Detecting Extra Relationships for Design Patterns Roles

Emiliano Tramontana
Dipartimento di Matematica e Informatica
University of Catania, Italy
tramontana@dmi.unict.it

ABSTRACT

When using design patterns, classes and their relationships should conform to prescribed solutions. However, several ad-hoc design choices are often made and, as a result, *variants* of the design pattern appears, which, though keeping most of the original structure intact, could alter the intended aim. Variants are more likely to emerge in large software systems and detecting them during development is not trivial, due to the many classes and relationships.

This paper proposes a solution to automatically detect variants by leveraging the relationships that classes have with roles of design patterns. Our approach comprises some rules for detecting the *extra* relationships and provides a support to check an implementation. The checks put into place act as an executable documentation for the desired design patterns, warning developers should any change occur.

1. INTRODUCTION

While developing large software systems, some of the arising design issues are solved by resorting to well-known design patterns and, accordingly, some classes play the prescribed design pattern roles [9]. However, the design must accommodate many other issues and further classes and relationships emerge, which are superimposed on the chosen design patterns. As a result, some relationships between classes could be arranged in such a way to become detrimental to some features the design patterns try to achieve: e.g. relationships between classes could appear in places where the design patterns would instead prescribe a clean separation between classes. A class could also play multiple roles in one or several design patterns, which could indicate that the class intertwines several concerns [26]. Therefore, though some of the classes are mapped according to design patterns, some *spurious* (i.e., unwanted) features could emerge that could alter the aim of some design patterns.

Generally, refactoring techniques are used to perform *preventive* changes, hence improving the quality of design and code, and some of these techniques have been tailored to detect code smells that indicate a suitable design pattern [8, 17]. Some other techniques have been proposed to detect occurrences of design patterns within systems, e.g., [27]. However, such approaches are not aiming at analysing the extra relationships that classes may have with design patterns roles, hence are not concerned with finding *spurious* relationships. Similarly to detection, a verification approach checks that required characteristics are exposed by the respective roles of a design pattern on a minimal implementation [3]. However, further (system-wide) extra relationships with related classes, possibly deviating from the proper structure, are not considered.

We propose to set up some *rules* for classes playing a role in a design pattern and other related classes to automatically check system-wide relationships and assess deviations (from the proper version) on relationships. The proposed approach is novel, as it aims at finding spurious relationships that might appear as a side-effect of some design issues, i.e., once other characteristics typical of a design pattern have been properly put into place. Moreover, our approach lets developers define when a design pattern variant can be considered appropriate.

Rules to be checked can be implemented as *pointcut*-based expressions. *Pointcuts* are language constructs available for the *aspect-oriented programming* (AOP) paradigm [18] and AspectJ language [19]. They can be used to find matchings between defined rules and a design. Therefore, AOP greatly reduces the effort otherwise needed for developing a tool that analyse design and code. Nevertheless, our approach can be considered general and independent of the language used for implementing design patterns and the analysing tool.

Our approach provides developers with an *executable documentation* for the desired design choices, i.e., defined design rules can be automatically and continuously checked at design time, hence without any overhead at runtime. Detecting variations prevents introducing a design solution that can bring problems in the long term.

The rest of the paper is structured as follows. The next section describes the proposed approach. Sections 3, 4, and 5 express the rules for design patterns *Adapter*, *Observer*, and *Factory Method*, respectively. Section 6 shows the results obtained on a case study. Section 7 analyses the related work, and finally conclusions are drawn in Section 8.

2. DETECTING EXTRA RELATIONSHIPS

For each design pattern, the solution describes the roles, with names and responsibilities, and their relationships. Roles are mapped into classes of the software system and class re-

lationships describe the dependencies to be created, which comprise generalisation/specialisation, method call, and instantiation [9]. Generally, for a large software system, further portions of code appear that represent additional responsibilities and, often, further relationships. Approaches detecting occurrences of a design pattern can find extra relationships, e.g., [27], however their aim is to recognise the minimum set of satisfied relationships not extra, spurious relationships.

