Aspect oriented software development

The implementation of software applications using GOAD techniques results in a better implementation structure which has an impact on many important software qualities such as enhanced risibility and reduced complexity. In turn, these software qualities lead to an improved software development lifestyle and, hence, to better software. This report introduces to management and software development staff to the concepts of aspect-orientation software development.

It presents why aspect-orientation is needed in modern software development and what its contributions are to the improvement of software design and implementation structure. The report also highlight AAAtechnologydetails though without probing much in particular, as it present the various concepts of GOAD. After reading this introduction, the reader will understand what GOAD is about, know its key concepts and terminology engaged to elaborate 2. Introduction As software systems becomes more complex developers use new technologies to help manage development. The development of large and complex software applications is a challenging task. Apart from the enormous complexity of the software's desired functionality, software engineers are also faced with many other acquirement that are specific to the software development lifestyle. Requirements such as risibility, robustness, performance, believability, etc. Re requirements about the design and the implementation of the software itself, rather than about its functionality. Nevertheless, these non-functional requirements cannot be neglected because they contribute to the overall software quality, which is eventually perceived by the users of the software application. For example, a better believability will ensure that future

maintenance tasks to the implementation can be carried out relatively easily and consequently also with fewer errors.

Building software applications that adhere to all these functional and non-functional requirements is an ever more complex activity that requires appropriate programming languages and development paradigms to adequately address all these requirements throughout the entire software development lifestyle. To cope with this ever-growing complexity of software development, computersciencehas experienced a continuous evolution of development paradigms and programming languages. In the early days, software was directly implemented in machine-level assembly languages, leading to highly omelet implementations for even simple software applications.

The introduction of the procedural and functional programming paradigms provided software engineers with abstraction mechanisms to improve the design and implementation structure of the software and reduce its overall complexity. An essential element of these paradigms is the ability to structure the software in separate but cooperating modules (e. G. Procedures, functions, etc.). The intention is that each of these modules represents or implements a well-identified subpart of the software, which renders the individual modules better reusable and evolvable.

Modern software development often takes place in the object-oriented programming paradigm that allows to further enhance the software's design and implementation structure through appropriate object-oriented modeling techniques and language features such as inheritance, delegation,

encapsulation and polymorphism. Aspect-oriented programming languages and the entire aspect-orientation paradigm are a next step in this ever continuing evolution of programming languages and development paradigms to enhance software development and hence, improve overall software quality 3. 0

Fundamental ideas underlying aspects and aspect-oriented software development The notion behind aspects is to deal with the issue of tangling and scattering. According to Ian Somerville (2009), tangling occurs when a module in a system includes code that implements different system requirements and scattering occurs when implementation of a single concern (logical requirement or set of requirements) is scattered across several components in a program. 3. 1 What an Aspect is. Aspect is an abstraction which implements a concern. Aspects are completely specification of where it should be executed.

Unlike other abstractions like methods, you cannot tell by examining methods where it will be called from because there is clear separation between the definition and of the abstraction and its use. With Aspects, includes a statement that defines where the aspect will be woven into the program. This statement is known as a pinpoint. Below is an example of a pinpoint (Ian Somerville, 2006) before: call (public void update* (..)) This implies that before the execution of many method whose starts with update, followed by any other sequence of characters, the code in the aspect after the induct definition should be executed.

The wildcat (*) matches any string characters that are allowed in the identifiers. The code to be executed is known as the advice and is implementation of the cross-cutting concern. In an example below of an aspect authentication (let's say for every change of attributes in a payroll system requires authentication), the advice gets a password from person requesting the change and checks that it matches the password of currently logged -in user. If not user is logged out and update does not proceed. Aspect authentication before: call (public void update* (.. // this is a pinpoint { // this is the advice that should be executed when woven into // the executing system into tries = O; string swearword = Password. Get (tries) ; while (tries < 3 && userPassword thisuser. password ( )) { // allow 3 tries to get the password right tries = tries +1 ; userPassword = Password. Get (tries) ; if (userPassword thisuser. password (then //if password wrong, assume user has forgotten to logout System. Logout (thisUser. uid) ; } // authentication (Ian Sommerville, et al. , 2006) 3. 2 Aspect Terminology Advice: the code implementing a concern

Pinpoint: defines specific program events with which advice should be associated (I. E. , woven into a program at appropriate Join points) Events may be method calls/ returns, accessing data, exceptions, etc. Weaving: incorporation of advice code into the program (via source code preprocessing, link-time weaving, or execution time weaving) 4. 0 Why Separation of Concerns a good guiding principle for Software Development Separation of concerns is a key principle of software design and implementation. Concerns reflect the system requirements and the priorities of the system stakeholders.

