Chapter 1 : Aspect Oriented Refactoring of J2EE Applications A Case Study | joseph nyirenda - racedaydv

Otto-von-Guericke-UniversitÃ¤t Magdeburg School of Computer Science Department of Technical and Business Information Systems Diplomarbeit Aspect-Oriented Refactoring of Berkeley DB.

Typical problems of a requirements document, regarding its contents and organization, involve deficient modularization where requirements artifacts may deal with too much information, duplication of requirements, scattering requirements, tangled problems, among other problems. In this paper we describe how to improve requirements documents by removing duplication of information using aspect-oriented refactoring. Refactorings and Early Aspects. These approaches describe the functionalities to be provided and the interactions between the users and the system. Requirements are structured and described using, for example, use cases, viewpoints, textual descriptions, activity diagrams or sequence diagrams [2]. Over the past years, we have observed that a set of typical issues seems to plague the requirements specifications, such as requirements that are abandoned and no longer relevant, descriptions that are unnecessarily long and complex, and information that is duplicated. Inspired in the code refactoring literature [9], we set off to identify a bad smell related to duplicated requirements that could indicate potential refactoring opportunities. These bad smells decrease reuse, not only during implementation, but throughout the development process [1] and can be minimized by the identification of their symptoms and the removal of their causes. These symptoms may indicate potential problems with the software [2] and can be removed using appropriate refactoring transformations. The removal of these bad smells in the early stages of a software development process reduces the costs associated with software changes. This cost reductions could be three to six times bigger in later stages than during requirements activities [10]. Opdyke, in [9], initially coined the term refactoring as the process of improving the design of existing software using transformations, without changing its observable behaviour. The term is also used to refer to program restructuring operations aiming to support the design, evolution and reuse of object-oriented software. Refactorings express ideas of good style, which can have a significant impact on the maintainability and evolvability of code bases. There are some tools for refactoring requirements documents [21] , [22], [16] but they focus on specific techniques, such as use case models and do not directly address textual descriptions or other mechanisms used to detail requirements. Moreover, these approaches do not provide any guidelines about how to identify the problems. As such, they say nothing about which refactoring can be used to address those issues. In this paper we describe a generic approach to identify duplicated, tangled and scattered requirements as a refactoring opportunity in requirements descriptions, together with an associated early aspect refactoring. The approach can be instantiated with any of the existing requirements description techniques e. In this paper we will demonstrate our ideas using Multi- Dimensional Separation of Concerns [8] and use cases approaches. In summary, the main contributions of this paper are twofold: We describe a how to identify occurrences of these problems and b indicate how to use the refactoring that can to minimize the effects of the problem. This refactoring contains the context that suggests the application of the refactoring, the type of solution provided by the refactoring, a motivation for the transformations, its mechanics a set of well defined steps and an example of a refactored description. We are using early aspects for having had demonstrated a good alternative to deal with duplicated, tangled and scattered requirements. The Early Aspects [3] focuses on managing crosscutting properties at the early development stages of requirements engineering and architecture design using the aspect oriented paradigm [15]. This paper is structured as follows: Section 2 introduces the concept of refactoring opportunity. Section 3 presents the extract early aspect refactoring to manipulate requirement descriptions. Section 4 discusses related work. In Section 5, conclusions are discussed. The Refactoring Opportunity Fowler in [4] introduces bad smells as indications of deficiencies that can appear in existing software artifacts. The use of refactoring techniques might address the causes of these deficiencies. In this paper, we use the term refactoring opportunity to refer to the types of bad smells that can appear in requirements artifacts. Refactoring opportunities should not be seen as exact rules to the automatic application of refactorings. The requirements engineer needs to decide about the interest in changing the requirements and needs to choose which