Our approach aims at finding the following categories of *extra relationships*. We analyse for a class playing a design pattern role whether the class: (i) implements one or several interfaces, (ii) extends a superclass, (iii) is instantiated or used by several client classes, (iv) depends on other roles of the same design pattern. Which extra relationships, among the above categories, are to be considered inappropriate, hence spurious, depend on the specific design pattern. The following sections provide the definitions of inappropriate relationships for a few design patterns, the approach is however viable for others as well.

Some relationships are not mentioned by the solutions prescribed by the GoF's catalogue [9], hence for classifying some extra ones as inappropriate (or otherwise correct), we have tapped into the experience gained in using design patterns and developing components having a low degree of *coupling* and internal *complexity*, while having a high degree of *functional cohesion* [4, 6, 11, 12, 14, 22, 23, 24]. Additionally, our approach can bring the developer's point of view in, as s/he can express quality requirements for the system at hand. Hence, though some extra relationships are filtered out automatically as inappropriate, developers can fine-tune the provided filter mechanism.

Detection of extra relationships is performed automatically. Automation is needed because the relationships that a few roles for a design pattern play potentially extend to several dozens of classes in industrial software systems. Once detection rules have been conceived, a static analysis is required for checking conformance and some analysing tool can be developed for the desired programming language. In this paper, we express the rules finding extra relationships in terms of AspectJ constructs and its standard compiler suffices, hence no ad-hoc tools are needed. Software *aspects* are components allowing the developer to implement *pointcuts* and *advices*. The former provide developers with means to define rules for intercepting some points during the compilation and execution of a system, the latter are similar to methods, hence allow a sequence of operations to be implemented, which are executed when the associated pointcut matches the currently executed point [12, 13, 14, 18, 19]. Aspects and classes are compiled together by means of a *weaver*. In AspectJ, the most robust and feature-ready aspect-oriented language, based on Java, the weaver generates standard bytecode that can be executed on any JVM [1].

Using AspectJ, the developer can drive the weaver to perform some checks at compile-time, i.e., the compiler matches pointcuts and portions of a system. This is possible only when the pointcuts depend on the structure of the classes, i.e., pointcuts are independent of values of attributes, execution flow, etc. For the developers to have the results of the matching, pointcuts must be inserted into a declare construct. Therefore, our rules are described by means of the declare construct, as we aim at checking structural dependencies, which can be found at compile-time, and find

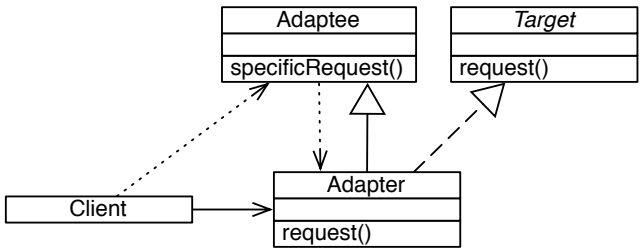

Figure 1: UML class diagram showing extra relationships, as dotted lines, between roles for Adapter

all existing occurrences of extra relationships. The following provides for each design pattern analysed the pointcuts capturing extra relationships. Pointucts are implemented in terms of the names of the roles for a design pattern. For the actual version of the aspects that can be weaved to a software system, it suffices to substitute the name of the role with that of the corresponding class playing the role.

3. ADAPTER

Design pattern *Adapter* aims at letting a client class interact with a server class, even though the interface provided by the server class is not that expected by the client class. The suggested solution consists of roles Target, Adapter, and Adaptee. Target defines the interface that the client class expects. Adapter implements the Target interface and redirects calls to the corresponding operation provided by server class Adaptee.

The following provides the extra (i.e., spurious) relationships that we propose to check for validating the implementation of design pattern *Adapter*. Figure 1 shows such extra relationships as dotted directed lines.

- *Condition 1.* Client classes should not call operations on Adaptee, only Adapter is allowed.

