Modularity Analysis of Use Case Implementations

Fernanda d'Amorim and Paulo Borba
Informatics Center, Federal University of Pernambuco
Recife, Brazil
Email: {frsa, phmb}@cin.ufpe.br

***Abstract*—Component-based decomposition can result in implementations with use cases code tangled and scattered across components. Modularity techniques such as aspects, mixins, and virtual classes have been recently proposed to address this problem. One can use such techniques to group together code related to a single use case. This paper analyzes qualitatively and quantitatively the impact of this kind of use case modularization. We apply one specific technique, Aspect Oriented Programming, to modularize the use case implementations of the Health Watcher system. We extract traditional and contemporary metrics, including cohesion, coupling and separation of concerns and analyze modularity in terms of quality attributes such as changeability, support for parallel development, and pluggability. Our findings indicate that the results of modularity analysis depends on other factors beyond the chosen system, metrics and the applied technique.**

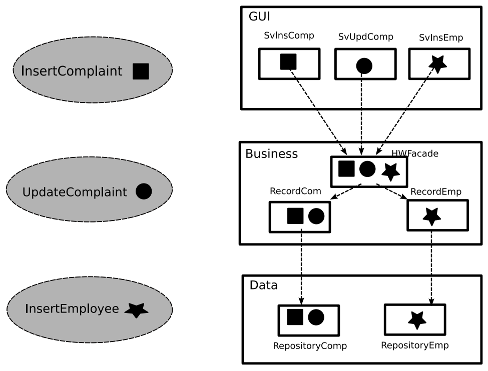


Fig. 1. Tangling and scattering when realizing use cases.

I. Introduction

Component-based decomposition [1] is one of the most used modularity technique to develop complex software systems. It relies on a process where the main goal is the assignment of requirements to components. Traditional modularity techniques, like Object Orientation, are applied to allow the system's decomposition into well-defined components. Components are crucial to design, implement, test and to understand the system. They encapsulate requirements that are more likely to change together and they can keep some concerns separate. Ideally, we want to be able to isolate all kinds of concerns into separate components in order to develop and maintain each one independently. However, we find that although some concerns can be realized by distinct components, in general, we have many concerns that impact multiple components. They are known as crosscutting concerns.

There are different kinds of crosscutting concerns. Logging, persistence, and distribution are classical examples of such concerns. They are typically used to address nonfunctional requirements. However, a crosscutting concern can deal with functional requirements as well. We find that the implementation of functional requirements, which can be specified as use cases, is fundamentally a crosscutting concern. That happens because traditional modularity techniques [2], [3], [4] decompose use case implementations across components. This is illustrated in Fig. 1

Fig. 1 shows use cases on the left side. On the right side, we have components distributed throughout the system layers. Each shape (square,circle and star) inside a component represents the code that implement the respective use case. Notice the limitation of components to keep separate use case implementations; as such they become scattered and tangled over the codebase. Tangling and scattering are the two main causes of modularity loss resulted by the implementation of crosscutting concerns.

Tangling happens when each component contains the implementation (code) to satisfy distinct use cases. For example, in Fig. 1, we see that the HWFacade component is involved with the realization of three different use cases. The component instead of being dedicated to a particular use case implementation, participates in many. This can make the code hard to understand as the developer has to be aware of code that is not specific to the use case he is dealing with.

Scattering happens when code that realize a particular use case is spread across multiple components. For example, in Fig. 1, we see that the realization of UpdateComplaint enforces additional code on four components. So, anytime the requirements associated with that use case change, we must update many components.

It is well known that crosscutting concerns reduce modularity, making programs harder to understand, develop in parallel, maintain, and evolve [5], [6]. All these problems are also present in the context of use case implementations.

In a use case driven development use cases guide the whole development process. Use cases start at requirements, are translated to realizations during analyzes and design and to test cases in testing. *Each* use case in the use case model is translated to *one* use case realization in the design model. But, during implementation, one use case realization can not be mapped to just one component. At this point, a use case unit cannot be seen any more. The 1:1 mapping from use cases to assets we had in the previous phases of the development process is over. Use case implementations impose a N:M relation between use cases code and components causing traceability loss. For instance, it is not easy to uncover requirements just looking at the code of each component or a set of components. This impacts understandability hindering the learning curve for developers. Poor understandability leads to poor maintainability

In a typical software environment, changes are requested even while the software is still being developed. Thus, a simple addition or modification on a use case specification leads to a very invasive process of change, where a developer has to deal with many source files, classes, methods, etc. Such situation increases maintenance and evolution costs.

Pluggability is another important issue to consider. In this situation, where a use case implementation cut across multiple components, it is not easy to turn on/off a specific use case in the system. Being able to plug or unplug a use case with minimum effort is an essential capability for some particular contexts, for example, in a software product line [7], [8].

So, the main goal of our work is to evaluate if *modularity of use case implementations improve the overall system's modularity*. With this purpose, we (i) analyze how uses cases are structured at the code level; (ii) compare different forms of use case implementations; (iii) and, finally, point out which form of use case decomposition is more suitable to accomplish improvement on the overall system's modularity.

In addition, we answer the following research 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?

It is well known that AOP is effective to modularize 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. We find some works [9], [10], [11] showing how AOP can be used to enable such modularization; they also report some benefits, such as, improvement on parallel development, traceability, and changeability. However, their observations are not based on empirical evaluations. We were not able to uncover empirical evidence of such effects. In fact, a quantitative evaluation is missing in order to provide a fair analysis regarding the real modularity gain when we modularize use case implementations.

To this end, we run a case study where we analyze the impact of use case modularization on some traditional and recent internal quality metrics, including cohesion, coupling and separation of concerns. We apply AOP to isolate the use case implementations of a benchmark system for AOP studies: Health Watcher [12]. We compare and evaluate use cases implemented as aspects (with AspectJ) and use cases implemented in the traditional object oriented way (with Java).