refactoring is more adequate for each opportunity. In this section, we describe the duplicated, tangled and scattered requirements refactoring opportunity as well as associated refactoring. The Extract Early Aspect refactoring is described in more detail in section 3. Duplicated Requirements A duplicated requirement is a situation that occurs when i the same requirement is duplicated in different places in a requirements document or ii the same requirements or the same pre-post conditions appear in several requirements structures. A requirement that appears in more than one place offers an opportunity for refactoring. In the simplest case, the requirement is duplicated in one requirements structure. A possible solution is to use the Extract Early Aspect refactoring Section 3. Another common duplication problem occurs when an individual requirement appears in several requirements structures. This duplication could be removed using Extract Early Aspect refactoring as mentioned above. If the requirements are similar but not exactly the same, we may need to separate the duplicated piece from the rest of the requirement. Tangled Requirements A tangled requirement is a situation that occurs when a requirements unit contains descriptions of several properties or different functionalities [15]. A requirement unit that contains several different properties could be hard to understand. Thus it offers an opportunity for refactoring. This solution needs to be analyzed with caution, because it may increase the number of the units. Another possible solution is to use the Move Activity [14] see more details in Section 4. With this solution requirement units are not created, they are just grouped in existing units that already express the same requirements. Thus, this second solution could be an alternative when the first, with aspects, were so expansive. An example of Move Activity refactoring is showed in Figure 1. In the Order Processing System [19] described with use case, consider a Login use case that is concerned with user authentication and also with the functionalities selected by the user two properties in the same unit. The shaded lines show the activities that could be moved to another use case, responsible for the functionality selection flow. Main flow of events: While the user does not select Exit loop 1. The use case starts when the user 9. If the user selects Place Order then starts the application. The system will display the Login The user enters a username and Use Cancel Order. The system will display the Main The user will select a function. The user will select a function The use case ends. Scattered Requirements A scattered requirement is a situation that occurs when the specification of one property is not encapsulated in a single requirements unit [15]. When the requirements that treat of the same goal are disperse by the requirement document the general localization is more difficult harder and in consequence difficult they reuse and maintenance, this situation could be solved with the Refactoring application. Refactoring Requirements In this section we define and describe the requirements refactoring Extract Early Aspect. This refactoring was mentioned in the previous section to provide solutions to: For the refactoring we provide a context, a solution, the motivation for the application of the refactoring, a set of mechanics to apply the refactoring and an example illustrating the application of the refactoring. We use the format recommended by Fowler in [4] and we add some figures and a description of the solution for each mechanism steps. The solutions described in this section are abstract, therefore we have the intention of being able uses them in any situation that occurs "bad smells". Thus, to a requirement engineer to uses them, first he already must have chosen the early aspect oriented technique to later applying the solutions. The example used is described using the Multi-Dimensional Separation of Concerns approach [8]. The requirements are represented in XML format. A set of duplicated information is used in several places. This could be better modularized in an early aspect. This refactoring is also used when a requirement is too large or contains information related to a feature that is scattered across several requirements and is tangled with other requirements. Extract that information to a new requirement and name it according to the context. This refactoring should be applied when there is duplicated information that can be split into two or more new requirements. The duplicated information could increase the costs of the requirements document maintenance and the potential for errors insertion. Every time that a change is needed, it is necessary to modify all the places where the duplicated requirements appear. The use of aspect-oriented requirements engineering [15] provides a good mechanism to modularize these requirements that are scattered among several places in a requirements document. The use of aspectual requirements [15], [5] promotes better modularization of the requirements artifacts. The following steps should be performed: Create a new early aspect and name it. Figure 2 shows in detail how to create a new early aspect. Also it is considered when does not have to create a

new early aspect. Select in the original requirement the information you want to extract. Figure 3 illustrate an example with 10 requirements structures that only in 3 had been found the information desired. Illustration of the requirements document and the selected segments. Figure 4 shows in detail how to select in the original requirement the information you want to extract. The detailing mechanism to select the information you want to extract. Add the selected information to the new early aspect. Figure 5 illustrate the situation of the 3 selected information in the previous step to being added to the new early aspect. Illustration of the addition of the selected information in the new early aspect. To add the information in the new structure it is important to analyze the existence of a priority between the selected information. Figure 6 shows how to add the selected information in the new early aspect. To all structures of the requirement document that exists the information selected do: Verify IF exist a priority order:

Chapter 2 : CiteSeerX â€” Citation Query Aspect-Oriented Refactoring: Classification and Challenges

AOP has several direct antecedents A1 and A2: The examples in this article use AspectJ. It is scattered by virtue of the function such as logging being spread over a number of unrelated functions that might use its function, possibly in entirely unrelated systems, different source languages, etc. That means to change logging can require modifying all affected modules. Aspects become tangled not only with the mainline function of the systems in which they are expressed but also with each other. That means changing one concern entails understanding all the tangled concerns or having some means by which the effect of changes can be inferred. For example, consider a banking application with a conceptually very simple method for transferring an amount from one account to another: A version with all those new concerns, for the sake of example, could look somewhat like this: Transactions, security, and logging all exemplify cross-cutting concerns. Now consider what happens if we suddenly need to change for example the security considerations for the application. AOP attempts to solve this problem by allowing the programmer to express cross-cutting concerns in stand-alone modules called aspects. Aspects can contain advice code joined to specified points in the program and inter-type declarations structural members added to other classes. For example, a security module can include advice that performs a security check before accessing a bank account. The pointcut defines the times join points when one can access a bank account, and the code in the advice body defines how the security check is implemented. That way, both the check and the places can be maintained in one place. Further, a good pointcut can anticipate later program changes, so if another developer creates a new method to access the bank account, the advice will apply to the new method when it executes. So for the example above implementing logging in an aspect: Advice should be reserved for the cases where you cannot get the function changed user level [6] or do not want to change the function in production code debugging. Join point models[edit] The advice-related component of an aspect-oriented language defines a join point model JPM. A JPM defines three things: When the advice can run. These are called join points because they are points in a running program where additional behavior can be usefully joined. A join point needs to be addressable and understandable by an ordinary programmer to be useful. It should also be stable across inconsequential program changes in order for an aspect to be stable across such changes. Many AOP implementations support method executions and field references as join points. A way to specify or quantify join points, called pointcuts. Pointcuts determine whether a given join point matches. Most useful pointcut languages use a syntax like the base language for example, AspectJ uses Java signatures and allow reuse through naming and combination. A means of specifying code to run at a join point. AspectJ calls this advice , and can run it before, after, and around join points. Some implementations also support things like defining a method in an aspect on another class. Join-point models can be compared based on the join points exposed, how join points are specified, the operations permitted at the join points, and the structural enhancements that can be expressed. AspectJ The join points in AspectJ include method or constructor call or execution, the initialization of a class or object, field read and write access, exception handlers, etc. They do not include loops, super calls, throws clauses, multiple statements, etc. Pointcuts are specified by combinations of primitive pointcut designators PCDs. One such pointcut looks like this: For example, this Point This pointcut matches when the currently executing object is an instance of class Point. Pointcuts can be composed and named for reuse. It can be referred to using the name "set ". Advice specifies to run at before, after, or around a join point specified with a pointcut certain code specified like code in a method. The AOP runtime invokes Advice automatically when the pointcut matches the join point. All advice languages can be defined in terms of their JPM. Join points are all model elements. Pointcuts are some boolean expression combining the model elements. The means of affect at these points are a visualization of all the matched join points. Inter-type declarations[edit] Inter-type declarations provide a way to express crosscutting concerns affecting the structure of modules. Also known as open classes and extension methods , this enables programmers to