  Calls from client classes to Adaptee would introduce an extra relationship, bypassing Adapter, which is inappropriate because Adapter acts as *intermediary*. Dependencies between client classes and Adaptee would unexpectedly break should Adaptee change its interface. Moreover, such relationship would let a client be directly *coupled* with Adaptee role. From the functional point of view, Adapter might provide additional functionalities, such as caching, guarding, etc., bypassing it would give unexpected results.

- *Condition 2.* Client classes should not create instances of Adaptee, only Adapter is allowed.

  This follows from the same reason as above of having Adaptee hidden by Adapter and no direct *coupling* between the former and client classes. When the *Class* version of *Adapter* is used, i.e., Adapter is a subclass of Adaptee, Adapter should not instantiate Adaptee. Otherwise, for the *Object Adapter* version, only Adapter can instantiate Adaptee.

- *Condition 3.* Adaptee should neither call methods of Adapter, nor instantiate Adapter.

  This condition expresses the decision by which Adaptee should have no knowledge of its Adapter, hence no cou-

```
1  public aspect AdapterCheck {
2    // Condition 1: check calls to Adaptee
3    declare warning :
4      call(* Adaptee.*(..)) && !within(Adaptee)
5      && !within(Adapter)
6      : "Disallow calls to Adaptee's methods";

8    // Condition 2: check instantiation of Adaptee
9    declare warning :
10     call(Adaptee.new(..))
11     && !within(Adapter)
12     : "Only for the Object Adapter solution,
13     Adapter creates an instance of Adaptee";

15   // Condition 3: check calls from Adaptee to Adapter
16   declare warning :
17     (call(Adapter.new(..)) || call(* Adapter.*(..)))
18     && within(Adaptee)
19     : "Adaptee should not know about Adapter";
20 }
```

Figure 2: Aspect with pointcuts detecting extra relationships for Adapter

pling. This lets Adapter be changed without triggering changes to Adaptee.

In our solution, checks for the above conditions have been implemented as corresponding pointcuts in AspectJ. Figure 2 shows the aspect-oriented code. Each pointcut finds, at compile-time, the classes violating the corresponding condition. Violation of *Condition 1* is traced by pointcut in lines 4 to 5; *Condition 2* as pointcut in lines 10 to 11, for the *Object Adapter* version, and as line 10 only for the *Class Adapter* version; *Condition 3* as pointcut in lines 17 to 18. Designator call() matches method calls toward the signature given as a parameter and wildcards allow any occurrence to match. Designator within() matches the points found inside the class given as a parameter. Matching expressions are combined using boolean operators. Section 6 shows the findings when using the above pointcuts on a system.

4. OBSERVER

For design pattern *Observer*, the solution prescribes an interface Observer that classes named ConcreteObservers implement to be notified by a change of state within an observed class ConcreteSubject. Moreover, a Subject holds a list of Observers and notifies instances when the state of ConcreteSubject changes. ConcreteSubject is a subclass of Subject and communicates its state changes to Subject by calling method setChanged(), likewise triggers updates by calling method notifyObservers().

Figure 3 shows, as dotted directed lines, the undesired extra relationships. Each dotted line represents any category of relationships, i.e., method call, instantiation, inheritance, etc. The extra relationships are described, along with conditions and motivations, in the following.

- *Condition 1.* No instances of Subject should be created.

  This condition follows from the decision that Subject is only responsible for handling Observers. Having no other functionalities, no classes need to handle its instances, instead ConcreteSubject inherits from Subject. Violations of such a condition reveal that Subject has been given several responsibilities, hence its internal *complexity* could be considered high, as several concerns are mixed in it.

