Design Pattern Density Defined

Dirk Riehle
SAP Research, SAP Labs LLC
3412 Hillview Ave, 94304 Palo Alto, CA, U.S.A.
+1 650 215 3459

dirk@riehle.org, www.riehle.org

ABSTRACT

Design pattern density is a metric that measures how much of an object-oriented design can be understood and represented as instances of design patterns. Expert developers have long believed that a high design pattern density implies a high maturity of the design under inspection. This paper presents a quantifiable and observable definition of this metric. The metric is illustrated and qualitatively validated using four real-world case studies. We present several hypotheses of the metric's meaning and their implications, including the one about design maturity. We propose that the design pattern density of a maturing framework has a fixed point and we show that if software design patterns make learning frameworks easier, a framework's design pattern density is a measure of how much easier it will become.

Categories and Subject Descriptors

D.1.5: Object-oriented Programming, D.2.8: Metrics – Complexity Measures, D.2.11: Software Architectures – Patterns.

General Terms

Measurement, Design.

Keywords

Design patterns, design pattern density, role modeling, collaboration-based design, inheritance interface, object-oriented framework, object-oriented design, framework maturity, JUnit, JHotDraw, object-oriented case study.

1 INTRODUCTION

JUnit is a widely-adopted unit testing framework for Java, developed by Kent Beck and Erich Gamma. In the discussion of JUnit 3.8's design, the authors state:

> "Notice how TestCase, the central abstraction in the framework, is involved in four patterns. Pictures of mature object designs show this same 'pattern density'. The star of the design has a rich set of relationships with the supporting players." [1]

This statement articulates the long-held belief of expert software developers that mature frameworks consist of a higher than average number of design pattern instances, or, in short, "exhibit a high design pattern density". However, this claim has never been validated empirically nor has it been formalized to allow for such validation. The reason is simple: It is difficult to define and measure a metric like design pattern density as relevant case studies are hard to come by and laborious to carry out one oneself.

This paper presents a quantitative definition of design pattern density so that we can track its value in the evolution of a given framework. The metric is applied to four case studies which are then interpreted based on the results. The paper presents multiple hypotheses using this metric, including the one about design maturity, and discusses these hypotheses using the instrument developed in this paper. This paper, however, *does not* validate these hypotheses but rather leaves this to future work [34].

The paper presents an enhanced definition of collaboration-based design [2] a.k.a. role modeling [4] to define a quantitative measure of functionality in a class model. Object collaborations are used as the atomic unit of functionality. This makes it easy to assess the number of design pattern instances in a given design. The calculation of a framework's design pattern density becomes the percentage of collaborations that are pattern instances.

The contributions of this paper are:

- An enhanced definition of collaboration-based design that can cope with inheritance interfaces;
- A quantitative (and measurable) definition of a new metric called 'design pattern density';
- A qualitative evaluation of the metric using four real-world case studies;
- The discussion of multiple relevant hypotheses about object-oriented frameworks.

Again, the hypotheses themselves are not being validated in this paper. The main contribution of the paper is to be the first to precisely define the metric, provide an instrument for assessing it, and illustrate its potential usefulness.

OOPSLA '09, October 25-29, 2009, Orlando, Florida, U.S.A.

In *Proceedings of the 2009 Conference on Object-Oriented Programming Systems, Languages, and Applications* (OOPSLA Onward! '09). ACM Press, 2009. Forthcoming.

Section 2 introduces the enhanced version of collaboration-based design that is used in this work. Section 3 introduces the design pattern density metric and applies it to a running example. Section 4 presents three more case studies. Section 5 presents multiple hypotheses that can now be defined more precisely than before and discusses conclusions that can be drawn from the metric, the case studies, and their data. Section 6 discusses the limitations of this work and addresses future work. Section 7 discusses related work and how it addresses the issues presented in this paper. Section 8 concludes the paper.

2 PATTERNS AND COLLABORATIONS

Section 3 defines 'design pattern density' to be the percentage of a framework's functionality that can be explained as design pattern instances. For this, we need a measure of functionality on the level of granularity of design patterns so that we can represent and measure that functionality. We can then determine which parts of a design are design pattern instances and which are not.

This section defines the notion of object collaboration as the atomic unit of functionality with which to measure the number of design pattern instances in a given framework.

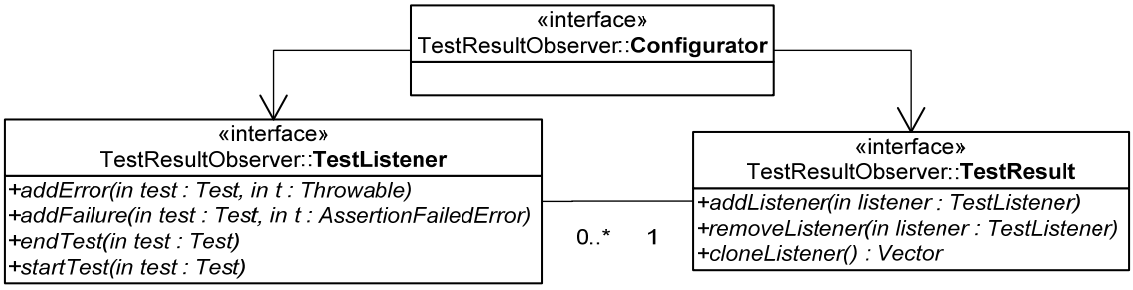

Figure 1: The TestResultObserver collaboration from JUnit 3.8.