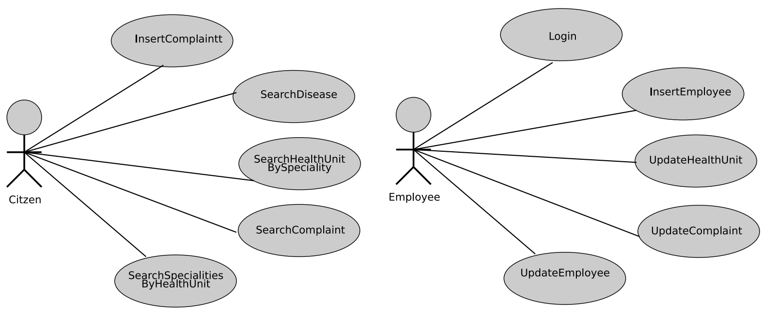


Fig. 2. Use Case Model for the Original HW Implementation.

II. CASE STUDY

We conducted a case study to evaluate research questions Q1-Q3. The study is based on a typical evolution and maintenance environment. From a base implementation, we evolve the target system through a set of change requests. We select changes from different natures (type and scope) that are commonly applied to a system under maintenance and evolution. Regarding their scope, we classify changes as use case scoped and system scoped. Use case scoped are changes that affect directly use case specifications: adding, removing and modifying a particular functionality on the system. This type of change aims to evaluate how use case implementations behave when changes are derived directly from them. While system scoped changes deal with modifications of nonfunctional features affecting various use case implementations at the same time. Regarding their types, we classify changes as corrective, perfective, and adaptive as in [13]. On selecting diverse kinds of changes, we can analyze the effects of modularizing use case implementations from different points of view, assuring that the results cover a wide range of possibilities and are not just built upon changes that can favor or degrade a specific concern in our evaluation.

A. Target System

The HealthWatcher (HW) [12] is a real web-based information system with available OO and AO implementations. The main goal of the HW system is to improve the quality of services provided by public health care institutions. For this purpose, citizens use the HW system to register complaints regarding health issues, and health care institutions use the HW system to investigate the reported complaints and take the required actions. The HW original implementation is composed of ten use cases. Fig. 2 shows its use case model.

B. System Selection

We choose the Health Watcher (HW) system as the subject of this study because of the following:
1. Its design has a significant number of crosscutting and non-crosscutting concerns, such as, persistence, concurrency, and distribution.

2. A use case document is available, which is essential in the definition of the concerns used to guide the extraction of concern based metrics; As stated in [5], the selected concerns should represent the logic of implementation and cover most of the code. For example, *persistence* should not be selected as a concern if the developer did not consider it. This reduces sample bias since all concerns are considered, not just those that are crosscutting and in the focus of analysis [5].

3. Qualitative and quantitative studies of the HW system have been conducted [14], [15] which allows us to compare our results with them.

C. System Design

The AO implementation of the HW system is developed with AspectJ. Its architecture is based on the Layer pattern having the system classes structured in three main layers: View, Business and Data. The crosscutting concerns distribution, persistence, and concurrency are implemented as aspects.

Fig. 3 shows a simplified class diagram of the AO original implementation, highlighting the system's architecture through its main components. Note that use case implementations cut across components defined in the hierarchy of layers.

The View layer is related to the HW web user interface. The Java Servlet API is used to implement this layer. The Business layer is responsible for implementing the classes that define the system business rules. The Data layer is responsible for abstracting the functionality of database persistence using the JDBC API. Components in the Model package are responsible for implementing the domain objects. These objects represent the core concepts of the application; they transit among all architectural layers and have simple implementation logic. Complaint and Employee are examples of core concepts in the HW system. The Distribution aspects distributes the system services provided by the Business layer; it is implemented using the RMI technology. The Persistence aspects modularize transaction control and connection pool. The Concurrency aspects deal with code that implements concurrency control policy. The abovementioned design decisions have shown to be effective to modularize most of the nonfunctional system concerns: distribution, persistence and concurrency [12]. However, code relative to use case implementations is still tangled and scattered throughout the system's components.

D. Setup

Fig. 4 illustrates the steps used to execute the case study. It is organized in 3 phases. In the first phase, we apply the use case modularization technique to generate the base implementations of the study: (imp1) use cases implemented following the Use Case as Aspect (uc_asp) technique where all methods, fields and extra behavior needed to implement the functionality of a use case are grouped in an aspect, constituting a use case module; and (imp2) Use cases OO (uc_oo), where use cases are implemented in the traditional object oriented way with Java (all code related to a use case is scattered over the components of the system). In the second

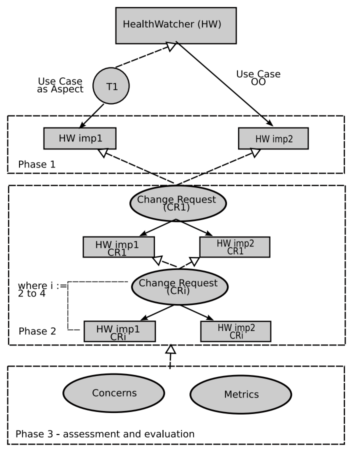


Fig. 4. HealthWatcher experiment setup.

phase, we evolve each implementation based on a set of change requests (described below) producing four releases. Each change request is applied at the most recent release, for example, CR2 is applied in the release generated after the implementation of CR1. In the third phase, we assess the peer releases applying metrics to compare and evaluate the overall system's modularity on accomplishing the required changes. Prior to the assessment, we define the set of chosen concerns that guides the application of concern-based metrics.