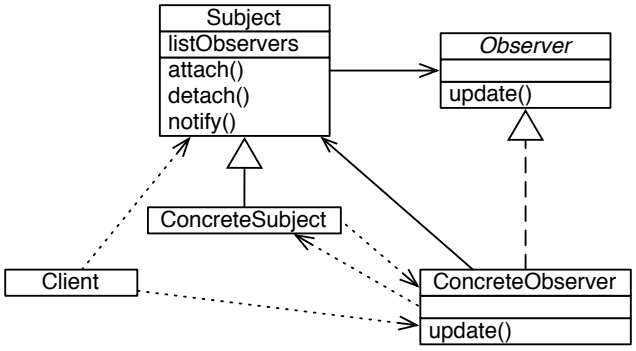

Figure 3: UML class diagram showing extra relationships, as dotted lines, between roles for Observer

- *Condition 2.* Only ConcreteSubject should invoke methods setChanged() and notifyObservers(), inherited from Subject.

  This indicates the decision to have only ConcreteSubject manage its state and notifications. Violating such a condition reveals that other classes are handling the state of ConcreteSubject, which could contrast with the information hiding principle. A client class would be better call a proper method of ConcreteSubject instead, to preserve the logic that the final decision for updating Observers is taken within ConcreteSubject. Subclasses of ConcreteSubject are considered as playing the same role as their superclass.

- *Condition 3.* ConcreteObserver should not call ConcreteSubject's methods and instead should call Subject's methods.

  The aim of *Observer* is to have a loose coupling between ConcreteObserver and ConcreteSubject, hence, when interaction is needed, ConcreteObserver should call methods by means of a variable having type Subject.

- *Condition 4.* Only Subject should invoke the notify method update() on ConcreteObserver.

  For having the desired loose coupling, calls to ConcreteObserver's methods are restricted. This condition applies, among other classes, to ConcreteSubject, i.e., the latter should not call ConcreteObserver's methods.

- *Condition 5.* ConcreteObserver should not call update() on an implementation of Observer. If a class exists that violates this condition and satisfies *Condition 4* at the same time, then the class is both a subclass of Subject and implements Observer.

  Violation of this condition means that the same class has been given the two roles: ConcreteSubject and ConcreteObserver, hence it is mixing portions of code related to different responsibilities. Possibly, the complexity of such a class is high.

```
1  public aspect ObserverCheck {
2    // Condition 1: check Subject's instantiations
3    declare warning :
4      call(Subject.new(..))
5      : "Subject should not be instantiated ";

7    // Condition 2: check state changes and updates
8    declare warning :
9      (call(* Subject.setChanged()) ||
10     call(* Subject.notifyObservers(..)))
11     && !within(Subject+)
12     : "ConcreteSubject only should change its state or
13       start notification ";

15   // Condition 3: check calls back to ConcreteSubject
16   declare warning :
17     call(* ConcreteSubject.*(..)) && within(Observer+)
18     && !within(ConcreteSubject+)
19     : "ConcreteObserver should call Subject's methods";

21   // Condition 4: check calls to ConcreteObserver
22   declare warning :
23     call(* Observer+.update(..)) && !within(Subject)
24     : "Only Subject should call ConcreteObserver";

26   // Condition 5: check ConcreteObserver's calls to
27   // other ConcreteObservers
28   declare warning :
29     call(* Observer+.update(..)) && within(Observer+)
30     : "A ConcreteObserver calls Observer";

32   // Condition 6: check ConcrObserver's calls to Subject
33   declare warning :
34     (call(* Subject.setChanged()) ||
35     call(* Subject.notifyObservers(..)))
36     && within(Observer+)
37     : "A ConcreteObserver calls Subject";
38 }
```

Figure 4: Pointcuts detecting extra relationships for Observer

- *Condition 6.* ConcreteObserver should not call Subject's methods for updating the state and start notifying Observers (other methods can be called). A class violating this condition and satisfying *Condition 2*, is both a subclass of Subject and implements Observer.

  Similarly as *Condition 5*, this would reveal a class being both the originator and destination of a notification of state change, hence two roles for the same class.

Figure 4 shows the aspect-oriented code to detect whether the above conditions have been violated by an implementation. A pointcut matching one or more classes reveals the extra relationships expressed by the corresponding condition. Each implemented pointcut follows from the negation of the condition to be satisfied.

