Working More Effectively with Legacy Code Using Domain Knowledge and Abstractions: A Case Study

Igor Osetinsky and Reuven Yagel

Software Engineering Department, Azrieli - The Jerusalem College of Engineering,
POB 3566, Jerusalem, 91035, Israel

Abstract. We describe how using abstractions based on domain knowledge can improve the process of changing legacy code. The central claim of this paper is that ontologies are a desirable starting point, not only for development of new code, but also for effective refactoring of legacy code. We start from a standard refactoring method from the literature of working with legacy code, and show how it has been improved into knowledge-based refactoring. A use case illustrates the practical application in an industrial project setting.

1 Introduction

Maintenance is a major phase in the lifecycle of software products. The maintenance part consumes typically about 60% of time and budget of software projects [10]. One reason for such high cost is the state of the code, which is frequently maintained badly, thus leading to a code-base that collapses under its heavy complexity followed by ever increasing costs of repairing bugs and adding new features.

Agile development practices put emphasis on testing and test coverage techniques as a way to maintain high code quality over time. Thus one might try to use test coverage as a way of working with code that is already in production phase, commonly referred to as legacy code.

This paper goes a step further beyond agility, by introducing domain knowledge, in the form of ontologies, in an early stage of legacy code refactoring, leading to knowledge-based refactoring. This guides the refactoring programmer at a much higher and effective abstraction level than customary.

1.1 Working with Legacy Code

Legacy code, is a title sometimes attributed to old and brittle code which is hard to work with, but is still in production. For the purpose of this work, and in agreement with the agile viewpoint on testing which we mentioned above, we follow here Michael Feathers's book "Working effectively with Legacy code" [6], which defines: "*legacy code* is simply code without tests". The book goes on to suggest a process or general algorithm for working with legacy code and brings many methods and examples of how to do that in various scenarios.

Osetinsky I. and Yagel R..
Working More Effectively with Legacy Code Using Domain Knowledge and Abstractions: A Case Study.
DOI: 10.5220/0005168400630070
In *Proceedings of the 5th International Workshop on Software Knowledge* (SKY-2014), pages 63-70
ISBN: 978-989-758-051-2

1.2 The Problem – Too Low Level Refactoring Applied to Legacy Code

The problem is that the programmer which is equipped with those excellent methods and techniques still needs more guidance on how to apply them to the actual mission at hand. In particular, *Code refactoring* [7] methods are being collected and described as patterns for improving code in incremental ways. Most of these patterns are rather at a quite low level of coding and in fact today are being automated by IDEs.

1.3 Related Work

Agility, among others terms and practices relevant to our work, includes unit testing [8], and test-driven-development [1].

There is a wide literature concerning generic higher abstraction level ideas that can enhance the process of working with legacy code. We mention Ontologies [2], Knowledge Management [3], Software Knowledge [5], Domain Driven Design [4], and more broadly, Patterns and Principles, e.g. [9, 12] and their use in object oriented programming and software engineering. It is out of scope for this paper to describe in detail the various ideas, and the interested reader is invited to check the above references and other resources.

In [14] it is shown how ontology can guide the development of a new system, in contrast to the current work in which the emphasis is on existing software. Yang et al. [15] proposes extracting ontologies from legacy code, to improve the understanding and eventually the re-engineering of the respective code. Their emphasis is on the ontology extraction, while ours is on the ontology utilization in the refactoring process.

Specific ontologies relevant to the use case in this work include, e.g. ITSMO [11] a service management ontology, referring to software utilities, and OWL-S: Semantic Markup for Web Services [13].

The remaining of this paper is organized as follows: In Section 2 we detail the original proposed process and how we elaborate on it, in Section 3, we bring one use case for demonstrating this improved method and then in section 4 we conclude with some remarks and future work.

