Modularity Analysis of Use Case Implementations

Fernanda d'Amorim   Paulo Borba

Informatics Center, Federal University of Pernambuco, Recife, Brazil

{frsa, phmb}@cin.ufpe.br

1. Introduction

Use cases is a very popular technique for requirements specification. In use case driven development one can isolate a use case abstraction in a single modular unit in all development phases except implementation [5]. Traditional modularity techniques decompose use case implementations, that is, how use cases are realized into code, across components; as such they become scattered and tangled over the codebase. That happens because a use case implementation is fundamentally a crosscutting concern.

It is well known that crosscutting concerns reduce modularity, making programs harder to understand, develop in parallel, maintain, and evolve [2, 6]. Modularity techniques have been recently proposed to address this problem. For example, Aspect-Oriented Programming (AOP) [6] and Virtual Classes [8]. They promote separation of crosscutting concerns, and can be used to create modules with code related to a single use case implementation [5, 4].

This paper analyzes the impact of use case modularization on some traditional and recent internal quality metrics, including cohesion, coupling and separation of concerns. In this study, we apply AOP to isolate the use case implementations of a benchmarking system for AOP studies: Health Watcher [9]. It is well known that AOP is effective to modularize conventional crosscutting concerns, such as, persistence, logging and transaction. But to the best of our knowledge it is not yet clear how AOP behaves for modularizing use case implementations and yet improves design stability, changeability, parallel development, traceability and pluggability. To that end, we compare and evaluate use cases implemented as aspects (with AspectJ) and use cases implemented in the traditional object oriented way (with Java).

This study attempts to answer the following questions:

Q1. Does modularity of use case implementations reduce modularity of non use case concerns?

Q2. Does modularity of use case implementations improve overall maintenance and evolution?

Q3. Which modularity dimensions benefit from modularized use case implementations?

2. Study Settings

We conducted a case study to evaluate research questions Q1-Q3. We describe next the subject used, the experimental setup, the change requests considered, and the metrics analyzed.

Target System Selection. We choose the Health Watcher (HW) system as the subject of our study because of the following: (i) Its design has a significant number of crosscutting and non-crosscutting concerns. (ii) A use case document is available, which is essential in the definition of the concerns used to guide the extraction of concern based metrics; the selected concerns should cover most of the code, this way, we reduce bias in the analysis process [2]. (iii) Qualitative and quantitative studies of the HW system have been conducted [3, 7] which allow us to correlate our results with them.

Setup. Our evaluation uses 5 releases of the HW system. The initial release is derived from the public AspectJ implementation of the HW. The others 4 are characterized by one change request expressing a maintenance task explained below. For each release, we have 2 versions: (i) use cases implemented with AspectJ (all methods, fields and extra behavior needed to implement the functionality of a use case is grouped in an aspect) and (ii) use cases implemented with Java (all code related to a use case is scattered over the components of the system).
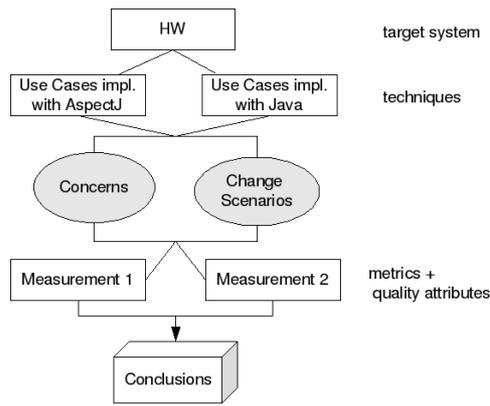
Change Requests. We evolve both versions of the initial release according to the following change requests:

CR1. Addition of 6 new functional use cases.

CR2. Modification in 2 use cases - *change on data manipulation and change on functional behavior*.

CR3. Change of the persistence mechanism.

CR4. Removal of 1 use case.

