ISSN 1884-0760

GRACE TECHNICAL REPORTS

Proceedings of the Third International Workshop on Software Patterns and Quality (SPAQu'09)

Hironori Washizaki, Nobukazu Yoshioka, Eduardo B. Fernandez and Jan Jürjens (editors)

GRACE-TR 2009–07 October 25, 2009

CENTER FOR GLOBAL RESEARCH IN
ADVANCED SOFTWARE SCIENCE AND ENGINEERING
NATIONAL INSTITUTE OF INFORMATICS
2-1-2 HITOTSUBASHI, CHIYODA-KU, TOKYO, JAPAN

WWW page: http://grace-center.jp/

Overview of the 3rd International Workshop on Software Patterns and Quality (SPAQu'09)

Hironori Washizaki
Waseda University / GRACE Center, National Institute of Informatics
3-4-1, Okubo, Shinjuku-ku, Tokyo, Japan
washizaki@waseda.jp

Nobukazu Yoshioka
National Institute of Informatics
2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, Japan
nobukazu@nii.ac.jp

Eduardo B. Fernandez
Florida Atlantic University
777 Glades Road, Boca Raton, FL 33431, USA
ed@cse.fau.edu

Jan Jurjens
TU Dortmund / Fraunhofer ISST
jan.jurjens@cs.tu-dortmund.de

Abstract

We will discuss here the theoretical, social, technological and practical issues related to quality aspects of software patterns including security and safety aspects. The workshop will provide the opportunity for bringing together researchers and practitioners, and for discussing the future prospects of this area. As for the workshop format, first, we will have short talks on what software patterns are, and how they are related to quality. Second, we will have accepted position paper presentations to expose the latest researches and practices on software patterns and quality. Finally, we will discuss several topics related to these presentations in small groups. Newcomers, interested researchers and practitioners are free to attend the workshop to facilitate their understandings, researches and practices on software patterns and quality.

Categories and Subject Descriptors D.2.10 [*Software Engineering*]: Design; D.2.11 [*Software Engineering*]: Software Architectures; D.2.13 [*Software Engineering*]: Reusable Software; D.3.3 [*Programming Languages*]: Language Constructs and Features

General Terms Design, Experimentation, Measurement

Keywords Software Patterns, Software Quality, Design Patterns, Security Patterns

1. Main Theme and Goals

As requirements for software products and processes have become more complex, larger scale and have begun to include higher reliability, demand is increasing for a system of technologies to capture, share, enhance, apply and evaluate software patterns. Especially, although numbers of pattern catalogs have been published, little known is about how to specify, measure and evaluate those patterns themselves and/or their application results from the viewpoint of quality. Such conditions make it difficult to see the nature of software patterns and pattern-oriented development ways.

To overcome such conditions, the first workshop of this series was held on December 2007 collocated with the Asia-Pacific Software Engineering Conference (APSEC)[1], and it attracted more than 30 people. The second one was held on October 2008 collocated with the Pattern Languages of Programs Conference (PLoP)[2], and it attracted around 10 people. These previous workshops were successful to discuss the theoretical, social, technological and practical issues related to quality aspects of patterns including security and safety aspects.

However we believe there is a still room to gain an improved understanding and for further research on these topics, and thus continuous efforts for holding the workshop are necessary. This workshop will provide the opportunity for bringing together researchers and practitioners, and for discussing the future prospects of that area. The tone of the workshop will be such that a newcomer to the field of software patterns will receive an introduction of what software patterns are, and how they fit in with their research.

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.
ACM 978-1-60558-768-4/09/10.

2. Possible topics

"Quality" is defined as *the totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs* in ISO 8402. An important property of software quality is that quality requirements are not limited to functionality and reliability. For example, typical software quality characteristics are classified in ISO/IEC 9126 as the followings: functionality, reliability, usability, efficiency, maintainability and portability. To these we can add security and safety. Quality requirements (as part of non-functional requirements) can be specified for each quality characteristic.

Software patterns can be reused to fulfill software requirements including functional and non-functional ones. Currently how to specify quality aspects of patterns applications or of themselves is a remaining big research challenge. Typical existing approaches are the followings:

- Qualitative analysis of relationships among quality attributes (characteristics) and patterns, such as software quality assessment[4] and architecture trade-off analysis[5].
- Requirements engineering for quality aspects of patterns, such as the goal-oriented analysis of patterns for finer representation and selection[6].
- Quantitative measurements of quality aspects of patterns, such as the design complexity[7] and defect frequency[8] in design patterns application results.
- Emerging quality-specific patterns such as security patterns[9]

However, we believe there is still room to gain an improved understanding and further research development on these topics (e.g. how to validate pattern analysis and/or application results?).

3. Post-workshop activities

After the workshop, we will display a poster summarizing the workshop results at the OOPSLA conference site. Moreover, we have a plan to make and put a detailed report on the workshop website[3]. This report will include a summary of discussions so that it will provide a brief summary of the state of the art and future perspectives in the area of software patterns and quality. Therefore, it should facilitate each participant's and non-participant reader's understanding and future research/practice on this area.

Acknowledgments

The workshop will be co-sponsored by the IPSJ/SIGSE Patterns Working Group and the GRACE Center of the National Institute of Informatics of Japan (NII).

References

[1] 1st International Workshop on Software Patterns and Quality (SPAQu'07), 2007. `http://patterns-wg.fuka.info.waseda.ac.jp/SPAQU/result-2007.html`

[2] 2nd Workshop on Software Patterns and Quality (SPAQu'08), 2008. `http://patterns-wg.fuka.info.waseda.ac.jp/SPAQU/result-2008.html`

[3] 3rd International Workshop on Software Patterns and Quality (SPAQu'09), 2009. `http://patterns-wg.fuka.info.waseda.ac.jp/SPAQU/`

[4] Eelke Folmer and Jan Bosch, "A Pattern Framework for Software Quality Assessment And Tradeoff Analysis," International Journal of Software Engineering and Knowledge Engineering, Vol.17, No.1, 2007.

[5] Len Bass, Paul Clements and Rick Kazman, "Software Architecture in Practice," Addison-Wesley, 2003.

[6] Ivdm Araujo and Michael Weiss, "Linking Patterns and Non-Functional Requirements," Proc. of the 9th Conference on Pattern Language of Programs (PLoP 2002), 2002.

[7] Hironori Washizaki, Kazuhiro Fukaya, Atsuto Kubo, Yoshiaki Fukazawa, "Detecting Design Patterns Using Source Code of Before Applying Design Patterns," Proc. of the 8th IEEE/ACIS International Conference on Computer and Information Science (ICIS 2009), 2009.

[8] Marek Vokac, "Defect Frequency and Design Patterns: An Empirical Study of Industrial Code," IEEE Transactions on Software Engineering, Vol.30, No.12, 2004.

[9] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann and Peter Sommerlad, "Security Patterns: Integrating security and systems engineering," Wiley 2006.

Program Committee

Yijun Yu, The Open University, UK
Yasuyuki Tahara, The University of Electro-Communications, Japan
Soon-Kyeong Kim, The University of Queensland, Australia
Linda Rising, Independent Consultant, US
Eric Platon, Cirius Technologies, Japan
Somjai Boonsiri, Chulalongkorn University, Thailand
Naoyuki Nagatou, Ritsumeikan University, Japan
Yann-Gaël Guéhéneuc, Canada Research Chair on Software Patterns and Patterns of Software, École Polytechnique de Montréal, Canada
Kenji Tei, Waseda University, Japan
Atsuto Kubo, National Institute of Informatics, Japan
Katsuhisa Maruyama, Ritsumeikan University, Japan
Fuyuki Ishikawa, National Institute of Informatics, Japan
Michael VanHilst, Florida Atlantic University, US

External Reviewers:
Foutse Khomh, DIRO, Universite de Montreal, QC, Canada
Weimin Ma, The University of Texas at Dallas, US

Table of Contents

New Patterns and Quality of Patterns

Defining a Catalog of Programming Anti-Patterns for Concurrent Java

Jeremy S. Bradbury, Kevin Jalbert
Software Quality Research Group
Faculty of Science (Computer Science)
University of Ontario Institute of Technology
Oshawa, Ontario, Canada
jeremy.bradbury@uoit.ca, kevin.jalbert@mycampus.uoit.ca

***Abstract*—Many programming languages, including Java, provide support for concurrency. Although concurrency has many benefits with respect to performance, concurrent software can be problematic to develop and test because of the many different thread interleavings. We propose a comprehensive set of concurrency programming anti-patterns that can be used by Java developers to aid in avoiding many of the known pitfalls associated with concurrent software development. Our concurrency anti-patterns build upon our previous work as well as the work of others in the research community.**

***Keywords*-concurrency, anti-patterns, bug patterns, Java, deadlock, race conditions, static analysis.**

I. Introduction

The widespread adoption of multi-core technologies has made concurrency an essential characteristic of many traditionally sequential programs. The use of concurrency with multi-core systems can provide an increase in performance over sequential code because it allows programs to have multiple threads executing simultaneously. Although concurrency is beneficial, it can also be problematic. For example, the possibly many different ways to interleave threads in concurrent code make it very difficult to test. Concurrency bugs can be hard to find due to the non-deterministic nature of thread interleavings and because some bugs may occur in only a small subset of the entire interleaving space. It is also challenging to reproduce these bugs and determine if a bug has been fixed or not. In general, concurrency bugs exhibit consequences not present in sequential source code, including deadlock and race conditions. These consequences typically occur because of problems with accessing shared data or controlling access to shared data.

In an effort to improve the quality of concurrent programs there has been considerable effort invested by researchers in developing new programming models, new testing and analysis tools and in identifying concurrency-related design patterns. The development of new concurrent programming models [1] has the potential to make programming with concurrency easier and less error prone. The development of new testing and analysis techniques, as well as the improvement of existing techniques, is aimed at identifying more concurrency bugs prior to deployment. The identification of concurrency design patterns complements the previous two research topics by focusing on how to improve concurrency programming in *existing* languages in an effort to *reduce* bugs prior to testing and analysis.

A pattern is defined as something that *"...describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice"* [2]. A pattern should include details such as the pattern name, the problem, the solution to the problem and the consequences of using the pattern [3]. Alternatively, an anti-pattern defines a recuring bad design solution [4]. The goal of our research is to produce a set of low-level anti-patterns for improving concurrent source code. We have focused our efforts on low-level anti-patterns to complement the previous work on identify concurrency design-level patterns [5], [6].

In Section II we will discuss how to write concurrent programs in Java. We present our concurrency anti-patterns catalog in Section III and provide an example to illustrate the use of the catalog. We also discuss how our catalog can be used to improving concurrency programming, testing and analysis. In Section IV we address how to automatically detect potential anti-patterns in source code before presenting our conclusions in Section V.

II. Java Concurrency

Concurrent Java programs are often called multi-threaded programs. During execution an active thread can be runnable or not runnable and a number of methods exist that can affect a thread's status:

- sleep(): will cause the current thread to stop executing for a certain amount of time.
- yield(): will cause the current thread that is running to pause and yield the processor to another thread.
- join(): will cause the caller thread to wait for a target thread to terminate.
- wait(): will cause the caller thread to wait until a condition is satisfied. Another thread notifies the caller that a condition is satisfied using notify() or notifyAll().

Prior to J2SE 5.0, Java provided support for concurrency primarily through the use of the synchronized keyword. Java supports both synchronization methods and synchronization

blocks. Additionally, synchronization blocks can be used in combination with implicit monitor locks. In J2SE 5.0, additional mechanisms to support concurrency were added as part of the `java.util.concurrent` package [7]:

- Explicit Lock: Provides the same semantics as the implicit monitor locks but provides additional functionality such as timeouts during lock acquisition.
- Semaphore: Maintains a set of permits that restrict the number of threads accessing a resource.
- Latch: Allows threads to wait until other threads complete a set of operations.
- Barrier: A point at which threads from a set wait until all other threads reach that point.
- Exchanger: Allows two threads to exchange objects at a given synchronization point.

To reduce the overhead of developing concurrent software J2SE 5.0 also provides a number of other resources:

- Concurrent collection types: ConcurrentHashMap, BlockingQueues.
- Built-in thread pools: FixedThreadPool and an unbounded CachedThreadPool.
- Atomic variable types: Types that can be used in place of synchronization since each atomic variable type contains special atomic methods. For example, AtomicInteger contains a methods getAndSet().

III. A Catalog of Concurrency Anti-Patterns

Prior to J2SE 5.0, Farchi, Nir, and Ur developed a bug pattern taxonomy for Java concurrency [8]. The bug patterns are based on common mistakes programmers make when developing concurrent code in practice. Furthermore, the taxonomy has been expanded and used to classify bugs in an existing public domain concurrency benchmark maintained by IBM Research [9]. Bradbury, Cordy and Dingel further extended the taxonomy in their concurrency mutation research [10]. We will use this bug taxonomy as the basis for our concurrency anti-patterns – in fact many of the problems we identify were included in this previous work.

An anti-pattern catalog for Java multithreaded software has already been developed by Hallal et al. [6]. In their work, Hallal et al. distinguish between design anti-patterns and error or bug patterns. The former category focuses on the syntactic design within a program while the latter category focuses on *"patterns of erroneous program behavior correlated with programming mistakes"* [6]. The Hallal et al. anti-pattern catalog primarily contains design anti-patterns, including anti-patterns related to efficiency, quality and style, while our work focuses on the identification of anti-patterns based on bugs and includes anti-patterns related to the correctness of the program. Therefore, we believe that the Hallal et al. catalog and our catalog are complementary.

Table I and II provide an overview of all the concurrency anti-patterns included in our catalog[1]. For each anti-pattern we provide the following information:

- *pattern name:* the anti-pattern name is based on the corresponding bug's name. For example, the two-state access anti-pattern corresponds to the two-state access bug.
- *problem:* the problem describes the corresponding bug that is being addressed.
- *context:* the context in which the problem often occurs.
- *solution:* the solution describes general steps that can be taken to correct the anti-pattern. We have made an effort to keep the solutions as general as possible and it is expected that the developer will have the appropriate level of knowledge to understand how to apply the solution in a specific context.

We have not included the consequences of fixing each anti-pattern because in most cases these are evident from the problem section of the anti-pattern. For example, the consequences of applying the solution in the *Deadlock anti-pattern* are that locks will now be released and the threads will no longer halt.

Our catalog of concurrency anti-patterns provides several benefits:

1) The catalog is language specific – it is focused on anti-patterns that can occur in Java and not anti-patterns that occur in general.
2) The catalog is comprehensive – it includes the bug definitions from several different sources [8], [9], [10].
3) The catalog provides solutions – in addition to enumerating different kinds of concurrency bugs as anti-pattern problems, we also provide solutions to each anti-pattern.

To demonstrate the use of the catalog we will now describe an example using the *Deadlock anti-pattern*. Consider the following two code fragments which are executed by different threads:

Code fragment #1:

```
public void methodA(){
   synchronized(lock1){
      synchronized(lock2){    }
   }
}
```

Code fragment #2:

```
public void methodB(){
   synchronized(lock3){
      synchronized(lock4){    }
   }
}
```

[1]In Table I and II we distinguish between the original bugs from [8] (*), the added bug used in the benchmark classification [9] (**) and the bugs included in [10] ($^{+}$).

Pattern name	Problem	Context	Solution
Nonatomic operations assumed to be atomic anti-pattern.*	*"...an operation that "looks" like one operation in one programmer model (e.g., the source code level of the programming language) but actually consists of several unprotected operations at the lower abstraction levels"* [8].	Trying to perform an operation on a shared data variable atomically.	Use the volatile keyword when using 64-bit variables.
Two-state access bug anti-pattern.*	*"Sometimes a sequence of operations needs to be protected but the programmer wrongly assumes that separately protecting each operation is enough"* [8].	Trying to protect access to operations involving shared data.	Combine the multiple critical regions into one critical region.
Wrong lock or no lock bug anti-pattern.*	*"A code segment is protected by a lock but other threads do not obtain the same lock instance when executing. Either these other threads do not obtain a lock at all or they obtain some lock other than the one used by the code segment"* [8].	Trying to protect access to operations involving shared data.	Identify all accesses to shared data and use the same lock object to protect these critical regions. This may involve added a new lock or replacing incorrect locks with the correct one.
Double-checked lock anti-pattern.*	*"When an object is initialized, the thread local copy of the objects field is initialized but not all object fields are necessarily written to the heap. This might cause the object to be partially initialized while its reference is not null"* [8].	Trying to initialize shared variables without using protection.	Use locks to synchronize all access to the object or use volatile. Do not perform lazy initialization on shared objects.
The sleep() anti-pattern.*	*"The programmer assumes that a child thread should be faster than the parent thread in order that its results be available to the parent thread when it decides to advance. Therefore, the programmer sometimes adds an 'appropriate' sleep() to the parent thread. However, the parent thread may still be quicker in some environment."* [8].	Trying to coordinate threads based on assumptions regarding thread timing.	*"The correct solution would be for the parent thread to use the join() method to explicitly wait for the child thread"* [8].
Missing or nonexistent signals anti-pattern.+	This pattern generalizes the losing a notify bug pattern to all signals. The losing a notify bug is defined as occurring *"If a notify() is executed before its corresponding wait(), the notify() has no effect and is "lost" ... the programmer implicitly assumes that the wait() operation will occur before any of the corresponding notify() operations"* [8]. Another example of this problem can occur at a barrier. If an await() from one thread never occurs then all of threads at the barrier may be stuck waiting.	Trying to coordinate threads based on assumptions regarding thread timing.	*In the case of a notify signal,* "One way of avoiding this bug pattern is to repeatedly execute the notify() operation until a condition stating that the notify() was received occurs"[8]. Use concurrent mechanisms such as barriers and join() to prevent thread timing issues. Analogous solutions exist for other signals.
Notify instead of notify all anti-pattern.**	If a notify() is executed instead of notifyAll() then threads with some of its corresponding wait() calls will not be notified [16].	Trying to coordinate threads.	Replace notify() with notifyAll().
A "blocking" critical section anti-pattern.*	*"A thread is assumed to eventually return control but it never does"* [8].	Using locks to try and protect access to operations involving shared data.	Ensure that every lock() acquisition has a corresponding unlock(). If it is possible to throw an exception inside a critical region the unlock() must be placed in a finally block. The finally block will be executed regardless if the exception is thrown.

Table I
CONCURRENCY ANTI-PATTERNS CATALOG *(Part 1 of 2)*

The above fragments are an example of the *Deadlock anti-pattern* if lock1 is the same lock object as lock4 while lock2 is the same lock object as lock3. If the above fragments are an example of the *Deadlock anti-pattern* then we have several

Pattern name	Problem	Context	Solution
The interference anti-pattern.**	A pattern in which *"...two or more concurrent threads access a shared variable and when at least one access is a write, and the threads use no explicit mechanism to prevent the access from being simultaneous."* [17]. The interference bug pattern can also be generalized from classic data race interference to include high level data races** which deal *"...with accesses to sets of fields which are related and should be accessed atomically"* [18].	Trying to use operations involving shared data without protecting the access to the shared data.	Use synchronization to protect both write and read access to shared variables.
The deadlock anti-pattern.**	*"...a situation where two or more processes are unable to proceed because each is waiting for one of the others to do something in a deadlock cycle ... For example, this occurs when a thread holds a lock that another thread desires and vice-versa"* [17].	Trying to protect access to operations involving shared data.	Remove unnecessary synchronization if possible. Remove unnecessary nested synchronization if possible. Ensure nested synchronization always occurs in the same order.
Starvation anti-pattern.+	This bug occurs when their is a failure to *"...allocate CPU time to a thread. This may be due to scheduling policies..."* [5]. For example, an unfair lock acquisition scheme might cause a thread never to be scheduled.	Trying to use concurrency independent of scheduling policies.	When available use fairness parameter for concurrent mechanisms like semaphores. This will ensure that no thread can unfairly acquire semaphore permits.
Resource exhaustion anti-pattern.+	*"A group of threads together hold all of a finite number of resources. One of them needs additional resources but no other thread gives one up"* [5].	Trying to optimize a concurrent program by limiting resources.	One solution is to consider allocating additional resources. Another solution is to limit all threads' access to resources.
Incorrect count initialization anti-pattern.+	This pattern occurs when there is an incorrect initialization in a barrier for the number of parties that must be waiting for the barrier to trip, or an incorrect initialization of the number of threads required to complete some action in a latch, or an incorrect initialization of the number of permits in a semaphore.	Trying to protect access to operations involving shared data.	Correct the count to the appropriate value.

Table II
CONCURRENCY ANTI-PATTERNS CATALOG *(Part 2 of 2)*

options regarding how to correct the code fragments. First we need to ensure that both locks are indeed necessary. If any lock is not necessary it should be removed. If all locks are necessary, we next consider whether the locks need to be nested. If not we rewrite the code to separate the critical region into two separate regions each protected by one of the locks. If nested synchronization is necessary, we need to ensure that the lock objects are always acquired in the same order. This example illustrates how a catalog of concurrency anti-patterns can aide in improving the quality of concurrent software. We believe that the benefits of this work fall into three key areas: programming, testing and static analysis.

Programming. There are many examples of software design patterns that have been adopted and used in industry [3], [5]. A benefit of these patterns is that they clearly show good ways (or in the case of anti-patterns bad ways) to design or implement software. The goal of our concurrency anti-patterns is to provide programmers an additional resource that will assist them in concurrent Java programming by sharing potential problems that should be avoided. The benefit of our anti-patterns is that they help to clarify bad concurrency practices which can assist developers in avoiding concurrency bugs and thus result in improved source code.

Testing. Sequential testing typically involves developing a set of test cases that provide a certain type of code coverage (e.g., path coverage) and executing these tests on the code to detect possible bugs and failures. Due to the non-determinism of the execution of concurrent source code and the high number of possible interleavings, concurrency testing can not rely on coverage metrics alone to guarantee that code is correct. Concurrency testing must also provide increased confidence that bugs that manifest themselves in only a few of the interleavings are found. For example,

since a race condition or deadlock may only occur in a small subset of the possible interleaving space, the more interleavings we test the higher our confidence that the bug that caused the race condition or deadlock will be found. An example of a tool for executing different thread schedules is ConTest [11].