2 The Solution – Refactoring with Domain Knowledge

2.1 Standard Refactoring Process

The main process suggested by Feathers for adding new features is as follows:

1. **Identify Change Points:** that is, finding the actual place(s) in the code where the change is going to be applied. This can be achieved in many ways, e.g., identifying the related and relevant areas in the code, consulting (original) developers, reading documentation and error reports, etc.
2. **Find Test Point:** sometimes and especially with legacy code, it is not so easy to determine what, where and how to test the code. Legacy code, as described,

can become over time less and less modular with high coupling which makes it hard for isolation, thus sometimes there's a need to:

3. **Break Dependencies:** the code we would like to change might depend on another code which makes it hard for testing. Breaking dependencies is a way to isolate the targeted code.
4. **Write Tests:** after isolating the code, it is presumably more testable and thus we sometimes try to cover it with tests to demonstrate the current behavior, followed by writing tests for the new behavior. These new tests specify the required change and should be kept for future verification and validation / regression testing.
5. **Make Changes and Refactor:** we can now go ahead and make the changes and can even improve or refactor it, counting on the tests to insure code correctness. Overtime the system is being covered with many tests that bring with them also the confidence to continue improving the code on a regular basis – thus keeping its quality and ability to develop and adapt to changes.

2.2 Knowledge-based Refactoring

Our proposed way to improve the standard refactoring process above, is to add steps to be carried out at a higher abstraction level. Steps a) and b) below can be carried out in parallel with above steps 1 and 2. The other steps are augmented as follows:

a. **Specify the Relevant Domain:** either manually or by usage of a relevant software tool, one should define the knowledge domain relevant to the legacy code.
b. **Obtain the Relevant Ontology:** once the domain has been specified, one should either obtain a relevant ontology from the literature (as e.g. [11]) or explicitly formulate such an ontology.
3. **Break Dependencies:** use the ontology to detect the main abstractions and concepts, preferring dependency on interfaces rather than concrete implementations, e.g. by applying the Dependency Inversion Principle [12] (see the case study below).
4. **Write Tests:** tests should refer to the more abstract concepts, thus more concise.
5. **Refactor:** explicitly introduce the higher level abstractions first in UML and then in the code, e.g. the new interfaces, with the respective needed inheritances.

3 A Case Study

We demonstrate here the improved process with an actual code of a real project. This is part of a stress/load testing suite for an online library service in the industry. Here we show two Java classes which are tightly coupled. The code in Figure 1 is pulling running statistics from a server and the code in Figure 2 is analyzing this data.

```java
/**
 * extracts jstat info from server using sshUtil
 *
 *
 */
public class JStatUtils{


    private JStatAnalyzer _jStatAnalyzer;
    private String jstatResult;
    private SSHUtil _sshUtil;

    public void setJStatAnalyzer(JStatAnalyzer jStatAnalyzer){
        _jStatAnalyzer = jStatAnalyzer;
    }

    public void getJstatFromServer(){
        this.jstatResult = _sshUtil.getJstatResult();
    }

    public void set_sshUtil(SSHUtil _sshUtil) {
        this._sshUtil = _sshUtil;
    }
```

Fig. 1. JStatUtils.java – Holder class for needed utilities, e.g. secure connection for getting statistical results.

```java
/**
 * analyzes jstatResults
 *
 *
 */
public class JStatAnalyzer {


    private JStatUtils _jStatUtils;

    public JStatAnalyzer(JStatUtils jStatUtils){
        _jStatUtils = jStatUtils;
        jStatUtils.setJStatAnalyzer(this);
    }

}
```

Fig. 2. JStatAnalyzer.java – The main logic of the presented code.

The code in the class JStatAnalyzer.java of Figure 2 depends on services, which are encapsulated in the class JStatUtils.java of Figure 1 (the class logic itself is omitted and not relevant here, only the dependencies are presented). In this code arrangement JStatAnalyzer is hard to test in isolation, especially since JStatUtils relies on an actual server for obtaining the results.