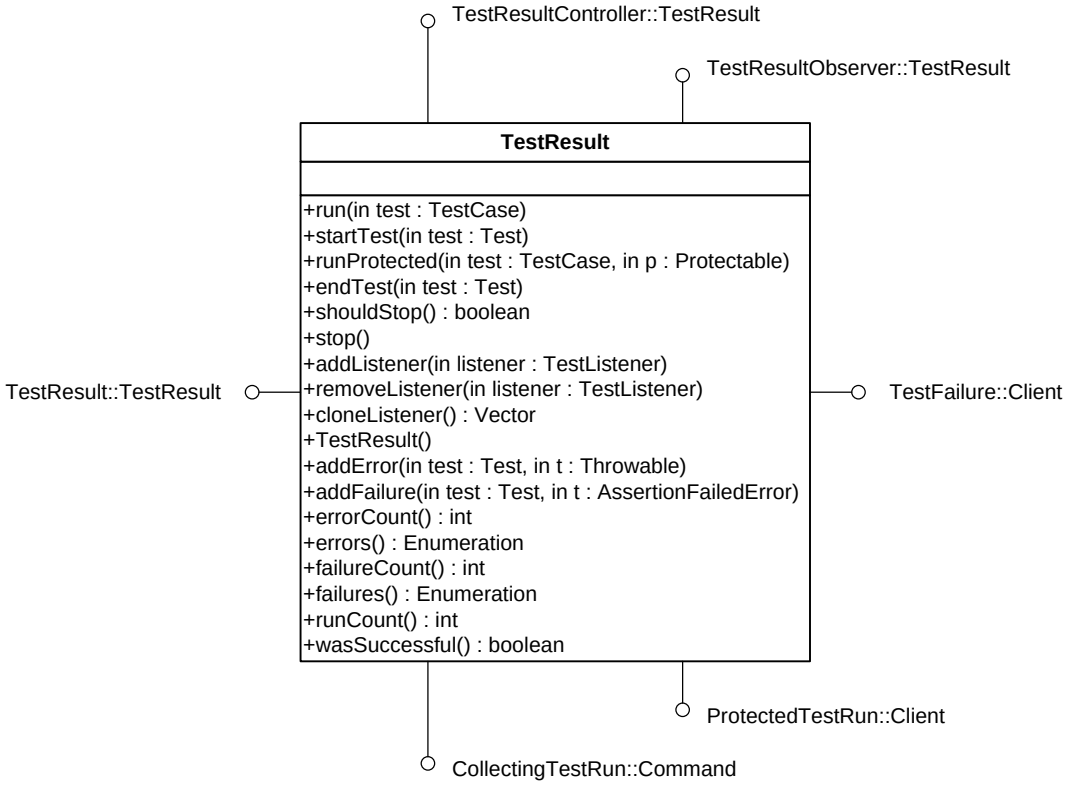

Figure 2: The TestResult class and the different roles its instances play.

2.1 Patterns and Granularity

Software patterns vary greatly in their granularity (from architectural styles through design patterns to programming idioms) as well as in what aspect of a design they address (object flexibility, concurrency, persistence, etc.) Here, we focus on the level of granularity of the classic design patterns (hence *design* pattern density) [5] and we ignore aspects like concurrency and persistence. Section 6 discusses the consequences of this limitation.

The level of granularity addressed by the classic design patterns is the class and method level. This level is of a finer granularity than architectural styles like pipes and filters [6] and it is of a coarser granularity than programming idioms like how to write a for-loop in a given programming language [7]. Thanks to the Design Patterns book, this is the best investigated level of patterns.

Most patterns (but not all) in the Design Patterns book are about object collaborations: How responsibilities are distributed across classes such that by configuring their instances a specific purpose can be achieved. By admission of their authors [8] and as increasingly found in follow-up work, the notion of participant in many pattern descriptions is more accurately called a role, and the focus is on object collaboration rather than class structure. We have previously shown how to reinterpret the classic design patterns using a collaboration-based approach [9] [10] [11].

However, not all design patterns are about object collaborations; some are about the structural aspects of a design and how flexibility is reached using inheritance (for example, Factory Method) or how behavior can be made pluggable (for example, Null Object). In the next subsections, we present an enhanced version of collaboration-based design that can be used to represent design patterns even in such situations.

2.2 Collaboration-Based Design

Collaboration-based design was born as role modeling [4]. It is related to (but independent of) the CRC (Class Responsibility Collaboration) card approach to design [3]. Collaboration-based design has made its way into UML where it is called collaborations [12]. These modeling techniques vary significantly in their details and no single technique dominates collaboration-based design. The presentation here is based on [13] which in turn is based on [4] with its formalization in [32].

Table 1. Data from the collaboration-based design view of the JUnit 3.8 framework.

Collaboration name	Number of Collaborations	Number of Roles in Collaboration	Number of Methods in Collaboration	Is it a pattern? If so which?
TestCase	1	2	4	-
TestSuite	1	2	4	-
TestSuiteTestCreation	1	2	4	-
TestRun	1	2	1	Command
TestCaseTestRun	1	2	2	-
TestSuiteTestRun	1	2	1	-
TestHierarchy	1	3	13	Composite
TestResult	1	3	9	Collecting Parameter
TestResultController	1	2	2	-
TestResultObserver	1	3	7	Observer
CollectingTestRun	1	2	4	Command
ProtectedTestRun	1	3	2	Adapter
TestRunMethod	1	2	4	Template Method
Assertions	1	2	38	-
TestFailure	1	2	6	-
AssertionFailedError	1	2	2	-
ComparisonFailure	1	2	3	-
ComparisonCompactor	1	2	2	Strategy
CompactMethod	1	2	6	Composed Method
Total	**19**	**42**	**114**	**9**