E. Change Requests (CR)

We evolve both implementations of the base release according to the following change requests:

- **CR1** - Addition of 6 new functional use cases (InsertMedicalSpecialty, UpdateMedicalSpecialty, InsertHealthUnit, InsertSymptom, UpdateSymptom, and InsertDiseaseType). The functionalities introduced by these new use cases represent typical operations encountered in the maintenance of information systems. They naturally involve modification of modules implementing several system concerns. The new use cases require changes in classes of all the system's layers. This is a use case scoped and perfective type of change.
- **CR2** - Modification in 2 use cases. In this request we contemplate different kinds of changes: change on data manipulation and change on functional behavior.
  In the first situation, the InsertEmployee use case is changed to add a new field on the Employee record. The original Employee data includes name, login and password. Now, we want to store the date on which the employee is registered in the HW system.
  In the second, the UpdateMedicalSpecialty use case is modified to allow a Medical Specialty to be updated by directly entering its code. In the original implementation, to update a medical specialty we first list all medical specialties registered in the system; then, from this list we select the one we want to update. Now, we have the option to update a medical specialty using its code instead

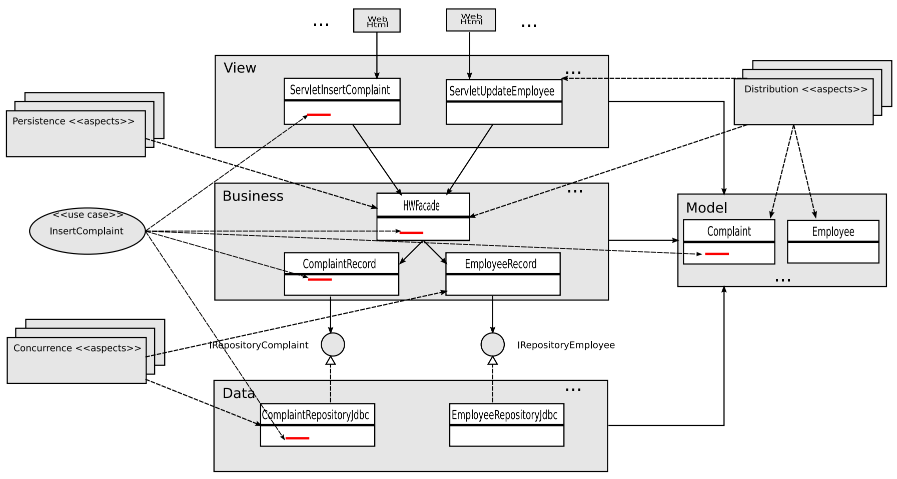


Fig. 3. HW AO original architecture design.

of selecting it from a list. These are use case scoped and perfective type of change.

- **CR3** - Removal of 1 use case. This modification removes the SearchMedicalSpecialty use case. It excludes all specific files and comment all lines of code related to the use case. This change aims at providing a specific evaluation on how easy is to plug/unplug a functional use case in the system. This is a use case scoped and perfective type of change.
- **CR4** - Change of the persistence mechanism. This modification alter the persistent mechanism to store data on a XML file instead of a database (JDBC). This type of change aims at evaluating how an adaptation of a non-functional concern affects the functional use cases in the system. This is a system scoped and adaptive type of change.

F. Assessment Procedure

The main goal of our evaluation process is to answer how a system behaves regarding modularity improvement when modularized by use case implementations. To this end, we analyze the impact on modularity through a set of software quality attributes that are affected directly by the system's modular implementation: complexity of source code, separation of concerns, changeability, support for parallel development, traceability, and pluggability.

For each attribute, we define a set of metrics that enables its quantitative evaluation. Based on that, we setup a Goal-Question-Metric (GQM) model [16] to drive the selection process. Using such approach, we can choose meaningful metrics correlating them with quality factors. Table I shows the GQM model where we can see all selected metrics used in the evaluation of the respective software quality attribute. Most metrics we choose have already been successfully used in several case studies [14], [17], [18], [19], [15]. Some of them are refinements of classical metrics used in the OO context [20] adapted to be used in the AO context [21]. Others are newly defined, as the ones for evaluating separation of concerns [19], [22].

TABLE I
GQM TO ASSESS MODULARITY IMPROVEMENT.

Goal: Increase system's modularity	
Purpose	Increase
Issue	Modularity of use case implementations
Object	System
Viewpoint	Software Engineer
Questions and Metrics	
Q1 - What is the potential crosscutting level of use case implementations?	
M1.1 - CDC	Concern diffusion over components
M1.2 - DOSC	Degree of scattering over classes
M1.3 - ADOSC	Average degree of scattering
M1.4 - DOT	Degree of tangling
M1.5 - ADOT	Average degree of tangling
Q2 - How modularized use case implementations impact the complexity of source code?	
M2.1 - CBC	Coupling between components
M2.2 - LCO	Lack of cohesion over operations
M2.3 - VS	Vocabulary size
M2.4 - DIT	Depth of inheritance tree
M2.5 - LOC	Lines of Code
Q3 - What is the effort to modify use case implementations? (Changeability issue)	
M3.1 - NAC	Number of added components
M3.2 - NCC	Number of changed components
M3.3 - NALOC	Number of added lines of code (LOC)
Q4 - What is the effort to trace changes in use case specifications from requirements to implementation? (Traceability issue)	
M1.1 - CDC	Concern diffusion over components
M4.1 - NSFC	Number of source files related to the change
Q5 - What is the the effort to plug/unplug a use case in the system? (Pluggability issue)	
M3.2 - NCC	Number of changed components
M5.1 - NBDLOC	Number of blocks with dead LOC
M5.1 - NDLOC	Number of dead LOC
Q6 - What is the impact on parallel development caused by modularizing use case implementations?	
M6.1 - CI	Core Influence - System's core LOC / Total LOC use cases
M6.2 - NAC	Number of adjustments in the Core

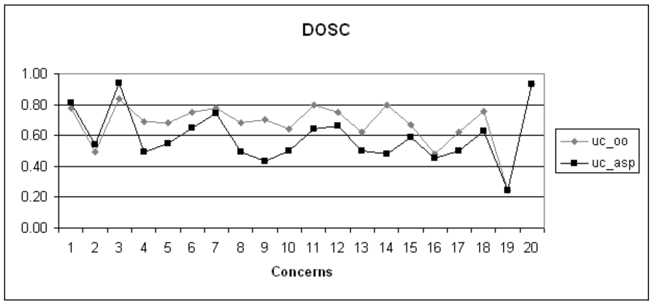
We use the *AOP metrics* tool [23] to collect some of the metrics. We select and assign concerns to code by hand, however, we follow the guideline proposed by Eaady et al [22] in order to make these tasks more systematic.