Some examples of concerns are performance, security, specific categorized in several types. Functional concerns, quality of service concerns, Policy concerns, System concerns and Organizational concerns. Functional: related to specific functionality to be included in a system. Quality of service: related to the nonfunctional behavior of a system (e. G. , performance, reliability, availability). System: related to attributes of the system as a whole (e. G. , maintainability, configurability). Organizational: related to organizationalgoalsand priorities (e. G. , staying within budget, using existing software assets).

In other areas concerns has been categorized according to different areas of interest or properties I. E. High level implies security and quality of service, Caching and buffering are Low level while Functional includes features, business rules and Non Functional (systematic) implies synchronization, transaction management. By reflecting the separation of concerns in a program, there is clear traceability from requirements to implementation. The principle of separation of concerns states that software should be organized so that each program element does one thing and one thing only.

In this case it means each aerogram element should therefore be understandable without reference to other elements. Program abstractions (subroutines, procedures, objects, etc) support the separation of concerns. Core concerns relate to a system's primary purpose and are normally localized within separate procedures, objects, etc. And other concerns tend to scatter and cross multiple elements. These cross-cutting concerns are managed by aspect since they cannot be localized resulting in problems when changes are required due to tangling and scattering.

Separation of concerns provides modular dependency between aspects and components. For instance we would like to maintain a system that manages payroll and personnel functions in our organization, and there is a new requirement to create a log of all changes to an employee's data by management. It would mean that changes will include in payroll, number of deduction, raises, employee's personal data and sass of many other information associated with employee. This implies that there are several codes that will require changes.

This process could be tedious and you might end up forgetting changing other codes as well even not understanding each and every code. With aspects you old deal with a particular element only. In this case there won't be redundancy of multiple codes doing the same thing. An update function could be implemented that would be called whenever you would want to implement a particular method. 5. 0 Aspect-oriented Approach 5. 1 Requirement Engineering In requirements engineering there is need to identify requirements for the core system and the requirements for the system extensions.

Viewpoints are a way to separate the concerns ofdifferent stakeholdersthat are core and secondary concerns. Each viewpoint represents the requirements of related groups of stakeholder. The requirements are organized according to stakeholder viewpoint then they are analyses to discover related requirements that appear in all or most viewpoints. These represent the core functionality of the system. There could be other viewpoint requirements that are specific to that viewpoint these then can be implemented as extensions to the core functionality.

These requirements (secondary functional requirements) often reflect the needs of that viewpoint and may not share there are non-functional requirements that are cross-cutting concerns. These generate requirements of to some or all viewpoint for instance requirements for security, performance and cost. 5. 2 Software Design Aspect Oriented Design is the process of designing a system that makes use of aspects to implement the cross-cutting concerns and extensions that are identified during the requirements engineering process.

ADD focuses on the explicit representation of cross-cutting concerns using adequate design languages. ADD languages consist of some way to specify aspects, how aspects are to be composed and a set of well-defined composition semantics to describe the details of how aspects are to be integrated. (Chitchat, Awls Rashes, Pete Sawyer, Alexandra Garcia, Monica Pinto Larson, Jotter Beaker, Bedim Ticonderoga, Skibobs Clarke, Andrew Jackson, 2005) Like in object orientation, several aspect-oriented extensions to ML design language to represent aspect-oriented concepts at the design level.

One of these ML extensions is ATOM. ADD in ML requires a means of modeling aspects using ML stereotypes. Is an approach of specifying the Join points where the aspect advice is to be composed with the core system. The high-level statement of requirements provides a basis for identifying some system extensions that may be implemented as aspects. Developing these in more details to identify further extensions and understanding the functionality required is to identify a set of use cases associated with each viewpoint. Each use case represents an aspect.