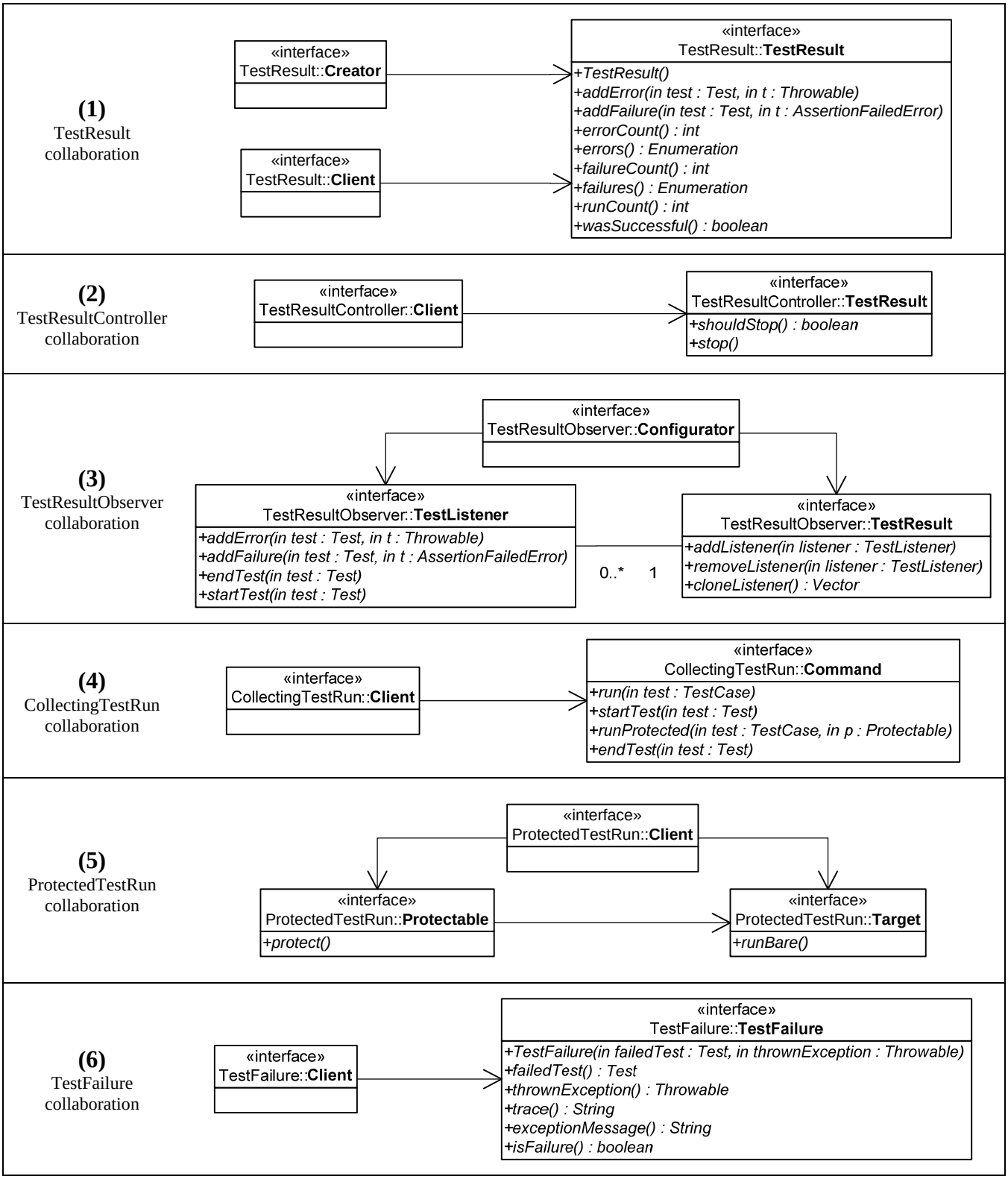

Figure 3: Six collaborations in which the TestResult class is involved in.

This paper uses the JUnit framework as a running example [33]. JUnit is a framework for writing unit tests in Java. It is available in source code form. The framework has had significant industry impact yet is small enough to make for a good case study. We use version 3.8, the last version for which an official design discussion by the authors is available [1]. We focus on the junit.framework classes only. The discussion in this paper is based on our own method-by-method documentation of JUnit 3.8 using collaborations [14].

This paper uses the UML concept of interface to represent a role and the UML concept of package to scope a collaboration (keeping its constituents, the roles, together). Compared with UML 2.x as well as our own definition [13] this is a simplification; however, it helps focus the paper and the metric definition.

In collaboration-based design, objects play roles that define how these objects collaborate to achieve exactly one well-defined purpose. A role is scoped by its collaboration and it cannot see outside its boundaries.

A role is a type that defines the behavior of an object within a collaboration and a collaboration is a grouping of roles that defines how objects behind these roles are allowed to interact.

For example, Figure 1 shows the TestResultObserver collaboration from JUnit 3.8. This collaboration defines how a TestResult object allows for registration and unregistration of TestListener objects interested in what's happening with the TestResult object. For that purpose, TestListener objects provide callback methods that the TestResult object can invoke. It does so when a test run starts, when it ends, and when a failure occurs. This collaboration is an application of the Observer pattern, where the TestResult role represents the Subject participant and the TestListener role represents the Observer participant. An additional Configurator role handles the registration process.

Earlier attempts viewed role modeling as a competitor to class-based approaches [4]. In contrast to this, we cast it as an extension of class-based modeling. In our definition, a class model like the JUnit framework can be described by composing collaborations. The composition is carried out by assigning roles to classes and by defining how a class composes and implements the roles assigned to it.

The roles define the visible behavior of instances of the class within a particular collaboration, and the class defines how its instances bring together the expected behavior in these different contexts as one integrated whole.

Thus, the class view is complementary to the role view. The role view defines how an instance behaves in one particular context (collaboration) and the class view defines how state models and control flow is integrated between the different collaborations. Basically, the class composes the roles to form the class [13].

The class TestResult in JUnit 3.8, for example, provides the TestResult::Testresult role. This denotes a role called TestResult that is part of a collaboration called TestResult. (Following UML, the collaboration is named first, followed by double colons, followed by the role name.) The collaboration name disambiguates the role name by scoping it, so there can be several roles called TestResult. The role name, in this case TestResult, is the same as the class name. This typically happens if the service provided by the role is exactly the primary service associated with the class. In such a case, the collaboration represents a domain-specific client/service collaboration. This double and triple use of the same name may at first appear confusing but works well within context.

Other roles assigned to the TestResult class are the TestResultController::TestResult role, the TestResultObserver::TestResult role, the TestFailure::Client role, the CollectingTestRun::Command role, and the ProtectedTestRun::Client role. One may wonder again about the frequent use of TestResult. For example, in the TestResultObserver collaboration, the TestResult role represents the applied Subject participant of the Observer pattern, so why not call it Subject rather than TestResult? This is a decision left to the developer, and we follow the advice to be specific (TestResult) rather than generic (Subject).

The TestResult class is shown in Figure 2, and its roles and their collaborations are shown in Figure 3. The class interface shown in Figure 2 is the sum of the different methods defined by the roles in the participating collaborations.

Please note that using UML interfaces to represent a role doesn't imply that on the code-level any such interface exists. More often than not, in JUnit the methods defined by roles are directly embedded in a Java interface or Java class, as Figure 2 illustrates.

Of the six collaborations that TestResult participates in, three are instances of design patterns: The TestResultObserver collaboration is an instance of the Observer pattern, the CollectingTestRun is an instance of the Command pattern, and the ProtectedTestRun is an instance of the (Object) Adapter pattern.