A catalog of concurrency anti-patterns can benefit concurrency testing by helping to direct the testing effort. A good understanding of concurrency bugs can provide a tester with more insight into the problems he or she may encounter as well as help a tester focus his or her testing effort within the interleaving space.

Static analysis. Static analysis can be used throughout the software development life cycle and provides useful information about the possible presence of bugs in software. For example, a static analysis tool that detects a match of the *Deadlock anti-pattern* may help a programmer improve his or her code during implementation or may help catch a bug during testing. Existing static analysis tools, including FindBugs [12], JLint [13] and the Otto-Moschny tool [14], already utilize some concurrency bug patterns in an effort to identify potential problems in concurrent Java source code. Our catalog of concurrency anti-patterns will aide in improving existing tools as well as in the development of new static analysis tools.

IV. Detecting Concurrency Anti-Patterns

In addition to developing our concurrency anti-pattern catalog we have also developed several tools to assist programmers in managing anti-patterns and in identifying potential anti-pattern matches in source code.

A. Concurrency Anti-Pattern Creator

Concurrency anti-patterns can be created and managed using the Concurrency Anti-Pattern Creator tool (see Figure 1). In this tool we define a concurrency anti-pattern as consisting of a name, a problem (with context), a solution, one or more fragments of code as well as a rule about how the fragments interact to cause undesired behaviour. Our experience has shown that many concurrency bugs result in a combination of different code fragments executing in different threads. The interaction of code fragments from different threads is specified in the anti-pattern definition using a rule. Specifically, the rule explains how the code fragments interact to produce erroneous output.

B. Clone-Based Detection of Concurrency Anti-Patterns

One of our goals in creating our concurrency anti-pattern catalog was to also create a static analysis tool to detect possible anti-pattern matches. In our tool, a potential match in source code is made to a known anti-pattern only if all code fragments are present and the rule is satisfied. Our approach takes program source code and anti-patterns as input. The source code is normalized and input to the clone

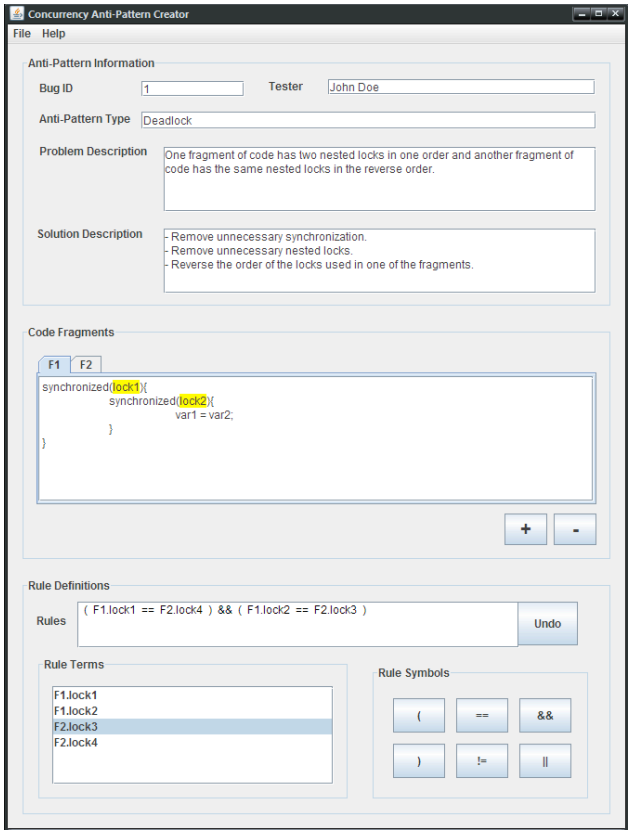


Figure 1. Concurrency Anti-Pattern Creator

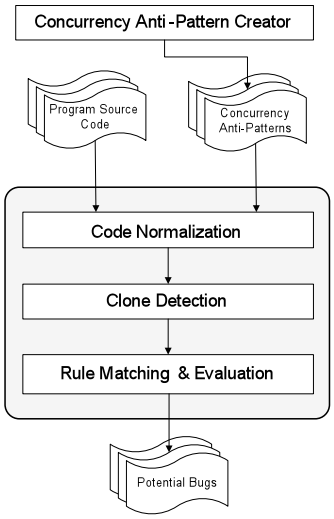


Figure 2. Detection of anti-pattern matches in source code

detection tool ConQAT [15]. We use the pattern matching in ConQAT to determine if code fragments in the source code match the code fragments in our anti-pattern. In cases where ConQAT finds matches for all code fragments in an

anti-pattern we use rule matching to determine if a particular combination of code fragments satisfy the anti-pattern rule. If the rule is satisfied we identify the code fragments as a potential match to our anti-pattern. At this point the developer can use the anti-pattern catalog to determine the appropriate fix (if the match is not a false positive).

An important feature of our detection tool is that it is designed to detect any anti-pattern specified using the Concurrency Anti-Pattern Creator. This feature ensures that the detection tool is flexible enough to be extended to any future anti-patterns that could be added to the catalog. It also means that the catalog can be customized to a particular project or source code repository.

V. Conclusion

In this paper we have presented a catalog of programming anti-patterns for concurrent Java that are comprehensive with respect to the programming features available in the Java programming language and comprehensive with respect to an existing concurrency bug pattern taxonomy. We will be making our catalog available publicly[2] and providing the community an opportunity to both use and contribute anti-patterns.

In the future we hope to conduct additional research on the benefits of the catalog with respect to static analysis and testing. We are also interested in studying how these anti-patterns can be utilized in combination with more high-level design patterns [3], [5] and the Hallal et al. anti-patterns [6].

Acknowledgment

The authors would like to thank Shmuel Ur for providing access to IBM Haifa Lab's Concurrency Bug Benchmark. We would also like to thank the Natural Sciences and Engineering Research Council of Canada (NSERC) for funding this research.

References

[1] H. Sutter and J. Larus, "Software and the concurrency revolution," *Queue*, vol. 3, no. 7, pp. 54–62, 2005.

[2] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel, *A Pattern Language*. Oxford University Press, 1977.

[3] E. Gamma, R. Helm, R. Johnson, and J. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, ser. Addison-Wesley Professionl Computing Series. Addison Wesley, 1995.

[4] M. Meyer, "Pattern-based reengineering of software systems," in 13^{th} *Working Conference on Reverse Engineering (WCRE '06)*, 2006, pp. 305–306.

[5] D. Lea, *Concurrent Programming in Java: Design Principles and Patterns, Second Edition*. Addison Wesley, 2000.

[2]http://svilab.science.uoit.ca/concurr-catalog/

[6] H. Hallal, E. Alikacem, W. Tunney, S. Boroday, and A. Petrenko, "Antipattern-based detection of deficiencies in java multithreaded software," in 4^{th} *International Conference on Quality Software (QSIC 2004)*, 2004, pp. 258–267.

[7] "`java.util.concurrent` documentation," Web page: http://java.sun.com/j2se/1.5.0/docs/api/java/util/concurrent/package-summary.html.

[8] E. Farchi, Y. Nir, and S. Ur, "Concurrent bug patterns and how to test them," in *Proc. of the* 1^{st} *International Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2003)*, Apr. 2003.

[9] Y. Eytani and S. Ur, "Compiling a benchmark of documented multi-threaded bugs," in *Proc. of the* 2^{nd} *International Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2004)*, Apr. 2004.

[10] J. S. Bradbury, J. R. Cordy, and J. Dingel, "Mutation operators for concurrent Java (J2SE 5.0)," in *Proc. of the* 2^{nd} *Workshop on Mutation Analysis (Mutation 2006)*, Nov. 2006, pp. 83–92.

[11] O. Edelstein, E. Farchi, Y. Nir, G. Ratsaby, and S. Ur, "Multithreaded Java program test generation." *IBM Systems Journal*, vol. 41, no. 1, pp. 111–125, 2002.

[12] "Findbugs - find bugs in Java programs," Web page: http://findbugs.sourceforge.net/.

[13] *Jlint Manual: Java program checker*, Web page: http://artho.com/jlint/manual.html, Jan. 2002.

[14] F. Otto and T. Moschny, "Finding synchronization defects in java programs: extended static analyses and code patterns," in *IWMSE '08: Proceedings of the 1st international workshop on Multicore software engineering*. New York, NY, USA: ACM, 2008, pp. 41–46.

[15] F. Deissenboeck, E. Juergens, B. Hummel, S. Wagner, B. M. y Parareda, and M. Pizka, "Tool support for continuous quality control," *IEEE Software*, vol. 25, no. 5, pp. 60–67, 2008.

[16] B. Long, R. Duke, D. Goldson, P. A. Strooper, and L. Wildman, "Mutation-based exploration of a method for verifying concurrent Java components," in *Proc. of the* 2^{nd} *International Workshop on Parallel and Distributed Systems: Testing and Debugging (PADTAD 2004)*, Apr. 2004.

[17] B. Long, P. Strooper, and L. Wildman, "A method for verifying concurrent Java components based on an analysis of concurrency failures," *Concurrency and Computation: Practice and Experience*, vol. 19, no. 3, pp. 281–294, Mar. 2007.

[18] C. Artho, K. Havelund, and A. Biere, "High-level data races," in *Proc. of the* 1^{st} *International Workshop on Verification and Validation of Enterprise Information Systems (VVEIS'03)*, Apr. 2003.

[19] R. H. B. Netzer and B. P. Miller, "What are race conditions?: Some issues and formalizations," *ACM Lett. Program. Lang. Syst.*, vol. 1, no. 1, pp. 74–88, 1992.

Abstract Testability Patterns

Wanderlei Souza
State of Sao Paulo Institute for Technological Research
Sao Paulo, Brazil
wandi@uol.com.br

Reginaldo Arakaki
University of Sao Paulo, Polytechnic School
Sao Paulo, Brazil
reginaldo.arakaki@poli.usp.br

***Abstract*—Testability is a software quality characteristic that exposes the degree to which a software artifact facilitates the testing process. Software testing is a technical and economical problem, it is important to help identify patterns that would improve the industry's software testing capabilities. This position paper proposes five abstract patterns that improve software testability, which serves as a reference for testers and developers to evaluate the testability support for high reliable software.**

Keywords: Software quality; Design patterns; Design for testability; Software testing; Observability; Built-in testing.

I. INTRODUCTION

The component testability is an important quality characteristic to evaluate the degree to which a software artifact facilitates the testing process. A lower degree of testability results in increased test effort. Depending on the methods used, testing activities account for 25% to 90% of total project effort [1][2][3][4]. Thus, it is important to help identify patterns that would improve the software capabilities of the industry.

Controllability and observability are the key points to testability [5][6]. To test a component it is necessary to control the inputs (controllability) and observe the outputs (observability). Without these key points it is difficult to improve system testability.

Testability patterns proposed in this paper are based on abstract pattern concept described in [7] and correspond to mechanisms or services that increase overall system observability and controllability, these patterns are more general ideas and are not concerned with any specific implementation technique or software development platform. They should not be confused with testability factors or characteristics. Abstract testability patterns correspond to architectural mechanisms, not testability principles.

A software architect uses a distinct pattern collection to design a system. Architectural patterns provide a predefined set of structures, responsibilities, rules and guidelines to organize the relation between system components. A pattern implements a sequence of design decisions to manage certain system quality characteristics. The testability of the architecture was brought up by Nancy Eickelmann and Debra Richardson in [8]. The authors propose that the architectural decisions must be aligned to testing strategies. In this way, the testability of architecture is the combination between architectural patterns and the testing strategy.

II. ABSTRACT TESTABILITY PATTERNS

A. *Built-In Self-Testing (BIST)*

Problem: Internal components establish connections to external resources (HTTP connections, database systems, remote calls etc.) and do not have a standard interface to validate directly these integrations. The absence of a uniform way to verify all critical integration points after the system deployment process reduces the system testability.

Solution: Built-In Self-Tests (BIST) is a mechanism to self-report the status and health of individual system components. Built-In Self-Tests (BIST) adds standard interfaces to validate core system functionalities and provides many types of validation possibilities. For example, testing the interface between a component and a database system can be accomplished by invoking the BIST to validate the connection and permissions on system tables.

Consequences: Developers must implement a standard test interface in all BIST related classes. In general ways, it is a minimal overhead to development process, but implement a BIST could be more difficult depending on the complexity of the integration under test.

B. *Dependency Injection (DI)*

Problem: A business component is difficult to test in isolation because it has a direct reference to external dependencies (third-party components, database systems, web services, etc.) and it is not possible to replace the dependency without changing the source code. The main problem roots from the business component creating the external dependencies.

Solution: It is necessary to inject dependencies into a business component, rather than relying on the component to manage the dependencies itself. Dependency injection is a pattern that can be used to improve the software testability by removing the business component responsibility for instantiating its own external dependencies.

Consequences: There is added complexity to the source code and there are more elements to manage on test automation process. Testers must be able to manage mock objects creation and initialization in order to replace the original code dependencies when necessary.

C. *Dynamic Component Management Extension*

Problem: A test team has different mock components to inject and simulate external component behavior, but they do not have a dynamic way to change these components. To

choose the component concrete implementation dynamically is fundamental to test automation process.

Solution: The system must provide a standard extension to make the system components and services suitable. This extension defines a management architecture to testable components. Using a standard test extension to manage components increases the system testability by making applications more controllable.

Consequences: Dynamic Component Management Extension enables system components management in a test environment. Security barriers or component *undeploy* must be used to avoid undesired test behavior in a production environment.

D. Testability Logger

Problem: Application events and test related data must be logged for testing purposes. This can lead to redundant code.

Solution: Use the Testability Logger to provide centralized control of logging functionality and takes care of how the testable events are classified and logged. Testability Logger increases the system testability by making applications more observable.

Consequences: The Testability Logger operations (disk IO access, message digests, etc.) impact the system performance.

E. Testability Interceptor

Problem: A tester needs to intercept messages between components for the purpose of verifying the internal behaviors. System also must provide possibility to change application behavior in order to inject and simulate faults in a system.

Solution: Testability Interceptor offers a mechanism to enhance the observability and controllability of a software system by letting components monitor and dynamically change their behavior. Testers can observe and modify functionality without changing the internal logic of components. The Interceptor supports runtime system monitoring and control through Dynamic Component Management Extensions pattern, described in (C).

Consequences: A system design can get more complicated and hard to understand since the developer has to implement the intercepting points.

III. Related Work

Binder [6] presents the Built-In Test concept, a well-known technique for hardware testing, in a software context. We use an abstraction of this concept to describe the Built-In Self-Testing pattern.

Dynamic Component Management Extension is an abstraction layer over JMX technology [9]. We use the JMX ideas and capabilities in order to improve system testability.

The Dependency Injection pattern was first described by Fowler [10] as a specific form of Inversion of Control. We believe that DI can be used to improve the system testability by abstracting the dependencies out of a component.

The Testability Logger is based on Secure Logger pattern idea [11], but with a focus on system testing and evaluation instead of security reasons.

The Testability Interceptor pattern is related to the Interceptor pattern, which allows services to be added transparently and triggered automatically [12].

IV. Conclusion

In this paper we have introduced perspectives to improve system testability and propose a new architectural pattern category: testability patterns. Future work includes other ideas to architectural testability patterns and concrete implementations.

A Java based concrete implementation of these abstract patterns has been used to improve the testability of critical Internet systems in the main portal to content & the Internet provider in Brazil.

References

[1] S. Jungmayr, “Reviewing Software Artifacts for Testability”, Proc. of EuroSTAR’99, Barcelona, Spain, November 10-12, 1999.

[2] R. S. Pressman, “Software Engineering: Practitioner’s Approach”, European 3rd Edition, McGraw-Hill Book Company, Berkshire, England, 1994, pp. 609.

[3] Tim Koomen and Martin Pol, “Test Process Improvement: A Practical Step-by-Step Guide to Structured Testing”, Addison-Wesley, 1999.

[4] B. Beizer, “Software Testing Techniques”, International Thomson Computer Press, Boston, 1990.

[5] Roy S. Freedman, “Testability of Software Components”, IEEE Transactions on Software Engineering, Vol. 17, No. 6, 1991.

[6] Robert V. Binder, “Design for Testability in Object-Oriented Systems”, Communications of the ACM, v.37 n.9, 1994.

[7] Eduardo B. Fernandez, Hironori Washizaki and Nobukazu Yoshioka, “Abstract Security Patterns”, 2nd International workshop on software patterns and quality (SPAQu'08), 2008.

[8] Nancy S. Eickelmann and Debra J. Richardson, “What makes one software architecture more testable than other?”, Joint proceedings of the second international software architecture workshop (ISAW-2) and international workshop on multiple perspectives in software development (Viewpoints '96) on SIGSOFT '96 workshops, San Francisco, California, 1996, pp. 65.

[9] H. Kreger, “Java management extensions for application management”, IBM Journal of Research and Development, IBM Corp. Riverton, NJ, 2001.

[10] Martin Fowler, “Inversion of Control containers and the Dependency Injection pattern”, http://martinfowler.com, 2004.

[11] Christopher Steel, Ramesh Nagappan and Ray Lai, “Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management”, Prentice Hall PTR, 2006, pp. 577.

[12] Douglas Schmidt, Michael Stal, Hans Rohnert and Frank Buschmann. “Pattern-Oriented Software Architecture, Vol. 2 - Patterns for Concurrent and Distributed Objects”. John Wiley and Sons, Ltd., 2000.

Towards an Assessment of the Quality of Refactoring Patterns

Norihiro Yoshida, Masatomo Yoshida, Katsuro Inoue
Graduate School of Information Science and Technorogy, Osaka University
1-5 Yamadaoka, Suita, Osaka, 565-0871, Japan
Email: {n-yosida, mstm-ysd, inoue}@ist.osaka-u.ac.jp

***Abstract*—Refactoring is a well-known process that is thought to improve the maintainability of object-oriented software. Although a lot of refactoring patterns are introduced in several pieces of literature, the quality of refactoring patterns is not always discussed. Therefore, it is difficult for developers to determine which refactoring patterns should be given priority. In this paper, we propose two quality characteristics of refactoring pattern, and then describe an open source case study on assessing those quality characteristics.**

***Keywords*-refactoring; quality of software pattern; object-oriented programing; software maintenance;**

I. Introduction

Refactoring [1] is the process of changing a software system in such a way that it does not alter the external behavior of the code yet improves its internal structure. That is to say, refactoring is a process to improve the maintainability of software systems.

Several practitioners introduce a lot of **Refactoring Patterns (RP)** [1][2]. Each RP includes both a description of a **refactoring opportunity (RO)** (i.e., a set of code fragments that should be refactored) and the corresponding procedure to perform refactoring (i.e., how to perform refactoring). However, the quality of each refactoring pattern is mostly never assessed. Therefore, it is difficult for developers to determine which refactoring patterns should be given priority.

In this paper, we propose two quality characteristics of RPs, and then describe a case study on assessing those quality characteristics.

II. Proposed quality characteristics of refactoring patterns

We introduce the following two quality characteristics of RP.

- **Number of ROs:** Because a lot of refactoring patterns exist and developers have only a limited time, it is desirable to choose RPs that have a lot of ROs.
- **Ease of Refactoring:** It means that ease of applying each RP to ROs in source code. When the ease of refactoring of a RP is high, it means that software systems involve a lot of ROs that can be easily performed refactoring. A RP that is difficult to apply often leads to time-consuming refactoring. Because the aim of refactoring is to reduce maintenance cost, time-consuming refactoring is not desirable. There are two kinds of refactoring pattern. The first one requires developers to apply only steps described in its description. On the other hand, another sometimes requires developers to apply not only steps described in its description but also additional steps.

III. Case Study

In this section, we assess the quality characteristics of RP which is named **Introduce Polymorphic Creation with Factory Method (IPCFM)** [2].

We introduce IPCFM and an automated method to identify ROs in software systems for IPCFM. Then, we discuss the ROs in several software systems from proposed quality characteristics of RP.

A. Introduce polymorphic creation with factory method

IPCFM is a kind of **Pull up Method** [1] pattern that is aimed at merging similar methods from different classes into a common superclass. Figure 1 shows an example of IPCFM. The aim of IPCFM is to merge similar methods except for object creation statements by introducing factory methods. An RO for IPCFM is defined as *"Classes in a hierarchy implement a method similarly except for an object creation step"* [2].

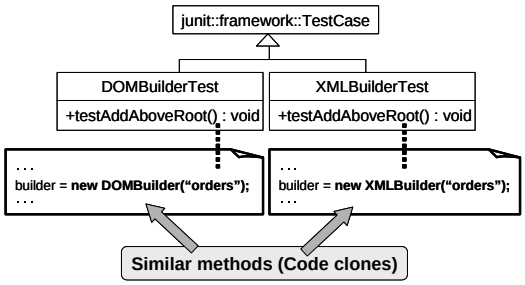
As shown in Figure 1(a), the targets of the refactoring are the test classes DOMBuilderTest and XMLBuilderTest for testing DOMBuilder and XMLBuilder, respectively. Because the target classes have similar methods except for an object creation step, they indicate an RO for applying IPCFM. This refactoring is comprised of following two steps.

Step1 As shown in Figure 1(b), a common superclass (AbstractBuilderTest) for the target classes is introduced, and similar methods in the target classes are merged into new method in the common superclass.

