Electronic Communications of the EASST
Volume 48 (2012)

Proceedings of the Fifth International Workshop on on Foundations and Techniques for Open Source Software Certification (OpenCert 2011)

Using antipatterns to improve the quality of FLOSS development

Antonio Cerone and Dimitrios Settas

16 pages

Guest Editors: Luís Soares Barbosa, Dimitrios Settas
Managing Editors: Tiziana Margaria, Julia Padberg, Gabriele Taentzer
ECEASST Home Page: http://www.easst.org/eceasst/ ISSN 1863-2122

Using antipatterns to improve the quality of FLOSS development

Antonio Cerone[1] **and Dimitrios Settas**[2]

[1] antonio@iist.unu.edu [2] settdimi@iist.unu.edu
UNU-IIST — International Institute for Software Technology
United Nations University, Macau SAR China

Abstract: Antipatterns have been mostly reported in closed source software environments. With the advent of Free/Libre Open Source Software (FLOSS), researchers have started analysing popular FLOSS projects, seeking vitality indicators and success patterns. However, an impressively high percentage of FLOSS projects are unsuccessful. Moreover, even in the successful cases of FLOSS there can be found tracks of failed attempts, dead-ends, forks, abandonments etc. FLOSS antipatterns can help developers to improve their code and improve the communication and collaboration within the FLOSS community. In this paper, we present some example of FLOSS antipatterns and discuss the benefits that they bring to various FLOSS user roles. Furthermore, we present ontology-based technology and software tools that can be used to assist FLOSS developers and community users to identify, document, share antipatterns and use these mechanisms to assist FLOSS projects conform to specified requirements. Finally, we propose a framework for the quantitative identification of the antipatterns to use as quality indicators in the certification of FLOSS products.

Keywords: FLOSS development, antipatterns, certification, ontology

1 Introduction

An antipattern is a new form of pattern that has two solutions [BMMM98]. The first is a problematic solution with negative consequences and the other is a refactored solution, which describes how to change the antipattern into a healthy solution. The second solution is what makes antipatterns beneficial. The difference is in the context: An antipattern is a pattern with inappropriate context and is particularly useful in the case of knowledge representation, because it captures experience and provides information on commonly occurring solutions to problems that generate negative consequences [LN06]. The process that is followed by a pattern to change its solution into a better one is called refactoring. This solution has an improved structure that provides more benefits than the original solution and refactors the system toward minimised consequences.

FLOSS anti-patterns are not yet explored to the same extent as in closed source. In addition, because FLOSS and closed source software produce code using very different development processes, FLOSS antipatterns are quite different in nature from their closed source counterparts. There exist different categories of antipatterns. According to the literature [BMMM98, LN06] closed source software antipatterns exist at a development, architectural and managerial level. FLOSS antipatterns mostly exist at a community level and describe social and managerial issues regarding communication, interaction and coordination among developers that participate

in FLOSS projects. However, closed source software development antipatterns are also applicable in FLOSS projects and can greatly affect the quality of both closed and open source software projects.

This paper describes how antipatterns can be used in FLOSS projects by defining the sources of these antipatterns and the different user roles of FLOSS antipatterns. While antipatterns cannot be used as a formal certification approach, different kinds of antipatterns (i.e. development, community level) can be used

1. within the FLOSS development process to directly overcome problems that may affect the certification of the FLOSS product;

2. as part of the certification process to define indicators of the quality of the development process and the resultant FLOSS product.

Development antipatterns may help developers overcome commonly occurring coding problems. For example, the "Spaghetti Code" antipattern [BMMM98] can be used to describe code that has a complex and tangled control structure, especially one with several exceptions, threads, or other "unstructured" branching constructs. Spaghetti code can be caused by several factors, including inexperienced programmers and a complex program which has been continuously modified over a long life cycle. A solution proposed to resolve the antipattern is using a formal and predictable style of coding such as that of Structured Programming. Community antipatterns can help developers overcome problematic FLOSS practices, such as participation and motivation problems, which are crucial to FLOSS development. For example, "The Big Show" antipattern [Nea11] describes the scenario in which companies will work for several months on a software project behind closed doors before announcing it to the public. This behaviour negatively impacts the ability of a FLOSS community to grow outside corporate walls. The project members are assigned to "secret" projects and interact less with the community, and the end result is a big code drop which has not had public peer review and was not listed on any roadmap before its announcement, resulting in people outside the company feeling like second class citizens.

Since encountered coding problems and problematic community practices highlighted by such antipatterns affect the quality of the FLOSS product, once an antipattern has been identified, it may be incorporated in the certification process as a negative quality indicator.

One problem is that identifying antipatterns is essentially a qualitative process, in which symptoms are associated to antipatterns either directly or indirectly by means of existing causal relationships between antipatterns. Such a qualitative nature of antipattern identification is not adequate to the accuracy required by a formal certification process. In order to overcome this problem, metrics for technical quality, based on the ISO/IEC 9126 standard, that have been successfully applied to proprietary software as well as to FLOSS [CV08] could be customised to detect specific antipatterns that describe coding problems such as the Spaghetti Code antipattern.

