-	-	-
TestRunner	TestRunnerResultPrinter	Parent	-1	-	-	-	-	-

[5] These values were measured without the interference of the static methods. They were removed from the before and after situation. With the static methods TCC decreased to almost zero.

Appendix G Effect of First-Class Relation on Observer and Composite Pattern

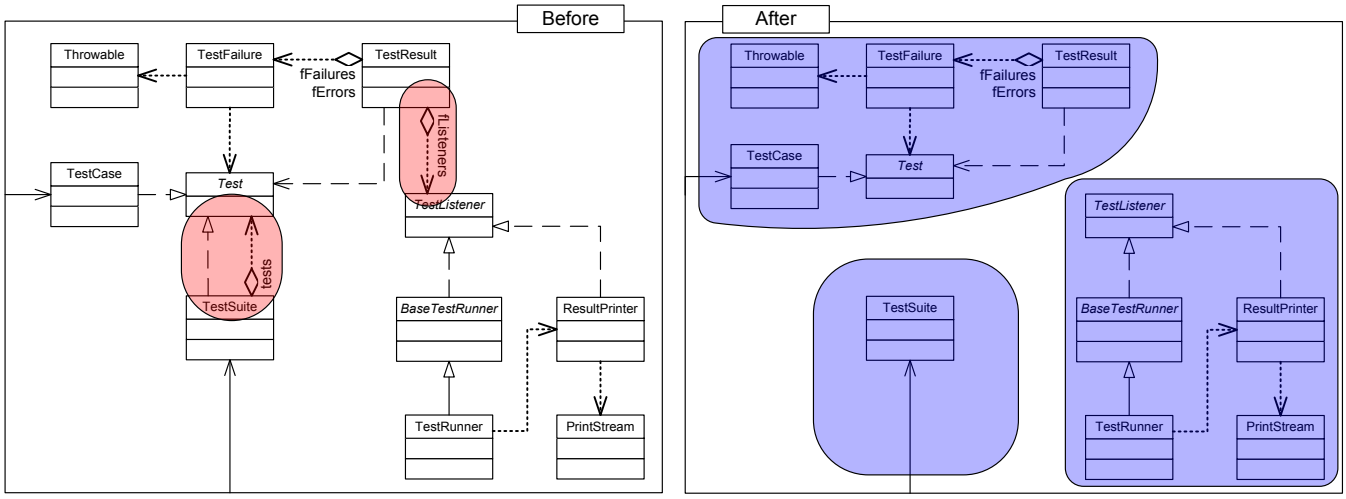

Figure 36 UML-like Class Diagram of Part of JUnit after Refactoring Observer and Composite pattern with First-Class Relations