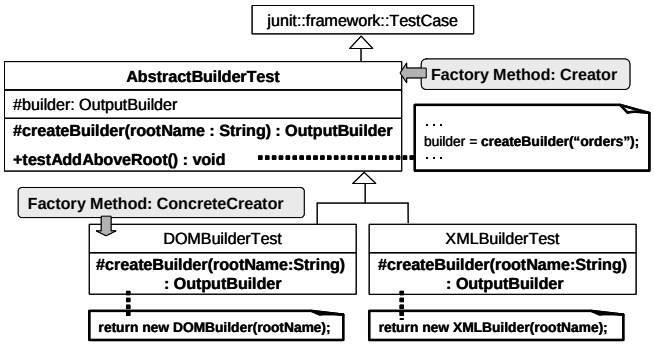
Step2 A factory method is introduced in each of the common superclass (AbstractBuilderTest) and the subclasses (DOMBuilderTest and XMLBuilderTest).

B. Assessment Method

For our case study, we have developed the tool that identifies ROs for the target RP by the steps below.



(a) Before refactoring



(b) After refactoring

Figure 1. Introduce Polymorphic Creation with Factory Method

Step1 Detect similar methods using a code clone detection tool CCFinder [3][1].

Step2 Evaluate whether detected methods belong to classes that have common superclasses in target source code and whether they include object creation statements.

We apply the target RP to the ROs in Ant and ANTLR. To assess the ease of refactoring, we confirm the steps that are not described in the description of the target RP.

C. Results

Table I includes the result of identifying ROs for IPCFM in several software systems. For comparison, in Table I, we show the number of ROs for Pull up Method (PM). We identify ROs for PM by detecting code clones belonging to classes that have common superclasses. We should note that because IPCFM is kind of PM, an RO for IPCFM is counted towards the number of ROs for PM. According to Table I, 17.9% of the ROs for PM are the ROs for IPCFM. We can say that ROs for PM includes more than few ROs for IPCFM. This indicates that when developers found RO for PM, they should inspect whether those RO are also for IPCFM.

When we apply IPCFM to all ROs in Ant and ANTLR, we did not have to apply additional steps that are not described in the description of IPCFM. This result indicates that the ease of refactoring of IPCFM is high.

[1]In our case study, we set 30 tokens as the minimum length of code clone.

IV. Related Works

Hsueh, et al.[4] and Huston[5] focus on the quality of design patterns. We focus on the quality of RPs, and propose the two novel quality characteristics of RPs.

V. Summary and Future work

In this paper, we proposed two quality characteristics of RP, and then described a case study on assessing those quality characteristics of IPCFM. To compare the quality characteristics of RP, we are planning to assess other RPs. We should discuss not only proposed quality characteristics but also change in maintainability because the aim of refactoring is to reduce maintenance cost.

Acknowledgment

We thank the anonymous SPAQu'09 reviewers for useful feedback on earlier versions of this paper. This reseach was supported by JSPS, Grant-in-Aid for Scientific Research (A) (No.21240002) and Grant-in-Aid for JSPS Fellows (No.20-1964).

References

[1] M. Fowler, *Refactoring: improving the design of existing code*. Addison Wesley, 1999.

[2] J. Kerievsky, *Refactoring to Patterns*. Addison Wesley, 2004.

[3] T. Kamiya, S. Kusumoto, and K. Inoue, "CCFinder: A multilinguistic token-based code clone detection system for large scale source code," *IEEE Trans. Sofw. Eng.*, vol. 28, no. 7, pp. 654–670, 2002.

[4] N.-L. Hsueh, P.-H. Chu, and W. Chu, "A quantitative approach for evaluating the quality of design patterns," *Journal of Systems and Software*, vol. 81, no. 8, pp. 1430–1439, 2008.

[5] B. Huston, "The effects of design pattern application on metric scores," *Journal of Systems and Software*, vol. 58, no. 3, pp. 261–269, 2001.

Table I
Number of ROs for IPCFM

name	LOC	#classes	#opportunities	
			IPCFM	PM
Ant 1.7.0	198K	994	2	23
ANTLR 2.7.4	32K	167	1	33
Azureus 3.0.3.4	538K	2226	20	42
JEdit 4.3	168K	992	0	1
JHotDraw 7.0.9	90K	487	1	26
SableCC 3.2	35K	237	0	1
Soot 2.2.4	352K	2298	5	53
WALA 1.1	210K	1565	7	22

On the Symbiosis between Quality and Patterns

Pankaj Kamthan
Department of Computer Science and Software Engineering
Concordia University
Montreal, Quebec, Canada H3G 1M8
kamthan@cse.concordia.ca

***Abstract*—This position paper proposes that the impact on the quality of a software system by using patterns is intrinsically related to the quality of the patterns themselves. In doing so, it presents some of the challenges being faced and hints towards potential resolutions.**

Pattern Description, Pattern Stakeholder, Semiotic Quality

I. INTRODUCTION

For the sake of this paper, a pattern is an empirically proven solution to a recurring problem in a particular context. For simplicity, two classes of stakeholders of a pattern, namely pattern producers and pattern consumers, are considered. The rest of the paper explores the interdependence of the notions of patterns for quality and quality of patterns.

II. A PERSPECTIVE ON QUALITY

For the sake of this paper, quality is defined using the ISO/IEC 9126-1:2001(E) as "the totality of [attributes] of an entity that bear on its ability to satisfy stated and implied needs." The notion of quality is multifaceted. These facets include the entity of interest, the viewpoint on that entity, and quality attributes of that entity.

There are a number of possible viewpoints of quality. In one of the earliest approaches towards perceptions of quality [4], the five views of quality are identified: (QV1) the transcendental-based view (quality is perfective), (QV2) the product-based view (quality is measurable), (QV3) the manufacturing-based view (quality is conformance), (QV4) the economics-based view (quality is benefit for cost), and (QV5) the user-based view (quality is satisfaction).

Indeed, multiple views may need to be satisfied in addressing quality of a software system. For example, the ISO/IEC 9126-1:2001(E) presents an intersection of QV2, QV4, and QV5. Furthermore, due to practical considerations, QV4 constrains QV1. Therefore, any initiatives towards quality assurance or evaluation need to end if the cost exceeds the benefit. There are no studies in the current literature on the return on investment (ROI) of using patterns, for the purpose of aiding quality of a software system or otherwise.

III. PATTERNS FOR QUALITY

There are a number of approaches for quality assurance and evaluation. The use of a pattern is a preventative approach to quality, as opposed to inspections or testing that are curative approaches.

Let S be a software system under development. Let FR be a functional requirement for S. A realization of FR is constrained by certain expectations of quality. If a pattern P is to be selected for S, then a number of conditions must be satisfied:

- **Context.** The context of P must subsume that of S. This means that the description of P must make the context explicit.
- **Problem.** The problem of P must be aligned with FR. This means that the description of P must make the problem explicit.
- **Forces.** A quality model is useful for creating an understanding of quality. There is currently no single, universal quality model that is applicable to all software systems. Let QM be a quality model associated with S. Let the quality attributes in QM be prioritized as QA1 ≥ … ≥ QAn, where ≤ is some total ordering. The forces of P must to be aligned with QM. In other words, the highest priority forces that the solution of P resolves must also be the highest priority in QA1 ≥ … ≥ QAn. Thus, P may aid some but not other quality attributes of S. The view of quality of the software engineers of S and producers of P may not be identical. Indeed, it has been shown empirically [5] that the relationship between quality of S and the patterns used in the development of S is equivocal. This also means that the description of P must make the forces explicit [1].

In turn, these usually imply that the description of P is structured in some way. It is possible to impose a structure on a pattern by adopting a pattern form. There is currently no single pattern form followed by pattern producers. This makes a systematic comparison of patterns in general and an assessment of their impact on quality of a software system in particular difficult.

IV. QUALITY OF PATTERNS

There is guidance available for describing patterns [6]. However, currently there is no acceptable definition of quality of a pattern and no general quality model for patterns.

There are a number of possible views of a pattern. From an epistemological viewpoint, a pattern is implicit knowledge made explicit by means of a pattern description. From a semiotic viewpoint, a pattern can be viewed at six levels:

physical, empirical, syntactic, semantic, pragmatic, and social. In this paper, the interest is in pragmatic quality of a pattern, which is a contract between a pattern and its stakeholders.

The quality of a pattern needs to be studied at two levels: (1) at the pattern description level and (2) at the individual pattern element level.

A. *Quality of a Pattern at the Description Level*

The quality attributes such as accessibility/usability and maintainability are part of pragmatic quality, and apply to a pattern description as a whole. There are a number of issues that can arise at the description level:

- **Accessibility/Usability.** These are of concern to pattern consumers. A pattern description, especially that made available only via repositories on the Web, can have accessibility (as per ISO 9241-171:2008)/usability (as per ISO/IEC 9126-1:2001(E)) issues. For example, there are pattern descriptions that do not pass the W3C Web Content Accessibility Guidelines (WCAG) and users have found difficulties in reading and navigating pattern descriptions available on pattern repositories [7].
- **Maintainability.** This is of concern to pattern producers. A pattern description may need to evolve for a number of reasons including discovery of errors that need to be rectified, modification in technologies illustrating the solution, presentation on a device not targeted originally, and so on. For example, software engineers trained in this century may be more familiar with the Unified Modeling Language (UML) than with the notations used in describing the solutions of the patterns of the 1990s.

B. *Quality of a Pattern at the Element Level*

A pattern form can have a number of mandatory and optional elements [2, 6]. The elements that are considered mandatory are: (pattern) name, context, problem, forces, and solution. The optional elements that can be useful include examples and resulting context. There are a number of issues that can arise at the element level:

- **Name.** The name of a pattern may not be evocative [2] or pronounceable. This can be an impediment on the selection of patterns and use of patterns for communication by its consumers.
- **Context.** The context of a pattern may not be explicitly stated. This can be an obstacle towards the selection of patterns. For example, the COLOR-CODED SECTIONS pattern [8] is not suitable for situations where the users have some form of color deficiency.
- **Problem.** The problem of a pattern may not be explicitly stated [3] and/or may not be context-free. In general, the problem description can suffer from the issues that affect a software requirement statement.
- **Forces.** The forces of a pattern may not be explicitly highlighted. For example, it is possible to make the forces explicit by listing them individually. For a given problem in the development of S, let there be multiple candidate patterns, say, pattern complements [2]. Then, the absence of forces makes it difficult to select the appropriate pattern.
- **Solution.** It is evident that the quality of the solution of a pattern will directly affect the quality of S as this is the place where conceptual reuse is realized. However, the solution of a pattern could contain errors [3].
- **Examples.** The solution of a pattern may not have gone through an evaluation, at one of the *PLoP 'family' of conferences or otherwise, and may not include three examples as suggested by the 'patternity test' [2].
- **Resulting Context.** The consequences of applying a pattern may not be discussed. In such a case, the forces that the solution resolves, partially or completely, and the ones it does not resolve, may not be known.

V. CONCLUSION

If patterns are to be considered as entities of conceptually reusable knowledge that lead to the development of 'high-quality' software systems, then the quality of these patterns themselves must be considered as a first-class concern, and should be treated as such. It is the exploration of the semiotic quality of a pattern that is of particular interest. Indeed, a semiotic quality model for pattern descriptions could be useful in the selection of the appropriate pattern from a given set of patterns that span multiple pattern collections such as different pattern languages.

ACKNOWLEDGMENT

The author is thankful to Peter Grogono for useful discussions, and to the reviewers for detailed feedback and suggestions for improvement.

REFERENCES

[1] I. Araujo and M. Weiss, "Linking Patterns and Non-Functional Requirements," The Ninth Conference on Pattern Languages of Programs (PLoP 2002), Monticello, U.S.A., September 8-12, 2002.

[2] F. Buschmann, K. Henney, and D. C. Schmidt, "Pattern-Oriented Software Architecture, Volume 5: On Patterns and Pattern Languages," 2007, John Wiley & Sons.

[3] M. Fowler, D. Rice, M. Foemmel, E. Hieatt, R. Mee, and R. Stafford, "Patterns of Enterprise Application Architecture," 2003, Addison-Wesley.

[4] D. A. Garvin, "What does Product Quality Really Mean?" MIT Sloan Management Review, Vol. 26, No. 1, 1984, pp. 25-43.

[5] F. Khomh and Y. -G. Guéhéneuc, "Perception and Reality: What are Design Patterns Good For?" The Eleventh ECOOP Workshop on Quantitative Approaches in Object-Oriented Software Engineering (QAOOSE 2007), Berlin, Germany, July 31, 2007.

[6] G. Meszaros and J. Doble, "A Pattern Language for Pattern Writing," in: Pattern Languages of Program Design 3. R. C. Martin, D. Riehle, and F. Buschmann, Eds. Addison-Wesley, 1998, pp. 529-574.

[7] K. Segerståhl and T. Jokela, "Usability of Interaction Patterns," The ACM CHI 2006 Conference on Human Factors in Computing Systems (CHI 2006), Montreal, Canada, April 22-27, 2006.

[8] J. Tidwell, "Designing Interfaces: Patterns for Effective Interaction Design," 2005, O'Reilly Media.

Pattern-based Design

Generic Patterns: Bridging the Contextual Divide

Marc Boyer
Computer Science Department
University of Manitoba
Winnipeg, MB, Canada
marc.boyer@shaw.ca

Vojislav B. Mišić
School of Computer Science
Ryerson University
Toronto, ON, Canada
vmisic@ryerson.ca

***Abstract*—Correct application of design patterns requires bridging the cognitive gap between the problem and implementation domains, as well as identifying the proper pattern amongst many to use in correctly modeling the user domain. As a result, pattern-based design is neither as efficient nor as effective as it might be. Hence, we propose an improved two-step pattern design process: first pick a pattern that matches domain requirements from a small number of generic, context-free patterns; then concretize the pattern further into one of the industry-standard, context-dependent patterns. In this manner, identifying patterns in requirements tightly bound to their context becomes both faster and more accurate, as demonstrated in a real-world example.**

Keywords-software design, design patterns, design quality

I. INTRODUCTION

Ever since their introduction in the mid-nineties [3][6], design patterns and their siblings at various levels of abstraction [4][5] have been advertised as being one of the most efficient ways to reuse design knowledge: "each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem …" [2]. The traditional approach to using design patterns, as advocated by the original authors [3][6], relies on analyzing requirements, identifying the patterns therein, and then using the patterns to model the software's concrete elements: functional classes, instantiated objects, and their interactions. In this manner, design patterns lead to reuse of software engineering knowledge, and ultimately lead to an increase in designer productivity and an improvement in the quality of software products.

Yet despite extensive training and a wealth of pattern-related information (including a number of catalogs [7]), proper use of patterns is still problematic [1][9][11], and advertised benefits of patterns and pattern-based approaches to software design are still hard to realize in practice. What is the cause of this discrepancy: are people misusing available design patterns? Are requirements too complex to model using available design patterns? Or is the main reason for the discrepancy between promise and reality the cognitive gap between the domain requirements and the pattern descriptions [17], and perhaps even the insufficient discriminatory power of the pattern descriptions themselves?

To remedy this situation, we propose a set of generic or context-free patterns that can be more easily identified within domain requirements, and a two-stage design process through which the generic patterns can be concretized to one of the already-known, industry-standard, context-dependent patterns.

The paper is organized as follows: in Section II we discuss briefly the limitations of the traditional pattern-based techniques used to design software, and illustrate the problem with experimental findings. We also discuss more recent techniques used to address these limitations, and explain why they may not deliver the anticipated rewards of using design patterns to model software. Section III outlines the proposed approach to mapping requirements to models using context-free patterns, with a small example. Section IV concludes the paper and discusses areas of further research.

II. PATTERN PROBLEMS AND SOLUTIONS

In the traditional pattern-based design approach, designers analyze the requirements and use a design pattern to map them to a software model [3][6]. Patterns are sought and identified in the domain requirements, thus providing the foundation upon which full-fledged software designs are developed. The right pattern is identified by matching the contextual elements discerned in the domain requirements with the problem context elements provided by the pattern description. Once identified and applied, patterns are subsequently enriched by adding appropriate detail, and rendered concrete so as to facilitate their implementation through coding.

The success of this process—the selection of the best pattern to apply in a specific domain context—is critically dependent on the accuracy of the mapping between the domain elements and the pattern elements. However, the current practice of pattern use relies heavily on the context in which the problem and the pattern (i.e., the solution to the problem) are defined. For the correct pattern to be applied, the context in which the domain requirement is defined must match the context of the pattern definition. Depending on the degree to which the two semantically match, bridging the semantic gap between domain and design may or may not be easy. Moreover, if model designers must re-define the pattern to accommodate domain-context idiosyncrasies, an activity which more often than not will require substantial skill in abstraction or pattern-creation on the part of the designers, the expected benefit of using a supposedly

reusable pattern may in fact be entirely lost – a problem that lies at the very heart of the design process.

Equally problematic are the situations in which domain and pattern contexts do not seem to match (and no suitable pattern can be found to model the requirements), or the designer identifies two or more equally viable candidate patterns. In either case, a non-conforming pattern may eventually be selected, or an existing pattern may be modified (possibly incorrectly) to match the problem, leading to further complications down the road.

A. *The Cognitive Divide*

As an illustration of the difficulties inherent in using the traditional approach to pattern identification in requirements, a group of 46 fourth-year students taking the Software Engineering II course at the University of Manitoba were presented with a list of some fifteen common patterns, and then asked to (a) identify those patterns they thought they were capable of defining or describing; and (b) identify which of those patterns were present in a simplified description of a real life system, shown in Fig. 1 below. The students were fairly uniform in their prior education: all of them had taken earlier courses in basic programming, object orientation, and software engineering; in these courses, they were exposed to several of the design patterns found in the catalog [6]; in addition, 30 of them had taken the basic database course and were familiar with the design patterns specific to this knowledge domain.

Table I shows some of the patterns present in the questionnaire, the results of the students' self-assessment of pattern-knowledge (answer to the question "I can describe … the following patterns"), and the level of their usable knowledge (answer to the question "I detected … the pattern … in user requirements"). As can be seen, students were quite confident regarding the Adapter/Wrapper and Client-Server patterns, less so with regard to the Shared Database and Façade patterns, and not very confident at all with regard to the Bridge pattern. However, their definitional knowledge is ill reflected in their practical or usable knowledge of the patterns. About 80% of students managed to correctly identify the Client-Server pattern, but only about one quarter of them (or less than a third of those who claimed to know the proper description) succeeded in identifying the Adapter/Wrapper pattern. Similar success rates were obtained for the Bridge pattern. The discrepancy between the ability to define or describe a pattern and the ability to actually identify it in the domain requirements is likely due to the *cognitive divide* between the context of the domain requirement and the context in which the pattern is defined. Describing the pattern within its own contextual domain was not sufficient to identify it in another domain; only when the domain context matched the pattern context to a large degree, could a high accuracy of pattern identification be expected.

For instance, the Adapter/Wrapper pattern, according to [6], "converts the interface of a class into another interface clients expect", while the specification in Fig. 1 does not

> GreenEarthWay Ltd. hires you to design a new computer system to run their grocery stores, which are to be connected using the existing network.
>
> Staff will increase stock counts in the new system when there is an inventory delivery to a store. Cash registers will decrease stock counts when there is a sale to a customer. Store managers will run reports on inventory counts in the computer system. Each store is connected to the central Winnipeg office computer for all credit card transactions.
>
> GreenEarthWay wants the new system to interact with an existing Microsoft Exchange application using Office protocols to cut costs. However, the company also wants you to design the system so that it can easily switch to using an open-source email component in the future.

Figure 1: Example specification for a new system.

talk about classes and interfaces; instead, it asks for the system to be designed "so that it can easily switch to using an open-source email component in the future." Obviously, there is a significant cognitive distance or gap between the business requirements context and the pattern definition context. (Other, even more striking examples of this gap can easily be found in the specification of Fig. 1, despite its apparent brevity.) To bridge such a gap, the designer would have to perform an accurate mapping from one context (the domain context) to another (the pattern context) directly – a process that offers abundant opportunity for error [16].

However, this gap is only part of the problem. Note that about half of the students claimed to be able to define/describe the Façade pattern which, according to [6], "defines a higher-level interface that makes the subsystem easier to use." The requirements call for an adapter to isolate external components from the rest of the software, rather than for a façade to isolate architectural tiers. Yet almost a quarter of the entire student group has (incorrectly) found the Façade pattern in the specification of Fig. 1. This type of error in pattern identification may be attributed to the similarity—in other words, *too small cognitive distance*—between the Façade pattern and other patterns which are actually present in the specification (Adapter/Wrapper, in this case). In fact, similarity between pattern definitions is quite common. For example, Bridge is said to "decouple an abstraction from its implementation so that the two can vary independently", whilst Proxy "provides a surrogate or placeholder for another object to control access to it" [6].

Note that these pattern definitions again talk about OO-domain specific classes and objects, rather than about higher-level concepts that could be usefully mapped to a problem domain (i.e., one or more different portions of user requirements), and it is not too difficult to mistake one for another. In fact, looking at the Intent sections of the 23 patterns in [6], one can't help noticing that more than two-thirds are defined using implementation-oriented terms like "object", "class", or "interface". It should come as no surprise, then, that future designers (in this case, fourth-year students) seem to have difficulty identifying such patterns in functional requirements where no such "objects", "classes," or "interfaces" are present.

Table I. Some of the patterns identified in the example specification of Fig. 1.

Requirement	Pattern	"I can describe or give a definition of the following design patterns"	"I detected the following design patterns in the user requirements above"
Each store is connected to the central Winnipeg office computer.	Client-Server	96.77%	80.65%
The new system must use the existing network.	Bridge	19.35%	6.45%
Store managers will run reports on inventory counts.	Shared Database	54.84%	54.84%
Easy switch to another component in the future.	Adapter/ Wrapper	87.01%	25.81%
—not present—	Façade	51.61%	22.58%

Thus it seems safe to conclude that *learning a pattern using its contextual definition does not of itself guarantee error-free use of that pattern*. Available design patterns are very strongly coupled to the domain context which gave birth to them – but this can also limit their utility in domains with other contextual requirements.

The validity of this observation is witnessed by a number of aids that have been developed to help designers perform the contextual mapping process more efficiently and more reliably. Yet, as we shall see, even these more modern techniques still fail to successfully address both the domain-to-model cognitive distance and pattern-similarity problems described above.