Figure 1. Analysis process

Quality Attributes and Metrics After each release, we collect a set of metrics and compare both versions (use cases implemented with AspectJ and use cases implemented with Java) to evaluate the impact of change on modularity.

Also, we link internal and external quality attributes with these metrics as follows:

- Design Stability - Coupling, Cohesion, Depth of inheritance tree, Lines of code (LOC), Weighted operations over methods and Vocabulary size. Separation of Concerns Metrics [2]: Degree of scattering (DOS) over components, Degree of tangling (DOT) over components, Average DOS, Average DOT.

- Changeability - Changed components, Added components, Added Lines of Code, Changed Lines of Code.

- Pluggability - Changed components, Blocks with dead LOC, Total dead LOC

- Parallel Development - Core influence (Basecode LOC/ Total LOC), Adjustments in the basecode.

- Traceability - Changed components, Changed files.

Previous works [3, 7] guided the selection of most metrics adopted in this study. Also, the Goal Question Metrics (GQM) approach [1] was used to choose the set of metrics and the correspondent quality attribute addressed by it.

3. Results and Discussion

Figure 1 shows the dependence relation among factors that influence the analysis process. As depicted, we can see that modularity analysis depends on others factors beyond the chosen system, metrics and the applied technique. In this specific case, the selected concerns and the change scenarios have strong impact on the study results.

The main outcomes of our initial analysis are the following:

(1) An use case implementation is a concern that will, likely, be related with others concerns - *container concern*. In this study, they are linked with persistence and distribu-

tion. So, isolating use case implementations in the HW system degraded the modularization of the persistence and distribution concerns. There is evidence that isolating container concerns can affect negatively the modularization of concerns that have a dependence relation with them, producing higher degree of scattering. For this reason, we just have a fair evaluation on modularity gain if we analyze the results considering a set of concerns that covers most of the code, as pointed out in the previous section, and not just a specific group of them.

(2) Impact on maintenance and evolution tasks has strong relation with types of change. For example, if a system is under constantly addition of new functional requirements (perfective change), isolating use case implementations brings significant gain on modularity. Although, considering changes in the application environment (adaptive change), modularizing use cases can scatter code of others concerns (e.g., persistence) often leading to a more invasive process of change.

(3) As might be expected, modularized use case implementations lead to a system where is easier to plug/unplug a specific use case and where is easier to track use case code creating direct links to requirement specification.

From these observations, we conclude that it is hard (or even impossible) to find a design (using language constructs that physically separate concerns) where all considered concerns are modularized and do not produce negative side effects when exposed to maintenance and evolution tasks. Depending on a combination of factors, such as, the set of concerns under evaluation, the types of change, the application architecture and, even, the developers goals, one design (decomposition) will be chosen over others.

References

[1] V. Basili, G. Caldiera, and H. Rombach. The goal question metric approach. *Encyclopedia of Software Engineering*, 1:528–532, 1994.

[2] M. Eaddy et al. Do crosscutting concerns cause defects? *IEEE Transactions on Software Engineering*, pages 497–515, 2008.

[3] P. Greenwood et al. On the impact of aspectual decompositions on design stability: An empirical study. *Lecture Notes in Computer Science*, page 176, 2007.

[4] S. Herrmann et al. Mapping use case level aspects to object teams/java. In *OOPSLA Workshop on Early Aspects*, 2004.

[5] I. Jacobson. Use cases and aspects - working seamlessly together. *Journal of Object Technology*, 2(4):7–28, 2003.

[6] G. Kiczales et al. Aspect-oriented programming. In *ECOOP '97*, 1997.

[7] U. Kulesza et al. Quantifying the effects of aspect-oriented programming: A maintenance study. In *ICSM'06*, 2006.

[8] M. Mezini et al. Conquering aspects with caesar. In *AOSD '03*, pages 90–99, 2003.

[9] S. Soares et al. Implementing distribution and persistence aspects with aspectj. In *OOPSLA '02*, 2002.