For *Condition 3*, the implemented pointcut checks whether a call from a ConcreteObserver is performed to a ConcreteSubject (line 17). Additionally, the check excludes calls from ConcreteSubject (line 18), this latter check would be unnecessary if the two roles are implemented as two different classes. However, the case study that we used showed that this is not always true. Section 6 shows our findings.

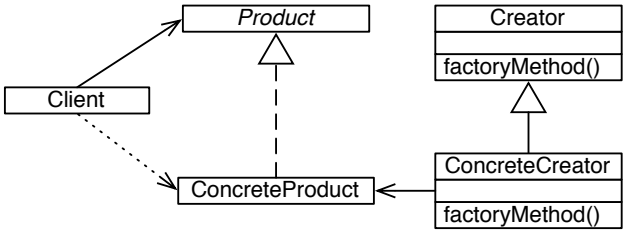

Figure 5: UML class diagram showing extra relationships, as dotted lines, between roles for Factory Method

```
1  public aspect FactoryMethodCheck {

3    // Condition 1: check instantiation of ConcreteProduct
4    declare warning :
5      call(ConcreteProduct.new(..)) && !within(Creator+)
6      : "Only Creator should instantiate a ConcreteProduct";

8    // Condition 2: check use of ConcreteProduct
9    declare warning :
10     call(* ConcreteProduct.*(..)) && !within(Creator+)
11     : "Client classes should only interact with Product";
12 }
```

Figure 6: Pointcuts detecting extra relationships for Factory Method

5. FACTORY METHOD

For design pattern *Factory Method*, a class ConcreteCreator extends an abstract class Creator and implements a method that selects and instantiates one among several ConcreteProduct classes that implement an interface Product. Figure 5 shows the extra relationships, whereas the rules provided to find them are given in the following.

- *Condition 1.* Client classes, other than a ConcreteCreator, should not instantiate a ConcreteProduct.

  This condition has been derived from the aim of the design pattern, i.e., to confine within ConcreteCreator only the decision about which ConcreteProduct to instantiate, hence supporting the information hiding principle.

- *Condition 2.* Client classes should not hold a variable of type ConcreteProduct.

  This condition checks that, according to the aim of *Factory Method*, classes ConcreteProducts are all implementing the same interface Product, without adding any public method, in such a way that ConcreteProducts are hidden and interchangeable.

Figure 6 shows the code of pointcuts that detect violations for the said conditions as lines 4 and 9 for *Condition 1* and *Condition 2*, respectively, and the finding are in Section 6.

6. ANALYSING JHOTDRAW

The above conditions have been checked for JHotDraw version 6.0 [2], a software system comprising 600 classes and 72 K LOC, which is considered a reference implementation

Adapter, Adaptee, Target	
Condition	Matching classes
DiamondFigureGeometricAdapter, DiamondFigure, Figure	
Condition 2	DiamondFigureTest, JavaDrawApp, JavaDrawApplet, UndoableToolTest
EllipseFigureGeometricAdapter, EllipseFigure, Figure	
Condition 2	EllipseFigureTest, JavaDrawApp, JavaDrawApplet, NothingApp, NothingApplet
RoundRectangleGeometricAdapter, RoundRectangleFigure, Figure	
Condition 1	RadiusHandle
Condition 2	JavaDrawApp, JavaDrawApplet, NothingApp, NothingApplet, RoundRectangleFigureTest
PolygonFigureGeometricAdapter, PolygonFigure, Figure	
Condition 1	ChopPolygonConnector, DiamondFigure, PolygonHandle, PolygonHandleTest, PolygonScaleHandle, PolygonTool, TriangleFigure
Condition 2	PolygonFigureTest, PolygonHandleTest, PolygonTool
TriangleFigureGeometricAdapter, TriangleFigure, Figure	
Condition 1	TriangleRotationHandle
Condition 2	JavaDrawApp, JavaDrawApplet, TriangleFigureTest