B. Better Descriptions

Intuitively, providing a better description of a design pattern, i.e., more information about its use in context, ought to allow the designer to better understand the pattern and, by extension, improve the likelihood of the designer selecting the pattern that correctly addresses the problem at hand.

However, too detailed a description may also mean that the pattern is burdened with too much contextual information. The designer may find a "pattern" bound to a context in this way much more difficult to apply to patterns discerned in a very different requirement context. For instance, a pattern like Master-Slave, if too tightly bound to hardware-domain contextual descriptions, might with difficulty be used to describe the interactions of, say, a number of employees and of their (profit-seeking) employer. Therefore, pattern descriptions can be improved only so much, because further pattern specification might in fact hamper, rather than facilitate, pattern identification and application, since additional pattern-contextual information may only reduce the pattern's applicability to other, possibly very different, application domains.

C. Pattern Languages

The push for better pattern descriptions has also produced several proposals for more formal (or at least better structured) pattern definition or description languages [18]. A pattern description language is essentially a system of constraints on the words and word-relationships used to define patterns. While a more structured language might seem to reduce the potential and likelihood for designer pattern-matching error, because the pattern descriptions will now be more rigidly defined, the very rigidity in the pattern description might actually make it more difficult for a designer to match the pattern's contextual description to the highly differentiated requirements defined for real world interactions. As a result, pattern languages have limited use in bridging the contextual gap between domain requirements and software model that we have described above.

D. Pattern Catalogs

A more straight-forward and popular solution seems to be the compilation of pattern catalogs. Gamma *et al*. [6] provide the description for 23 design patterns. A number of much more elaborate pattern catalogs have been published since – a recent survey [7] found that catalogued patterns numbered in the hundreds. An exhaustive pattern catalog might seem to assist designers in the task of pattern identification by increasing the number of patterns immediately available to them. However, an unfortunate consequence of so many choices is reduced cognitive distance between them – which in turn makes it more difficult for the designer to identify the correct pattern to use to model a given set of domain requirements. Thus, the proliferation of patterns effectively eliminates two of the main advantages of design patterns: in the words of Agerbo and Cornils, it "will make it too laborious to find and use the encapsulated experience, and [it] will make the common vocabulary too large to be easily comprehended" [1].

E. Pattern Classification

To make catalogs more accessible to designers, catalog providers oftentimes provide some kind of classification or grouping of the patterns within their catalogs. Gamma *et al*., for instance, provide a simple categorization of patterns into Structural, Creational, and Behavioral groups [6]. However, most of the patterns in their catalog have both structural and behavioral aspects, and quite a few—including Structural and Behavioral ones—have creational aspects as well. Compounding the problem is the fact that quite a few patterns within each group are so very similar that they can be easily misidentified: what is the cognitive distance between Façade and Wrapper, for instance? Consequently,

an a priori categorization of patterns by the 'most obvious' pattern feature may not be of much help to designers at all.

F. Refactoring to Patterns

Finally, we mention a radically different approach known as refactoring to patterns or R2P [8]. This approach differs from the ones previously mentioned. R2P seeks primarily to identify design patterns a posteriori, that is, after software code has been created and refactored, rather than before. The technique does not seek to map requirements to a design model. Instead, it starts with the code and reverse-engineers it into an (ever-improving) design model. The technique assumes that the design model matches the domain requirements, because unit tests are available to validate that every domain requirement is in fact implemented in a specific code artifact.

In practice, this means that if the designer can easily identify a pattern in the requirements suitable for use in structuring the code to match real-world relationships, then it is applied and concretized. If not, coding of the software proceeds without any design or pattern at all. Once the software product is fully operational, designers seek to identify within, and impose design on, the existing code base using existing design patterns as a guide to the refactoring effort.

In light of our discussions, the success of this approach stems from the fact that the cognitive distance between a portion of the code and the pattern that implements it, is much smaller than the corresponding distance between the original domain requirement and the concrete pattern in question. Smaller cognitive distance, as we have seen, leads to simpler and more accurate identification of patterns. Yet even the R2P approach is critically dependent on existing pattern definitions which, as we have seen, may be difficult to distinguish and use given their semantic overlap. For example, the class diagram for the Strategy pattern is surprisingly similar to that for the State pattern [6]; if such homonyms are discerned in the code base, they may look like an excellent candidate for refactoring – but even then, choosing the right target pattern may be difficult if the class, variable, and method names fail to provide clear clues as to the exact function of the class within its own environment.

As a result, the problem of finding the right design pattern is not completely eliminated by starting from the code base rather than from the requirements. If we don't have a suitable pattern to map to, we cannot be assured that we have in fact bridged the requirement to model contextual divide. If the code base is poorly organized, over-engineered, and/or excessively patched [9], the right target pattern for the code base might be as difficult to find as it is to discern in a poorly defined set of requirements.

Still, the concept of adjusting the development process to eliminate the dual problems of cognitive distance and pattern similarity seems to have more potential for improving the effectiveness of the pattern-based design process than a simple modification of pattern definitions. In fact, most improvement may be obtained by a judicious combination of the two. As will be seen in the following section, this is the essence of our proposed approach.

III. Generic Patterns and the Two-Stage Pattern-Based Design Process

The discussions above highlight the fact that design patterns must possess two main aspects to lend themselves to successful use. First, the pattern must have a conceptual aspect that can easily map to the contextual information found in different domains (the domain-contextualized requirements). Second, the pattern must also have a concrete aspect that can easily map to software models and (subsequently) to code specifications (the implementation-contextualized constructs). If the design patterns are too conceptual or conversely, too concrete, their utility to the designer may be significantly reduced.

A. The Two Stages

To reduce the cognitive gap between domain requirements and software model, we propose a two-stage process similar to the transformation method outlined in [12]. In the first stage, pattern-discovery, the designer focuses on mapping the domain requirements to a pattern that is generic or context-free. Since the number of these generic patterns will be small, identification should be simpler and faster; as a result, the risk of making an erroneous pattern choice will be much reduced.

In the second stage, that of iterative pattern specialization or contextualization, the designer focuses on mapping the domain requirements to ever more precise pattern contexts, until one or more concrete patterns (or concrete pattern specializations) are found that might be of use in modeling the requirements. At this point, one of the concrete patterns in the generic group—the one that provided the best contextual match to the domain requirements—would be selected and subsequently elaborated in the traditional manner to produce the actual software design.

The expected benefit to using this two-step approach is the reduced risk of a designer picking the wrong concrete pattern to model the requirements, a risk that we have shown to be very real given the contextual constraints inherent in existing patterns. Placing the concrete patterns in a generic pattern or sub-pattern set requires the designer to iteratively match subsequently more specialized patterns to the requirements, in a step-wise process which eventually leads the designer to a more judicious concrete-pattern choice. This removes the need for the designers to sift through voluminous pattern catalogs for a pattern that may (but probably will not) fit their requirements.

B. Generic, Sub-Generic, and Concrete Patterns

Let us now try to identify some of the generic patterns in question. For example, consider the patterns from the student survey above: Façade, Bridge, Wrapper, and Proxy. What is the feature common to all of them that clearly distinguishes them from patterns such as Shared Data or Client-Server?

The answer: they all provide *isolation* of some sort between two real-world entities. Therefore, all four of these patterns are, in fact, specializations or contextualizations of the same generic pattern which we may call Isolate.

Not all kinds of isolation operate in the same way: sometimes isolation merely provides a simple *transfer* of information between different interfaces; at other times, the isolating element uses a specific protocol to control and *mediate* between the entities. It makes perfect sense, then, in this case as well as in others, to add a sub-class or, rather, additional important pattern qualifications or features, in-between the top-level generic and concrete patterns. The sub-generic patterns in this layer capture additional information about the manner in which the generic pattern is expected to fulfill its duties: in the example given, the more specialized generic patterns we have identified are Isolate/Transfer (a pattern where the isolation features only a transfer of information) and Isolate/Mediate (a pattern where the isolation features mapping or information mediation services).

We use features in the concrete patterns to identify important pattern commonality of use in specifying a generic. We prioritize the features to ensure that we are classifying the generics according to their semantic importance for accurately describing the predominant characteristics of the pattern. We build a hierarchy of generic patterns from the bottom up, so that it can be used by designers from the top down.

In this manner, the designers are provided with a conceptual tool to help them bridge the contextual divide: the generic is firmly rooted in the concrete patterns from which they draw their most distinctive high-priority features and commonality; and yet due to the more conceptual nature of the generic, it is much easier for designers to discern these patterns in other contextual environments, that is, in different requirement domains.

C. A Preliminary Classification

An initial evaluation of the more commonly used concrete patterns yields the classification presented in Table II. Note that this classification is but a preliminary effort, and that we are currently working on a more precise and more refined classification scheme, as well as a pattern-evaluation procedure that can help ensure a proper classification within the generics of the patterns currently compiled in the catalogs.

A few comments are in order here. First, we note that there is no conceptual limit on the number of levels that may be used to qualify the generic and sub–generic patterns before committing to a fully context-dependent pattern. In practice, however, one might want to limit the levels to high-value pattern features only, since this has a direct bearing on the complexity of the design process.

Second, the separation of concrete from generic patterns into a multi-level categorization structure on the basis of the common features of concrete patterns allows for more precise design by defining similar yet distinct concrete patterns from the same generic or sub-generic pattern. For example, most of the literature treats the Wrapper and Adapter patterns as synonyms, but some authors consider the two to be different patterns [15]; yet others consider the Wrapper to be more generic than Adapter, Decorator, or even Proxy [13]. In our view, a Wrapper should isolate a component from a system, just as a Façade isolates one tier from another. An Adapter true to its name should provide an endpoint-to-endpoint mapping that enables a component to adapt to its environment. As a result, the Wrapper pattern belongs to the Isolate/Transfer pattern set, and the Adapter to the Isolate/Mediate pattern set. This precision and clarity in pattern names and descriptions would not be possible without the distinguishing power of generics.

D. Related Work

Pattern layering has been proposed elsewhere [19], but only in the context of interdependencies of concrete patterns, rather than in the hierarchical sense adopted in the current paper. Also, the proliferation of design patterns has been criticized in [1], and "restriction in the formation of Design Patterns, leading to a reduction in [their] number" was proposed as a possible remedy. Still, the discussion remains firmly within the realm of programming languages in general – which makes it part of the problem, not of the solution.

Kniesel, Rho, and Hanenberg discuss the need for generic patterns [10] but in the context of a language for defining generic aspects, which is again much closer to the implementation domain than the approach described here.

Riehle and Züllighoven [14] propose a hierarchical layering of patterns, and emphasize the distinction between conceptual patterns ("whose form is described by means of the terms and concepts from an application domain") and design patterns ("whose form is described by means of software design constructs"). Yet they stop short of describing the actual design process for transforming the requirements into conceptual patterns, into design patterns, and, ultimately, into programming patterns which are also discussed in their paper.

The classification scheme proposed by Tahvildari and Kontogiannis [16] attempts to establish relationships between patterns and to organize them into hierarchies; yet it does not venture beyond the categorization from [6], and

Table II: A preliminary classification of generic and specialized patterns.

Isolate	Transfer	Façade, Wrapper, Proxy
	Mediate	Bridge, Broker, Adapter, Mediator
Share	Provide	Repository, Database, Client-Server
	Collaborate	Producer-Consumer, Publisher-Subscriber, Blackboard
Entity	Generate	Factory, Builder
	Provide	Singleton, Pool, Queue, Stack, Tree
Activity	Detect	Listener, Observer
	Control	Master-Slave, Pipe-and-Filter

Table III. The two-stage pattern-based design process: mapping proceeds from left to right.

requirement	generic pattern	sub-generic pattern	concrete pattern
The new system must use the existing network.	Isolate	Mediate	Bridge
Each store is connected to the central Winnipeg office computer.	Share	Provide	Shared Database, Client-Server
Store managers will run reports on inventory counts.	Share	Provide	Repository or Shared Database
The new system should interact with an existing Microsoft Exchange application using Office protocols.	Isolate	Mediate	Bridge
Easy switch to another component in the future.	Isolate	Transfer or Mediate	Wrapper or Adapter
Cash registers will decrease stock counts when there is a sale to a customer.	Activity	Detect	Observer

focuses only on the relationships between patterns, without any link to the requirements they are meant to implement. Their approach is also dependent on subjective assessments of pattern relation, unlike the generic identification process that uses the intrinsic features of related concrete patterns to find the relationships between a pattern and its parent sets.

E. How Does It Work?

The process of pattern identification and its results are summarized in Table III, where the requirements in the leftmost column are mapped to a generic and sub-generic pattern to the right, and ultimately to a concrete pattern in the rightmost column. Due to space limitations, we show only the final results, rather than the details of the process.

IV. CONCLUSION

Current approaches to pattern-based software design suffer from the cognitive divide between the domain context in which requirements are specified and the implementation context in which suitable design patterns are to be found. To bridge the divide, we have proposed a two-stage process in which the domain context is first matched by generic pattern concepts, and subsequently concretized to standard design patterns and implemented accordingly. Initial experience indicates that this approach offers a better chance of delivering on the promise of pattern-based design: namely, an efficient and accurate mapping of domain requirements to the design model with significantly reduced risk of errors to all designers.

REFERENCES

[1] E. Agerbo, A. Cornils. "How to preserve the benefits of Design Patterns," *Proc. OOPSLA '98*, Vancouver, BC, pp. 134-143, 1998

[2] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel. *A Pattern Language*. Oxford University Press, NewYork, 1977.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal. *Pattern-oriented software architecture: a system of patterns*. John Wiley & Sons, 1996.

[4] M. Fowler, *Analysis Patterns: Reusable Object Models*, Addison-Wesley, 1997.

[5] M. Fowler, *Patterns of enterprise application architecture*, Addison-Wesley, 2003.

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

[7] S. Henninger and V. Corrêa, Software Pattern Communities: Current Practices and Challenges, in *Proceedings of the 14th PLoP*, Allerton Park, IL, 2007.

[8] J. Kerievsky, *Refactoring to Patterns*. Addison-Wesley, 2005.

[9] F. Khomh and Y.-G. Guéhéneuc, Do Design Patterns Impact Software Quality Positively? *Proc. CSMR 2008*, Athens, Greece, pp. 274—278, 2008.

[10] G. Kniesel, T. Rho, and S. Hanenberg, Evolvable Pattern Implementations need Generic Aspects. *Proc. RAM-SE'0*, Oslo, Norway, pp. 111-126, 2004

[11] W. B. McNatt and J. M. Bieman. Coupling of design patterns: Common practices and their benefits. *Proc. COMPSAC 2001*, Chicago, IL, pp. 574-579, October 2001

[12] H. J. Nelson and D. E. Monarchi. Ensuring the quality of conceptual representations. *Software Quality Journal* **15**:213-233, 2007

[13] J. Noble and R. Biddle, Patterns as signs, *Proc. ECOOP 2002*, Málaga, Spain, pp. 368-391, 2002.

[14] D. Riehle and H. Züllighoven, Understanding and using patterns in software development. *Theory and Practice of Object Systems* **2**(1):3-13, 1996.

[15] D. Roberts and R. Johnson, Evolving frameworks: A pattern language for developing object-oriented frameworks. In *Pattern Languages of Program Design* (*Proc. PLoP'94*), Vol. 3, pp. 471-486, Addison-Wesley, 1998.

[16] L. Tahvildari and K. Kontogiannis, On the Role of Design Patterns in Quality-Driven Re-engineering, *Proc. Euro. Conf. Software Maint. and Reeng. CSMR'02,* pp. 230-240, Budapest, Hungary, March 2002

[17] S. Wagner and F. Deissenboeck, Abstractness, Specificity, and Complexity in Software Design. *Proc. ROA'08*, pp. 35-42, Leipzig, Germany, 2008.

[18] L. Welicki, O. San Juan, and J. M. Cueva Lovelle, A Model for Meta-Specification and Cataloging of Software Patterns, *Proceedings of the 12th PLoP*, Allerton Park, IL, 2005.

[19] W. Zimmer. Relationships between design patterns. In *Pattern Languages of Program Design* (*Proc. PLoP'94*), pp. 345-364, Addison-Wesley, 1994.

Reporting the Implementation of a Framework for Measuring Test Coverage based on Design Patterns

Kazunori Sakamoto, Hironori Washizaki, Yoshiaki Fukazawa
Dept. Computer Science and Engineering
Waseda University
3-4-1 Okubo, Shinjuku-ku, Tokyo 1698555, Japan
kazuu@ruri.waseda.jp, washizaki@waseda.jp, fukazawa@waseda.jp

***Abstract*—Fault-free software is highly desirable, and so sufficient software testing plays an important role in attempts to realize a fault-free state. Test coverage is an important indicator of whether software has been tested sufficiently. However, existing measurement tools are associated with several problems, such as the cost of new development, the cost of maintenance, and inconsistent and inflexible measurement. In this paper, we propose a consistent and flexible test coverage measurement framework that supports multiple programming languages. We implemented our framework based on design patterns such as Template Method pattern and Macro Command pattern. Thus we report the success of the implementation of our framework based on design patterns, and we confirm the benefit of design patterns.**

***Keywords*-Design pattern; Framework; Software testing; Test coverage; Code coverage; Metrics**

I. Introduction

Test coverage (code coverage) is an important measure used in software testing. It refers to the degree to which the source code of a program has been tested and is an indicator of whether software has been tested sufficiently. Design pattern is an important software pattern which is a general reusable solution to a commonly occurring problem in software design. Pattern formulates the know-how of solution to a commonly occurring problem to be reused by people. There are multiple levels in test coverage, such as statement coverage, decision coverage and condition coverage. Developers select a suitable level according to the purpose of their software testing[1].

Measurement tools are necessary in order to measure the coverage of various programs accurately, and test coverage measurement tools have become widely available. Many measurement tools are offered for major languages such as C or Java. However, measurement tools for legacy languages such as COBOL and minor languages such as Lua are not readily available and only at some considerable expensive. Moreover, it is more difficult to have access to measurement tools for newly defined languages and for existing languages with some language specification changes because each existing tool is specific to a certain language specification. Such a situation drives the need for the development of some framework or tool that corresponds to a variety of languages including new languages in the future.

In this paper, we propose a consistent and flexible test coverage measurement framework that supports multiple languages. Our framework extracts commonalities among multiple languages, and disregards variability by focusing on the syntax of the languages. We implemented our framework based on design patterns such as Template Method pattern and Macro Command pattern, thus we confirm the benefit of design patterns.

Our framework is now freely available via the Internet[2].

II. Problems in conventional approaches

The following summarizes the problems with existing measurement tools. The problems are in cost of new development, in cost of maintenance, in inconsistent measurement, in inflexible measurement and in Incomplete measurement but we focus only the cost of new development.

The variety of languages is becoming more diverse. Moreover, coverage measurement tools are often unavailable for a number of legacy and/or minor languages due to a lack of community or non-commercial efforts. So, measurement tools for these languages are necessary. A measurement tool consists of the following 4 functions: a syntactic analyzer that interprets syntax from source code, a semantic analyzer that interprets the meaning of syntax such as a statement and a conditional branching, a measurement function for test coverage, and a display function for measurement results. Generally, it is difficult to implement these functions. Therefore, the cost necessary for development is high.

III. Coverage measurement framework supporting multiple programming languages

We propose a test coverage measurement framework that supports multiple languages, and which will solve and alleviate the problems described above.

The framework is a reusable software architecture and provides a generic design as some similar applications. The application can be implemented by adding application-specific code as user code to the framework[4].

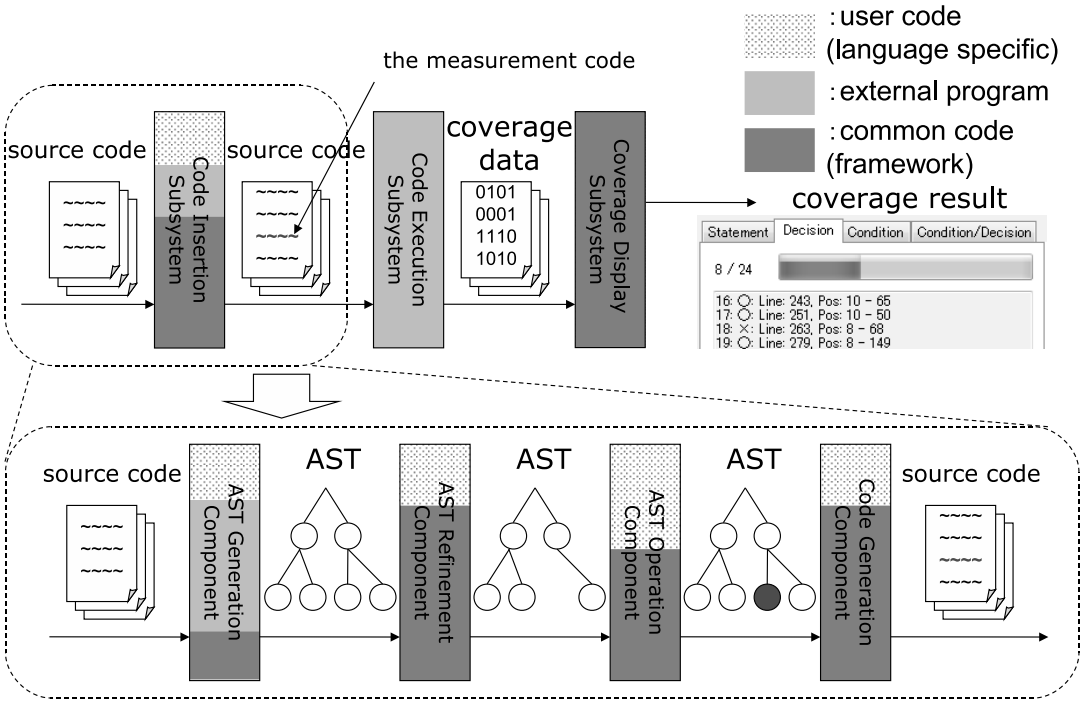


Figure 1. The entire design of our framework

The entire design of our framework and the processing flow is shown in Figure 1. Our framework consists of three subsystems: the code insertion subsystem, the code execution subsystem and the coverage display subsystem. Moreover, the code insertion subsystem consists of four components: the AST (Abstract Syntax Tree) generation component, the AST refinement component, the AST operation component and the code generation component. We implemented their with design patterns, so we get high reusability and reduce the cost of new development.