G. Results

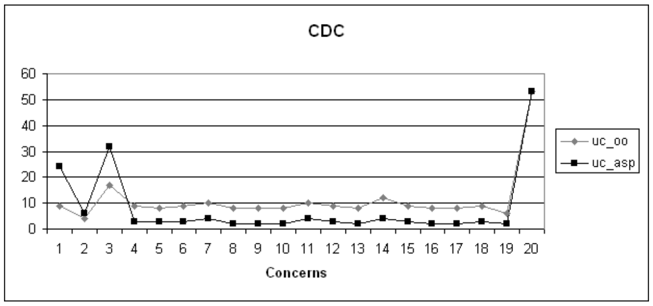
In this section, we evaluate and analyze the results obtained after the assessment phase. We organize our analysis based on

TABLE II
HEATHWATCHER AO IMLEMENTATION - SEPARATION OF CONCERNS

	uc_oo			us_asp		
Concern Name	**dosc**	**cdc**	**loc**	**dosc**	**cdc**	**loc**
1 - Distribution	0.78	9	347	0.81	24	370
2 - Concurrency	0.49	4	187	0.54	6	187
3 - Persistence	0.84	17	2144	0.94	32	1815
4 - SearchDisease	0.69	9	239	0.49	3	270
5 - UpdateHealthUnit	0.68	8	263	0.55	3	185
6 - SearchSpecialitiesByHealthUnit	0.75	9	129	0.65	3	137
7 - UpdateMedicalSpeciality	0.78	10	201	0.74	4	216
8 - InsertDiseaseType	0.68	8	112	0.49	2	127
9 - InsertNewEmployee	0.70	8	100	0.43	2	105
10 - InsertNewSymptom	0.64	8	96	0.50	2	113
11 - UpdateSymptom	0.80	10	291	0.64	4	329
12 - UpdateEmployee	0.75	9	93	0.66	3	77
13 - InsertNewHealthUnit	0.62	8	92	0.50	2	105
14 - InsertNewComplaint	0.80	12	417	0.48	4	555
15 - UpdateComplaint	0.67	9	255	0.59	3	273
16 - SearchComplaint	0.48	8	178	0.45	2	189
17 - InsertNewMedicalSpeciality	0.62	8	93	0.50	2	106
18 - SearchHealthUnitBySpecialities	0.76	9	138	0.63	3	155
19 - Login	0.24	6	70	0.24	2	71
20 - Core	0.93	53	1919	0.93	53	1919
Average DOSC	**0.709**			**0.602**		



(a) Degree of Scatering over classes



(b) Concern Diffusion over classes

Fig. 5. Separation of Concerns

the quality software attributes that are under evaluation: separation of concerns, complexity of source code, changeability, traceability, pluggability and support for parallel development

1) Quantifying Separation of Concerns (SoC): Table II exhibits the absolute values collected after CR2. It shows, for each implementation, the degree of scattering over classes (DOSC), the concern diffusion over components (CDC), and the lines of code (LOC) metrics for all use case implementations in the system, and also for others equally important concerns.

As mentioned before, we chose the concerns following the guideline to obtain a set of concerns that represents a logic of the system implementation and that covers most of the code. Thus, for the HW case study, the selected concerns comprehends all functional use cases in the system, the crosscutting concerns: persistence, distribution and concurrency (that are already modularized in the HW base implementation of the study), and the system's core (elements that are shared among all the use case implementations).

Since the main goal of the uc_asp solution is to provide the modularization of use case implementations, we could expect superior SoC outcomes in favor of the uc_asp implementation. In fact, the curves in Fig. 5 confirm this impression where the uc_asp implementation exhibits better results with respect to all use cases (concerns 4-19) investigated. However, when analyzing the distribution (1), concurrency (2), and persistence (3) concerns we see that their modularization is degraded. It happens because these concerns are directly related to use cases, for instance, all use cases contain persistence and distribution code in their implementation; and two use cases contain concurrency code. When modularizing use case implementations, we extract the code related to persistence, distribution, and concurrency to its own use case aspect, thus, scattering these concerns over all use case modules.

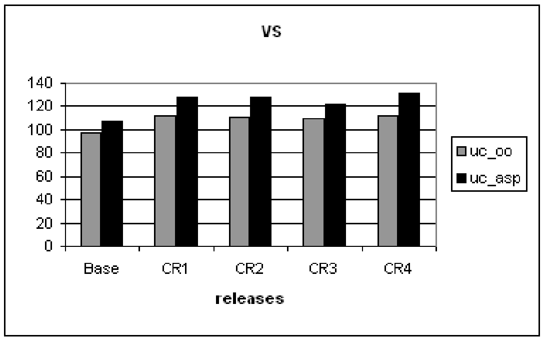
Regarding the Average Degree of Scattering (ADOSC) metric, we see in the last line of Table II that the ADOSC value is slightly better in the uc_asp implementation. One could expect a much better advantage in favor of the uc_asp implementation, since a big set of concerns are modularized. However, the small gain is due to the fact that while improving functional use cases implementation, we affect negatively the modularity of all concerns that have a relation with them; in this case, Persistence and Distribution.

Now, let's look at the LOC values associated with each concern on Table II. The LOC values are very similar for both implementations. We have almost no gain with quantification while modularizing use cases implementation due to its heterogeneous nature. This effect contributes to maintain the LOC values almost the same.

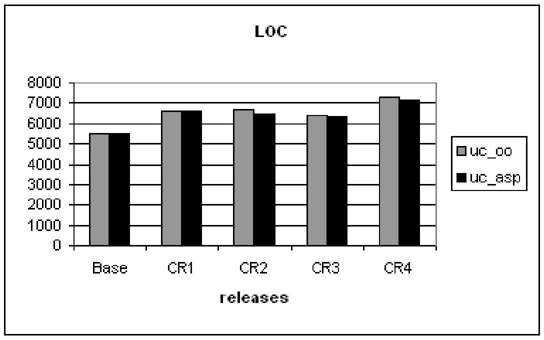
Table III shows the Average Degree of Tangling (ADOT) metric and the consolidated data on the number of dedicated components in the system. Lower ADOT value indicates a system with less tangled code, which implies on better modularity. The results show that the uc_asp implementation has a significant advantage regarding code tangling. This occurs mainly because we extract all code related to use case implementations making components, like `HWFacade, RecordXX, RepositoryXX`, that have high DOT values in the uc_oo implementation, dedicated to the Core concern in the uc_asp implementation.