Table 1 shows the statistics of the junit.framework classes. It lists 19 collaborations that were found when analyzing JUnit 3.8's junit.framework classes (TestCase, TestSuite, TestSuiteCreation, etc.) Of those, 9 collaborations were identified as instances of design patterns using the original authors' [1] and our own judgment [14]. The table also lists the number of roles a collaboration offers: Typically it is two, sometimes three; atomic collaborations with more than three roles are rare. The number of methods per collaboration is fairly evenly distributed from 1 to 9, with one anomaly, the Assertions collaboration. The Assertions collaboration provides the methods from the Assert class, which is using a shopping list approach to interface design [15] to let developers express assertions about the program being tested.

2.3 Collaborations and Granularity

Collaborations are design elements of the same granularity as the classic design patterns. They have two important properties:

- Well-defined collaborations are orthogonal to each other and can be composed easily;
- Class models can be derived completely by composing collaborations; nothing falls between the cracks.

The JUnit 3.8 documentation using collaborations accounts for every single method in the framework and identifies it as part of a specific collaboration [14].

Any given collaboration has to serve one defined purpose. From this, an upper level of granularity follows. (Composite collaborations and patterns [11] are outside the scope of this paper.)

2.4 Collaborations and Inheritance

Traditional and collaboration-based design typically focuses on the collaboration between objects. Self-delegation, intra-object communication, and class inheritance structures have played little or no role. However, the inheritance interface that superclasses define as a contract of interaction with their subclasses is also important. We need to extend the notion of collaboration-based design with a way of specifying and using inheritance interfaces. Without such enhancement, we would not be able to completely capture white-box or gray-box frameworks like JUnit.

The key insight is that one object may play several roles within the same collaboration. For example, an object may be observing itself or it may be using some basic service using self-delegation. Hence, we can capture intra-object communication with the same approach as inter-object communication: We simply define roles and collaborations as done before, but allow for role assignments to the same class, even if the same instance of that class may end up playing multiple roles. Then, a role in that collaboration can represent all or parts of the inheritance interface.

Template Method is an example. In this pattern, a superclass defines a public method that spreads its work over multiple non-public methods within the class. These other methods are part of the class' inheritance interface and are visible only to subclasses.

Figure 4 shows an application of the Template Method pattern using a textual notation for collaborations. It defines three roles, two of which are assigned to the TestCase class. The public runBare method is a template method that makes use of primitive methods that constitute parts of the class inheritance interface.

The use of collaborations to define collaborations that are internal to class hierarchies is new to collaboration-based design. Yet, it is critical to fully account for the functionality of a design. Without it, we would be ignoring how class hierarchies work internally and would only be able to grasp the collaboration between separate objects. In short, the only frameworks we would be able to fully explain using collaborations would be black-box frameworks. However, most frameworks are a mixture between black-box and white-box frameworks.

3 DESIGN PATTERN DENSITY

Using the instrument laid out in Section 2, we can now provide a quantitative and measurable definition of design pattern density:

Table 2: Summary data from the JUnit 3.8 analysis.

JUnit 3.8 Case Study	
Number of classes/interfaces	11
Number of collaborations	19
Number of pattern instances	9
Number of roles in total	42
Ratio roles per class/interface	3.8
Design pattern density	**47%**

The design pattern density of an object-oriented framework is the percentage of its collaborations that are design pattern instances.

For example, as Table 1 shows, the core junit.framework classes are composed from 19 collaborations. Of these, 9 are instances of design patterns. Hence, as shown in Table 2, the design pattern density of JUnit 3.8 is 9/19 or 47%.

The metric 'design pattern density' is simple, precise, complete, and measurable. It is simple and precise because only a basic calculation is needed. It is complete and measurable due to the completeness properties of the enhanced collaboration-based design method described in Section 2.

One advantage of this metric is that it has a simple comparison relation already built-in. With values on a linear scale from 0 to 100%, comparing two density values becomes trivial.

Interpreting the metric and a comparing two of its values is not trivial. What does it mean to say that the pattern density of JUnit 3.8 is 47% while the pattern density of JHotDraw 5.1 (a case study in Section 4) is 71%? Section 5 discusses this further.

In general, it is safe to assume that the pattern density of a framework will remain well below 100%, because in any given framework there will be at least one client/service collaboration whose chief purpose is to provide a domain-specific service.

In addition, we need to distinguish the design pattern density of a framework's interface architecture from the design pattern density of its complete design.

The design pattern density of a framework's interface architecture is the percentage of those collaborations that are design pattern instances and that are defined in the framework's interface architecture.

The design pattern density of a framework's complete design is the percentage of all collaborations in the framework's complete design, including both interface architecture and implementation structures.

```
// collaboration definition
public collaboration TestRunMethod {

   free role Client {
      // no methods
   }

   role TemplateMethod {
      public void runBare() throws Throwable;
   }

   role PrimitiveMethods {
      protected void runTest() throws Throwable;
      protected void setUp() throws Exception;
      protected void tearDown() throws Exception;
   }
}

// role to class assignments
public class TestCase provides TemplateMethod,
    PrimitiveMethods ...
```

Figure 4: A collaboration-based definition of the Template Method application in JUnit.

The interface architecture of a framework defines its interfaces and interface classes and structures their core collaborations. If someone wants to understand how to use a framework, he or she typically turns to the interface architecture first.

The implementation architecture consists of the implementation classes and their class hierarchy and how these classes implement the interface architecture. The implementation architecture becomes relevant mostly if someone wants to extend the framework.

It makes sense to measure both the design pattern density of the interface architecture and the complete design. The implementation cannot be understood without the interface architecture, so we don't measure the density of the implementation structures only. Thus, we distinguish between the design pattern density of the interface architecture and the complete design.

4 CASE STUDIES

In addition to JUnit, we used the metric and its underlying instrument to gather data from three other frameworks.

1. The Geo system, a metalevel-architecture-based implementation of a distributed object system [13].
2. The KMU Desktop framework used to build tools in a financial risk assessment application [13].
3. The JHotDraw framework for building graphical editors [13] [16] [17].

The Geo framework and the KMU Desktop framework are the result of a major revision and hence in their second release JHotDraw had already undergone several major releases and was analyzed in version 5.1.

Table 3: Summary data from three case studies.