Table 1: JHotDraw classes violating Adapter's rules

Subject, Observer ConcreteSubject	
Condition	Matching classes
StandardDrawing, DrawingChangeListener	
Condition 1	DrawApplet, DrawApplication, DrawingChangeEventTest, JavaDrawViewer, JDOStorageFormat, NullDrawingView, StandardDrawingTest, StandardDrawingViewTest
AbstractFigure, FigureChangeListener	
Condition 2	PolygonScaleHandle, TriangleRotationHandle
Condition 3 DecoratorFigure BouncingDrawing	
Condition 4	CompositeFigure, DecoratorFigure, FigureChangeEventMulticaster, TextAreaFigure, TextFigure
Condition 5	CompositeFigure, DecoratorFigure, FigureChangeEventMulticaster, TextAreaFigure, TextFigure
Condition 6	ElbowConnection, GraphicalCompositeFigure, HTMLTextAreaFigure, LineConnection, NodeFigure, PertFigure, TextAreaFigure, TextFigure

Table 2: JHotDraw classes violating Observer's rules

for the design patterns in GoF's catalogue [9]. We have chosen JHotDraw because it can be considered an industrial software system that reveals many more interactions for design pattern roles than the minimum documented, hence it is prone not to conform to some of the conditions proposed in this paper. A trivial implementation of design patterns, such that that can be found, e.g., in [25], would instead easily conform to all conditions.

Table 1 shows the classes playing a role for design pattern *Adapter* as well as the classes that have been found to violate the conditions described in Section 3. Very few classes violate *Condition 1*, i.e., calling methods of a class playing as *Adaptee*. The small number of found classes confirms that developers aimed to have the *Adaptee* shielded by *Adapter*.

Several classes violate *Condition 2*, i.e., creating an instance of a class playing as *Adaptee*. Among such classes, the ones having the suffix Test are likely to have been implemented to test the functionalities of the *Adaptee* class, hence such a violation can be considered not a relevant problem. By visually inspecting the code of some of the classes violating *Condition 2* though not *Condition 1*, we can see that the created instance is not used.

Table 2 shows JHotDraw classes playing the roles for design pattern *Observer* and the classes matching the checks for the *Observer*'s, hence violating the conditions described in Section 4. For class StandardDrawing playing as Subject and BouncingDrawing playing as ConcreteSubject, some classes violate *Condition 1*, i.e., they create instances of class StandardDrawing. By inspecting the code, we observe that StandardDrawing offers several functionalities and is a subclass of CompositeFigure and the latter is a subclass of AbstractFigure. StandardDrawing implements more than the Subject role and for this reason it has been instantiated. A suggested refactoring would be to better isolate the Subject role, by having a separate extra class for handling additional responsibilities. Such an extra class could be acting as ConcreteSubject by using subclassing.

For class AbstractFigure playing as Subject, some classes violate *Condition 2* to *Condition 6*. For *Condition 2*, i.e., change state methods on Subject are called by two matching classes, not having a hierarchy relationships with ConcreteSubject nor Subject. It would be suggested that the change state method is redefined in ConcreteSubject to avoid this direct call.

Condition 4 and *Condition 5* are matched by the same classes because of the class hierarchy in JHotDraw. Class AbstractFigure acts as Subject and has several children classes acting as ConcreteSubject: AttributeFigure, CompositeFigure, DecoratorFigure, PolyLineFigure. Class CompositeFigure implements interface FigureChangeListener, i.e., the Observer role. This means multiple roles are played by CompositeFigure. A better separation between concerns would be achieved by separating into different classes the functionalities pertaining to the two roles, then using the notify mechanism already in place, however for additional method pairs.

The classes matching *Condition 5* or *Condition 6* are playing both roles ConcreteSubject and ConcreteObserver. This has been confirmed by visually inspecting the code.

Table 3 shows the classes playing a role for *Factory Method* and the results of the checks described in Section 5 for JHotDraw. Only three classes violate *Condition 1*, i.e., create an instance of a ConcreteProduct. Two of the classes have the suffix Test, hence it would be advisable to have the other class (StandardDrawingView) revised to use the factory method.