The process of the coverage measurement is as follows.

1) Generation of AST from source code
2) Insertion of code for measurement on AST
3) Generation of source code from AST
4) Execution of generated source code and collection of measurement information
5) Display of measurement results from test coverage

Our framework inserts the measurement code into the source code, and the test coverage is measured by executing the program. When our framework inserts the measurement code, it collects information such as the location information of the measurement elements in the source code.

Our framework is designed as an object-oriented framework with object-oriented programming and design patterns. Our framework provides common code for language independent processing and also provides structure to help to write user code for language dependent processing. Moreover, the insertion on AST simplifies the insertion processing. In this way, our framework reduces the cost of new development and maintenance. However, our framework targets only procedure-oriented languages due to the mechanism used for measurement which involves inserting the measurement code.

IV. Implementation of our framework

We implemented our framework in .NET Framework 3.5 SP1. Our framework enables the implementation of language specific processing by adding user code such as assembly files that run in .NET Framework 3.5 SP1 or older, or script files in languages supported by Dynamic Language Runtime (DLR)[13]. DLR is .NET library that provides language services for several different dynamic languages. In this way, our framework helps to add user code.

We now show sample code as a sample measurement tool implementation that measures test coverage in Java, C and Python by using our framework.

A. Code insertion subsystem

The code insertion subsystem consists of the following components: the AST generation component, the AST refinement component, the AST operation component and the code generation component.

1) AST generation component: converts the obtained source code into an AST as an XML document. In this sample, this component consists of two functions: AST builder and the caller of AST builder. AST builder is user code which is deployed as an external program. AST builder

is implemented using compilers such as SableCC[5] for Java, ANTLR[6] for C and Python standard library for Python. The caller of AST builder is common code which is designed by using Template Method pattern[7].

The Template Method pattern reorganizes the processing steps between the coarse-grained process flow and fine-grained concrete processing steps. The former is placed in a superclass method and the latter is placed in subclass methods. The latter is triggered by the former by calling superclass abstract methods which are actually implemented in subclasses.

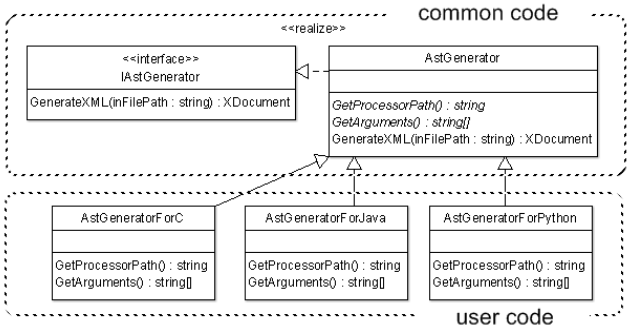


Figure 2. The class diagram of the AST generation component

The class diagram of UML[8] related to this component is shown in Figure 2. The `AstGenerator` is an abstract class that is designed by applying the Template Method pattern. The sample user code of this component for Java is shown in List 1.

List 1. AstGeneratorForJava.cs

```
using System.ComponentModel.Composition;

namespace CoverageFramework.AstGenerator.Java {
  [Export(typeof(IAstGenerator))]
  public class AstGeneratorForJava : AstGenerator {
    private static readonly string[]
        _arguments = new[] {
      "-jar", "../Java/Java.jar",
    };
    protected override string FileName {
      get { return "java"; }
    }
    protected override string[] Arguments {
      get { return _arguments; }
    }
  }
}
```

Therefore, the use of the compiler compilers and common code eases the implementation of this component.

2) AST refinement component: changes the structure of AST to operate AST easily. In the sample, this component removes the unnecessary nodes of AST such as nonterminal nodes which have only nonterminal nodes as child nodes. Moreover, this component converts single-line if statements into multi-line if statements. Our framework provides the almost processing as common code.

3) AST operation component: has roughly three functions: the enumeration of subtrees, the generation of subtrees and the replacement of subtrees. The enumeration function locates the position in which the measurement code is inserted. For example, this function locates the position of all atomic logical terms in conditional expressions in Python. Our framework provides a large part of this function as common code which is designed by using the Template Method pattern.

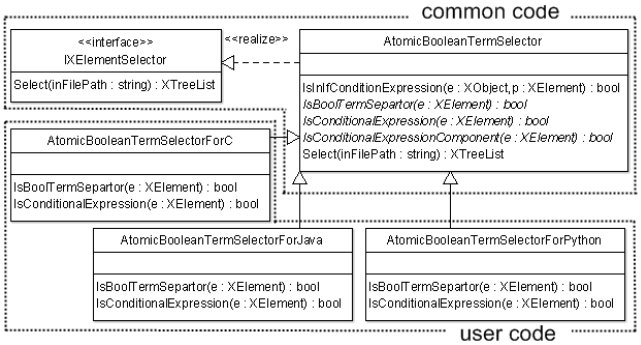


Figure 3. The class diagram of the AST operation component

The class diagram related to this function is shown in Figure 3. The `AtomicBooleanTermSelector` is an abstract class that is designed by applying the Template Method pattern. The sample user code that enumerates the atomic logical terms for Python is shown in List 2.

List 2. AtomicBooleanTermSelectorForPython.cs

```
using System.Linq;
using System.Xml.Linq;
using System.ComponentModel.Composition;

namespace CoverageFramework.Element.Selector.Python {
  [Export(typeof(IXElementSelector))]
  public class AtomicBooleanTermSelectorForPython
      : AtomicBooleanTermSelector {
    private static readonly string[]
      _condComponentNames = new[]
    { "or_test", "and_test", };
    private static readonly string[]
      _condNames = new[]
    { "or_test", "and_test", };
    private static readonly string[]
      _condOpValues = new[]
    { "or", "and", };
    protected override bool
        IsBoolTermSepartor(XElement e) {
      return !e.HasElements &&
        _condOpValues.Contains(e.Value);
    }
    protected override bool
        IsConditionalExpression(XElement e) {
      return _condNames.Contains(e.Name.LocalName);
    }
    protected override bool
        IsConditionalExpressionComponent(XElement e) {
      return _condComponentNames
        .Contains(e.Name.LocalName);
    }
  }
}
```

By implementing processing that judges whether the given

node is the measurement element, this code enumerates the atomic logical terms.

In addition, our framework provides some other classes as common code, such as the `XElementSelectorUnion` class, which integrates some enumeration results, and the `XElementSelectorPipe` class, which enumerates subtrees in other enumeration results. These classes are designed by applying the Command pattern[7].

The Command pattern is the design pattern that encapsulates a request and the parameters in an object. A command object that is combined with certain other command objects is called a Macro Command.

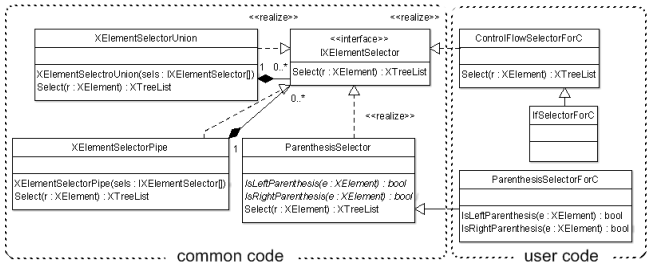


Figure 4. The class diagram according to enumeration subtrees

The class diagram related to the enumeration function is shown in Figure 4. The `XElementSelectorPipe` is a class as a Macro Command by applying the Command pattern. The usage of this class is shown in List 3.

List 3. The usage example of XElementSelectorPipe

```
1  var ifSelector = new XElementSelectorPipe(
2    new IfSelectorForC(),
3    new ParenthesisSelectorForC());
```

By combining the instance of the `IfSelectorForC` class, which enumerates the subtrees corresponding to the conditional sentence, and the instance of the `ParenthesisSelectorForC` class, which enumerates the subtrees corresponding to the parenthetic expression, this code enumerates the subtrees corresponding to the conditional expression for C.

Therefore, our framework reduces the size of the classes and promotes code reuse. Moreover, flexible measurement is achieved by adding processing that locates subtrees.

In addition, the generation functions are used to generate the subtrees corresponding to the measurement code. Our framework requires user code for this function. The replacement functions are used to insert the subtrees of the measurement code into the source code on AST. Our framework provides this function completely as common code.

4) Code generation component: converts the obtained AST into source code. When the AST has memorized almost all of the tokens corresponding text in source code, this component can be implemented simply by adding user code that outputs the tokens as they are without exception. Our framework provides common code that outputs the memorized tokens by applying the Template Method pattern.

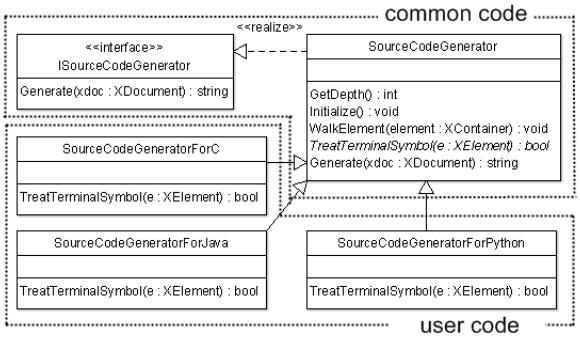


Figure 5. The class diagram of the code generation component

The class diagram related to this component is shown in Figure 5. The `SourceCodeGenerator` is an abstract class that is designed by applying the Template Method pattern. The sample user code of this component for Python is shown in List 4.

List 4. SourceCodeGeneratorForPython.cs

```
1  using System.Xml.Linq;
2  using System.ComponentModel.Composition;
3
4  namespace CoverageFramework.CodeGenerator.Python {
5    [Export(typeof(ISourceCodeGenerator))]
6    public class SourceCodeGeneratorForPython
7        : SourceCodeGenerator {
8      protected override bool
9          TreatTerminalSymbol(XElement element) {
10       switch (element.Name.LocalName) {
11       case "NEWLINE":
12         WriteLine();
13         break;
14       case "INDENT":
15         Depth++;
16         break;
17       case "DEDENT":
18         Depth--;
19         break;
20       default:
21         return false;
22       }
23       return true;
24     }
25   }
26 }
```

Neither the linefeed nor the indent is memorized in AST for Python. Accordingly, this component requires user code to output the linefeed and the indent to the corresponding terminal nodes.

Therefore, in our framework, most of this component is common code.

V. Evaluation

We evaluate our framework by comparing sample implementation that is developed by using our framework

with typical existing measurement tools, namely, Cobertura supporting Java and Statement coverage for Python[9] supporting Python.

We evaluate the new development cost by using the LOC (Lines of Code) of the program that inserts the measurement code and by the number of supporting coverage levels.

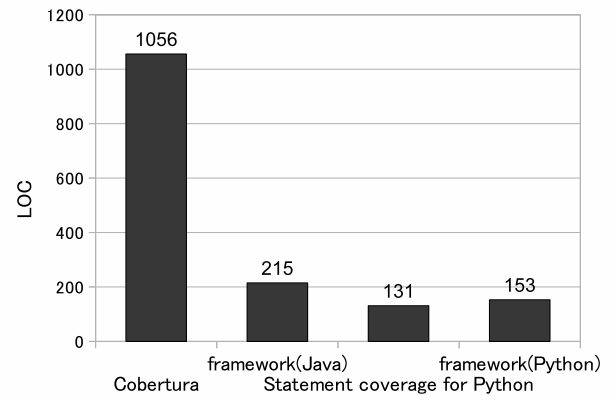


Figure 6. The LOC

Figure 6 shows the comparison results obtained with LOC. The LOC of Cobertura is 1056 lines, and the LOC of the sample implementation for Java is 215 lines. Cobertura uses BCEL[10] to insert byte code into the Java class file. BCEL is a library that conveniently provides users with the option to analyze, create, and manipulate (binary) Java class files. However, the sample implementation does not use the library except for our simple helper methods and the .NET standard library.

On the other hand, the LOC of Statement coverage for Python is 131 lines, and the LOC of the sample implementation with our framework for Python is 153 lines in Figure 6. Statement coverage for Python uses only the Python standard library. In addition, the LOC of the language independent and reusable part in the framework is 654 lines. Our framework can support new language with less cost than new development of the measurement tool.

Cobertura supports statement coverage and decision coverage, and Statement coverage for Python supports only the statement coverage. On the other hand, the sample implementation with our framework supports statement coverage, decision coverage, condition coverage and condition/decision coverage. The same functionality can be implemented with fewer LOCs.

Therefore, our framework succeeded in alleviating the problem of high cost for new development using design patterns and we confirm high reusability of design patterns.

VI. Related work

Here, we explain the ideas of Kiri et al.[12] as other researches that relate to the mechanism and purpose of our framework.

Kiri et al. propose the idea of developing a measurement tool which inserts the measurement code into source code. Their idea measures statement coverage, decision coverage and a special coverage called RC0. RC0 is special statement coverage for only the revised statement. However, their idea can measure only statement coverage and decision coverage because their idea measures coverage by simply inserting a simple statement. Moreover, though their idea can measure the coverage of four languages, including Java, C/C++, Visual Basic, and ABAP/4, it cannot support any other languages. Conversely, our framework cannot measure RC0. However, our framework can support new coverage such as RC0 easily by adding user code.

VII. Conclusion and future work

In this paper, we have proposed a coverage measurement framework for multiple languages and report the implementation of our framework based on design patterns. We achieved reduction of cost by reusing common code because we implemented our framework based on design patterns. Thus we conclude design patterns produce high reusability.

We plan to evaluate more completely our framework, achive more reusability using design patterns, and improve the framework in order to support languages other than procedure-oriented languages, such as functional programming languages.

References

[1] Lee Copeland, ”A Practitioner’s Guide to Software Test Design”, Artech House, 2003.

[2] Kazunori Sakamoto, Open Code Coverage Framework, http://sourceforge.jp/projects/codecoverage/.

[3] Cobertura, http://cobertura.sourceforge.net/.

[4] Mohamed Fayad and Douglas C. Schmidt, ”Object-Oriented Application Frameworks”, the Communications of the ACM, Special Issue on Object-Oriented Application Frameworks, Vol. 40, No. 10, October 1997.

[5] Etienne M. Gagnon, Ben Menking, Mariusz Nowostawski, Komivi Kevin Agbakpem and Kis Gergely, SableCC, http://sablecc.org/.

[6] ANTLR, http://www.antlr.org/.

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

[8] OMG, Unified Modeling Language (UML) specification, version 2.2, http://www.omg.org/spec/UML/.

[9] Gareth Rees, Statement coverage for Python, http://garethrees.org/2001/12/04/python-coverage/.

[10] Apache Software Foundation, The Byte Code Engineering Library, http://jakarta.apache.org/bcel/.

[11] Haruhiko Okumura, Houki Satoh, Kazuo Turu, Kazuyuki Shudo and Tutimura Nobuyuki, ”Algorithm cyclopedia by Java”(in Japanese), Gijutuhyoronsya, 2003.

[12] Takashi Kiri, Tatuya Miyoshi, Satoru Kishigami, Tatuo Osato, Tuyoshi Sonehara ”About the source code insertion type coverage tool”, The 69th Information Processing Society of Japan National Convention, 2003.

[13] Microsoft, http://dlr.codeplex.com/.

Architectural and Design Patterns in Multimedia Streaming Software

Yanja Dajsuren
Dept. of SoC Architectures and Infrastructure
NXP Semiconductors
Eindhoven, the Netherlands
yanja.dajsuren@nxp.com

Mark van den Brand
Dept. of Mathematics and Computer Science
Eindhoven University of Technology
Eindhoven, the Netherlands
m.g.j.v.d.brand@tue.nl

***Abstract*—Software developers typically build streaming applications using multimedia streaming frameworks like DirectShow, GStreamer, and Symbian MMF. Although a significant amount of work has been done on architectural and design patterns in software engineering, there is a limited notion of patterns in the development of multimedia streaming software. This article explores architectural and design patterns in the field of multimedia streaming software to facilitate the understanding process of multimedia frameworks and development of streaming applications.**

Keywords-streaming pattern; multimedia framework; streaming application;

I. INTRODUCTION

Since it is costly and time-consuming to build multimedia streaming software from scratch, multimedia frameworks enable streaming applications to be assembled by integrating pluggable media components. In addition, multimedia frameworks isolate applications from a variety of complex tasks such as handling of the complex multimedia acceleration hardware, data transport, and synchronization between various tasks.

Multimedia frameworks are available on different operating systems e.g. Windows supports DirectShow [1], Linux provides GStreamer [2] and Symbian enables Multi-Media Framework (MMF) [3]. Since frameworks are concrete realizations of groups of patterns that enable reuse of code [4] and there is a limited notion of architectural and design patterns in the multimedia streaming software, it is essential to identify patterns used in the multimedia streaming software.

In this article, we present design patterns that are based on the GStreamer, DirectShow, and Symbian MMF multimedia frameworks. We use class diagram and streaming notations of the UML 2.0 [5] to illustrate the patterns.

II. MULTIMEDIA STREAMING SOFTWARE

One of the broadly recognized approaches in the development of the multimedia streaming software is to structure the streaming software as Pipes and Filters. The Pipes and Filters architectural pattern [6] divides a complex functionality into several sequential processing sub-functionalities forming a streaming graph as shown in Fig 1. The nodes of the graph are the media components that process the data. The output of one media component can be used as input for another media component. The edges of the graph are (mostly) data buffers that establish connections between the media components.

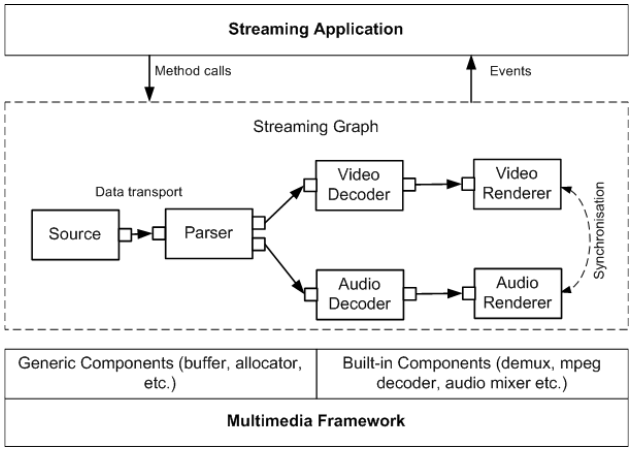


Figure 1. Overview of multimedia streaming software

We summarize below a list of main tasks that are explored from the GStreamer, DirectShow, and Symbian MMF multimedia frameworks:

- **Building a streaming graph.** Every streaming application starts by building a streaming graph mostly by instantiating media components and connecting them using the functions provided by a multimedia framework.
- **Streaming data in the graph.** Primitives for moving media data through the streaming graph are usually provided by the framework designers.
- **Responding to events.** Besides facilitating mechanisms for an interaction between application and streaming graph such as seeking to a position in a media file, there is also an event handling needed between media components such as End of Stream.

Multimedia frameworks enable developers to build custom media components by providing specific APIs or a set of base classes that provide the developer with a default implementation for certain tasks.

III. DESIGN PATTERNS

We present three composite patterns as depicted in the directed acyclic graph of Fig 2.

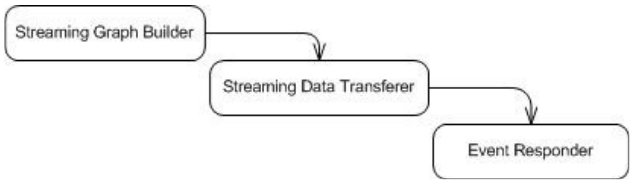


Figure 2. Graph of streaming design patterns

A. *Streaming Graph Builder*

Complex streaming software is time consuming and cumbersome to be developed from scratch or procedural way. The Streaming Graph Builder pattern is similar to the Builder Pattern. Its intention is to abstract steps of construction of a specific streaming graph so that different implementations of these steps can construct different type of streaming graphs (e.g. video capturing graph). The Streaming Graph Builder pattern is depicted in Fig 3.

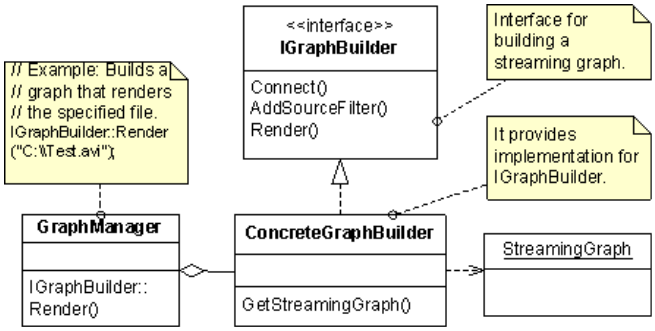


Figure 3. Streaming Graph Builder pattern

As a result, it can cope with many variations in building a streaming graph and enable clients to treat media processing components constituting the streaming graph uniformly.

B. *Streaming Data Transferer Pattern*

The Streaming Data Transferer pattern consists of Transport Data, State Transition, and A/V Sync patterns. The Transport Data pattern is elaborated in Fig 4. It enables moving media data through the streaming graph using common data transfer protocol and mechanisms like Media Data pool.

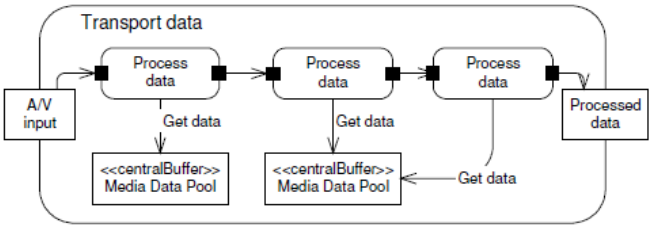


Figure 4. Transport Data pattern

A streaming application, which uses general mechanisms to transfer data through the graph independently of the media format. State changes are handled consistently between application and media component or streaming graph. A/V synchronization is handled overall.