Case study	[1]	[2]	[3]
Number of interfaces and interface classes	17	13	20
Number of collaborations	34	20	28
Number of pattern instances	20	12	20
Number of roles assigned to classes	75	44	66
Ratio roles per class/interface	4.4	3.4	3.3
Design pattern density (interface architecture)	**59%**	**60%**	**71%**
[1] - The Geo framework [2] - The KMU Desktop framework [3] - The JHotDraw core framework			

4.1 Case Study Data

All case studies were documented using collaboration-based design. The referenced documentation provides the details on the collaborations, the class models, and how the class models are composed from the collaborations.

Table 3 shows summary data and the design pattern densities from the three new case studies, excluding the JUnit case study. The three new case studies were assessed on an interface architecture level, so the numbers given in Table 3 are interface architecture design pattern densities.

Table 4 summarizes the pattern densities and assigns a maturity level to each framework. The maturity level is a simple integer value 1-3, where 1 represents "new", 2 means "revised" and 3 means "mature". These values are based on this paper's author's assessment of the frameworks and are kept simple, because they can only give a qualitative maturity indication.

These frameworks exhibit a high design pattern density.

4.2 Collaborations and Functionality

The design pattern density metric is based on collaborations. In terms of granularity, collaborations are between methods and use cases. Thus, collaborations can serve as a measure of functionality in a framework on a medium-granularity level: Collaborations are not as small as methods and not as coarse-grained as use cases. Moreover, how many methods there are in a framework can depend strongly on a developer's programming style. Use cases on the other hand can vary drastically in terms of the actual functionality they provide.

Tables 2, 3, and 4 and the documentation data show that collaborations are of fairly even size. The number of roles per collaboration is between 2 and 3 in the given frameworks, and the number of methods per role is between 2 and 4. The number of case studies available is too small to measure the variance in these numbers; however, it seems unlikely that they vary drastically.

Table 4: The maturity level, pattern density, roles per collaboration data of the case studies.

Case study	Maturity Level (1-3)	Design Pattern Density	Number of roles per collaboration
Assessed on the interface architecture level			
Geo System	2	59%	2.21
KMU Desktop	2	60%	2.20
JHotDraw	3	71%	2.36
Assessed on the complete design level			
JUnit	3	47%	2.21

Thus, we have a strong indicator that collaborations can serve as a more reliable measure of functionality in framework design than other approaches like method count or use cases.

We pick up this observation in Section 5 below where we frame it as the hypothesis that the design pattern density metric is a measure of ease of learning a framework.

While not explored further in this paper, this also shows that collaborations might be a good proxy for complexity and estimated effort in implementing a framework.

5 HYPOTHESES

The definition of the new metric 'design pattern density' is simple, precise, complete, and measurable. But is it useful?

To show that a metric is a reliable proxy of something, we need to develop a sufficiently large body of quantitative data and correlate the metric with the property of interest, assuming that this property is known for the case studies. This correlation can then be used to predict the property for future case studies, assuming that it is easier to assess the metric than the property.

The body of case studies presented in this paper is too small to derive statistically significant conclusions from it. As a consequence, we can only use the metric, its underlying instrument, and the qualitative indicators from the case studies to discuss hypotheses of interest. This in itself is valuable if the hypotheses are new or can now be framed in more precise terms than before.

This section looks at both old and new hypotheses that have become tractable for the first time. Previously, these hypotheses (so they had been postulated) were unclear and not open to validation. We can frame them now in such a way that we can validate (or invalidate) them in future work.

5.1 Framework Maturity

The original motivation of this work came out of the expert opinion that "a high design pattern density indicates a high maturity of a design." (This is our framing of the introducing quotation of Kent Beck and Erich Gamma).

Hypothesis 1: As a framework matures, the design pattern density of a framework increases.

We equate age with maturity, so this can only be a statistical relationship, as it is (easily) imaginable that an aging framework's design pattern density drops temporarily through intermediate revisions.

Given that the design pattern density will always stay below 100%, and given that a framework can't grow beyond any boundaries, we can sharpen hypothesis 1:

Hypothesis 2: As a framework matures, its design pattern density approaches a fixed point value.

This fixed point would be the design pattern density of the "perfect design" for the framework at hand. We would expect (following Hypothesis 1) that the sequence of design pattern density values created by successive revisions of a framework approaches the fixed point from below.

This hypothesis isn't saying anything about a specific framework's fixed point for its design pattern density though. Moreover, the assumption is that this fixed point value varies from framework to framework.

A corollary is that the fixed points of the design pattern densities of all conceivable frameworks follow a probability distribution:

Hypothesis 3: The fixed points of the design pattern densities of all possible frameworks form a random variable that follows a probability distribution.

Here, the distinction between the design pattern density of a framework's interface architecture and the design pattern density of a complete design becomes important:

Hypothesis 4: The distribution of the random variable 'design pattern density of a framework's interface architecture' has a higher mean than the distribution of the random variable 'design pattern density of a framework's complete design'.

This should be easy to see, since the design pattern density of any interface architecture is generally higher than the design pattern density of a complete design. This is because most of a framework's flexibility is expressed in the interface architecture. Adding implementation classes proportionally increases the number of non-design-pattern instance collaborations in a framework over the number of pattern instances. For example, adding a new subclass like Rectangle to the Figure abstraction in the JHotDraw framework for graphical editors does not change the interface architecture. It does add, however, a client/service collaboration that lets clients make use of the specifics of Rectangle objects.

It is the author's best guess that for the design pattern densities of interface architectures the mean is likely to be at around 70% with a standard deviation of around 5%.

It seems safe to assume that both densities are highly correlated:

Hypothesis 5: The random variable 'design pattern density of a framework's interface architecture' is highly correlated with the random variable 'design pattern density of a framework's complete design'.

For this reason most hypotheses apply equally to both densities and there is usually no need to explicitly say which density we are talking about.

These hypotheses finally lead us to a precise formulation of the expert opinion quoted in the beginning of this paper:

Hypothesis 6: The difference between the design pattern density of a specific framework version and its fixed point is a measure of the framework's maturity.

We suggest that if the value is far below the fixed point, the framework is still in its early stages. On the other hand, if the value is higher than the fixed point, the framework may have been over-engineered. Over-engineering can happen when developers learn about design patterns for the first time and apply them everywhere, appropriate or not.

Hypothesis 6 is a precise framing of the original motivation of this paper and one that can be validated. Once we determine the fixed point distribution through further case studies, we will be

able to measure how mature a given framework version is and can make decisions with more confidence than before.