TABLE III
DEGREE OF TANGLING

	# Components	**Tangled**	**Not Tangled**	**% Not Tangled**	**ADOT**
uc_oo	107	78	29	72%	0.158
uc_asp	124	15	109	88%	0.064

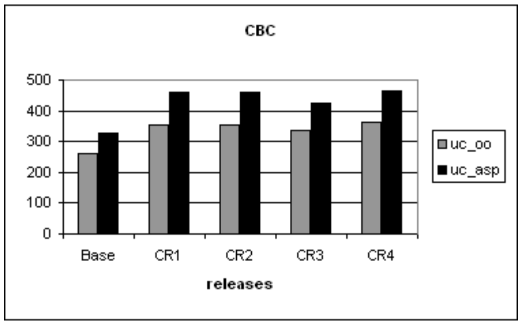


(a) Vocabulary Size

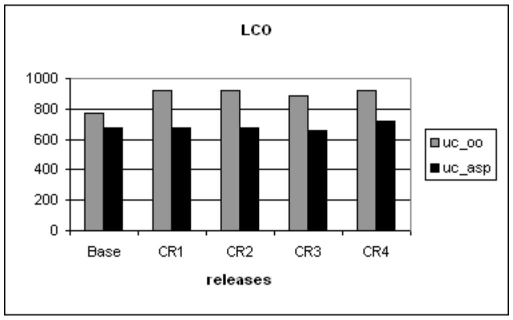


(b) Lines of Code

Fig. 6. Vocabulary Size and Lines of Code



(a) Coupling



(b) Cohesion

Fig. 7. Coupling and Cohesion

After analyzing all SoC metrics we find that they are very important as a modularity indicator, but also extremely dependent of the set of chosen concerns. For example, if we have just considered the set of functional use cases we would state that we had an absolute modularity gain. However, while analyzing all important concerns, regarding the HW implementation, we obtain a relative improvement on the overall system's modularity.

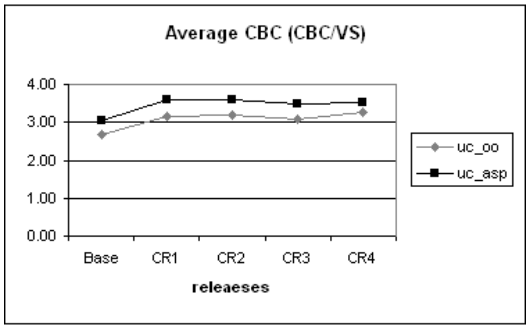
2) Quantifying Complexity of Source Code: In this section, we discuss how modularized use case implementations impact positively or negatively on the complexity of source code by quantifying its effects on coupling, cohesion, and size metrics. All measures were gathered according to the system perspective; that is, they represent the total of metric values associated with all the classes and aspects for the system implementation. Fig 6 shows the absolute values for Vocabulary Size (VS - i.e. number of components) and Lines of Code (LOC) throughout the releases.

The VS graphic shows that the uc_asp implementation tend to be worse in all releases. This happens because of the introduction of aspects that are used to modularize the functional use cases. For each use case in the system, we need to define, at least, one new aspect containing the specific code related to its implementation.

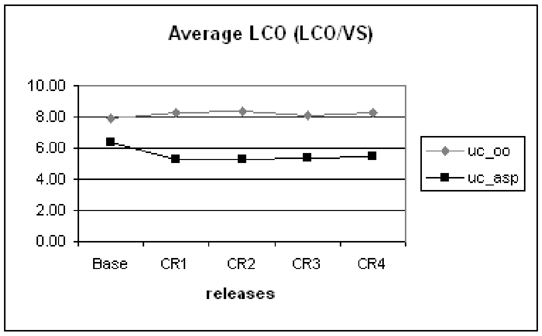
Analyzing the Lines of Code (LOC) values we have an interesting outcome; previous works [12], [15] observed that although defining more components, AO solutions tend to have less LOC. However, in the context of use case implementations we do not see this effect. This happens because of the heterogeneous nature of this specific crosscutting concern. Heterogeneous crosscutting concerns imply almost on no gain with quantification, which implies no reduction on LOC values. In our case, we find similar values for both implementations throughout all releases.

Now, let's analyse Coupling Between Components (CBC) and Lack of Cohesion over Operations (LCO) metrics. Fig. 7(a) shows that there is a considerable difference in favor of the uc_oo implementation with respect to the absolute value of the coupling metric (CBC). It happens mainly because, although many of the aspects reduce the coupling of system classes by modularizing their crosscutting concerns, they still reference the classes in which they introduce some state or behavior. Due to the class hierarchy of the HW system, a use case implementation has methods spread over, at least, 6 components; therefore, a use case aspect needs to introduce these methods in their respective classes, contributing to a significant increase on the coupling metric. This effect is quite similar to the one produced when we break a big class into two new classes (main and auxiliary); this action causes an increase on coupling, however, we improve the component design.

Fig. 7(b) shows the superiority of the uc_asp implementation with respect to the cohesion metric (LCO). The uc_asp implementation is, on average, 27% superior in the absolute value and also 45% superior in the average values per component (LCO/VS). The production of components with higher cohesion was an expected effect in the uc_asp implementation



(a) Coupling



(b) Cohesion

Fig. 8. Average Coupling and Cohesion

of use case implementations; it can be explained by the fact that when we modularize use case code into an aspect, we separate methods that access different fields (mainly because they address different use cases code) to distinct aspects. This action contributes to increase the cohesion of the uc_asp implementation, i.e. decrease the value of LCO.

Fig. 8 displays the graphics for the average cohesion and coupling per component. Observing the curves, uc_asp implementation has better result for cohesion while the uc_oo implementation presents superior outcome for coupling. Despite having more components (classes and aspects), demonstrated by the VS metric, we can observe that the uc_asp implementation has produced, on average (CBC/VS), more coupled classes and aspects.

Fig. 9 shows that the Depth of Inheritance Tree (DIT) metric has similar results for both implementations and present also uniform behavior among all releases.