C. *Event Responder Pattern*

Event handling is needed between media components as well as application. The Event Responder pattern consists of Vertical Event Handling and Horizontal Event Handling patterns. The Vertical Event Handling pattern is elaborated in Fig 5.

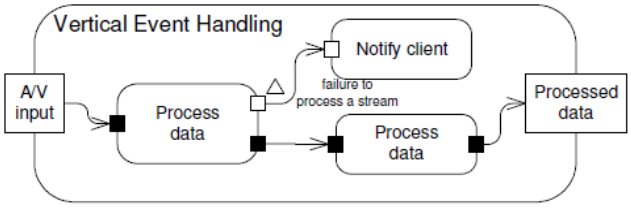


Figure 5. Vertical Event Handling pattern

Besides facilitating mechanisms for an interaction between application and streaming graph such as seeking to a position in a media file, an event handling mechanism between associate media components is provided. This uniform solution improves the quality of streaming software development.

IV. Concluding Remarks

Multimedia frameworks ship with a large collection of media processing components by default, making the fast development of a large variety of streaming applications possible. The key capabilities of existing multimedia frameworks include building of streaming graphs, streaming data through the graph, and responding to events invoked in the graph and the application. However, these tasks are realized differently in the multimedia frameworks.

Therefore, developers need to understand the concepts and mechanisms of multimedia frameworks to build quality-streaming applications cost effectively, particularly when it comes to the development for different platforms. We have presented sample design patterns based on the GStreamer, DirectShow, and Symbian MMF frameworks to facilitate the development of multimedia streaming software.

Future work will be focused on a case study illustrating how we apply design patterns to enhance the understandability and extensibility of multimedia frameworks and evaluate how much the application and framework developer's effort is facilitated by the streaming design patterns.

References

[1] Microsoft, "Microsoft DirectShow 9.0" http://msdn2.microsoft.com/en-us/library/ms783323.aspx

[2] GStreamer open source multimedia framework, http://gstreamer.freedesktop.org/

[3] Symbian, "Multimedia Framework" http://www.symbian.com/

[4] D. C. Schmidt, F. Buschmann, "Patterns, Frameworks, and Middleware: Their Synergistic Relationships", IEEE, 2003.

[5] D. Pilone, N. Pitman, "UML 2.0 in a Nutshell", O'Reilly Media Inc., 2005

[6] F. Buschmann et al., "Pattern-Oriented Software Architecture", John Wiley and Sons, pp. 53-70, 1996

Security Patterns

Building a Concept Grid to Classify Security Patterns

Michael VanHilst, Eduardo B. Fernandez
College of Engineering and Computer Science
Florida Atlantic University
Boca Raton, Florida, USA
mike@cse.fau.edu, ed@cse.fau.edu

Fabrício Braz
Departamento de Ciência da Computação
Universidade de Brasília
Brasília, Brasil
fabraz@unb.br

Abstract— In security, good solutions are comprehensive in their coverage. In this paper we discuss a method of classifying patterns based on coverage of the overall problem space. The method defines regions in a continuous space, and associates patterns with different regions of concern. Similar to faceted classification, the approach aids finding patterns that address immediate problems. But it also brings attention to what is not being addressed. The approach allows for a meaningful comparison of security patterns based on the size and shape of the areas they address. In this paper we discuss the conceptual basis of the approach, using George Kelly's psychological construct theory to divide the conceptual space along clearly defined axes, and describe methods borrowed from operations research to display the classifications in a graphical form.

Keywords-patterns; security; classification

I. Introduction

To build secure systems, security must be addressed for all components in all activities in every phase of the product lifecycle. For complex applications, a comprehensive approach to security – top-to-bottom, beginning-to-end, and everywhere-in-between – is a requirement. The problem is vast. Inspection and testing is only a small part. Building secure systems involves more than plugging holes.

The security challenge is compounded by the way real world systems are built. Consider the issues of component source in software development. Components can come from new code, open-source, runtime script, model transformation, wizard code generation, legacy application, reuse library, outsourced development, commercial-off-the-shelf, and remote web service. It is a rare and inefficient project that doesn't leverage more than a couple of these sources.

The patterns community creates and collects patterns in the belief that patterns help users to solve problems and perform tasks. Specifically, patterns are intended to save users time and improve the quality of the results. By now we have hundreds, if not thousands of patterns, including security patterns, for a wide variety of problems and situations. In choosing which patterns to study and apply, the potential user of patterns must narrow the choice to only those patterns likely to be of value to the task at hand. While finding the one best pattern is an ideal, in this case, it is not the primary goal. The user may wish to see collections of patterns related to the problem, the context, and each other in various ways. By studying alternative and complementary patterns, the user gains knowledge and understanding.

In this paper we present a method to organize and search patterns based on multiple dimensions of classification. Each dimension divides a conceptually continuous space into multiple regions of classification. For each region, patterns are classified by whether or not they play a meaningful role, or have value, in that region. Multiple orthogonal dimensions are combined to form an n-dimensional space. Patterns occupy regions within that space, and can be found by searching and navigating among the regions.

Our method builds on the ideas of Personal Construct Theory, first described by George Kelly in 1955 [1]. In Personal Construct Theory, conceptual dimensions are bi-polar – defined as a continuum between two opposite poles. Each dimension (or construct) embodies an aspect of one's intentions and understanding. Conceptual space is formed by the combination of many such dimensions.

In the following section we describe the problem being addressed and existing approaches to pattern classification. We then describe our method of pattern classification, with examples from our own work. We discuss its strengths and advantages, and some variations that can be applied for even greater utility. We also describe the use of tools, common in Operations Research, that can be used to display and explore this form of pattern classification. Finally, we present a sample walk-through to illustrate how the proposed approach to classification might be used.

II. Prior Work

Several groups have published collections of security patterns to address the problem of secure software development [2][3][4]. They group their patterns into rough categories, typically based on the type of solution being presented. The original GoF patterns book [5], for example, grouped patterns into three categories: creational, structural, and behavioral.

Many of the existing pattern classifications use a hierarchical approach. Hierarchical classification mimics classification in biology, where the hierarchical structure enables identification by following a decision tree (e.g. Does it have a backbone?) and conforms to a view of speciation by mutation from a parent gene. Hafiz, et al. [6], for example, propose a hierarchical categorization for

security patterns with three branches corresponding to Core security, Perimeter security, and External security. Hierarchical classification serves a pattern collector's need to determine the extent to which a new pattern is the same as other patterns in the collection, and are typically based on the solution being presented. But hierarchical classification offers less value to users who know more about the problem than the solution.

Classification for component reuse has a long history. Prieto-Diaz made a strong case for faceted classification over hierarchical classification. In the paper, hierarchical is called enumerative, while facets are essentially property tags [7]. Prieto-Diaz provides seven criteria for the classification scheme:

1. It must accommodate continually expanding collections,
2. It must support finding components that are similar, not just exact matches,
3. It must support finding functionally equivalent components across domains,
4. It must be very precise and have high descriptive power.
5. It must be easy to maintain, that is, add, delete, and update the class structure and vocabulary without need to reclassify.
6. It must be easily usable by both the librarian and end user, and
7. It must be amenable to automation.

Our matrix approach is equivalent to facets and satisfies the first six criteria. In our case, it is not clear how automation would apply. Prieto-Diaz's earlier component work concerned only finding a close match, classified largely by elements of the solution. Our matrix emphasizes properties of the problem and also serves an education and knowledge purpose for navigating the problem space and identifying gaps. In the Prieto-Diaz solution, support for navigation was limited. It said nothing about what was missing.

Sarmah et al. [8], recently presented a model for categorizing security patterns based on a concept lattice. Their approach of mapping patterns is similar to ours. Their use of a lattice structure to support scaling is an interesting aspect that we have not considered. However, their presentation is brief and discusses neither the use of their categorizations, nor the choice of concepts. Both concerns are emphasized in the work here.

Rosado et al. [9], use matrix diagrams to evaluate and compare security patterns using relative measures of appropriateness by criteria. Other related work, such as by Munoz-Arteaga, et al. [10], simply map patterns to individual levels or criteria. The work of Rosado et al., is complementary to our own and focuses more on the means of making comparisons than on the choice of criteria. The work here focuses more on the design and selection of dimensions and categories, and on the use of matrices for pattern coverage, selection, and navigation.

III. Construct Theory

We address pattern classification and problem coverage through the use of a multi-dimensional matrix of concerns. Each dimension of the matrix presents a range of distinctions along a single axis, with a common concept or theme. The categories along each axis should be easily understood and represent widely used and accepted classifications with respect to the concept of that axis. For example, one dimension would be a list of stages along a lifecycle axis spanning the life of an application from initial conception to final end of use. Distinct stages would include domain analysis, requirements, problem analysis, design, implementation, integration, deployment (including configuration), operation, maintenance and disposal. A pattern applies to a lifecycle stage if a developer could use knowledge from the pattern in performing tasks at that stage. The list of component source types, using a dimension from no control of details to full control as described earlier, forms another dimension. Types of security response from intent to attack to attack aftermath could form yet a third dimension, covering avoidance, deterrence, prevention, detection, mitigation, recovery, and investigation (or forensics).

The design of the matrix is motivated by a notion of coverage of concerns. For security, coverage must be comprehensive. Information is not secure if it can be compromised at any point in any way. We express coverage as a grid or matrix of concerns, where comprehensive coverage would mean there is something for every cell in the grid. Thus we are concerned not only with patterns that exist, but also to identify gaps where patterns do not exist. Our approach starts with a complete problem space, and then carves it into different concerns along different dimensions. The idea of dividing up psychological space can be traced back to Euclid's elements. Its use here builds on the ideas of George Kelly [1]. In Kelly's personal construct theory, a construct is a reference axis of two opposing poles. Wealth, for example is an axis of rich and poor. The space between the poles defines a "range of convenience" which gains further relevance with additional planes of distinction. "A construct is a dichotomous reference axis. It defines a family of planes orthogonal to it that divide the space." [11]. In our case, we are not as interested in the planes of distinction, which create the separations, as with spaces between two planes, which provide a convenience of classification. Kelly described a matrix of concepts that embodies a person's intentions and shapes their response. He called it a "role repertory grid." A more formal treatment of the division of psychological space can also be found in Brown's "Laws of Form" [12].

IV. Matrix Classification and Display

We use Kelly's approach to define a matrix of concepts to embody the intention to secure information, and to use that matrix to classify patterns. In defining dimensions or axes for classification, each axis should correspond to a single logical construct. In Kelly's model, a construct is defined by dichotomous poles. We strive to do the same. For each of the primary axes we try to find two poles that define its continuum in problem space. The axis or dimension is then divided into regions of concern. Because we are not concerned with uniqueness, regions along a dimension can be loosely defined and be hierarchical, disjoint, or overlap. Regions, or classifications of concern, should be based on distinctions that are reasonably understood by target users – in our case software developers and security practitioners. Defining the regions is much like defining concerns in top-down decomposition. In logic, a distinction defines both that which is included, and that which is not [12]. In comparison, when classification starts with a known, but unstructured collection of items, and puts them into groups, there is no way to know what is missing.

Operations research proposes 7 management tools for organizing non-quantitative information and ideas [13]. These tools are: relationship diagram, affinity diagram, tree diagram, matrix diagram, prioritization matrices, arrow diagram, and process decision program chart. In organizing collections of patterns, the patterns community already uses three of these tools: relationship diagram, affinity diagram, and tree diagram. Here, we propose to use a fourth tool, the matrix diagram. We find the use of matrix diagrams to be a convenient way to improve the quality and usability of pattern collections for the consumers of patterns.

Figure 1 shows a grid that maps patterns with a single dimension of the problem space. For a single mapping, we use an L-Shaped Matrix. The dimension for type of protection partitions the problem space of an attack into stages along a continuum from its initial conception to the aftermath of its having happened. Each category identifies the type of response appropriate to the corresponding stage of attack. From this matrix it is easy to see which stages of attack are not addressed.

Figure 2 shows a mapping between patterns and three dimensions of the problem space. Because we are not interested in relationships between different dimensions of the problem space, we can present this view as a T-shaped matrix. By extension, the vertical axis can be stacked with additional dimensions of the problem space.

The dimension for lifecycle stage partitions the lifecycle along a continuum from pre-project preparation to the final disposal of all artifacts. The National Security Administration's guidelines for information systems management requires all of these stages to be covered. The narrower lifecycle view common in software development is not sufficient when it comes to security.

Security pattern vs. Type of protection	Check-pointed System	ProtectedSystem	StatefulFirewall	WiMax Security
Avoidance				
Deterrence				
Prevention		X	X	X
Detection			X	
Mitigation	X			
Recovery	X			
Forensics			X	

Figure 1: An L-shaped matrix of relationships between patterns and protection type

Protection Type	**Avoidance**				X
	Deterrence				X
	Prevention		X	X	X
	Detection			X	
	Mitigation	X			X
	Recovery	X			X
	Forensics			X	
Security Pattern		**Check-pointed System**	**Protected System**	**Stateful Firewall**	**Virtual Machine**
Lifecycle Stage	**Domain Analysis**				
	Requirements				
	Analysis	X		X	X
	Design	X	X	X	X
	Implementation	X			X
	Integration				X
	Deployment				X
	Operation				X
	Maintenance				X
	Disposal				X
Architecture Layer	**Network**				
	Transport			X	X
	Operating System	X	X	X	X
	Distribution			X	X
	Data	X	X		X
	Business Logic	X	X		X
	Client/Application	X	X		X

Figure 2: An extended T-shaped matrix with patterns and three dimensions of the problem space.

In today's systems-of-systems view of applications, many development projects address a single system or level within a larger stack of infrastructure and components. Threats, strategies, and mechanisms are different at different levels of this architecture. We define a dimension for the level of

system architecture to address these differences. Because there are several different views of the system stack, depending on the domain and application, this dimension has a number of overlapping partitions. The continuum of this dimension is defined between the lowest physical level of abstraction – the wire, and the highest semantic level of abstraction – the business task. We chose the following classifications: network, transport, distribution (including gateways and brokers), platform and operating system, data, business logic, and client. A simpler notion called application spans the last three. Network, transport, and distribution may also be grouped as communication. Distribution and operating system overlap since gateways and brokers often sit on top of, and depend upon, the operating system. Since patterns can be placed in more than one cell, there is no real need for exact or disjoint classification.

Studies by Leveson of safety failures at NASA [14] show that accidents are caused by failures of constraints at higher levels than just mechanisms and developers. We define another dimension for level of constraint. This dimension defines the continuum from simple device mechanisms to societal levels of regulation and oversight. Following Leveson's work, we partition the axis for level of constraint into mechanism, operator, developer, organizational, and regulatory. In our own work, we create patterns for standards and protocols, like WiMax Security [15], that map to these higher levels of constraint. The Common Criteria standard requires a number of organizational level practices, and are themselves a regulatory level constraint.

V. Discussion

In security, there is a growing realization that we cannot solve the problem by layering on more and more piece-meal solutions. Examples of piecemeal approaches are, patches for each new vulnerability, separate mechanisms for each level and protocol, and ever growing checklists of vulnerabilities to address. So, at least in the field of security, good solutions are solutions that address many issues in a comprehensive way. The matrix provides a convenient way of representing that comprehensiveness. More area coverage on the matrix means more comprehensive.

An example of a solution pattern with good comprehensiveness is the Virtual Machine Operating System Architecture pattern. A single managed image addresses all of the connections at every level above the hardware and network. A replaceable machine image also addresses concerns from analysis through implementation, deployment, operation, maintenance, and ultimately disposal. Its replaceable and isolated environment also addresses many stages of the attack lifecycle

Recently, software engineering has shown a growing interest in Donald Schön's work on reflection-in-action [16]. In looking at a situation, the reflective practitioner considers multiple, alternative views or settings for interpreting the situation and framing problems. Each setting defines things to be considered, problems to be found, and available solutions. Schön's observations are based on a critique that any single point of view might overlook important aspects of the situation and solution opportunities. By encouraging the use of multiple views on different dimensions, our approach is less likely to miss concerns that might be overlooked in a single classification hierarchy or scheme.

Our approach to problem classification specifically supports multiple, alternative settings of the same problem space. Not surprisingly, pattern authors and classifiers are often not comfortable with all the dimensions presented in our scheme. The approach allows settings, goals and perspectives that are attack, implementation, regulatory, organizational, military, network, code, and device centric to coexist in a single classification system. Patterns can be classified separately in every setting to which they can be applied. The practitioner exploring a new situation can identify patterns from one perspective, and then explore related patterns along other, perhaps less familiar, dimensions thereby gaining new understanding and perspective. New settings can be added at any time, simply by defining a new axis.

In an earlier article [17], we discussed other types of dimensions that augment the types of axes described here. These secondary or auxiliary views can be lists or collections and do not necessarily divide a bipolar continuum. Because our mappings are not unique, they can be combined with other mappings without loss of meaning. For example, additional matrix dimensions can be used to provide finer levels of distinction, or to filter selections by other criteria, such as the criteria in a standard, or the application domain. Classifications based on the solution space can also be combined with the problem classifications shown here to select solution components for a particular architecture or strategy. In that paper, we suggested that checklists could be used for judging patterns. The approach here takes a more top-down view mindful of being comprehensive.

Garbe et al. also used Kelly's construct theory to map psychological space for the classification of patterns, though not specifically for security patterns [18]. In their case, the purpose of the work was to find the pattern that most closely matched a set of properties. In a learning phase, psychological constructs are discovered by asking pattern experts to compare different patterns, and recording what they say. Using Kelly's repertory grid technique to extract the terms, and formal concept analysis to cluster the results, they discover both the bipolar dimensions and the classification properties (clusters of similarity) on those dimensions. In the usage mode their system asks a user to

choose one of the properties for each dimension and returns the closest pattern.

Although the paper by Garbe et al. does not discuss the resulting dimensions in their system, we found in our own experience that pattern experts almost always classify patterns with properties of the solution space. That was part of our motivation for mapping the problem space. We took a synthetic approach to construct dimensions that correspond to the types of coverage issues we have encountered in our teaching and research. It would however, be an interesting exercise to ask security experts to classify the problem space using the same repertory grid technique.

Since Garbe et al. asked for properties of existing patterns, it is unlikely that they would discover named properties for which no patterns exist. The repertory grid technique does, however, use a scale for each dimension. Respondents are asked not only to give a distinguishing property, but to assign a value from 1 to 7 for that dimension. Using the scale values, regions could be defined by the range of values included in each property's cluster. Values between 1 and 7, but not in any cluster, could indicate a gap.

VI. Sample Walk-Through

The use of the concept grid is very much like any organizing system with labels or tags. Unlike arbitrary tags, for example as used in Google Mail, our use of axes with bipolar constructs assures that we have partially ordered sets of tags to define regions and progressions along each concept axis. The lack of an imposed hierarchy allows arbitrary combinations.

Imagine an architect developing a system of active defense for a public utility. An active defense assumes an intelligent and engaged adversary cleverly able to overcome passive defenses. For the initial requirements and analysis, we will be interested in both high level analyses and specific deployable mechanisms. Here, the problem space for lifecycle could cover requirements and analysis, but also deployment and operation, since the dynamic aspects of the defense requires solutions that can address the attack in the system's deployment and operation. All layers of architecture could be under attack, so we select the entire range of architecture levels. On the constraint axis, the design of dynamic solutions involves mechanisms and operator behavior. A dynamic defense aims at the middle stages of an attack axis, namely prevention, detection, and mitigation.

The selections described so far cover a large region and return a significant number of patterns. Attack and abuse patterns, as well as high level analysis patterns, are of particular interest for the initial analysis. A look at the patterns in the deployment and operation regions gives some idea of the defenses that can be deployed against an adversary. Since part of the system involves small embedded devices, we might use an application domain axis, described in our earlier paper [17], to separately select for network and server elements or embedded sensor and control elements. Few patterns are specifically linked to the domain of small, low power systems. In response, we may wish to add a new axis, and classify the remaining patterns for their minimum required device capability, ranging from the smallest devices (no state and limited compute cycles), to the biggest devices (essentially unlimited resources).

After reviewing the options, our architect is attracted to the capabilities of the virtual machine pattern, and decides to design a system around this model. Looking at the T-shaped matrix view, as shown in Figure 2, the architect can immediately see that the virtual machine pattern does little to protect the network and does not address the problem of detecting attack. Thus, at a minimum, additional pieces will be needed to fill in these areas. To address detection, again looking at the T-matrix, a number of different agent patterns are considered for coverage and capability.

VII. Conclusion

Our objective in the work described here was to define a method of classification for security patterns that addressed the needs of end users and developers. In particular, we were concerned with showing pattern coverage of problem concerns, and supporting pattern selection and navigation based on applicability to a developer's immediate concerns. We made no assumptions about the developer's familiarity with existing patterns or solutions.

We applied a matrix mapping technique that maps patterns to concerns and matrix diagramming tools, found in quality management and operations research, to visualize the results. Regions of problem concern were defined along multiple independent dimensions. To preserve a complete view of the problem space, we defined each dimension as a continuum between two opposing poles, and created categories by partitioning the space into regions along that continuum. The approach is grounded in a theory of psychology called Construct Theory.

We then showed how the approach could be applied to judging pattern quality based on a concrete representation of scope of coverage or comprehensiveness.

Although space limitations prevent discussion here, early experience indicates that the approach is feasible and offers many other desirable properties. We hope that the ideas presented here will stimulate more interest and further work in the classification and evaluation of patterns from the user's point of view.

Experienced developers with security expertise may prefer traditional classifications based on solution type or elements. Solution based classifications can be used in conjunction with the types of dimensions proposed here – even as additional dimensions of the same matrix. But even

for experts, our approach can offer value for exploring coverage and discovering gaps.

References

[1] G.A. Kelly. The Psychology of Personal Constructs. Norton, 1955.