5.2 Ease of Learning Frameworks

In Section 4, we discussed how collaborations can be used as a measure of functionality in a framework. As a consequence, the design pattern density of a framework measures how much of that framework's functionality can be captured as instances of design patterns.

For example, in JUnit 3.8's case, with a design pattern density of 47%, that very percentage of its functionality can be understood as design pattern instances.

Assuming that design patterns make learning, using, and documenting frameworks easier, faster, and less error-prone, the design pattern density metric also becomes a measure of how much easier, faster, etc. patterns make it for developers to work with frameworks.

Hypothesis 7: The closer a framework version's design pattern density is to its fixed point, the easier on average the framework is to learn and use.

For example, if consistent documentation of a framework using patterns is known to make learning a framework 50% faster with respect to those parts that are described using patterns, learning JUnit would be sped up by 23.5% (47% of its functionality is being learned at twice the speed over not using patterns).

A key assumption is that patterns have been applied only where sensible, meaning the framework has not been over-engineered to artificially inflate its design pattern density.

Thus, a framework's design pattern density may not only be a measure of the framework's maturity, but equally importantly, a measure of how much easier it will be for developers to learn the framework and put it to use.

6 LIMITATIONS AND FUTURE WORK

The work presented in this paper makes a number of assumptions that restrict the applicability of the metric and its underlying instrument. This section discusses these assumptions as well as future work to follow this paper. In this discussion, we need to distinguish the metric from the instrument from the case studies.

6.1 Definition of Metric

It is a strength of the design pattern density metric that it is precise and explicit about what it is based on, namely collaborations. This is also its largest restriction, as there are patterns that cannot be neatly captured using collaborations.

Concurrency is one example. Synchronization patterns, work distribution patterns, and different concurrency models can not be captured well using the instrument presented in this paper.

The first question is whether they should be. Some of these patterns are clearly not on a design level, but rather on an architecture or programming level.

More importantly though, it does not seem sensible to mix (largely) orthogonal design aspects into one metric. A better approach might be to have a pattern density metric for each major type of aspect in a given system, and to define an overall design pattern density metric as a composite metric based on these different aspect metrics.

Under this assumption, the design pattern density metric presented in this paper covers the aspect "flexibility" (by object composition and distribution of responsibilities between classes), while concurrency, persistence, and other aspects will be assessed using different metric definitions.

This forms a base for future work, as it will be interesting to define aspect-specific patterns, a pattern density metric for these patterns, and instruments for assessing these densities in a given framework.

Another issue is pattern granularity. What about those programming idioms that developers routinely apply? We argue that they are on a different level of abstraction and should be assessed independently from a design level metric like design pattern density.

A final restriction of this paper is that the metric definition focuses on frameworks rather than the more general notion of class model. This choice was deliberate as the design pattern density metric provides most of its value when applied to reusable code components like frameworks.

6.2 Application of Instrument

The main instrument used in assessing the metric in a given framework is an enhanced method for using collaboration-based design as presented in this paper.

This instrument has the following two shortcomings with respect to assessing the design pattern metric:

1. Few developers actually use collaboration-based design, so any such documentation is after the fact;
2. The recognition of design patterns in a collaboration-based documentation may be subjective.

Shortcoming 1 about the limited use of collaboration-based design is a problem as it may be difficult to document a framework using collaborations after the fact. However, we contend that as a framework matures, the different purposes why objects collaborate become clearer, and the deconstruction of the framework into its constituting object collaborations will become easier.

An indicator of such a progress is the recognition of design patterns itself: Since a pattern has one well-defined purpose, recognizing patterns goes lock-step with a collaboration-based decomposition (and reconstruction) of a framework.

A more severe problem is the recognition of a pattern itself. One might argue that if this is left to subjective opinion, anything goes, and an assessed metric's value is not worth much. (In particular if commercial interests develop around this metric.)

Most design pattern descriptions are done in prose and remain ambiguous. While experts can generally determine which pattern has been applied by looking at the context of the pattern and matching pattern intent with code, a machine may remain confused about whether it is looking at a Bridge, Strategy, or (Object)

Adapter, if all it has is the Structure Diagram from the Design Patterns book.

This problem can only be alleviated through more formal and precise definitions of what constitutes a (design) pattern. Gil and Maman's work on automated recognition of micro-patterns is a step into the right direction [18]. However, their patterns are not on a design level and extending it to that level seems difficult. Zdun and Avgeriou's work on primitives for modeling design patterns has the advantage of being able to capture the many variants in which patterns can come, but their work has not yet been applied to automated recognition of design patterns [19].

To the extent that we make progress towards formalizing design patterns (without taking away their inherent variability) we will be able to make progress towards automated calculation of the metric design pattern density.

6.3 Case Studies and Hypotheses

The case studies discussed in this paper represent a non-trivial amount of work. However, since they only provide four data points (design pattern densities), more are needed to validate the hypotheses presented in Section 5.

One type of studies that needs to be done are longitudinal studies that track the design pattern density of a framework over its many versions. Using such studies, it should be possible to determine whether framework-specific fixed points of the design pattern density metric exist, and how the metric approaches these fixed points.

Another study that should be done is one that determines the probability distribution of the fixed points of framework design pattern densities. The underlying assumption is obviously that such fixed points exist. Further work should then investigate how the distribution relates to the fixed points of evolving frameworks.

Also, the design pattern density needs to be correlated with other properties of frameworks under investigation, most notably maturity, quality, and ease of learning.

All of these studies represent non-trivial efforts. Still, they are necessary to validate the hypotheses presented in this work. Undertaking these studies seems worthwhile given the conjectured power of design pattern density in predicting such important qualities as framework maturity and ease of learning.

Thanks to the open source movement, today we have sufficiently large quantities of materials at hand (frameworks in their many versions) which we can analyze so that these studies have become feasible.

7 RELATED WORK

Related work falls mainly into three categories: design metrics, design patterns, and modeling techniques.

Henderson-Seller's early work on object-oriented metrics provides a set of fundamental metrics useful in a wide variety of circumstances [20]. None of those, however, are about design patterns and object-oriented frameworks. A 2003 survey by Purao and Vaishnavi provides a collection of 375 different object-oriented metrics [21]. Among those metrics is not a single one that is about design patterns or object-oriented frameworks.