7. RELATED WORK

Previous works proposed some approaches to recognise which classes play a role for a design pattern [7, 15, 23,

ConcreteProduct, Creator	
Condition	Matching classes
HandleEnumerator, Figure	
Condition 1	HandleAndEnumeratorTest, HandleEnumeratorTest, StandardDrawingView

Table 3: JHotDraw classes violating Factory Method's rules

27]. Such approaches essentially match the relationships that classes have with the relationships that design pattern roles should have.

In [3], authors described the characteristics that a design pattern should exhibit by means of conditions to be checked on the source code using Prolog-like statements. In [21], authors provided a support for checking that the source code conforms to the intended structure, separately documented. Both the above works aim at representing the essential structural features that have to be found, whereas unwanted relationships are not considered, which, instead, provide leverage for our proposed checks.

The importance of indicating design decisions, which however cannot be easily documented, has been stated in [16]. Some approaches used architectural languages for documenting design choices. However, in such approaches, architecture description and code are separated and the description is at a high-level of abstraction. Hence, the produced code can not be automatically checked [5, 10].

In [20], a machine learning technique unveiled occurrences of anti-patterns, after having trained a classifier with class-related metrics and tagged classes as an anti-pattern occurrence or not. Unlike the said anti-pattern detection, our approach is very fine-grained, i.e., uncovers a single extra relationship indicating a significant variation for the design pattern. Moreover, our approach need not training set, thus is protected from a possible bias that could originate in the amount and variety of tagged classes.

In [4], authors described an approach to allow developers express many different design decisions to which classes should conform. Such design decisions are not concerned with extra relationship, nor with design patterns. Checks on the existing code is possible thanks to their fine-grained description using Java annotations and aspect-orientation.

Unlike the above related works, the approach proposed here assumes that the role played by a class in a design pattern is known and proposes to describe unwanted relationships. We focus on the extra relationships that could be found and aim at stating whether the resulting design is still conforming to the main aim of the design pattern. Moreover, our approach aims at analysing relationships that client classes have with design pattern roles, not only between roles themselves.

8. CONCLUSIONS

This paper provided definitions of rules that allow checking the proper implementations of a few well-known design patterns. Devised rules make sure that additional design choices and continuous changes during development and evolution are not adding extra relationships and roles. Although for a simple implementation of a design pattern all our rules confirm that relationships are as expected, for larger software systems this is not always true, because it is much harder to avoid conflicting design choices. Moreover, the proposed approach provides developers with a means to automatically check the correctness of a design, hence a framework that they can customise. This framework allows writing tests targeting the design and is helpful for achieving high-quality code.

Our related future work aims at conceiving useful rules finding extra relationships for additional design patterns and checking occurrences of the rules presented here, and further ones, in other large software systems, such as e.g., JUnit and ArgoUML.

Acknowledgment

This work has been supported by project JACOS funded within POR FESR Sicilia 2007-2013 framework, and project PRISMA PON04a2 A/F funded by the Italian Ministry of University and Research within PON 2007-2013 framework.

9. REFERENCES

[1] AspectJ Home page. http://www.eclipse.org/aspectj.

[2] JHotDraw Home page. http://jhotdraw.org.

[3] A. Blewitt, A. Bundy, and I. Stark. Automatic verification of design patterns in Java. In *Proceedings of international Conference on Automated software engineering (ASE)*, pages 224–232. ACM, 2005.

[4] A. Calvagna and E. Tramontana. Delivering dependable reusable components by expressing and enforcing design decisions. In *Proceedings of Computer Software and Applications Conference (COMPSAC) Workshop QUORS*, pages 493–498. IEEE, July 2013.

[5] P. C. Clements. A survey of architecture description languages. In *Proceedings of International workshop on software specification and design*. IEEE, 1996.

[6] A. Di Stefano, M. Fargetta, G. Pappalardo, and E. Tramontana. Metrics for Evaluating Concern Separation and Composition. In *Proceedings of Symposium on Applied Computing (SAC)*. ACM, March 2005.