The rest of the paper is organised as follows. In Section 2 we give a short introduction to antipatterns and to the way they are identified and used. Section 3 presents an overview of FLOSS community antipatterns whose descriptions are publicly available at the moment. Section 4 identifies FLOSS contributor types that could benefit from antipattern technology. Section 5 illustrates the tool support available for the antipattern technology. Section 6 proposes an approach to a quantitative identification of antipatterns and to the definition of an antipattern measure to

be used in the certification process of a FLOSS product.

2 Antipatterns

Brown *et al.* [BMMM98] define an antipattern as

> a literary form that describes a commonly occurring solution to a problem that generates decidedly negative consequences.

When properly documented, an antipattern describes a **general form** that defines the generic characteristics of the antipatterns as well as

causes which led to the general form;

symptoms describing how to recognise the general form;

consequences of the general form;

refactored solution describing how to change the antipattern into a healthier situation.

The general form of the antipattern provides an easily identifiable template for the class of problems addressed by the antipattern. Such general form can be further structured and enriched with additional template elements to best characterise the context for the existence of a particular antipattern in order to reduces the most common mistake in using design patterns: applying a particular design pattern in the improper context.

The first official template that can be used to document antipatterns was provided by Brown *et al.* [BMMM98]. The authors highlighted the importance of a template in order to define what can be considered as an antipattern. This antipattern template consists of 18 elements. Some of these elements are optional but others, such as the elements that characterise the **general form** and the **Refactored Solution**, are required to be completed. Although this is the most detailed template to describe antipatterns, it is not recommended for FLOSS antipatterns, as it requires background knowledge on completing specific sections such as the scale and the root causes of the antipattern, which are not available for the mostly informally documented FLOSS antipatterns.

Several other templates are available for documenting FLOSS antipatterns. The Pseudo-Antipattern template includes only the name of the antipattern and what problems are addressed [BMMM98]. This template is not very useful but has appeared on the Internet to quickly describe an antipattern. A more detailed version is the Mini-Antipattern template, which describes the two solutions of the antipattern, the problematic solution and the refactored solution [BMMM98]. These informal templates can be used in order to quickly describe an antipattern but they do not capture important elements of FLOSS antipatterns such as causes, symptoms and consequences.

An appropriate compromise to address both the informal presentation style of FLOSS antipatterns and the information completeness and structure needed in a certification process is the Laplante and Neil's template [LN06] illustrated in Table 1. Here **Dysfunction** provide a contextual description of **syntoms** while **Explanation** describes **causes** and **consequences**.

Table 1: Laplante's Antipattern template.

Name	A short name that conveys the antipattern's meaning.
Central Concept	The short synopsis of the antipattern in order to make the antipattern identifiable.
Dysfunction	The problems and symptoms with the current practice.
Vignette	The antipattern in a real or prototypical situation, it provides context and richness to the antipattern.
Explanation	The expanded explanation including causes and consequences.
Band-Aid	A short term coping strategy for those who don't have the influence nor time to refactor it.
Self-Repair	The first step for someone perpetuating the antipattern.
Refactoring	The required changes in order to remedy the situation and their rationale.
Observations	Optional section for additional comments or items of note.
Identification	An assessment instrument consisting of a list of questions for diagnosis of the antipattern.

We will consider a slightly simplified version of Laplante and Neil's template consisting of 8 rather than 10 elements. We remove the **Observations** element, which is anyway optional in Laplante and Neil's template, and the **Vignette** element, since in a freedom-centred ecosystems as a FLOSS community prototypical situations rarely occur and a single real situation may provide an incomplete and possibly misleading context.

In order to exemplify how our modified template can be used to document a FLOSS antipattern, the "Spaghetti Code" antipattern [BMMM98] has been documented (Figure 1) using this template. As already mentioned in the previous section this development antipattern can be used to describe code that has a complex and tangled control structure, especially one with several exceptions, threads, or other "unstructured" branching constructs. This one page description (Figure 1), provides a quick overview of this antipattern together with details on how it can be identified and resolved.

3 FLOSS Community Antipatterns

FLOSS community antipatterns are very relevant to the certification of FLOSS projects as they describe ways to build stronger project communities with effective communication, collaboration and management practices.

O'Brien [O'B] has recently proposed six OSS community antipatterns, which are characterised by six "personalities" that have emerged in the last years among FLOSS communities: Rule maker, Open Source Politician, Attack Dog, Non-contributing Pontificator, Back Room Dealer and (Apache Way) Ambassador. For each of these antipatterns, O'Brien describes the causes, symptoms and consequences as well as the refactored solution that can make the antipattern beneficial to a FLOSS project. For example, although it is important to have a few rule

Name Spaghetti Code

Central Concept Due to hurry or some other pressure, some developers and teams in organisations do not follow any standard software structure of the project work.

Dysfunction Several exceptions, threads, or other "unstructured" branching constructs. This leads to lack of clarity, to the extent that even the developer himself will not be able to follow it later if any problem arises during the software development cycle.

Explanation The main cause for an organisation to get into such a situation is due to either employing people without having a strong base on design or inadequate training provided to the developers.

Band-Aid Ruthless diligence combined with constant unit testing

Self-Repair If prevention of Spaghetti Code is an option, or if you have the luxury of fully engineering a Spaghetti Code application, the following preventative measures may be taken:

1. It is crucial that any moderate or large-size project develop a domain model as the basis of design and development.
2. After developing a domain model that explains the system requirements and the range of variability that must be addressed, develop a separate design model.