Summarizing our findings regarding complexity of source code, we see that throughout all releases, the uc_asp implementation presents better results with respect to lines of code (LOC) and cohesion (LCO) while the uc_oo implementation presents better results for coupling (CBC) and vocabulary size (VS) metrics.

3) Quantifying Changeability: In this section, we analyze the impact of changes based on a set of metrics presented in Table IV, such as, number of added or changed components (aspects/classes) and number of added LOC. The purpose of using these metrics is to quantitatively assess the propagation of change effects, when applying a set of change requests.

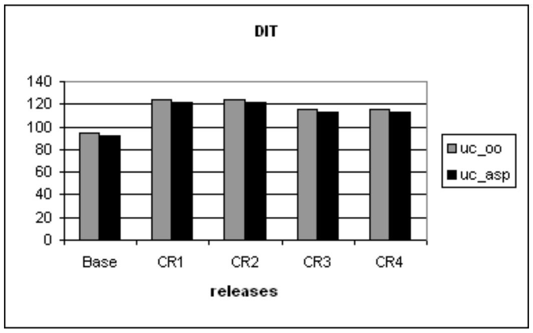


Fig. 9. HealtWatcher - Deph of Iheritance Tree

TABLE IV
HEALTHWATCHER - CHANGEABILITY ABSOLUTE VALUES

Scenario	Changed Comp.		Added Comp.		Added LOC	
	uc_oo	uc_asp	uc_oo	uc_asp	uc_oo	uc_asp
CR1	17	4	14	21	1104	1151
CR2	8	2	0	0	31	33
CR3	7	0	0	0	0	0
CR4	3	18	7	7	874	782
Total	35	24	21	28	2009	1966

Lower values of the metrics denote a less invasive and a more localized process of change; which is a desired side effect produced by well-modularized systems.

Table IV shows the absolute values for the changeability metrics. We find that the uc_asp vesrion conforms more strictly to the Open-Closed principle [24] which states that *"software should be open for extension, but closed for modification"*. It requires more new components and less modifications in existing components to implement a change (when we consider the total obtained in all releases). Up to 14% fewer components are changed in the uc_asp implementation throughout the releases, as seen in Table V.

Propagation of Use Case Scoped Changes is superior in the Use Case as Aspect implementation. When considering the change requests (CR1, CR2, and CR3) where the uc_asp is superior, they require fewer changes to components (in terms of modified components) and more added components. Considering the number of components manipulated at each change, the uc_asp implementation is, on average, 43% superior to the uc_oo implementation. This happens because these CR's are applied directly to use case implementations which are implemented, in the uc_asp implementation, as separate modules. While in the uc_oo implementation new fields and methods are added directly to the system's core classes in order to implement the same changes.

Propagation of System Scoped Changes is superior in the Use Case OO implementation. When considering the change request where the uc_oo implementation is superior (CR4), the inverse behavior is observed; that is, the uc_oo implementation now requires fewer changes to components and the same number of components to be added in the system. This happens because when implementing a system

TABLE V
HEALTHWATCHER - CHANGEABILITY PERCENTAGE

Scenario	% Unchanged Comp		% Changed Comp.	
	uc_oo	uc_asp	uc_oo	uc_asp
CR1	81.9	96.1	18.1	3.9
CR2	92.6	98.4	7.4	1.6
CR3	93.5	100.0	6.5	0.0
CR4	97.1	85.2	2.9	14.8
Total	65.1	79.7	34.9	20.3

TABLE VII
PARALLEL DEVELOPMENT

Scenarios	Core Influence (CI))		#adjustment in core	
	uc_oo	uc_asp	uc_oo	uc_asp
Base	1	0.52	0	0
CR1	1	0.44	0	0
CR2	1	0.44	0	0
CR3	1	0.46	0	0
CR4	1	0.42	7	7

scoped change in the uc_asp implementation, code related to the change is scattered across the use cases modules. In this case, the CR4 deals with the modification of the persistent mechanism which impacts all use cases in the system. In the uc_oo implementation, this change is restricted to `Repository` elements which are confined in the Data layer. On the contrary, in the uc_asp implementation, there is persistence code in all use case aspects, resulting in a more invasive change.

So, we have strong evidences that propagation of change effects is extremely dependent on the type of change. A system under a use case decomposition, generally, absorbs better the effects of functional (use case scoped) changes; having a more invasive process of change when subjected to broader-scoped changes (system scoped).

4) Quantifying Traceability, Pluggability and Support for Parallel Development: In this section, we analyze how the isolation of functional use case implementations affects traceability (from requirement to source code), pluggability and support for parallel development.

Table VI presents the metric values that quantify how much effort is needed to plug/unplug a use case implementation on both implementations of the system. We collect the results after the application of CR3 (Removal of 1 use case in the system). We consider that a use case code is completely removed from the system when we exclude the dedicated components, and comment the lines of code related to its implementation on classes that share code with others use cases. After that, we count how many components we need to modify; also, how many blocks and lines of code we need to comment in order to completely eliminate the use case implementation. Looking at the results, we can find that it is much easier to plug/unplug a use case in the uc_asp implementation of the system. We need less changes in components, and also fewer lines of code need to be commented to eliminate all code related to a use case from the system.

TABLE VI
PLUGGABILITY

Changed Comp		Blocks with Dead LOC		#Total Dead LOC	
uc_oo	uc_asp	uc_oo	uc_asp	OO	AO
7	1	7	0	42	0

Table VII presents the results for parallel development metrics. The metrics quantify the effects of such attribute by computing the Core Influence (CI), that is, the amount of code that is shared between all use cases in the system. In the uc_oo implementation, use cases code are spread over the system components, therefore, we consider the core as the system itself. So, the higher is the amount of code in the system's core, the higher is its influence on contributing to a more complex parallel development management. We also measure the Number of Adjustment in Core (NAC). This metric computes the number of necessary adjustments in the system's core to accomplish modifications as the system evolves. Most of the adjustments are due to the fact that when we introduce new use cases in the system, we need to factor out some methods that were specific to the realization of an existing use case, but in a further release turn to be shared with others. In the uc_oo implementation, we consider the core as being the whole system, because of that, we do not compute any *adjustment*, as all methods are already defined in the core.