[2] B. Blakley, C. Heath, and members of the Open Group Security Forum. Technical guide: security design patterns. The Open Group, UK, 2004.

[3] M. Schumacher, E.B. Fernandez, D. Hybertson, F. Buschmann, and P. Sommerlad. Security patterns: integrating security and systems engineering. Wiley, 2006.

[4] C. Steel, R. Nagappan, and R. Lai. Core security patterns: best practices and strategies for J2EE, web services and identity management. Prentice Hall, 2005.

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

[6] M. Hafiz, P. Adamczyk, and R.E. Johnson. "Organizing security patterns." IEEE Software, 24(4), 2007, doi: 10.1109/MS.2007.114

[7] R. Prieto-Diaz. "Implementing faceted classification for software reuse." Comm. of the ACM, (34)5, pp. 88-97, 1991. doi: 10.1145/103167.103176.

[8] A. Sarmah, S.M. Hazarika and S.K. Sinha. "Security pattern lattice: a formal model to organize security patterns." Proc. Int. Conf. on Database and Expert Systems Application, pp. 292-296, 2008. doi:10.1109/DEXA.2008.74.

[9] D.G. Rosado, C. Gutiérrez, E. Fernández-Medina and M. Piattini. "A study of security architectural patterns." Proc. Int. Conf. on Availability, Reliability, and Security, pp. 358-365, 2006. doi: 10.1109/ARES.2006.18

[10] J. Muñoz-Arteaga, R. Mendoza González, and J. Vanderdonckt. "A classification of security feedback design patterns for interactive web applications." Proc. Int. Conf. on Internet Monitoring and Protection, pp. 166-171, 2008, doi:10.1016/j.advengsoft.2009.01.024

[11] M.L.G. Shaw and B.R. Gaines. "Kelly's "Geometry of psychological space" and its significance for cognitive modeling." The New Psychologist, pp. 23-31, October 1992.

[12] G.S. Brown. Laws of Form. George Allen and Unwin, 1971.

[13] S. Mizuno. Management for Quality Improvement: The Seven New QC Tools. Productivity Press. 1988

[14] N. Leveson. "A new accident model for engineering safer systems." Safety Science, 42(4):237-270, 2004, doi:10.1016/S0925-7535(03)00047-X

[15] E.B. Fernandez, M. VanHilst and J.C. Pelaez. "Patterns for WiMax security." Proc. European Conference on Pattern Languages of Programming, 2007.

[16] D.A. Schön, The Reflective Practitioner: How Professionals Think in Action, Basic Books, 1983.

[17] M. VanHilst, E.B. Fernández and F.A. Braz. "A multi-dimensional classification for users of security patterns." J. of Research and Practice in Information Technology, 41(2), May 2009, pp. 87-97.

[18] H. Garbe, C. Janssen, C. MÖbus, H. Seebold and H. De Vries. "KARaCAs: Knowledge acquisition with repertory grids and formal concept analysis for dialog system construction." Proc. 15th International Conference on Knowledge Engineering and Knowledge Management (EKAW), pp 3-18, 2006, doi:10.1007/11891451.

Validating and Implementing Security Patterns for Database Applications

Arnon Sturm, Jenny Abramov, Peretz Shoval
Department of Information Systems Engineering
and Deutsche Telekom Laboratories
Ben-Gurion University of the Negev Beer Sheva, Israel
sturm@bgu.ac.il, jennyab@bgu.ac.il, shoval@bgu.ac.il,

Abstract. *Security in general and database protection from unauthorized access in particular, are crucial to organizations. Security and authorization patterns encapsulate accumulated knowledge and best practices in this area. Correct application of security and authorization patterns will ensure effective access control to the database. For example, the Role-Based Access Control (RBAC) security pattern describes a general solution regarding who is authorized to access specific resources and which access privileges they have, based on user roles. Unfortunately, patterns alone do not provide concrete guidance for their application, and thus there is a need for validating their correct usage. We propose a methodical approach for implementing security patterns for access control in database applications. This approach provides implementation guidelines to the designer of the application model, validation of the correct usage of the patterns, and automatic generation of secure database schemata.*

Keywords: security patterns; domain engineering; database access control; ADOM; UML

I. INTRODUCTION

The most valuable asset for an organization is data, as its survival depends on the correct management, security, and confidentiality of the data [7], [8]. Most organizational data are stored and managed using database management systems; consequently, protecting the data that are stored in those databases against unauthorized access is crucial for organizations.

As security is just one of the many non-functional requirements that developers have to handle during software development, they might not have a solid security background. This is a huge problem since there are many security concerns to handle. To overcome the knowledge gaps among developers in different domains, the notion of *design patterns* was introduced. Patterns enable to capture expert knowledge and make it more generally available. The origins of *design patterns* lie in a work done by the architect Christopher Alexander during the late 1970s. Alexander noted that "each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" [2]. Since then, the idea was adopted in the field of software design. The famous work of design patterns of Gamma, Helm, Johnson, and Vlissides [5] stated that a pattern addresses a recurring design problem that occurs in a specific context, and presents a well-proven solution to it. In this way, patterns help to promote good design practices.

To assist developers to handle security concerns, security patterns were proposed. These patterns capture extensive accumulated knowledge regarding security. In the past decade, many security patterns have been described; yet, in this paper we focus on authorization patterns such as the Role-Based Access Control (RBAC) [3]. Authorization patterns describe who is authorized to access specific resources in a system whose access has to be controlled.

Security patterns provide guidelines to be used in the early stages of the development lifecycle. Yet, to the best of our knowledge no work has been done on automatic validation of the correct application of these patterns. In addition, existing patterns do not provide concrete guidance for their application.

To address these limitations, we adopt a domain engineering approach called Application-Based Domain Modeling (ADOM), which enables specifying and modeling domain artifacts that capture the common knowledge and the allowed variability in specific areas, guiding the development of particular applications in the domain, and validating the correctness and completeness of applications with respect to their relevant domains [13], [14]. Regarding security patterns, the patterns are specified within a domain model, while the application model elements are classified by the domain model's elements (i.e., the pattern elements). In addition to the specification of patterns, we attach to patterns transformation rules that elaborate on the implementation of these patterns. The proposed approach enables the automatic validation of application models with respect to the relevant security patterns and the automatic generation of SQL scripts including the database scheme and the security constraints (in particular, authorization constraints) to be injected into the database.

The rest of this paper is structured as follows. Related work is presented in Section II. Section III sets the background for the presented approach: first, it provides an overview of the ADOM approach; then it presents the SQL privilege mechanism. Section IV describes the proposed approach. Finally, Section V concludes the paper, discusses the benefits and limitation of the proposed approach, and set the basis for future research direction.

II. Related Work

Since it has been recognized that security must be treated from early stages of the software development life

cycle, it is the task of the designer to ensure that all required security requirements are included in the specifications and that adequate protection mechanisms are implemented to refer those specifications. In the following sections we will review several approaches which refer to this demand.

A. Specification Techniques

Several specification techniques for representing different security policies in a model-driven software development process have been proposed. SecureUML [20] is a modeling language based on RBAC, used to formalize access control requirements and integrate them into application models. It is basically a RBAC language with authorization constraints that are expressed in Object Constraint Language (OCL).

UMLSec [17] is an UML extension that enables specifying security concerns in the functional model. It uses standard UML extension mechanisms; stereotypes with tagged values are used to formulate the security requirements, and the constraints are used to check whether the security requirements hold in the presence of particular types of attacks.

B. Access Control Patterns

An alternative to refer security policies is by using *security patterns*. Security patterns accumulate extensive security knowledge and provide guidelines for secure system development and evaluation.

Access control is one of the core issues in systems and database security. In an environment with resources whose access has to be controlled, authorization patterns can be used to describe, for each entity, the resources it may have access to, and which access privileges it has. Figure 1 describes the authorization pattern as defined in [19]. The *Authorization_rule* association, together with the *Right* association class, defines the access privileges of the *Subject* to the related *ProtectionObject*. The *Right* association class includes the type of access allowed (e.g. read, write, execute), a predicate representing a condition that must be true for the authorization to hold, and a copy flag signifying a condition that indicates whether the right can be transferred or not. An operation *checkRights* can be used in the *Subject* or *Object* to check the validity of a request.

The Role-Based Access Control (RBAC) pattern [19] is a specialization of the authorization pattern that has become the most commonly used for access control since it reduces the cost of administering access control policies and the amount of errors in the process. RBAC is derived from the notion that in organizations, users have different roles that require different skills and responsibilities, and therefore they should have different rights of access to data, which are based on their role. Consequently, the RBAC mechanism [3] describes for each user which privileges they can acquire based on their roles or their assigned tasks. To support the RBAC mechanism at the analysis and design stages of the development lifecycle, a corresponding pattern was developed [19]. The RBAC pattern is shown in Figure 2. *User*s are assigned to *Role*s, while *Role*s are given *Right*s that are permitted to *User*s in that *Role*. As in the authorization pattern, the association class *Right* defines the access types that a user within a *Role* is authorized to apply on the *ProtectionObject.* Correct implementation of the RBAC pattern will ensure effective and secure access control to the database.

C. Secure Software Development with Security Patterns

Security patterns alone are not sufficient for supporting the development lifecycle, since they do not provide systematic guidelines regarding to their application throughout the entire software lifecycle. In order to provide such information to the designers, several methodologies for developing secure software were proposed in the literature. Fernandez et al. [6] proposed a methodology for integrating security patterns into each one of the software development stages. Other methodologies present the use of the *aspect-oriented software design* approach to model security patterns as aspects and weave them into the functional model [9][12], or the use of *agent oriented security pattern language* together with the Tropos methodology to develop secure information systems [10][11].

D. Patterns Validation

Although some of the methods mentioned above provide tools for checking some aspects of the model, they do not have the ability to validate the correct application of the patterns, which will ensure generation of a secure application or a database scheme. Without systematic validation of the involved patterns, we risk in having design problems that will propagate throughout the development process.

To the best of our knowledge, the only work in this area is of Peng, Dong, and Zhao [21], which presents a formal verification method to analyze the behavioral correctness of a design pattern implementation. Their method exploits the partial order relationship between the sequence diagram of a general design pattern and that of its implementation. However, this method does not verify the structural correctness of the implementation. Therefore, there is a need to develop an approach to automatically and fully validate the implementation of patterns.

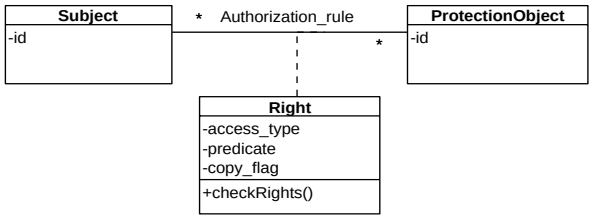


Figure 1. The general Authorization pattern (adopted from [19]).

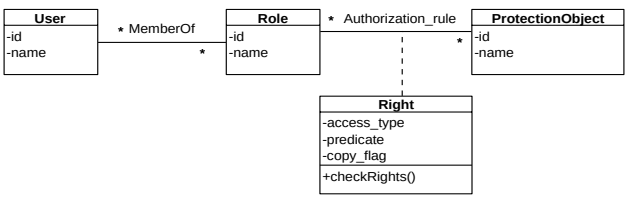


Figure 2. The basic RBAC pattern (adopted from [19]).

E. Secure Database Design Methods

There are several works related to implementation of specifications within a database, such as the work done by Fernández-Medina and Piattini [4] which propose a methodology to design multilevel databases[1] by introducing access control concerns into each one of the software development stages. The methodology allows to create conceptual and logical models of multilevel databases, and implements the models by using Oracle Label Security [16]. The resultant database access control imposes that access of a user to a particular row is allowed only if that user is authorized to do so by the DBMS, has the necessary privileges, and the label of the user dominates the label of the row. Following that methodology, the authors provide a way of transforming specification artifacts into implementation; however, they do not provide tools for validating the specification.

An additional model-driven approach for the development of secure databases was presented by Vela, Medina, Marcos and Piattini [1], which focuses on authorization and audit properties in XML databases. However, it does not apply any analysis techniques on the design of the system.

F. Limitations of Existing Methods

Since information security is crucial to many organizations, and since software project are big and complex, there is a need to assure that the security policies of database design within organization are not neglected during the development process. However, none of the methods provide means for enforcing that a database design complies with particular organizational security specifications. Some of the methods provide means for checking models; however, they do not have the ability to validate the correct application of the security policies. The proposed approach in this paper deals with both enforcing and validating the database design with the use of security patterns: the patterns that are specified in an upper model layer enforce the designers to apply them in the application model, and enable the validation of the correct usage of the defined patterns.

III. Background for the Proposed Approach

To set the background for the proposed approach, in this section we elaborate on the ADOM approach and the fundamental SQL mechanism for enforcing security over a database.

A. The ADOM Approach

The Application-based Domain Modeling (ADOM) [13], [14] is rooted in the domain engineering discipline, which is concerned with building reusable assets on the one hand, and representing and managing knowledge in specific domains on the other hand. ADOM supports the representation of reference (domain) models, construction of enterprise-specific models, and validation of the enterprise-specific models against the relevant reference models.

The architecture of ADOM is based on three layers:

(1) The ***language layer*** comprises metamodels and specifications of the modeling languages. In this paper we use UML 2.0 class diagrams as the modeling language.

(2) The ***domain layer*** holds the building elements of the domain and the relations among them. It consists of specifications of various domains; these specifications capture the knowledge gained in specific domains in the form of concepts, features, and constraints that express the commonality and the variability allowed among applications in the domain. The structure and the behavior of the domain layer are modeled using the modeling language defined in the language layer. In this paper we introduce the structure of each pattern in a domain model.

(3) The ***application layer*** consists of domain-specific applications, including their structure and behavior. The application layer is modeled using the knowledge and constraints presented in the domain layer and the modeling constructs specified in the language layer. An application model uses a domain model as a validation template. All the static and dynamic constraints enforced by the domain model should be applied in any application model of that domain. In order to achieve this goal, any element in the application model is classified according to the elements declared in the domain model using UML built-in stereotype. In this paper the application model elements are classified by the patterns (domain) model elements.

For describing variability and commonality, ADOM uses multiplicity stereotypes that can be associated to all UML elements, including classes, attributes, methods, associations and more. The multiplicity stereotypes in the domain model aim to define how many times a model element of this type may appear in an application model. This stereotype has two associated tagged values - min and max - which define the lowest and the upper most multiplicity boundaries. For clarity purposes, four commonly used multiplicity groups were defined: <<optional many>> (0:n), <<optional single>> (0:1), <<mandatory many>> (1:n), and <<mandatory single>> (1:1).

The relations between a generic (domain) element and its specific (application) counterparts are maintained by the UML stereotypes mechanism: each one of the elements that appears in the domain model can serve as a stereotype of an application element of the same type (e.g., a class that appears in a domain model may serve as a classifier of classes in an application model). The application elements are required to fulfill the structural and behavioral constraints introduced by their classifiers in the domain model. Some optional generic elements may be omitted and not be included in the application model, while some new specific elements may be inserted in the specific application model; these are termed application-specific elements and are not stereotyped in the application model.

[1] A multilevel database permits the classification of information according to its confidentiality, and considers mandatory access control.

ADOM also provides validation mechanism that prevents application developers from violating domain constraints while (re)using the domain artifacts in the context of a particular application. This mechanism also handles application-specific elements that can be added in various places in the application model in order to fulfill particular application requirements.

B. Granting Privileges Using SQL

While using SQL, users can access or manipulate data they do not own. To cope with this capability and enforce data security, SQL provides a mechanism of privilege access. This is done by specifying a set of access rules which define the required privileges. Syntactically, the access rules are defined using the GRANT statement; a short version of it is as follows: `GRANT [privileges] ON [table-name] TO [authorization-names].` *Privileges* can be one of the following: SELECT UPDATE, INSERT, and DELETE. *Authorization-names* refer to a list of users or roles. Naturally, creating roles and groups are also part of the security mechanism provided by SQL for managing databases.

IV. THE PROPOSED APPROACH

In this paper we propose an approach for validating the usage of security patterns and utilizing the knowledge encapsulated in these patterns for generating secure database schemata. For this purpose, we adopt the ADOM approach in which the security patterns are defined within the domain layer, along with transformation rules of how to inject the specification into a database scheme. The patterns will be enforced in the application model. The following stages are part of the sought approach:

1. A domain model should be developed by a security expert and a domain engineer. That model consists of the security patterns specification, as well as rules for their transformation into a database scheme. In this paper, we do not refer to this stage, and assume that the domain model containing the security patterns is correct. Yet, we refer to the stage outcomes.
2. An application model (in this paper, we refer to a class diagram based model) is specified by a developer.
3. The application model is classified according to the domain model (i.e., the security patterns) by a developer.
4. The classified application model is validated automatically against the domain model for the correct usage of the patterns and its fulfillment with respect to these patterns.
5. Having a valid classified application model, the model can be translated automatically into a database scheme.

In the following, we describe the domain model and the way according to which the security patterns are defined. Then, we discuss the procedure of applying the security patterns in a specific application, followed by an explanation of the validation algorithm of ADOM and its application in the context of the proposed approach. Finally, the transformation of the application specification into a database scheme is described and demonstrated.

A. The Domain Model

In Figure 3, the RBAC security pattern is specified using the terminology of ADOM. The *Role* is akin to external entity/user playing a specific function that needs an access to the database. In that case, it is required that in any application implementing or using the RBAC pattern, at least one role should be defined. The *ProtectionObject* is akin to a table in the database. The *Rights* association class determines the privileges of a *Role* with respect to a specific *ProtectionObject*. A class of that type within an application must include at least one privilege.

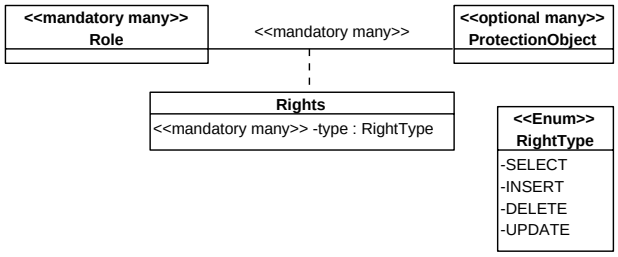


Figure 3. The RBAC security pattern residing within the domain layer.

In addition to the pattern specification, the pattern also refers to a transformation rule stating the way according to which a specification of an application should be transformed into a database scheme. Utilizing the access privileges mechanism as described in Section III.B, we propose the following transformation rules (that are related to privileges) to be applied on application models[2]:

1. `CREATE ROLE [role-class-name];`
   This rule means that for every class in the application model that is classified as role, this statement should be created.
2. `GRANT [rights.type*] ON [protectionObject-class-name] TO [role-class-name];`
   This rule means that a statement of that type will be produced for every association class in the application model classified as *Rights*.

B. The Application Model

For the specification of an application, we use a simple application of students and their course grades, denoted as GRADA. In that application, the approved users (i.e., roles) are a secretary and a student. Figure 4 presents the class diagram of this application along with the security specification as determined by the RBAC patterns described in Figure 3. The various elements that are relevant for the RBAC security pattern are classified (by stereotypes) with the pattern elements: *Role*, *ProtectionObject*, and *Rights*. Following the specification in the class diagram of the application, an *External-Student* has a SELECT privilege to the *Course*, *Grade*, and *Student* classes. In addition, an *External-Student* has an UPDATE privilege to *Student* class.

[2] Additional rules for creating database object such as tables should be defined as well.

The *Secretary* has SELECT and UPDATE privileges to the *Course* and *Grade* classes, and in addition, an INSERT and DELETE privileges to the *Grade* class.

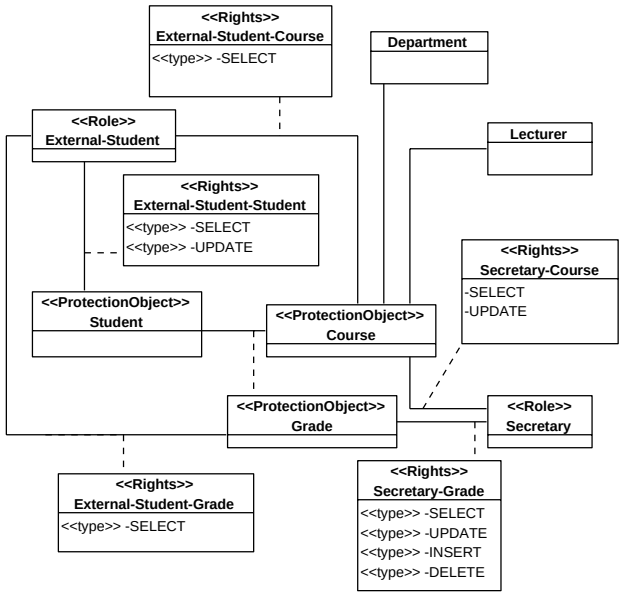


Figure 4. The classified application model of the GRADA system residing within the application layer.

C. Validating the Application Model

As noted before, the various elements that are relevant for the RBAC security pattern are classified (by stereotypes) with the pattern elements that are described in the domain model of the pattern. This enables the validation of the application specification with respect to the RBAC security pattern.

The validation of an application against its domain model is performed in three steps: element reduction, element unification, and model matching. In the ***element reduction*** step, classes that are not stereotyped by elements of the domain model are neglected. In the case of the GRADA application the classes of Department and Lecturer are ignored. During the ***element unification*** step, classes having the same domain stereotyped are unified, leaving only one class in the resultant model. The multiplicity of that class denotes the number of distinct classes in the application model having the same stereotype. In the example of GRADA application, the resultant model consists of three classes: *Role* with multiplicity of 2, *ProtectionObject* with multiplicity of 3, and *Rights* with multiplicity of 5. In the ***model matching*** step, the resultant model of the previous step is matched against the domain model. In the case of the GRADA application the model adheres with the domain model (i.e., the RBAC pattern). In case there were classes classified as *ProtectionObject* with no association classes *Right*s to classes classified as *Role*, it would be a violation of the domain model.