Refactoring Using a formal and predictable style of coding such as that of Structured Programming. Refactoring the software includes performing the following operations on the existing code:

1. Gain abstract access to member variables of a class using accessor functions. Write new and refactored code to use the accessor functions.
2. Convert a code segment into a function that can be reused in future maintenance and refactoring efforts. It is vital to resist implementing the Cut-and-Paste AntiPattern. Instead, use the Cut-and-Paste refactored solution to repair prior implementations of the Cut-and-Paste AntiPattern.
3. Reorder function arguments to achieve greater consistency throughout the code base.
4. Remove portions of the code that may become, or are already, inaccessible. Repeated failure to identify and remove obsolete portions of code is one of the major contributors to the Lava Flow AntiPattern.
5. Rename classes, functions, or data types to conform to an enterprise or industry standard and/or maintainable entity. Most software tools provide support for global renaming.

Identification The following identification instrument can help you determine if your organisation tends to use one-dimensional management techniques. Please respond to the following statements with Yes or No.

- My code has a complex and tangled structure
- The program has been continuously modified over a long life cycle.
- Interaction between OO objects is minimal.
- Inheritance and polymorphism aspects of OO methods are not used.

Figure 1: Spaghetti Code Antipattern.

makers within the community, when their growth in numbers of individuals and their attempts to advocate and enforce standards appear to hinder productive community activities, it may be necessary to split the community up into smaller, more focused teams, each governed by an active code contributor, who is expected to have a natural tendency to avoid needless bureaucracy [O'B].

Josh Berkus [Ber11] has listed ten ways on how to destroy a FLOSS community. These antipatterns are likely to arise when companies are involved or even are leading FLOSS projects. For example, a company leading a FLOSS project may permanently prevent anybody outside the company from having commit access, respond evasively to queries and/or choose employees who write no code as commiters on the project.

Leung [Leu08] has identified 22 FLOSS antipatterns and has provided a very short description of each one. Most importantly he identified the lack of management itself as an antipattern emphasising the need for managing FLOSS projects.

Dave Neary [Nea11] has developed a Community Management Wiki in which he maintains 18 FLOSS community antipatterns. These antipatterns are described in more detail and include the symptoms of each antipattern together with their refactored solution. For example, the "anarchy" antipattern highlights the problems associated with the "Free Software is all about absolute Freedom" way of thinking. This kind of thinking leads to a kind of anarchistic community where a substantial proportion of members affirm a total right to freedom of speech, freedom of expression, and other extreme libertarian principles. The result is a community where no indiscretion can easily be corrected, because any attempt to do so is perceived as a limit to the absolute freedom of speech. Another important antipattern identified by Neary is the "Big Show" antipattern, for which we give a detailed description in Figure 2 using Laplante and Neil's modified template. This antipattern describes the problematic practice in which a company will secretly operate on a FLOSS project behind closed doors for several months before announcing it to the public.

These 56 antipatterns contain valuable knowledge that can be used by the users of a FLOSS community in order to identify problematic practices or dysfunctional processes that have been previously documented as antipatterns.

The main source of FLOSS antipatterns is the Web, as no further FLOSS community antipatterns have been published elsewhere at the moment. These antipatterns do not use official antipattern templates proposed by Brown *et al.* [BMMM98] or Laplante and Neil [LN06]. They are documented using a short textual description in order to quickly describe them and allow users to memorise them easily. An open issue that has not been resolved for community antipatterns is the lack of formalisation of this knowledge, which, in fact, would be essential to enable the use of software tools to support this technology as well as to quantify the process of their identification. Furthermore, as far as the authors are aware there is no single knowledge base that documents these antipatterns and allows their use by software tools.

Our use of a modified version of Laplante and Neil's template to document FLOSS community antipatterns is an attempt to give structure and richness to taxtual descriptions available on the web. This approach can benefit several types of FLOSS contributors by enabling them to extend the use of support software tools for antipattern technology to FLOSS community antepattens. We will describe these different types of FLOSS contributors in Section 4 and we will illustrate the use of support software tools for antipattern technology Section 5.

Name Big Show

Central Concept Describes the scenario in which companies will work for several months on a software project behind closed doors before announcing it to the public.

Dysfunction The project members are assigned to "secret" projects and interact less with the community, and the end result is a big code drop, which has not had public peer review and wasn't listed on any roadmap before its announcement, resulting in people outside the company feeling like second class citizens.

Explanation This behaviour negatively impacts the ability of a FLOSS community to grow outside corporate walls.

Band-Aid Unfortunately, no short term coping strategy can be used for those with neither influence nor time to refactor it. A company can only choose to share a project with the FLOSS community or can choose to keep it internal.

Self-Repair Adopt an "Open" strategy for company-based FLOSS projects.

Refactoring Try to understand the benefits for the company by sharing a project from its beginning to its completion. Costs can be cut by increasing the numbers of developers working from outside the company and additional functionality can be implemented by having more developers working on the project. As long the license is chosen correctly the company can protect itself without keeping projects secret.

Identification The following identification instrument can help you determine if your organisation tends to use one-dimensional management techniques. Please respond to the following statements with Yes or No.

- My projects FLOSS community only communicates internally, within the company.
- No public peer review is taking place and no documentation is shared with other developers outside the company.

Figure 2: Big Show Antipattern.