Analyzing these metrics values, we have strong evidences that the uc_asp implementation is superior than the uc_oo implementation. This can be justified by the fact that modularizing use case implementations produces a system where it is easier to assign independent use cases code to different developers; thus, contributing to decrease the complexity on managing parallel development. Both approaches will still depend on a Version Control Tool and on a specific parallel development policy used by the development team. These actions deal with conflict resolutions that, in theory, tend to happen less in the AO implementation, even though, sometimes, adjustments in the system's core are still necessary.

Table VIII shows the result for traceability metrics. We want to be able to quantify how easy it is to trace changes from use case specifications to code. With this purpose, we select two metrics: CDC (number of components that implement a use case) and #SF (number of source files that implement use case). Lower values for these metrics imply on a system where it is easier to create traceability links from use cases specifications to code assets. As we can see, we need less components to implement a use case in the uc_asp implementation of the system. This fact decreases the complexity on the creation and maintenance of traceability links from requirement specification to code assets.

TABLE VIII
TRACEABILITY

Concern Name	uc_oo		uc_asp	
	cdc	#sf	cdc	#sf
SearchDisease	9	9	3	3
UpdateHealthUnit	8	8	3	3
SearchSpecialitiesByHealthUnit	9	9	3	3
UpdateMedicalSpeciality	10	10	4	4
InsertDiseaseType	8	8	2	2
InsertNewEmployee	8	8	2	2
InsertNewSymptom	8	8	2	2
UpdateSymptom	10	10	4	4
UpdateEmployee	9	9	3	3
InsertNewHealthUnit	8	8	2	2
InsertNewComplaint	12	12	4	4
UpdateComplaint	9	9	3	3
SearchComplaint	8	8	2	2
InsertNewMedicalSpeciality	8	8	2	2
SearchHealthUnitBySpecialities	9	9	3	3
Login	6	6	2	2

III. THREATS TO VALIDITY

In this Section, we discuss factors that impose threats to the reliability of our findings.

Target System - Our findings are based on just one real system, this is a limiting factor. However, the HealthWatcher has been extensively used as the subject of studies comparing AO and OO design, because of that, it has been considered as a benchmark system for AOP studies. As future work, we plan to involve others systems to collect more evidence of our findings.

Techniques/Language - Our study uses an AOP-based technique aiming at modularizing use case implementations. Although the technique is not tied to any specific aspect language, we have just explored AspectJ. This can interlace the results with the language and not with the technique itself. Trying to diminish this effect, we just use basic language resources, implementing the technique in the simplest possible way. Doing that, one can swap to other equivalent aspect language without having to change the concept behind the technique; and by so, expect equivalent results even using a different language.

Concerns Selection and Assignment - Concerns metrics can be unreliable because of the subjectivity inherent in the concern selection and assignment tasks. This can limit the consistency and repeatability of concern based measurements. Trying to reduce this effect, we use the concern selection and assignment guideline proposed by Eaddy et al. [22]. This way, we tie the scope of concerns selection and assignment increasing the chances of producing more repeatable and aligned results.

Metrics - Someone could argue that we have not assessed all possible internal software attributes affecting the system modularity. However, looking at previous works, we were able to identify relevant attributes that are frequently used on quantitative case studies: coupling, cohesion, size (VS, LOC), and SoC. We also included others metrics that are more specifically related to our goals, like the ones used to quantify traceability and pluggability of use cases implementations. One can add others indicators to adjust the criteria to particular settings on further case studies. Some of the metrics used in this work have often been questioned, such as, cohesion and coupling. We understand the issues raised on the application of such metrics, mainly, in the AO context. To tackle this problem, our results and conclusions are gathered from a set of metrics rather than just based on one specific.

Change Requests - As we observed, the type of change has strong influence on the studies results and this is a kind of bias that must be taken into account. We were not able to contemplate all kinds and combinations of changes due to time and scope limit of our work. For example, we did not apply changes that deal with refactoring or design patterns aiming at improving the system design. In order to minimize this effect, we prioritize changes that allow us to analyze from different points of views regarding their scope and nature.

IV. RELATED WORK

An increasing number of qualitative and quantitative assessments, comparing OO and AO designs, have been performed in the last five years or so. Most of them concentrate their efforts on the AspectJ language and do not consider use case implementations as the focus of analysis.

We find some works [9], [10], [11] showing how AOP can be used to enable the modularization of use case implementations. These studies focus mainly on the investigation on how the use of AO constructs support the separation of this kind of crosscutting concern. They do not analyze the effects on quality indicators. They report some benefits, such as, improvement on parallel development, traceability, and changeability. However, their observations are not based on empirical evaluations.

Some case studies analyze how AOP promotes superior modularity and separation of concerns in the implementation of commom crosscutting concerns such as distribution [12], [25], persistence [26], [25], and concurrency [25]. However, they also focus on how AO mechanisms are used to separate these usual crosscutting concerns; they do not quantify the positive and negative effects of AO techniques in the presence of widely-scoped changes.

These works [14], [15] analyze quantitatively the scalability of AOP (AspectJ) by performing a set of broader-scoped changes. Their analysis cover diverse aspects like concern interaction, design stability and affect on design principles. However, none of them address use case implementations as the main focus of analysis. Moreover, use case implementations are not considered as crosscutting concerns and are just used as a factor of change, that is, they only report how an introduction or modification of a use case impacts *commons* crosscutting concerns like distribution, persistence, concurrency, and etc.

V. CONCLUSION

This paper presented a case study to assess the effects on the overall system's modularity caused by modularized use case implementations. The main outcomes of our analysis are the following:

1. A use case implementation is a concern that will, likely, be related with others concerns, constituting what we call a *container concern*. For example, in the HW case study, use cases concerns are linked with persistence and distribution. Thus, isolating use case implementations in the HW system degraded the modularization of the persistence and distribution concerns. We find evidence that the isolation of container concerns affect negatively the modularization of concerns that have a dependence relation with them; producing higher degree of scattering of previously well-modularized concerns. For this reason, we can only have a fair evaluation on modularity gain if we analyze the results considering a set of concerns that covers most of the code and represents a rationale of the implementation, and not just a specific set of them.