D. Implementing the Application Model

Another aspect of using the patterns is the creation of SQL scripts that define the access privileges to the database of the application. In the GRADA example, the script shown in Figure 5 will be generated following the rule specified in Section IV.A. Note that in this work we assume that the class diagram of the application is automatically transformed to a relation database scheme, as discuss by [18]. In the example of GRADA application, the tables *Student*, *Course*, *Grade*, *Department*, and *Lecture* already exist, along with other tables that reflect the associations among the classes.

```
CREATE ROLE External-Student;
CREATE ROLE Secretary;
GRANT SELECT, UPDATE ON Student TO
External-Student;
GRANT SELECT ON Course TO External-
Student;
GRANT SELECT ON Grade TO External-
Student;
GRANT SELECT, UPDATE ON Course TO
Secretary;
GRANT SELECT, UPDATE, INSERT, DELETE
ON Grade TO Secretary;
```

Figure 5. The SQL script for enforcing authorization.

V. SUMMARY

In this paper we proposed a novel approach that utilizes security patterns for enforcing security over database application design and for injecting security constraints to the database. The advantages of the proposed approach stems from the two layering approach which enable the enforcement of the security patterns.

The limitations of the proposed approach lie in lack of expressiveness of security constraints for low level elements such as attributes. A possible solution for this limitation can be the usage of the extension mechanisms of UML, similarly to [4]. However, this requires a thorough examination.

Future research directions include the enforcement of more complex security patterns on an application design, and the implementation of the approach on more powerful database security mechanisms such as Virtual Private Database [15] and OLS [16] of Oracle.

REFERENCES

[1] B. Vela, E. F. Medina, E. Marcos, and M. Piattini, "Model Driven Development of Secure XML Databases," ACM SIGMOD Record, vol. 35, 3, Sept. 2006, pp. 22-27.

[2] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel, "A Pattern Language: Towns, Buildings, Construction," Oxford University Press, 1977.

[3] D. F. Ferraiolo, and D. R. Kuhn, "Role Based Access Control," 15th National Computer Security Conference, Baltimore, Maryland, Oct. 1992, pp. 554-563.

[4] E. Fernández-Medina and M. Piattini, “Designing Secure Databases,” Information and Software Technology, vol. 47 (7), 2005 , pp. 463-477.
[5] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, “Design Patterns: Elements of Reusable Object-Oriented Software,” Addison-Wesley Professional, 1994.
[6] E.B. Fernandez, M.M. Larrondo-Petrie, T. Sorgente, and M. VanHilst, “A Methodology to Develop Secure Systems Using Patterns,” Integrating Security and Software Engineering: Advances and Future Vision, H. Mouratidis and P. Giorgini Eds., IDEA Press, Ch. 5, 2006, pp. 107-126.
[7] G. Dhillon, “Information Security Management: Global challenges in the New Millennium,” Idea Group Publishing, 2001.
[8] G. Dhillon, and J. Backhouse, “Information System Security Management in the New Millennium,” Communications of the ACM, vol. 43 (7), 2000, pp. 125–128.
[9] G. Georg, I. Ray, and R. France, “Using Aspects to Design a Secure System,” Proc. of the Eighth IEEE International Conference on Engineering of Complex Computer Systems, (ICECCS 2002), ACM Press, Greenbelt, MD, Dec. 2002, pp. 117- 126.
[10] H. Mouratidis, and P. Giorgini, “Secure Tropos: A Security-Oriented Extension of the Tropos methodology,” International Journal of Software Engineering and Knowledge Engineering, vol. 27, no.2, 2007, pp. 285-309.
[11] H. Mouratidis, P. Giorgini, and G. Manson, “When Security Meets Software Engineering: A Case of Modeling Secure Information Systems,” Information Systems, vol.30, no.8, 2005, pp. 609-629.
[12] I. Ray, R. B. France, N. Li, and G. Georg, “An Aspect-Based Approach to Modeling Access Control Concerns,” Journal of Information and Software Technology, vol. 46, no. 9, 2004, pp. 575-587.
[13] I. Reinhartz-Berger, and A. Sturm, “Enhancing UML Models: A Domain Analysis Approach,” Journal of Database Management (JDM), vol. 19 (1), 2007, pp. 74-94.
[14] I. Reinhartz-Berger, and A. Sturm, “Utilizing Domain Models for Application Design and Validation,” Information & Software Technology, vol. 51 (8), 2009, pp. 1275-1289.
[15] J. Czuprynski, “Oracle 10g Security,” Part 2 - Virtual Private Database, The Database Journal, http://www.databasejournal.com/ ,2006.
[16] J. Czuprynski, “Oracle Label Security,” The Database Journal, http://www.databasejournal.com/, 2003.
[17] J. Jürjens, “Secure Systems Development with UML”, 2004, Springer.
[18] M. Blaha, W. Premerlani, and H. Shen, “Converting OO Models into RDBMS Schema,” IEEE Software vol. 11 , May 1994, pp. 28-39.
[19] M. Schumacher, E. B. Fernandez, D. Hybertson, F. Buschmann, and P. Sommerlad, “Security Patterns: Integrating Security and Systems Engineering,” John Wiley & Sons, 2006.
[20] T. Lodderstedt, D. A. Basin, and J. Doser, ”SecureUML: A UML-Based Modeling Language for Model-Driven Security,” Proc. of the 5th international Conference on the Unified Modeling Language, Lecture Notes In Computer Science, vol. 2460, J. Jézéquel, H. Hußmann, and S. Cook, Eds. Springer-Verlag, London, Oct. 2002, pp. 426-441.
[21] T. Peng, J. Dong, and Y. Zhao, “Verifying Behavioral Correctness of Design Pattern Implementation,” Proceedings of the 20th International Conference on Software Engineering and Knowledge Engineering (SEKE), CA, USA, July 2008, pp. 454-459.

Security patterns and quality

Eduardo B. Fernandez[1], Nobukazu Yoshioka[2], and Hironori Washizaki[3]

1 Dept. of Comp. Science and Eng., Florida Atlantic University, Boca Raton, FL, USA, ed@cse.fau.edu
2 National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, Japan
nobukazu@nii.ac.jp
3 Waseda University / GRACE Center, National Institute of Informatics, 3-4-1, Okubo, Shinjuku-ku,
Tokyo, Japan, washizaki@waseda.jp

Abstract. **Security patterns are increasingly being used to build secure systems. An important question is: How can we show that a system built in this way is secure in some sense? We discuss this question in this paper.**

I. INTRODUCTION

Patterns are normally evaluated by submitting them to some pattern conference, e.g. Pattern Languages of Programs (PLoP) or EuroPLoP. In these conferences, a pattern paper is developed with the help of a shepherd and then discussed in a workshop. The pattern is then published and exposed for criticism. Of course, the ultimate evaluation comes when developers use them in their designs. Formal modeling of patterns and combinations of patterns can prove some of the properties of the solution. A pattern that has gone through all these steps is believed to have some good level of quality, in the sense of being correct, reusable, understandable, and easy to tailor to specific requirements. This conclusion applies to security patterns as well.

Assuming that we have ways to show the quality of individual patterns, it is more meaningful to ask: what degree of security can a system reach by the use of patterns in its construction? Or similarly, how secure it is, according to some definition od security? Security is a quality property of a system architecture [1] and we need ways to evaluate the effect of patterns on improving this quality. Security is a quality for which there are no numerical measures. It can only be defined in a relative way with respect to another system or by showing that a system satisfies some requirements. In particular, we are developing a methodology to build secure systems [3,4], based on adding security patterns along the life cycle and in all the architectural layers of the system. How can we show that a system built in this way is secure? We discuss this question in this paper.

II. EVALUATING SECURITY

For our analysis we consider the effect of security patterns and misuse patterns. Security patterns can stop or mitigate specific threats and their consequent misuses. This means that each pattern added to the system could contribute to the total security of the system. Misuse patterns describe, from the point of view of the attacker, how a type of attack is performed (what units it uses and how), and analyzes the ways of stopping the attack by enumerating possible security patterns that can be applied for this purpose [5]. Threats are attacker goals and they can become misuses, described by misuse patterns. There may be more than one misuse pattern to realize a threat. Misuses include internal (insider) and external attacks (hackers). Misuse patterns describe how to realize the threats in T, the set of possible threats in a system; for example, a specific misuse could be an illegal access to a specific file which could be used to read a file with credit card information (the goal of the attacker). Threats can be enumerated systematically [2] and it is possible to build catalogs of misuse patterns [6].

If we consider all the threats to two specific systems, we can see how they handle their respective threats. If we have two versions of a system, following the same requirements, R, one built using security patterns, S_{1a}, and another without them, S_{1b} (Figure 1), we can compare them by enumerating the set of threats of the system, T, and seeing how the two systems can stop these threats. We can see how they handle the known security threats by analysis or by tests on the actual code.

To make the comparison more precise we can consider misuse patterns, M, which can be applied to see the effect of generic attacks on the system and see how the two systems can handle them. A misuse may involve low-level threats that would not show in an analysis of

application threats. In Figure 1, T is a set of threats, specific to a system, while M is a general set of typical misuses that applies to any system, although they must be tailored to the specific context of the misuse.

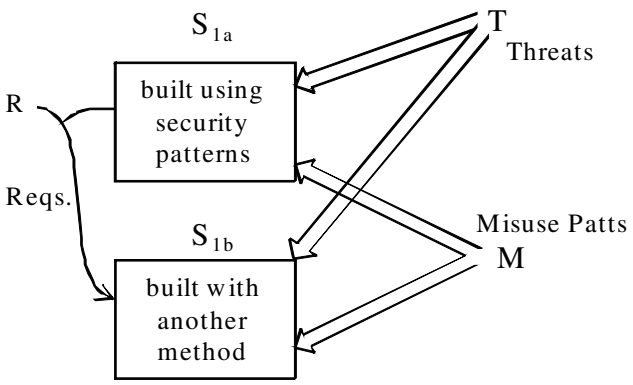


Figure 1. Comparing secure systems

Every application-level threat must be controlled. The same is not true for low-level threats because some of them may not lead to misuses or may lead to misuses we consider of small risk. A Denial of Service to access accounts in a financial institution is performed through a Denial of Service to the server which holds the accounts. A more complex example would be an illegal transfer of money from a customer account to a hacker account, which requires modifying some files that contain the relations with the account data. A policy that can stop this attack would be Need to know and some patterns to realize it in the specific platform: File Authorization, OS Authentication, and DBMS Authorization. In the same way, we can show how to stop each threat in T. If the system being considered contains the necessary patterns, we can say it is secure with respect to those threats.

III CONCLUSIONS

Certification is frequently performed by showing that a given process was followed and verifying that all steps have been performed. While this is clearly not sufficient for security, it adds up to the feeling of security. Following a systematic methodology such as ours, can enhance the confidence that the system is secure. At the end of each stage of the life cycle we can show that all threats have been handled. We can show this also for the complete system. For a system built using another methodology not using patterns we would need to analyze how that system can stop the threats, Applying the comparison to several representative cases we can get to some conclusion about the relative security obtained using patterns and not using them. We intend to perform some experiments comparing our methodology to others. We do not think that it is possible to formally prove security properties in a complex application; in addition the additional effect of the platform would make such an analysis impractical.

REFERENCES

[1] L.Bass, P. Clements, and R.Kazman, *Software architecture in practice* (2nd Ed), Addison-Wesley 2003.

[2] F. Braz, E.B.Fernandez, and M. VanHilst, "Eliciting security requirements through misuse activities" *Procs. of the 2nd Int. Workshop on Secure Systems Methodologies using Patterns (SPattern '07)*. Turin, Italy, September 1-5, 2008. 328-333.

[3] E. B. Fernandez, M.M. Larrondo-Petrie, T. Sorgente, and M. VanHilst, "A methodology to develop secure systems using patterns", Chapter 5 in "*Integrating security and software engineering: Advances and future vision",* H. Mouratidis and P. Giorgini (Eds.), IDEA Press, 2006, 107-126.

[4] E.B.Fernandez, N. Yoshioka, H. Washizaki, and J. Jurjens, "Using security patterns to build secure systems", position paper in the *1st Int. Workshop on Software Patterns and Quality (SPAQu'07),* Nagoya, Japan, December 3, 2007, collocated with the 14th Asia-Pacific Software Engineering Conference (APSEC),

[5] E.B. Fernandez, N. Yoshioka and H. Washizaki, "Modeling misuse patterns", *Procs. of the 4th Int. Workshop on Dependability Aspects of Data Warehousing and Mining Applications* (DAWAM 2009), in conjunction with the 4th Int.Conf. on Availability, Reliability, and Security (ARES 2009). March 16-19, 2009, Fukuoka, Japan.

[6] J. Pelaez, E.B.Fernandez, and M.M. Larrondo-Petrie, "Misuse patterns in VoIP", *Security and Communication Networks Journal.* Wiley.Published online**:** 15 Apr 2009

Extending a Secure Software Methodology with Usability Aspects

Eduardo B. Fernandez[1] and Jaime Muñoz-Arteaga[2],
[1]*Dept. of Comp. Science and Eng., Florida Atlantic University, Boca Raton, FL, USA,* ed@cse.fau.edu
[2] *Universidad Autónoma de Aguascalientes, México,* jmunozar@correo.uaa.mx

***Abstract*— A variety of security patterns have been presented, and there are several books about them. However, patterns are not very useful without a systematic way to apply them. For this purpose, we have developed a methodology to build secure systems. We have not until now considered usability aspects of the security mechanisms needed in those systems. We have also developed patterns for making security more usable. We are starting to add to our methodology some of the results of this work and to develop new aspects of their combination. We present here some preliminary ideas on how to combine these two approaches.**

***Index Terms*—interactive and design patterns, HCI-S, usability**

I. INTRODUCTION

Security patterns specify best practices to design and develop secure software. A variety of security patterns have been presented, and there are several books about them. However, patterns are not very useful without a systematic way to apply them. For this purpose, we have developed a methodology to build secure systems [2].

Most systems are interactive and the interaction usually occurs through a graphical user interface. The security of human computer interaction (HCI-S) considers how the security features of the user interface can be as friendly and intuitive as possible, to let users understand the available security features, thus avoiding errors in their use. A group of HCI-S patterns have been designed for this effect [4, 5]. In particular, privacy is a growing concern. In places where we need the users to provide personal information we should guarantee to them that this information is being sent to the right place and will not be misused. We have written some interface patterns for this purpose [3].

We intend to add to our secure development methodology some of the results of the HCI-S work and to develop new aspects of their combination. We present here some preliminary ideas on how to combine these two approaches. Section 2 summarizes the secure methodology while Section 3 considers some ways where these approaches may be synergistically combined.

II. A METHODOLOGY FOR SECURE SYSTEMS DESIGN

The main ideas of our methodology are that security principles should be applied at every stage of the software lifecycle and that each stage can be tested for compliance with security principles. Another basic idea is the use of patterns to guide security at each stage. Patterns are applied to cover all architectural levels. This methodology considers the following development stages:

Domain analysis stage: A business model is defined. This phase should be performed only once for each new domain. General security constraints, including regulations and institution policies, can be applied at this stage.

Requirements stage: Use cases define the required interactions with the system. Applying the principle that security must start from the highest levels, it makes sense to relate attacks to use cases. We study each action within a use case and see which threats are possible. We then determine which policies would stop these attacks. From the use cases we can also determine the needed rights for each actor and thus apply a need-to-know policy. The security test cases for the complete system are also defined at this stage.

Analysis stage: Analysis patterns can be used to build the conceptual model in a more reliable and efficient way. Security patterns describe security models or mechanisms. We can build a conceptual model where repeated applications of a security model pattern realize the rights determined from use cases. In fact, analysis patterns can be built with predefined authorizations according to the roles in their use cases. In that case, we only need to additionally specify the rights for those parts not covered by patterns.

Design stage: When we have the possible attacks to the system, design mechanisms are selected to stop these attacks. User interfaces should correspond to use cases and may be used to enforce the authorizations defined in the analysis stage. Components can be secured by using authorization rules for Java or .NET components. Distribution provides another dimension where security restrictions can be applied. Deployment diagrams can define secure configurations to be used by security administrators. A multilayer architecture is needed to enforce the security constraints defined at the application level. In each level we use patterns to represent appropriate security mechanisms. Security constraints must be mapped between levels.

Implementation stage: This stage requires reflecting in the code the security rules defined in the design stage. Because these rules are expressed as classes, associations, and constraints, they can be implemented as classes in object-oriented languages. In this stage we can also select specific security packages or COTS, e.g., a firewall product, a cryptographic package. Some of the patterns identified earlier

in the cycle can be replaced by COTS (these can be tested to see if they include a similar pattern).

III. A POSSIBLE COMBINATION

We can incorporate these usability patterns in our methodology in two basic ways:

• *For the construction of interactive applications*. Some interactions can be very sensitive, such as accessing a bank account by its owner. It is important for the users of such systems to be aware of the security attacks that are possible in the interaction. As an example, the activity diagram of Figure 1 shows the activities needed to open an account in a financial institution. For each activity we may need to use a specific screen, e.g. to create an account a manager would use a "Create Account" screen. The figure also shows possible attacks [1]; for each of them we need to warn the user in case it is happening or about the possibility of the attack happening if some precautions are not taken. From the scenarios and activity diagrams of the requirements stage we can see where we need views (screens) to interact with the system. These views are associated with conceptual model classes in the analysis stage. In the design stage we can use the MVC pattern to implement the views. For each view we can add a security subview, intended to provide feedback to the user in case of threats or if the user accidentally performs a potentially insecure action. Warnings about privacy can also be included in this way. For example, for the action "Provide personal information", the customer would receive warnings about privacy and a description of what security measures the system would take to protect this information. He would also be warned that his interaction with the institution requires checking that he is talking to the authentic web site.

• *To develop convenient facilities for security administrators*. Each security mechanism needs the definition of rules to indicate who can access specific resources of the system and what she can do with them. The person in charge of maintaining these rules is the security administrator. A confusing view of the authorization could result in errors and subsequent security violations. The administration interfaces should show clearly which roles have which rights, which users belong to a specific role, which are the rights of each role, etc. Of particular importance is the effect of new or changed rules; in this case the interface should display the effect of changes before the change becomes effective.

The security of human interaction considers how the security features of the user interface can be as friendly and intuitive as possible, to let users understand the security features, thus avoiding errors in their use. A group of patterns have been designed for this effect [4,5], which can be used in the corresponding user interfaces. These patterns can be applied to the two situations described above. Our approach can be described as a model-driven, pattern-based methodology and we believe that this is the appropriate level where security should be applied, not just in the code.

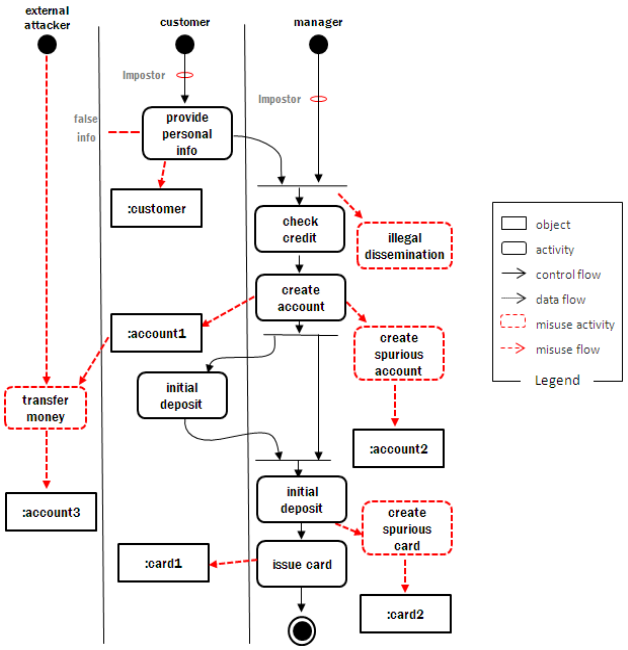


Figure 1. *Activities in use case "Open account"*

IV. CONCLUSIONS

Usability aspects are fundamental in any development methodology since almost all applications require some type of user interaction. Usability is usually reflected in the logical structuring of class operations and in the way to show to the users normal and exceptional operation of the system. In particular, the usability of the security administrator interfaces is basic for security. We have proposed here an approach to integrate a secure systems development methodology with a methodology to make user interfaces more usable for security purposes. We are still in the process of defining its specific details and scope and we expect to get ideas for further work from this workshop.

REFERENCES

[1] F. Braz, E.B.Fernandez, and M. VanHilst, "Eliciting security requirements through misuse activities" Procs. of the 2nd Int. Workshop on Secure Systems Methodologies using Patterns (SPattern'07). In conjunction with the 4th International Conference onTrust, Privacy & Security in Digital Busines(TrustBus'07), Turin, Italy, September 1-5, 2008. 328-333.

[2] E. B. Fernandez, M.M. Larrondo-Petrie, T. Sorgente, and M. VanHilst, "A methodology to develop secure systems using patterns", Chapter 5 in "Integrating security and software engineering: Advances and future vision", H. Mouratidis and P. Giorgini (Eds.), IDEA Press, 2006, 107-126.

[3] L.L. Lobato and E.B. Fernandez, "Patterns to support the development of privacy policies", Procs. of the First Int. Wokshop on Organizational Security Aspects (OSA 2009). In conjuction with ARES 2009.

[4] R. Mendoza, J. Muñoz-Arteaga, M. Vargas, and F. Alvarez-Rodriguez, "A pattern methodology to specify usable security in websites", Procs. of the Third Int. Workshop on Secure System Methodologies using Patterns (SPattern 2009). August 31-Sept. 4, Linz, Austria

[5] J. Muñoz-Arteaga , R.Mendoza-Gonzalez, J. Vanderdonckt, F. Álvarez-Rodriguez, "A methodology for designing information security feedback based on user interface patterns, Journal of Advances in Engineering Software (JAES), Elsevier, April 2009..