4 Antipatterns Users within the FLOSS Development Process

In addition to the 56 community level antipatterns considered in Section 3 different kinds of traditional antipatterns that have been documented in the literature [BMMM98, LN06] and the Web [Malb, Mala, WCa, WCb] can be used at either development level by FLOSS developers (active developers or contributors) or at managerial level (by project leaders or the company which leads the community). Therefore, defining the FLOSS contributor types that could benefit from this technology is very important. These are:

Developers who are the biggest part of the community by contributing and by reporting bugs. They can benefit from both development antipatterns that have been already documented in the literature and antipatterns that are specific to the FLOSS community.

Users of the software, who are also part of the community and download software or exchange

messages in mailing lists. They mainly benefit from community level antipatterns.

Companies FLOSS might be a crucial element in a product or service provided by a company and this company will have to be active or even lead the community [Saa]. Companies benefit from both community level antipatterns [Ber11] and management antipatterns that have been already documented in the literature.

Leaders or Managers of open source communities can benefit from the existence of project management antipatterns that exist in the literature and the Web.

Learners Students and free learners that participate in FLOSS projects in order to learn from more advanced developers or wish to experience FLOSS development. They can benefit from all kinds of antipatterns.

5 Supporting FLOSS antipatterns using software tools

The antipattern ontology [SMSB11a] has been developed using the Web Ontology Language (OWL) and defines antipatterns in terms of concepts and relationships. As a first step towards the use of tool-supported antipattern technology within FLOSS communities, the OWL antipattern ontology has been enriched with data from 13 FLOSS community antipatterns that exist on the Web.

Antipattern attributes (i.e. causes, symptoms and consequences) are treated as different ontology concepts and using ontology properties (i.e. cause-to-symptom) a contributor of an antipattern can define how antipattern concepts are related. The ontology currently contains both closed and open source software antipatterns. Anyone can become an antipattern author and contributor by accessing the Web-based version of the antipattern ontology.

This information is then used as the knowledge base of an intelligent system that can detect which antipatterns exist in a software project using a symptom-based approach [SMSB11b]. SPARSE, the intelligent system that uses this ontology has been released as an open source project [Sou]. The tool mimics the behaviour of an expert on antipatterns and can be used to support FLOSS projects by helping any FLOSS contributor type to identify, document and share antipatterns. In this sense antipatterns can benefit the certification of FLOSS projects by improving the quality of the software code, the FLOSS project practices and the community itself.

There are three underlying technologies involved in SPARSE [SMSB11b]:

Ontologies through the use of the OWL ontology language;

Description Logic (DL) reasoners through the use of the Pellet DL reasoner;

Production rule engines through the use of the CLIPS production rule engine.

The tool operates by importing the antipattern ontology file. FLOSS contributors can choose the visible symptoms that exist in a FLOSS project using the interface shown in Figure 3. This information can then be used from SPARSE in order to detect antipatterns using the reasoning mechanism of SPARSE. The result window displays two sets of antipatterns as shown in

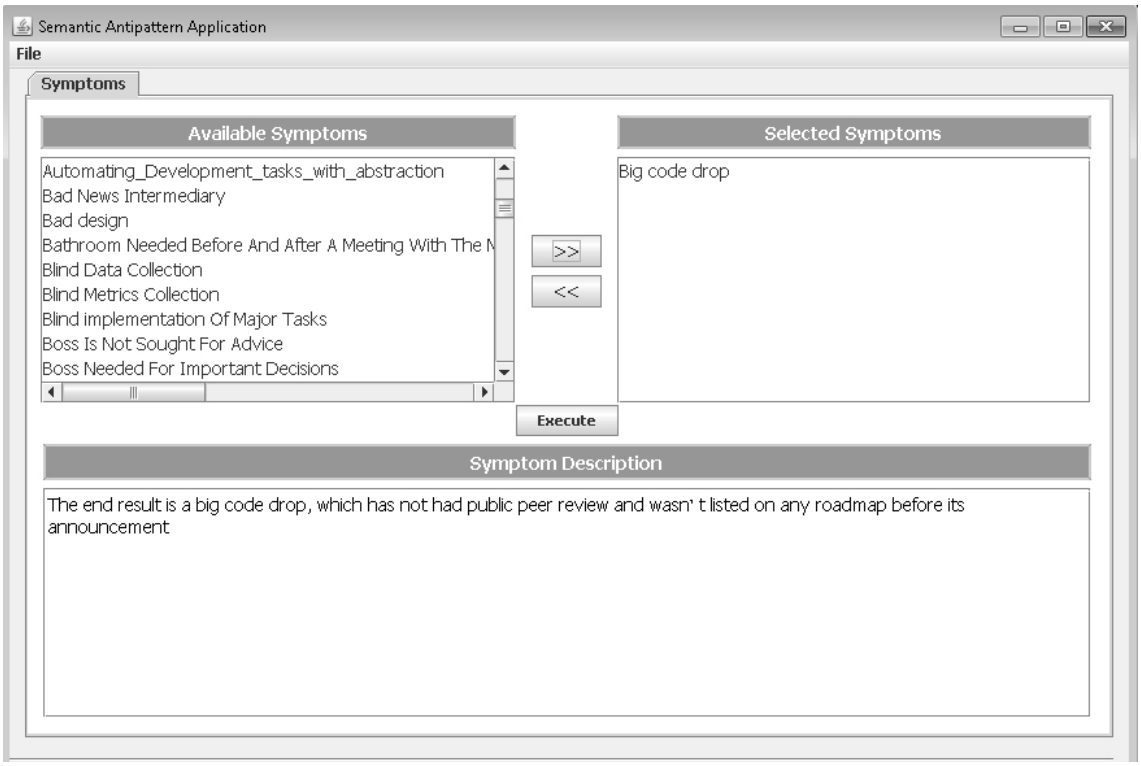

Figure 3: Selecting a set of symptoms that exist in a software project using SPARSE.