Extension use cases naturally fit the core and extensions architectural model of system. Jacobsen and Eng (2004)) 5. 2. 1 Aspect-oriented Design Process Below is fugue 1 that illustrate the design activities of generic aspect-oriented design process Core system design is where you design the system architecture to support the core functionality of the system. Aspect identification and design Starting with the extensions identified in the system requirements, you should analyses these to see if they are aspects in themselves or if they should be broken down into several aspects.

Composition design At this stage, you analyses the core system and aspect designs to discover where the aspects should be composed with the core system. Essentially, you are identifying the Joint points in a program at which aspects will be woven Conflict analysis and resolution Conflicts occur when there is a pinpoint clash with different aspects specifying that they should be composed at the same point in the aerogram Name design is the essential to avoid the problem of accidental pinpoints.

These occur when, at some program Join point, the name accidentally matches that in a pinpoint pattern. The advice is therefore unintentionally applied at that point. 5. 3 Programming The goal of aspect-oriented programming is to provide an advance modularization scheme to separate the core functionality of software system from system-wide concerns that cut across the implementation of this core functionality. (Kim Mess and Tom Tour©, 2007) APP must address both what the programmer can say and owe the computer system will realize the program in a program system.

APP system: mechanisms are conceptually straight forward and have efficient implementations. 5. 3. 1 Joint Point Model A Join point model defines the kinds of Join points available and how they are accessed and used. They are specific to each aspect-oriented programming language for instance Aspects. In Aspects, Joint point are defined by grouping them into pinpoints. 5. 3. 2 Pinpoint A pinpoint is a predicate that matches Join points. A pinpoint is a relationship 'Join point Boolean', where the domain of the relationship is all possible Join points. 3. 3 Advice 5. 4 Advantages and Disadvantages of APP APP promotes clear design and risibility by enforcing the principles of abstraction and separation of concerns. APP explicitly promotes separation of concerns, unlike earlier development paradigms. This separation of concerns provides cleaner assignment of responsibilities, higher modularization and easier system evolution, and should thus lead to software systems which are easier to maintain. The process is to collect scattered concerns into compact structure units, namely the aspects.

On the other hand, APP cannot be elegantly applied to every possible situation. . 0 Validation and verification Validation and Verification is the process of demonstrating that a program meets the real needs of its stakeholders and meets its specification. Validation or testing is used to discover defects in the program or to demonstrate that the program meets its requirements. Statement verification techniques focus on manual or automated analysis of the source code. Like any other systems, aspects-oriented systems can be tested as black-boxes using the specification to derive the tests.

However, program source code is problematic. Aspects also introduce additional testing (Ian Somerville (2006)) 6. 1 Testing problems with aspects To inspect a program in a conventional language effectively, you should be able to read it from right to left and top to bottom. Aspects make this as the program is a web rather than a sequential document. One can't tell from the source code where an aspect will be woven and executed. Flattening an aspect-oriented program for reading is practically impossible 6. Challenges with Aspect-oriented Systems One of the limitations of APP is that it is not supported by default on any programming platform. Although it seems to be gaining popularity, its implementation has been undertaken by third parties as extensions to development framework. This has resulted in some level of disparity on the features being implemented as some of the implementations only implement specific features making it difficult to use such frameworks in some situations in addition to creating some confusion over the feature.

AAA programs can be " black-box tested" using requirements to design the tests, but program inspections and " white-box testing" can be problematic, since you can't always tell from the source code alone where an aspect will be woven and executed. 7. 0 Recommendations Adopting Aspect Oriented Software development will reduce repetitions of coding or Component maintenance and reuse has a great impact to the company. On the part of cost, the company can determine whether it is easy to maintain its systems or not.

Using other development paradigm can be cumbersome hence increasing tangling and scattering. System performance will also be affected in such a way that there could be more codes doing the same thing. GOAD concepts

reduce redundancy and increase system performance. All functional and non-functional concerns are dealt with in GOAD. On implementation of security, Design flaws and code errors or bugs old be some of the causes of security flaws in software. Unlike SOD, GOAD approach made Software Development easy with the separation of concerns leading to modularization in reuse.