declare in one place members or parents of another class, typically in order to combine all the code related to a concern in one aspect. For example, if a programmer implemented the crosscutting display-update concern using visitors instead, an inter-type declaration using the visitor pattern might look like this in AspectJ: It is a requirement that any structural additions be compatible with the original class, so that clients of the existing class continue to operate, unless the AOP implementation can expect to control all clients at all times. Implementation[edit] AOP programs can affect other programs in two different ways, depending on the underlying languages and environments: The difficulty of changing environments means most implementations produce compatible combination programs through a process known as weaving - a special case of program transformation. An aspect weaver reads the aspect-oriented code and generates appropriate object-oriented code with the aspects integrated. The same AOP language can be implemented through a variety of weaving methods, so the semantics of a language should never be understood in terms of the weaving implementation. Only the speed of an implementation and its ease of deployment are affected by which method of combination is used. Bytecode weavers can be deployed during the build process or, if the weave model is per-class, during class loading. AspectJ started with source-level weaving in , delivered a per-class bytecode weaver in , and offered advanced load-time support after the integration of AspectWerkz in However, some third-party decompilers cannot process woven code because they expect code produced by Javac rather than all supported bytecode forms see also " Criticism ", below. Deploy-time weaving offers another approach. The existing classes remain untouched, even at runtime, and all existing tools debuggers, profilers, etc. Standard terminology used in Aspect-oriented programming may include: Cross-cutting concerns Main article: Cross-cutting concern Even though most classes in an OO model will perform a single, specific function, they often share common, secondary requirements with other classes. For example, we may want to add logging to classes within the data-access layer and also to classes in the UI layer whenever a thread enters or exits a method. Further concerns can be related to security such as access control [8] or information flow control. Advice programming This is the additional code that you want to apply to your existing model. In our example, this is the logging code that we want to apply whenever the thread enters or exits a method. Pointcut This is the term given to the point of execution in the application at which cross-cutting concern needs to be applied. In our example, a pointcut is reached when the thread enters a method, and another pointcut is reached when the thread exits the method. Aspect The combination of the pointcut and the advice is termed an aspect. In the example above, we add a logging aspect to our application by defining a pointcut and giving the correct advice. Comparison to other programming paradigms[edit] Aspects emerged from object-oriented programming and computational reflection. AOP languages have functionality similar to, but more restricted than metaobject protocols. Aspects relate closely to programming concepts like subjects , mixins , and delegation. Other ways to use aspect-oriented programming paradigms include Composition Filters and the hyperslices approach. Since at least the s, developers have been using forms of interception and dispatch-patching that resemble some of the implementation methods for AOP, but these never had the semantics that the crosscutting specifications provide written in one place. Though it may seem unrelated, in testing, the use of mocks or stubs requires the use of AOP techniques, like around advice, and so forth. Here the collaborating objects are for the purpose of the test, a cross cutting concern. Thus the various Mock Object frameworks provide these features. For example, a process invokes a service to get a balance amount. In the test of the process, where the amount comes from is unimportant, only that the process uses the balance according to the requirements. Adoption issues[edit] Programmers need to be able to read code and understand what is happening in order to prevent errors. Those features, as well as aspect code assist and refactoring are now common. Given the power of AOP, if a programmer makes a logical mistake in expressing crosscutting, it can lead to widespread program failure. Conversely, another programmer may change the join points in a program â€” e. One advantage of modularizing crosscutting concerns is enabling one programmer to affect the entire system easily; as a result, such problems present as a conflict over responsibility between two or more developers for a given failure. However, the solution for these problems can be much easier in the presence of AOP, since only the aspect needs to be changed, whereas the corresponding problems without AOP can be much more spread out. The obliviousness of application, which

is fundamental to many definitions of AOP the code in question has no indication that an advice will be applied, which is specified instead in the pointcut , means that the advice is not visible, in contrast to an explicit method call. This can be mitigated but not solved by static analysis and IDE support showing which advices potentially match. General criticisms are that AOP purports to improve "both modularity and the structure of code", but some counter that it instead undermines these goals and impedes "independent development and understandability of programs".

Chapter 3 : Aspect-oriented programming - Wikipedia

Refactoring, in spite of widely acknowledged as one of the best practices of object-oriented design and programming, still lacks quantitative grounds and efficient tools for tasks such as detecting smells, choosing the most appropriate refactoring or validating the goodness of changes.