Figure 4. On the left hand side there are the related symptom-based antipatterns, which are antipatterns linked through their symptoms. Antipatterns that appear on the right hand side are related through causes or consequences. By clicking on an antipattern SPARSE displays the description and the refactored solution of the antipattern. By selecting the explanations option, SPARSE provides explanations to the learner on the reasons that a specific antipattern has been proposed.

Figure 5 illustrates an example of the explanations that SPARSE provides on how a selected symptom is linked through causes and consequences of other antipatterns. In this example the symptom ”Big Code Drop” was selected (Figure 3), which is a primary symptom in the ”Big Show” antipattern presented in Figure 2. After executing SPARSE, the tool returned the ”Big Show” antipattern because the ”Big Code Drop” is defined as a symptom of the ”Big Show” antipattern in the antipattern ontology. However, the tool also returned other semantically retrieved antipatterns which relate antipatterns through causes and consequences. By selecting the ”Road to Nowhere” antipattern and selecting the explanations option, the tool displays the window shown in Figure 5. This window explains why the antipattern ”Road to Nowhere” might be a relevant result together with the relationships that made the tool reach this conclusion.

FLOSS contributors can further use the system by visiting the antipattern ontology Webprotege installation [SMSB11a] in order to participate in the enrichment of the content of the ontology or edit the existing antipatterns according to their user rights. Moreover, FLOSS contributors

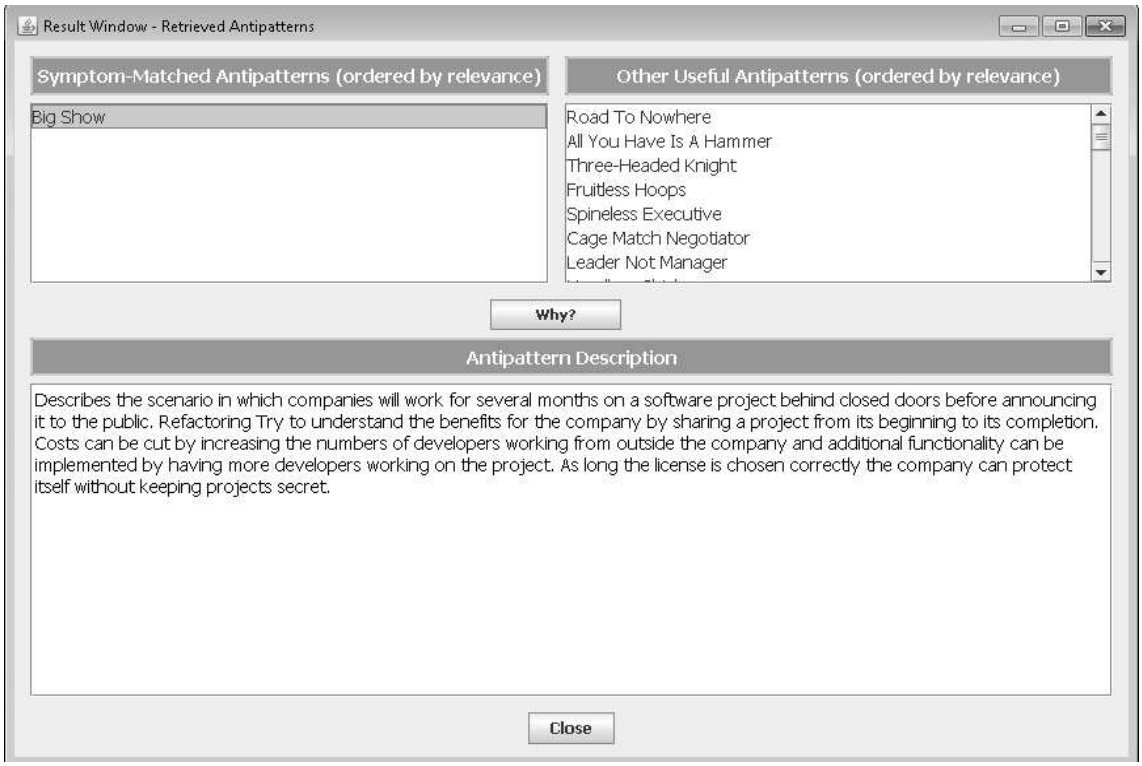

Figure 4: SPARSE result window for antipatterns related through causes or consequences.

are encouraged and asked to create discussions regarding enrichment and changes to the ontology in order to further increase the communication within FLOSS projects.

The antipattern ontology has recently been enriched [SCF12] with probabilistic information describing the certainty with which a cause, symptom or consequence of an antipattern exists in a software project. This makes SPARSE even more intelligent and effective as uncertainty is taken into account. As a result FLOSS developers are also given the option to populate the ontology with their belief on the degree of existence of an antipattern cause, symptom or consequence. This technique is described in detail elsewhere [SCF12].