3.1 Refactoring with Domain Knowledge

To fix this dependency we could use standard low level techniques for breaking dependencies, namely *Extract Interface* and *Parameterized Constructor* [6]. But still we need domain knowledge to find the right abstractions.

As a first step we look for the relevant domain, namely the *domain of services, and monitoring*. Then we formulate the dependency on several services or "utils" (utilities) and thus extract a utils interface. This can be a standalone abstraction or taken from a relevant Ontology such as [11] mentioned above.

We can now move forward and break the dependency using the found abstraction. Figure 3 demonstrates this change with a UML class diagram. On the left is the original design, while on the right is the improved design. Especially note the direction of the arrow which is inverted between the two designs. This encodes the inversion of dependencies which on the one hand, adds another entity to the design (the new *JStatUtils interface*) but makes the dependency rely on a more abstract concept and thereafter better testability.

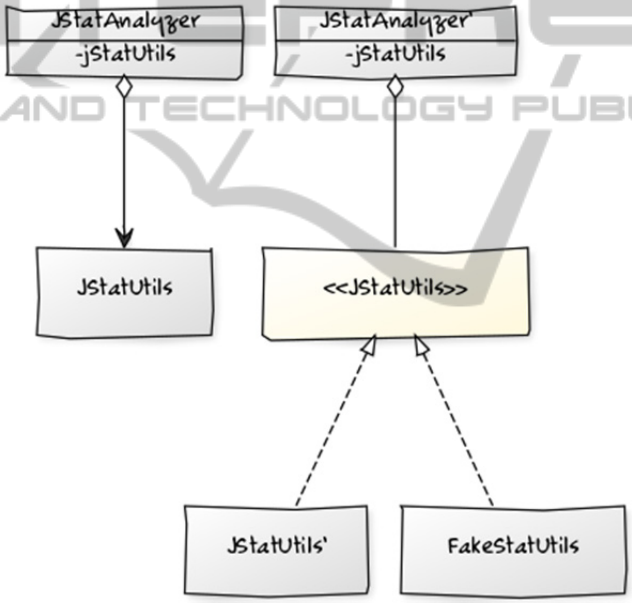

Fig. 3. UML Class Diagram, left: two legacy classes before refactoring; right: the new *JStatUtils interface* in between the two original classes. The inherited *FakeStatUtils* class can have dummy values for testing the Analyzer class logic.

Finally, we can refactor out the interface (Figure 4), and now the Analyzer class depends on this interface only. Even more importantly it depends now on the higher level concept of utilities.

As a result of this process, the Analyzer class is also more testable. Figure 5 shows how it can now be initiated with an anonymous simple class (omitting the implementation for simplicity), e.g., for testing purposes. This is also depicted in the UML diagram of Figure 3 by the inherited class FakeStatUtils which can be a class

with dummy values used for testing the logic of the analyzer class, supplying fixed values that allow fast and repeatable tests.

```
public interface JStatUtils {

    void setJStatAnalyzer(JStatAnalyzer jStatAnalyzer);

    void getJstatFromServer();

    void set_sshUtil(SSHUtil _sshUtil);
}
```

Fig. 4. Extracted Interface JStatUtils – according to domain knowledge.

```
JStatAnalyzer analyzer = new JStatAnalyzer(new JStatUtils() {

    public void set_sshUtil(SSHUtil _sshUtil) {}

    public void setJStatAnalyzer(JStatAnalyzer jStatAnalyzer) {}

    public void getJstatFromServer() {}

    });
```


Fig. 5. Easier Testing of the Refactored Class – is much easier, even with an anonymous class.

3.2 Refactoring: An Iterative Process

Once we have the refactored code, we can further use ideas from, e.g. Domain Driven Design (DDD) [4] to make sure that a service is defined appropriately, e.g., in case the service is not a natural part of the domain (entity in terminology of [4]), that the interface is defined in terms of other elements of the domain.