A functional aspect is an aspect that has the semantics of a transformation; it is a function that maps a program to an advised program. Functional aspects are composed by function composition. In this paper, we explore functional aspects in the context of aspectoriented refactoring. We show that re We show that refactoring legacy applications using functional aspects is just as flexible and expressive as traditional aspects functional aspects can be refactored in any order , while having a simpler semantics aspect composition is just function composition , and causes fewer undesirable interactions between aspects the number of potential interactions between functional aspects is half the number of potential interactions between traditional aspects. We analyze several aspect-oriented programs of different sizes to support our claims. Show Context Citation Context The essence of AOR can be exp Refactoring, in spite of widely acknowledged as one of the best practices of object-oriented design and programming, still lacks quantitative grounds and efficient tools for tasks such as detecting smells, choosing the most appropriate refactoring or validating the goodness of changes. This is a pro This is a proposal for a method, supported by a tool, for cross-paradigm refactoring e. Bieman , " Aspect-based refactoring, called aspectualization, involves moving program code that implements cross-cutting concerns into aspects. Such refactoring can improve the maintainability of legacy systems. Long compilation and weave times, and the lack of an appropriate testing methodology are two challe Long compilation and weave times, and the lack of an appropriate testing methodology are two challenges to the aspectualization of large legacy systems. We propose an iterative test driven approach for creating and introducing aspects. The approach uses mock systems that enable aspect developers to quickly experiment with different pointcuts and advice, and reduce the compile and weave times. The approach also uses weave analysis, regression testing, and code coverage analysis to test the aspects. We developed several tools for unit and integration testing. These annotations help when creating a pointcut. We also believe they should help during maintenance, since changes to core concerns could change the matching joinpoints. His approach does not help when Reducing Subjectivity in Code Smell Detection: However, such guidelines remain mostly qualitative in nature. As a result, judgments on how to conduct refactoring processes remain mostly subj As a result, judgments on how to conduct refactoring processes remain mostly subjective and therefore non-automatable, prone to errors and unrepeatable. The detection of the Long Method code smell is an example. The results of an experiment illustrating the use of this technique are reported. This finding supports the use of code smells as a systematic method to identify and refactor problematic classes in this specific context. One of the main problems in Aspect-Oriented Software Development is the so-called fragile pointcut problem. Uncovering and specifying a good robust pointcut is not an easy task. We present the tool chain we implemented to induce a pointcut given a set of identified joinpoints. Using several realistic medium-scale experiments, we show that our approach is able to automatically induce robust pointcuts for a set of joinpoints. As mentioned earlier, the major area of application of our technique lies in the automated refactoring of crosscutting concerns in pre-AOP code into aspects. However, these techniques do not consider the generation of pattern-based pointcuts. A large part of converting a pre-AOP application into an aspect-oriented one consists, next to as-pect mining, out of aspect refactoring.

Chapter 4 : CiteSeerX â€” Citation Query Aspect-Oriented Refactoring

34 Aspect Oriented Refactoring of J2EE Applications: A Case Study Listing Enforcement aspect for racedaydvl.com and racedaydvl.com The enforcement aspect in Listing ensures that all calls to racedaydvl.com and racedaydvl.com are flagged as a violation of the exception management policy.

DOWNLOAD PDF ASPECT ORIENTED REFACTORING

Chapter 5 : aop - Difference between Refactoring and Aspect Oriented Programming - Stack Overflow

Assessment of the Impact of Aspect Oriented Programming on Refactoring Procedural Software. Zeba Khanam, S.A.M Rizvi, Department of Computer Science.

Chapter 6 : Refactoring to Requirements Documents: An approach Aspect Oriented | R. Penteado - raceda

Refactoring Aspect-Oriented Programs Masanori Iwamoto Department of Computer Science and Engineering Fukuoka Institute of Technology Wajiro-Higashi, Higashi-ku.

Chapter 7 : Improving safety when refactoring aspect-oriented programs - researchr publication

aspect-oriented refactoring. Section 6 concludes the chapter with a summary. 2 Crosscutting Concerns Modern software is complex, and the complexity is increasing.