6 Using Antipatterns to support the FLOSS Certification Process

Community level antipatterns are strongly related to the notion of *quality by development* defined by Shaikh and Cerone [SC09] as specific to the FLOSS development process. Quality by development aims to measure the efficiency of all development and communication processes involved in the production, evolution and release of source code, its execution, testing and review, as well as bug reporting and fixing [SC09]. The idea of *quality by development*, therefore, is an attempt to measure the efficiency of such processes and the interaction between them.

Shaikh and Cerone identify factors that characterise the inherent quality aspects of these pro-

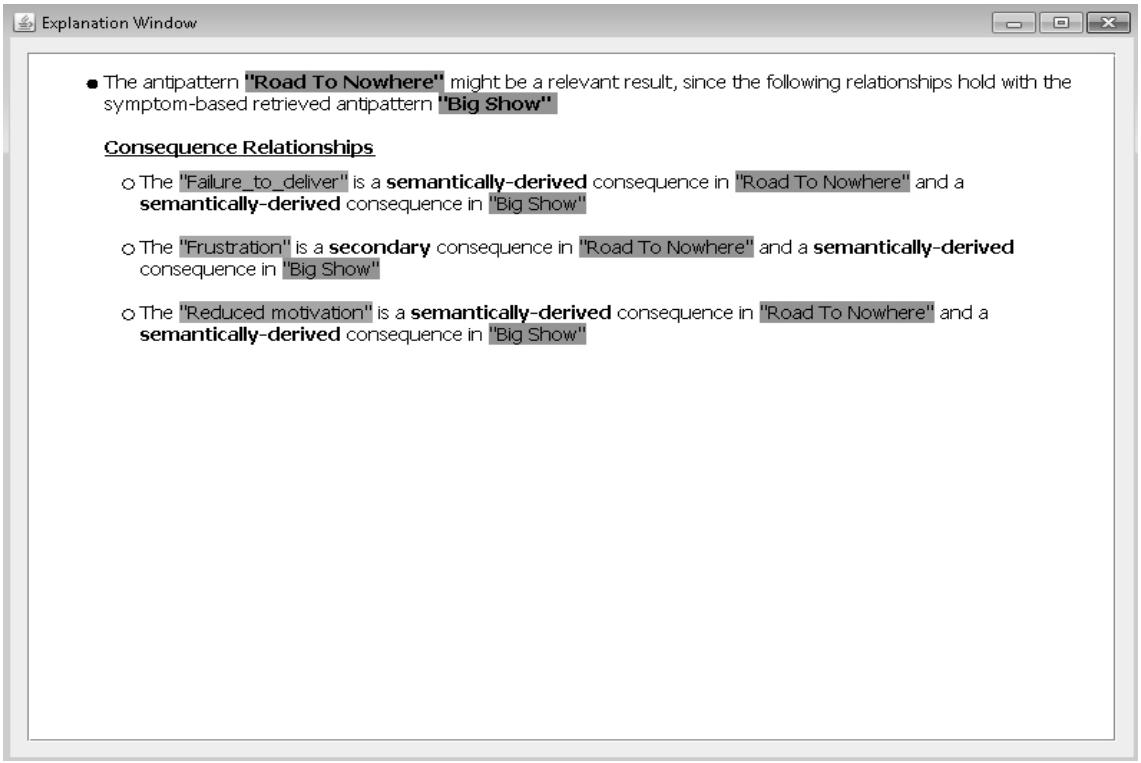

Figure 5: The explanation window of SPARSE describing why an antipattern might be a relevant result.

cesses, but also observe that it is the overall management of the project to play a more central role, with communication and coordination being the two key aspects of this. Management quality aspects that cannot be fully captured by Shaikh and Cerone's quality model, may instead be described in terms of antipatterns. "Make the project depend as much as possible on difficult tools", "provide no documentation", "employ large amounts of legalese", "governance obfuscation" and "don't answer queries" are examples of community level antipatterns that have been described by Josh Berkus [Ber11].

Symptoms of these antipatterns may be detected through the analysis of communications and the activities of the FLOSS project. Collaboration in FLOSS projects is highly mediated by the usage of tools, such as versioning systems, mailing lists, reporting systems, etc. These tools serve as repositories which can be data mined; data can be selectively collected and then analysed not only by using inferential statistics to identify activity patterns [SC10] but also by using ontology engineering formalisms that support the extraction of semantic information. Appropriate ontologies aiming to identify symptoms of antipatterns as well as the solutions applied to solve such antipatterns [SMSB11a], together with measures for the severity of the antipatterns and the effectiveness of the applied solution, can enable the extraction of quantitative information to be used as a measure for quality by development.

One problem in using antipattern in a formal certification process is that their identification is

essentially qualitative. Let us consider the “Spaghetti Code” antipattern illustrated in Figure 1.

The first statement in the identification instrument

> My code has a complex and tangled structure

refers to a code complexity that can be quantified using cyclometric complexity. In general, more sophisticated metrics for technical quality [CV08] could be customised to detect specific antipatterns that describe coding problems.