2. Impact on maintenance and evolution tasks has strong relation with types of change. For example, if a system is under constant addition of functional requirements (perfective and use case-scoped change), isolating use case implementations brings significant gain on modularity. However, considering changes in the application environment (adaptive and system-scoped 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 it is easier to plug/unplug a specific use case and where it is easier to track use case code creating direct links from requirement specification to use case code assets.

From these observations, we find that modularity is a relative concept. This happens because modularity analysis depend on a set of factors with inherent subjective characteristics. In our studies, for example, it depends not only on the chosen system, metrics and the applied technique, but, also on the selected concerns and change scenarios; modifying one of these variables could lead to completely different results regarding the real modularity gain. Indeed, 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 to others 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) should be chosen over others.

Acknowledgment

We would like to thank CNPq, a Brazilian research funding agency, and National Institute of Science and Technology for Software Engineering (INES), funded by CNPq and FACEPE, grants 573964/2008-4 and APQ-1037-1.03/08, for partially supporting this work.

References

[1] G. Heineman and W. Councill, *Component-based software engineering: putting the pieces together*. Addison-Wesley USA, 2001.

[2] E. Dijkstra, "Notes on structured programming," 1969.

[3] B. Liskov and S. Zilles, "Programming with abstract data types," in *Proceedings of the ACM SIGPLAN symposium on Very high level languages*. ACM, 1974, pp. 50–59.

[4] D. Parnas, P. Clements, and D. Weiss, "The modular structure of complex systems," in *Proceedings of the 7th international conference on Software engineering*. IEEE Press, 1984, p. 417.

[5] M. Eaddy, T. Zimmermann, K. Sherwood, V. Garg, G. Murphy, N. Nagappan, and A. Aho, "Do Crosscutting Concerns Cause Defects?" *IEEE Transactions on Software Engineering*, pp. 497–515, 2008.

[6] G. Kiczales, J. Lamping, C. Lopes, J. Hugunin, E. Hilsdale, and C. Boyapati, "Aspect-oriented programming," Oct. 15 2002, uS Patent 6,467,086.

[7] F. Van der Linden, K. Schmid, and E. Rommes, *Software Product Lines in Action: the Best Industrial Practice in Product Line Engineering*. Springer, 2007.

[8] K. Pohl, G. Böckle, and F. van der Linden, *Software Product Line Engineering: Foundations, Principles and Techniques*. Springer, 2005.

[9] S. Herrmann, C. Hundt, and K. Mehner, "Mapping use case level aspects to object teams/java," in *OOPSLA Workshop on Early Aspects*, 2004.

[10] I. Jacobson, "Use cases and aspects-working seamlessly together," *Journal of Object Technology*, vol. 2, no. 4, pp. 7–28, 2003.

[11] M. Bhole and K. Lieberherr, "Use Case Modularity using Aspect Oriented Programming," *Relation*, vol. 10, no. 1.57, p. 9687, 2008.

[12] S. Soares, E. Laureano, and P. Borba, "Implementing distribution and persistence aspects with AspectJ," *ACM SIGPLAN Notices*, vol. 37, no. 11, pp. 174–190, 2002.

[13] I. Sommerville, *Software Engineering*. Addison-Wesley, 2006.

[14] P. Greenwood, T. Bartolomei, E. Figueiredo, M. Dosea, A. Garcia, N. Cacho, C. Sant'Anna, S. Soares, P. Borba, U. Kulesza *et al.*, "On the impact of aspectual decompositions on design stability: An empirical study," *ECOOP 2007–Object-Oriented Programming*, pp. 176–200.

[15] U. Kulesza, C. Sant'Anna, A. Garcia, R. Coelho, A. von Staa, and C. Lucena, "Quantifying the effects of aspect-oriented programming: A maintenance study," in *22nd IEEE International Conference on Software Maintenance, 2006. ICSM'06*, 2006, pp. 223–233.

[16] V. Basili, G. Caldiera, and H. Rombach, "The goal question metric approach," *Encyclopedia of software engineering*, vol. 1, pp. 528–532, 1994.

[17] F. Castor Filho, N. Cacho, E. Figueiredo, R. Maranhão, A. Garcia, and C. Rubira, "Exceptions and aspects: the devil is in the details," in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering, November*. Citeseer, 2006, pp. 05–11.

[18] A. Garcia, C. Sant'Anna, C. Chavez, V. da Silva, C. de Lucena, and A. von Staa, "Separation of concerns in multi-agent systems: An empirical study," *Software Engineering for Multi-Agent Systems II*, pp. 343–344, 2004.

[19] A. Garcia, C. Sant'Anna, E. Figueiredo, U. Kulesza, C. Lucena, and A. von Staa, "Modularizing design patterns with aspects: a quantitative study," *Transactions on Aspect-Oriented Software Development I*, pp. 36–74, 2006.

[20] S. Chidamber, C. Kemerer, and C. MIT, "A metrics suite for object oriented design," *IEEE Transactions on software engineering*, vol. 20, no. 6, pp. 476–493, 1994.

[21] M. Ceccato and P. Tonella, "Measuring the effects of software aspectization," in *Workshop on Aspect Reverse Engineering*. Citeseer, 2004.

[22] M. Eaddy, A. Aho, and G. Murphy, "Identifying, assigning, and quantifying crosscutting concerns," in *Proceedings of the First International Workshop on Assessment of Contemporary Modularization Techniques*. IEEE Computer Society, 2007, p. 2.

[23] "Aop metrics tool," http://aopmetrics.tigris.org/.

[24] B. Meyer, *Object-Oriented Software Construction*. Prentice-Hall, 1988.

[25] S. Soares, P. Borba, and E. Laureano, "Distribution and persistence as aspects," *Software Practice and Experience*, vol. 36, no. 7, p. 711, 2006.

[26] A. Rashid and R. Chitchyan, "Persistence as an Aspect," in *Proceedings of the 2nd international conference on Aspect-oriented software development*. ACM, 2003, p. 129.