One explanation for the lack of design pattern metrics is given by Stein et al. who argue that design-level metrics cannot be derived from the code and are therefore more difficult to handle [22]. Based on Etzkorn and Delugach's work on so-called "semantic metrics" (design-level metrics) [23] they show how to derive such metrics from design documentation. This work is related to the work presented in this paper as we also work off design documentation. However, we derived this design documentation by analyzing the source code, so we do not agree with the assumption that design and implementation are completely separate.

Much of the work on object-oriented metrics is geared towards aiding refactoring of object-oriented designs [24]. Metrics are used as quality measures to indicate how to improve a legacy system [25]. Design pattern metrics, however, still have to enter this space.

More work has been spent on formalizing and automatically recognizing design patterns in existing code. Kramer and Prechelt provide one of the first implementations to automatically recognize design patterns in code [26]. However, with just the Structure Diagram information from the Design Patterns book, pattern definitions remain ambiguous and experimental results suffer. Gil and Maman therefore focus on "micro-patterns," which are structural patterns that can be defined precisely and that can be found in code [18]. However, micro-patterns are still a level of abstraction below design patterns.

Zdun and Avgeriou took a different approach by not focusing on comprehensive pattern specifications but rather on the primitives that can be employed to specify patterns [19]. This approach has the advantage that it can better cope with the breadth of variants that expert developers typically see in design pattern. Ideally, a combination of Gil/Maman's and Zdun/Avgeriou's work could lead to automated recognition of design patterns in code. Such work would benefit the metric assessment presented in this paper as it would reduce some of the subjectivity in the process.

Yet more work on detecting design patterns is available [36] [37] [38] [39]. The quality of the detection results depends as much on the quality of the chosen pattern formalization as it depends on the actual detection algorithm. We see two potentially useful use-cases of such tools for assessing design pattern densities: The first approach creates high-quality documentation by keeping the human in the loop. Human experts decide on which pattern instance is at hand, supported by the tool's provision of an automatically generated candidate pool. The design pattern metric of a framework assessed this way is likely to have high statistical significance. The second approach creates low-quality documentation automatically, but plenty of it. This second use case may be useful for establishing the overall design pattern density distribution of object-oriented frameworks.

A core aspect of the work presented in this paper is collaboration-based design, the employed modeling technique to capture framework functionality and design pattern instances. Our enhanced version is based on Reenskaug et al.'s original work and related to CRC cards [4] [3].

Role modeling and collaboration-based design are not the only approaches to breaking up frameworks into smaller composable pieces. Other more recent approaches are traits and fragments. A trait is a set of methods and their implementation that can be composed with other traits to form a class [27]. Most of the work on traits, much like with the work on subject-oriented programming [28] focuses on the difficulties encountered when composing code. Traits are like roles but do not come with a notion of collaboration, which is essential in dealing with design patterns.

Closer to collaborations is the work on fragments, which are descriptions of code fragments together with the code and how they can be composed to form or add to a framework [29]. Like collaborations, fragments can therefore be used to represent design pattern instances in framework design. As with traits and subject-oriented programming, fragments are more concerned with data and code in a bottom-up fashion, while collaborations focus on meeting domain modeling problems and worry less about implementation, coming top-down.

In Section 6.1 we argue that the design pattern density metric presented in this paper is really only the most prominent member in a family of pattern density metrics. The family members are defined by what aspect of the framework they address, be it flexibility, concurrency, etc. The most promising attempt at modeling and implementing such aspects is aspect-oriented programming (AOP) [30]. As Hannemann and Kiczales have shown, AOP can be used to implement design pattern instances in code [31]. Denier and Cointe's case study on JHotDraw shows that AOP can help capture design patterns; unfortunately, they do not provide an explicit definition of design pattern density [35].

Given the comprehensiveness of aspect-oriented programming in comparison with other approaches, we view it as the most promising implementation technology for traditional object-oriented frameworks. As the CaesarJ programming language shows, extending AOP with new constructs lets us make collaborations explicit in a natural way [40]. Making design patterns explicit is the next obvious step after this.

8 CONCLUSIONS

This paper presents a new metric called design pattern density. We investigate this metric because of common expert belief that this metric can serve as a reliable proxy for the maturity of object-oriented frameworks and can aid decision making about whether to use a framework or not.

The paper presents a novel extension of collaboration-based design as the instrument to calculate the metric's value in a given framework. The paper makes the metric not only precise but also measurable for the first time. To do so, we show how collaboration-based design cannot only be used to capture inter-object collaborations, but also how it can be extended to capture class inheritance interfaces.

The metric and its instrument are applied to four case studies, followed by a discussion of their quantitative assessment. Based on the case studies and their discussion, seven hypotheses are presented about design pattern density, framework maturity, and ease of learning of frameworks. It is left to future work, however, to actually validate these hypotheses.

Key hypotheses are that a framework's design pattern density has a fixed point and that the fixed pointfs of all conceivable frameworks form a random variable that follows a probability distribution. Thanks to the metric and its underlying instrument, these hypotheses have lost their vagueness and have become tractable in future studies.

ACKNOWLEDGEMENTS

I would like to thank Erich Gamma, Mario Lopes, James Noble, and Wolf Siberski for providing helpful comments and feedback for this paper.

REFERENCES

[1] Kent Beck and Erich Gamma. JUnit: A Cook's Tour. Available from http://junit.sourceforge.net/doc/cookstour/cookstour.htm

[2] Rebecca Wirfs-Brock and Brian Wilkerson. "Object-Oriented Design: A Responsibility-Driven Approach." In Proceedings of the 1989 Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '89). ACM Press, 1989: Pages 71-75.

[3] Rebecca Wirks-Brock, Brian Wilkerson, and Lauren Wiener. Designing Object-Oriented Software. Prentice Hall, 1990.

[4] Trygve Reenskaug, Per Wold, and O.A. Lehne. Working with Objects: The OOram Software Engineering Method. Prentice Hall, 1996.

[5] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1995.

[6] David Garlan and Mary Shaw. An Introduction to Software Architecture. Prentice Hall, 1994.

[7] James O. Coplien. Advanced C++ Programming Styles and Idioms. Addison Wesley, 1991.

[8] Ralph Johnson, John Vlissides. Personal Email Communication, 2002.