The second statement

> The program has been continuously modified over a long life cycle.

can be quantified in terms of the number of changes made to the code. FLOSS repositories, such as versioning systems, may be data mined to extract this quantitative information. The other two statements

> Interaction between OO objects is minimal.
>
> Inheritance and polymorphism aspects of OO methods are not used.

are very specific to OO programming style. Their quantification can be carried out by parsing the code to identify and count occurrences of interactions between OO objects as well as occurrences of usage of inheritance and polymorphism.

One way to foster such a quantitative process of antipattern identification is to incorporate quantitative measures and calculation mechanisms in the identification part of the antipattern description. This is especially important in community antitpattern, for which there is often no technical basis for identification. In addition, community antipatterns are often too context specific and usually describe very extreme situations. In reality, between such extreme situations and an ideal situation there is a whole range of variations, which can be adequately quantified and thus contribute to a measure of quality by development.

For example, the “Big Show” antipattern illustrated in Figure 2 describes a very extreme situation in which:

> companies will work for several months on a software project behind closed doors before announcing it to the public.

as documented in the **Central Concept** of the antipattern. Such a situation may be generalised to a wider range of cases as follows:

> companies limit communication and sharing with developers outside the company as well as the commit rights of such external developers.

In this enlarged context, we can identify a new, more general antipattern, which incorporate the “Big Show” antipattern as an extreme case and can be identified by considering several measures that characterise contributors and contributions such as:

- the ratio between internal and external developers;
- the number of external reviewers contributing to the project;

Name Shyness

Central Concept Describes the scenario in which companies limit communication and sharing with developers outside the company as well as the commit rights of such external developers.

Dysfunction ...

...

Refactoring ...

Identification Please calculate the following.

- $R =$ the ratio between internal and external developers;
- $ER =$ the number of external reviewers contributing to the project;
- $EC =$ the number of external contributors;
- ...

Then calculate $I = f(R, ER, EC, \ldots)$

Figure 6: Shyness Antipattern.

- the number of external contributors.

Such an antipattern can be called "Shyness" antipattern to highlight that developers from the company appear "shy" in communication and sharing with the external community. A partial description of such "Shyness" antipattern is given in Figure 6. The key differences with respect to the "Big Show" antipattern illustrated in Figure 2 are the **Central Concept**, which is more general and less extreme, and the **Identification**, which consists of a precise measuring mechanisms rather than a mere list of questions. The final calculation of the identification measure I is given by a function f, whose value increases when argument R increases and decreases when either arguments ER or argument EC increases.

Once the antipattern is characterised by an identification measure I with respect to the FLOSS project under certification process, we can define the weight W of the antipattern in the certification process as directly proportional to I and to a measure S of the severity of that antipattern and inversely proportional to the effectiveness E of the refactoring solution provided by the antipattern. In fact, a very effective refactoring solution represents a roadmap towards an improvement of the quality of the development process, thus decreasing the negative impact of the antipattern on software quality. Finally, it is important to notice that severity S is related to the context of the FLOSS project, including the structure and composition of the FLOSS community and the project aims and application domain.

We summarise our approach to the use of antipatterns within the FLOSS certification process in Figure 7: the questions-based identification of the template illustrated in in Figure 1 is turned into a quantitative identification based on precise measuring mechanisms, which operate by applying appropriate quality metrics to quantitative data that can be collected by data mining

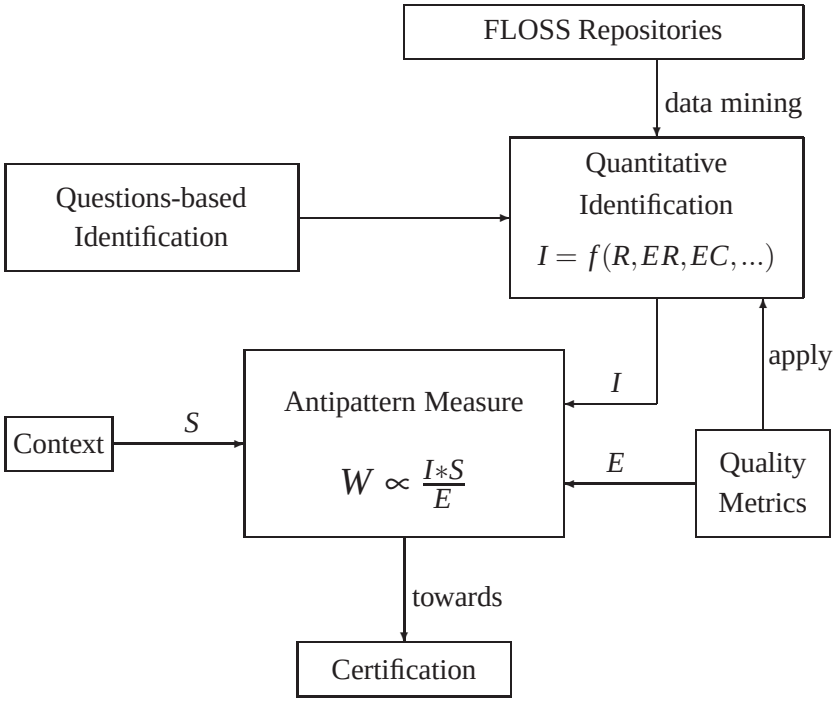

Figure 7: Quantitative Identification Process.

FLOSS repositories; the quantitative identification process provides an identification measure of the antipattern that together with the severity of the antipattern and the effectiveness of the antipattern refactoring solution result in the antipattern measure to be used in the certification process.