[7] J. Dong, Y. Zhao, and Y. Sun. A matrix-based approach to recovering design patterns. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 39(6), nov 2009.

[8] M. Fowler, K. Beck, J. Brant, W. Opdyke, and D. Roberts. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.

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

[10] D. Garlan, R. Monroe, and D. Wile. Acme: an architecture description interchange language. In *CASCON First Decade High Impact Papers*, pages 159–173. IBM Corp., 2010.

[11] R. Giunta, F. Messina, G. Pappalardo, and E. Tramontana. Providing qos strategies and cloud-integration to web servers by means of aspects. *Concurrency and Computation: Practice and Experience*, 2013.

[12] R. Giunta, G. Pappalardo, and E. Tramontana. Using Aspects and Annotations to Separate Application Code from Design Patterns. In *Proceedings of Symposium on Applied Computing (SAC)*. ACM, March 2010.

[13] R. Giunta, G. Pappalardo, and E. Tramontana. AODP: refactoring code to provide advanced aspect-oriented modularization of design patterns. In *Proceedings of Symposium on Applied Computing (SAC)*, pages 1243–1250. ACM, 2012.

[14] R. Giunta, G. Pappalardo, and E. Tramontana. Superimposing roles for design patterns into application classes by means of aspects. In *Proceedings of the ACM Symposium on Applied Computing, SAC*, pages 1866–1868. ACM, March 2012. DOI: 10.1145/2245276.2232082.

[15] Y.-G. Guéhéneuc and G. Antoniol. Demima: A multilayered approach for design pattern identification. *IEEE Transactions on Software Engineering*, 34(5):667–684, 2008.

[16] N. B. Harrison, P. Avgeriou, and U. Zdlin. Using patterns to capture architectural decisions. *Software, IEEE*, 24(4):38–45, 2007.

[17] J. Kerievsky. *Refactoring to patterns*. Addison-Wesley, 2005.

[18] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J. Loingtier, and J. Irwin. Aspect-oriented programming. In *Proceedings of ECOOP*, volume 1241 of *LNCS*, 1997.

[19] R. Laddad. *AspectJ in Action: Enterprise AOP with Spring Applications*. Manning, 2nd ed., 2009.

[20] A. Maiga, N. Ali, N. Bhattacharya, A. Sabane, Y.-G. Gueheneuc, and E. Aimeur. Smurf: a svm-based incremental anti-pattern detection approach. In *Proceedings of Working Conference on Reverse Engineering (WCRE)*, pages 466–475. IEEE, 2012.

[21] K. Mens and A. Kellens. Intensive, a toolsuite for documenting and checking structural source-code regularities. In *Proceedings of CSMR*. IEEE, 2006.

[22] C. Napoli, G. Pappalardo, and E. Tramontana. Using modularity metrics to assist move method refactoring of large systems. In *Proceedings of Complex, Intelligent and Software Intensive Systems (CISIS)*, pages 529–534. IEEE, 2013.

[23] G. Pappalardo and E. Tramontana. Automatically discovering design patterns and assessing concern separations for applications. In *Proceedings of Symposium on Applied Computing (SAC)*, pages 1591–1596. ACM, 2006.

[24] G. Pappalardo and E. Tramontana. Suggesting extract class refactoring opportunities by measuring strength of method interactions. In *Proceedings of Asia Pacific Software Eng. Conference (APSEC)*. IEEE, 2013.

[25] S. Stelting and O. Maassen. *Applied Java Patterns*. Prentice Hall, 2001.

[26] E. Tramontana. Automatically characterising components with concerns and reducing tangling. In *Proceedings of Computer Software and Applications Conference (COMPSAC) workshop QUORS*, pages 499–504. IEEE, 2013.

[27] N. Tsantalis, A. Chatzigeorgiou, G. Stephanides, and S. Halkidis. Design Pattern Detection Using Similarity Scoring. *Transactions on Software Engineering*, 32(11), 2006.