On the negative side, we note that the operations are not stateless, which goes against those DDD guidelines. This direction can be combined with using a selected ontology in order to find adequate names for, .e.g., subjects and entity names.

The refactored code can now also be augmented more easily with new features, for example replacing one of the utilities with yet another different implementation. This process is iterative and we can continue to refactor and enhance it. As yet another example if we need to use another service in order to fetch the data for the analyzer, we can apply the Interface Segregation Principle [12] and further separate the extracted interface into more specific domain abstractions, e.g., a security service.

4 Discussion

We have shown how that the currently standard low-level techniques for working with legacy code are much improved when aided by higher abstractions based on concepts from domain knowledge ontologies.

The higher abstractions still leave to the software developer decisions concerning the application of the suitable techniques at various abstraction levels, and choice of the appropriate principles, techniques and patterns.

The common wisdom and pragmatic way for improving code quality, is doing so only when changes are requested and implementing it in an incremental and iterative way.

The enhanced process described here was applied to an industrial large code base in a leading company of its domain. Actually, by the time of writing this paper, the example code has already been transformed and changed completely and the two originally described classes do not exist anymore.

4.1 Future Directions

In order to get a wider perspective and more value from knowledge-based refactoring, more extensive work on case studies should be done and shared knowledge should be gathered.

Another more advanced research direction is to try automating some of the steps in the process described in sub-section 2.2.

In future work we intend to focus and demonstrate the applicability of the improved refactoring process to writing maintainable tests.

4.2 Main Contribution

The main contribution of this paper is knowledge-based refactoring, viz. the idea that a domain knowledge ontology is a desirable starting point for effective refactoring of legacy code in a higher level of abstraction.

Acknowledgement

Finally, we thank the SKY anonymous reviewers and chairs for their helpful comments and most valuable aid.

References

1. Beck, K.: Test Driven Development: By Example, Addison-Wesley (2002).
2. Calero, C., Ruiz, F. and Piattini, M. (eds.): Ontologies in Software Engineering and Software Technology, Springer, Heidelberg, Germany, (2006).
3. Dalkir, K. and Liebowitz , J.: Knowledge Management in Theory and Practice. The MIT Press, 2nd Ed. (2011).
4. Evans E.: Domain-Driven Design: Tackling Complexity in the Heart of Software. Prentice Hall (2003).
5. Exman, I., Llorens, J. and Fraga, A.: "Software Knowledge", pp. 9-12, in Exman, I., Llorens, J. and Fraga, A. (eds.): Proc. SKY Int. Workshop on Software Knowledge (2010).

6. Feathers M.: Working Effectively with Legacy Code. Prentice Hall (2004)
7. Fowler M., Refactoring: Improving the Design of Existing Code, Addison-Wesley Professional (2001).
8. Freeman, S., and Pryce N.: Growing Object-Oriented Software, Guided by Tests, Addison-Wesley (2009).
9. Gamma, E., Helm R., Johnson R. and Vlissides J.: Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley (1995).
10. Glass, R. Software Conflict. Yourdon Press (1991).
11. ITSMO – IT Service Management Ontology (2012). See web site (last accessed August 2012).
12. Martin, R.: Agile Software Development, Principles, Patterns, and Practices, Pearson (2012).
13. W3C, OWL-S: Semantic Markup for Web Services, http://www.w3.org/Submission/OWL-S/ (2004).
14. Yagel, R., Litovka, A. and Exman I.: KoDEgen: A Knowledge Driven Engineering Code Generating Tool, The 4th International Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management (IC3K) - SKY Workshop, Vilamoura, Portugal, 2013.
15. Yang, H., Cui, Z. and O'Brien, P., "Extracting ontologies from legacy systems for understanding and re-engineering", in Compsac'99 23rd Annual International Computer Software and Applications Conference, pp. 21-26, (1999). DOI = 10.1109/ CMPSAC.1999.812512.