7 Conclusion and Future Work

We have presented the potential benefits of antipatterns usage within the FLOSS development process to allow various types of users to overcome problems that may affect the certification of the FLOSS product. The enrichment of the OWL antipattern ontology with data from 13 FLOSS community antipatterns provides a way of realising these potential benefits. There are clear advantages to the certification of FLOSS software that come from improving the quality of the developed software both at a development level and by overcoming FLOSS community problems. The strong social aspect of the collaborative development of the antipattern ontology will ultimately increase the communication among the different types of FLOSS contributors and will provide a platform in which FLOSS users can discuss their problems and possible solutions using antipatterns.

We have also suggested how to relate antipatterns, especially community level antipatterns, and their applied solutions to FLOSS quality by development as part of a FLOSS certification process. Development within a FLOSS projects is not dictated by prescriptive rules, but natu-

rally emerges as the product of community activities. FLOSS community antipatterns describe therefore dysfunctions that negatively affect the quality of the FLOSS product [Cer12].

Finally, we have proposed a framework to identify the antipatterns that may affect the quality of the FLOSS product and to provide a measure that quantifies the negative effect of such antipatterns on quality. Although we have proposed proportionality relationships between the negative weight of an antipattern for the quality by development and both the severity of the antipattern and the effectiveness of the applied solution, it is still an open problem how to fully characterise severity and effectiveness in order to completely define the antipattern measure. The use of such an antipattern measure in the definition of a metric for quality by development is part of our future work and would be essential in the creation of a FLOSS certification process.

Acknowledgements: This work has been supported by Macao Science and Technology Development Fund, File No. 019/2011/A1, in the context of the PPAeL project.

Bibliography

[Ber11] J. Berkus. How to destroy your community. 2011. http://lwn.net/Articles/370157/

[BMMM98] W. Brown, R. Malveau, H. McCormick, T. Mowbray. *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*. Wiley Computer publishing, 1998.

[Cer12] A. Cerone. Learning and Activity Patterns in OSS Communities and their Impact on Software Quality. In *Proceedings of OpenCert 2011*. Volume 48 of Electronic Communications of the EASST. 2012.

[CV08] J. P. Correia, J. Visser. Certification of Technical Quality of Software Products. In *Proceedings of the OpenCert and FLOSS-FM 2008 joint Workshop*. UNU-IIST Research Report 398, pp. 35–51. 2008.

[Leu08] T. Leung. OReilly OSCON Open Source Convention. 2008.

[LN06] P. Laplante, P. Neil. *Antipatterns: Identification, Refactoring and Management*. Taylor and Francis, 2006.

[Mala] N. Malik. Software Project Management Antipattern Blog, Pardon my dust. http://blogs.msdn.com/nickmalik/archive/2006/01/19/PMAntipattern-Pardon-My-Dust.aspx

[Malb] N. Malik. Software Project Management Antipattern Blog, Project Managers who write specs. http://blogs.msdn.com/nickmalik/archive/2006/01/03/508964.aspx

[Nea11] D. Neary. OSS community management Wiki. 2011. http://communitymgt.wikia.com/wiki/Category:Anti-patterns

[O'B] T. O'Brien. 6 Open Source Community Anti-patterns (or Less Talk. More Do.). http://www.discursive.com/2010/12/02

[Saa] M. Saastamoinen. Managing OSS As an Integrated Part of Business (OSSI), The linux foundation. https://fossbazaar.org/content/managing-oss-integrated-part-business-ossi-final-report

[SC09] S. A. Shaikh, A. Cerone. Towards a metrics for Open Source Software Quality. In *Proceedings of OpenCert 2009*. Volume 20 of Electronic Communications of the EASST. 2009.

[SC10] S. K. Sowe, A. Cerone. Integrating Data from Multiple Repositories to Analyze Patterns of Contribution in FOSS Projects. In *Proceedings of OpenCert 2010*. Volume 33 of Electronic Communications of the EASST. 2010.

[SCF12] D. Settas, A. Cerone, S. Fenz. Enhancing ontology-based antipattern detection using Bayesian networks. *Expert Systems with Applications*, 2012. doi:10.1016/j.eswa.2012.02.049. http://www.sciencedirect.com/science/article/pii/S095741741200293X

[SMSB11a] D. L. Settas, G. Meditskos, I. G. Stamelos, N. Bassiliades. Detecting antipatterns using a Web-based collaborative antipattern ontology knowledge base. In *Proc. of ONTOSE 2011*. Volume 83, pp. 478–488. Springer, 2011.

[SMSB11b] D. L. Settas, G. Meditskos, I. G. Stamelos, N. Bassiliades. SPARSE: A Symptom-based Antipattern Retrieval Knowledge-based System Using Semantic Web Technologies. *Expert Systems with Applications, Elsevier* 38(6):7633–7646, 2011.

[Sou] Sourceforge. SPARSE – Antipattern detection system. http://sourceforge.net/projects/sparse-antipatt/?source=directory

[WCa] Wiki-Community. Pattern Community Antipattern Catalogue. http://c2.com/cgi/wiki?AntiPatternsCatalog

[WCb] Wiki-Community. Wikipedia Antipatterns Community Catalogue. http://en.wikipedia.org/wiki/Anti-pattern