[9] Dirk Riehle. A Role-Based Design Pattern Catalog of Atomic and Composite Patterns Structured by Pattern Purpose. Ubilab Technical Report 97.1.1. Zurich: UBS AG, 1997.

[10] Dirk Riehle, Roger Brudermann, Thomas Gross, and Kai-Uwe Mätzel. "Pattern Density and Role Modeling of an Object Transport Service." ACM Computing Surveys 32, 1es (March 2000): Article No. 10.

[11] Dirk Riehle. "Composite Design Patterns." In Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages and Applications (OOPSLA '97). ACM Press, 1997: Pages 218-228.

[12] The Object Management Group (OMG). UML 2.x Specification. OMG, 2007. See http://www.uml.org.

[13] Dirk Riehle. Framework Design: A Role Modeling Approach. Ph.D. Thesis, No. 13509. ETH Zürich, 2000.

[14] Dirk Riehle. JUnit 3.8 Documented Using Collaborations. In Software Engineering Notes Volume 33, Issue 2 (March 2008), Article No. 5. ACM Press, 2008.

[15] Bertrand Meyer. Object-Oriented Software Construction. Prentice Hall, 1988.

[16] Erich Gamma. Advanced Design with Java and Patterns. Tutorial held at the 1998 JAOO Conference. Available from http://www.riehle.org/blogs/research/2007/2007-01-03.html

[17] Kent Beck and Erich Gamma. JHotDraw---Patterns Applied. Tutorial held at the 1997 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '97). ACM Press, 1997.

[18] Joseph (Yossi) Gil and Itay Maman. "Micro Patterns in Java Code." In Proceedings of the 2005 Conference on Object-Oriented Programming, Systems, Languages and Applications (OOPSLA '05). ACM Press, 2005: Pages 97-116.

[19] Uwe Zdun and Paris Avgeriou. "Modeling Architectural Patterns Using Architectural Primitives." In Proceedings of the 2005 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '05). ACM Press, 2005: Pages 133-146.

[20] Brian Henderson-Sellers. Object-Oriented Metrics: Measures of Complexity. Prentice-Hall, 1995.

[21] Sandeep Puraoand Vijay Vaishnavi. "Product Metrics for Object-Oriented Systems." ACM Computing Surveys Vol. 35, No 2 (June 2003). ACM Press: Pages 191-221.

[22] Cara Stein, Letha Etzkorn, and Dawn Utley. "Computing Software Metrics from Design Documents." In Proceedings of the 2004 ACM South East Conference (ACMSE '04). ACM Press, 2004: Pages 146-151.

[23] Letha Etzkorn and H Delugach. "Towards a Semantic Metrics Suite for Object-Oriented Design." In Proceedings of the 34th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS 2000). Pages: 71-80.

[24] Joshua Kerievsky. Refactoring to Patterns. Addison-Wesley, 2005.

[25] Serge Demeyer. Stéphane Ducasse, and Oscar Nierstrasz. "Finding Refactorings via Change Metrics." In Proceedings of the 15th Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2000). ACM Press, 2000: Pages: 166-177.

[26] Christian Krämer and Lutz Prechelt. "Design Recovery by Automated Search for Structural Design Patterns in Object-Oriented Software." In Proceedings of the Working Conference on Reverse Engineering. IEEE Press, 1996: Pages 208-215.

[27] Nathanael Schärli, Stéphane Ducasse, Oscar Nierstrasz and Andrew Black. "Traits: Composable Units of Behavior." In Proceedings of the European Conference on Object-Oriented Programming (ECOOP '03). Springer Verlag, 2003: Pages 248-274.

[28] Harold Ossher, Matthew Kaplan, William Harrison, Alexander Katz and Vincent Kruskal. "Subject-Oriented Composition Rules." In Proceedings of the 1995 Conference on Object-Oriented Programming, Systems Languages and Applications. ACM Press, 1995: Pages: 235-250.

[29] George Fairbanks, David Garlan, and William Scherlis. "Design Fragments Make Using Frameworks Easier." In Proceedings of the 2006 Conference on Object-Oriented Programming, Systems Languages and Applications. ACM Press, 2006: Pages 75-88.

[30] Gregor Kiczales, John Lamping, Anurag Mendhekar, Chris Maeda, Cristina Lopes, Jean-Marc Loingtier, and John Irwin. "Aspect-Oriented Programming." In Proceedings of the 1997 European Conference on Object-Oriented Programming (ECOOP 1997). Springer Verlag: Pages 220-242.

[31] Jan Hannemann and Gregor Kiczales. "Design Pattern Implementation in Java and AspectJ." In Proceedings of the 2002 Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA 2002). ACM Press, 2002: Pages 161-173.

[32] Egil Andersen. Conceptual Modeling of Objects: A Role Modeling Approach. Ph.D. Thesis, University of Oslo, 1997.

[33] Kent Beck and Erich Gamma. Source code available from http://www.junit.org.

[34] Dirk Riehle et al. "Design Pattern Density Validated." In preparation.

[35] Simon Denier and Pierre Cointe. "Understanding Design Pattern Density with Aspects: A Case Study in JHotDraw using AspectJ." In Proceedings of the 5th International Symposium on Software Composition (SC 2006). Springer Verlag, 2006.

[36] Dirk Heuzeroth, Thomas Holl, Gustav Högström, and Welf Löwe. "Automatic design pattern detection." In Proceedings of the 11th IEEE International Workshop on In Program Comprehension, 2003. IEEE Press, 2003. Page 94-103.

[37] Rudolf Keller, Reinhard Schauer, Sebastian Robitaille, and Peter Page. "Pattern-Based Reverse-Engineering of Design Components." In Proceedings of the 21st International Conference on Software Engineering (ICSE 1999). IEEE Press. Page 226-235.

[38] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T. Halkidis, "Design Pattern Detection Using Similarity Scoring," IEEE Transactions on Software Engineering, vol. 32, no. 11 (November 2006). Page 896-909.

[39] Zsolt Balanyi and Rudolf Ferenc, "Mining Design Patterns from C++ Source Code." In Proceedings of the 2003 International Conference on Software Maintenance (ICSM '03). IEEE Press, 2003. Page 305-314.

[40] Ivica Aracic, Vaidas Gasiunas, Mira Mezini, and Klaus Ostermann. "Overview of CaesarJ." Transactions on Aspect-Oriented Software Development I (LNCS vol. 3880). Springer Verlag, 2006. Page 135-173.