DOWNLOAD PDF IDENTIFYING TEST CASES FROM USE-CASE VARIABLES

Chapter 1 : Use Case Testing

After setting the traceability between scenarios and test cases, we can create a traceability tree that shows traceability all the way from use cases to the test cases. There are two options. The first option -- shown in Figure 13 -- is to trace out of the use case, which shows use cases on the top level and tracing to scenarios and test cases.

Buyer Places a Bid Description: An EBAY buyer has identified an item they wish to buy, so they will place a bid for an item with the intent of winning the auction and paying for the item. For our use case example, the basic flow should be to describe the happy day scenario for your use cases such as "placing a bid". For a consumer to play a successful bid, what is the primary flow when everything goes as planned. An effective use cases needs to have the basic flow before moving forward with writing the alternate flows. It really depends on the level of detail you wish to achieve. However, providing more detail to the consumers of your use case is always a good thing. The alternate flows providing the following: An exception or error flow to any line item in your basic flow An additional flow, not necessarily error based, but a flow that COULD happen A few examples of alternate flows are: While a customer places an order, their credit card failed While a customer places an order, their user session times out While a customer uses an ATM machine, the machine runs out of receipts and needs to warn the customer Tip Produce your effective use case document Recently at a new project assignment, I introduced a mid level developer to the concept of use cases which was totally foreign to him. Once walking him through the basic concepts and showing him the use case example, the lightbulb went off in his head on how convenient and simple it was to grasp the project. In several places in this document, I have stated "effective use cases" rather than just "use cases". The purpose of the use cases is for effective knowledge transfer from the domain expert to the software developer -- these use cases will serve as the software requirement specifications. There are several sources on the web for writing effective use cases including the book by Alistair Cockburn. See the image below for a sample of the use case model. The purpose of the use cases is for effective knowledge transfer from the domain expert to the software developer -- these use cases will serve as software requirements. With so many engineering teams making the paradigm shift from waterfall to Agile Software Development , people often get caught up in having a pure Agile process which would include the use of User Stories. What are they, how are they different from use cases, do I need them, and where do they fit in the process? What is a User Story? Simply put, written from the context of the user as a simple statement about their feature need. They should generally have this format. While a use case is highly structured and tells a story, the User Story sets the stage by stating the need. A User Story is the prelude to the use case by stating the need before the use case tells the story. How does the User Story fit into the process? User Stories are great as an activity in collecting and prioritizing the high level features. Getting this initial feedback from the customer is a simple way of trying to get all of their needs identified and prioritized. The User Stories will then morph themselves into the business requirements and use cases. Like anything else in life, nothing is black and white -- being Agile is really about smaller iterations, learning and adapting to the market. If you are using Agile, Scrum and moving away from waterfall, what you want to do is make sure to iterate with your use cases. All that means is that your flows will be smaller and less feature rich. While the theme of the use case may appear the same from iteration to iteration, what is changing is the level of detail and the features inside the particular sprint. Creating a use case to long winded with too many features can potentially put a product at risk. What happens is that you can extend your release to market from two weeks to several months without the ability to learn from the iteration and adapt to the market. Keep those use cases leaner!

DOWNLOAD PDF IDENTIFYING TEST CASES FROM USE-CASE VARIABLES

Chapter 2 : How to Write Test Cases: Sample Template with Examples

Test cases 4 and 7 would normally be verified at the unit test or integration test phase due to having to, presumably, use an intrusive method to fail the lock mechanism. There are, however, limitations of Using Use Cases For Test Case Generation.

Use case testing is a technique that helps us identify test cases that exercise the whole system on a transaction by transaction basis from start to finish. A use case is a description of a particular use of the system by an actor a user of the system. Each use case describes the interactions the actor has with the system in order to achieve a specific task or, at least, produce something of value to the user. Actors are generally people but they may also be other systems. Use cases are a sequence of steps that describe the interactions between the actor and the system. They often use the language and terms of the business rather than technical terms, especially when the actor is a business user. They serve as the foundation for developing test cases mostly at the system and acceptance testing levels. Use cases can uncover integration defects, that is, defects caused by the incorrect interaction between different components. Used in this way, the actor may be something that the system interfaces to such as a communication link or sub-system. Use cases describe the process flows through a system based on its most likely use. This makes the test cases derived from use cases particularly good for finding defects in the real-world use of the system i. Each use case usually has a mainstream or most likely scenario and sometimes additional alternative branches covering, for example, special cases or exceptional conditions. Each use case must specify any preconditions that need to be met for the use case to work. Use cases must also specify post conditions that are observable results and a description of the final state of the system after the use case has been executed successfully. We show successful and unsuccessful scenarios. In this diagram we can see the interactions between the A actor â€” in this case it is a human being and S system. From step 1 to step 5 that is success scenario it shows that the card and pin both got validated and allows Actor to access the account. But in extensions there can be three other cases that is 2a, 4a, 4b which is shown in the diagram below. For use case testing, we would have a test of the success scenario and one testing for each extension. In this example, we may give extension 4b a higher priority than 4a from a security point of view. System requirements can also be specified as a set of use cases. This approach can make it easier to involve the users in the requirements gathering and definition process.

Chapter 3 : How to Write Test Cases That Don't Suck: Tips from Software Testing Pros

The logical sequence and numbering of identifying the 'Test Case ID' will be very useful for a quick identification execution history of test cases. Also read => + sample ready to use test cases for web and desktop applications.

PDF Development teams can use business process model to visually document business work flows, and associate use cases with those business processes for modeling the desired features to be achieved by the system. In this tutorial, we will explain in detail how to make use of the Model Transitor function to establish traceability between use cases with business processes. BPMN provides business analysts with a set of graphical notations for modeling business processes. It is a flow-chart like diagram, which depicts the process flow, participants involved and message exchanges between participants. Business analysts draw BPD s to model how different participants collaborate together to accomplish a business objective. Having validated the completed business model against the end users, system analyst can then start planning the system. The following is a simple BPD of a registration process for an organization. It covers most of the typical modeling notations you would see. Notation Description Pool - Represents a participant within a process. In BPMN, both pools and lanes are used to represent participants. A lane is contained by a pool for modeling a sub-partition of the parent pool. Start event - The beginning of a process. Triggers can be defined to tell readers in what situation the process will be triggered. Tasks and other flow objects are connected together to form a complete business workflow. End event - The end of a process. A result can be defined to tell readers what will happen when the process ends. In this tutorial we are not going to focus heavily on BPD nor business process modeling. What is Use Case Diagram? Use case modeling refers to the technique of capturing high level user requirements using UML use case diagram. Use case model is designed for software or system designer, not for business people. There are three main elements in a use case diagram. Notation Description Use case - Each use case represents a user goal, which is an objective the user of the system wants to achieve. Note that use cases can only be used to show what the user wants to do instead of what the developer needs to develop, although they may be the same in some cases. Just keep in mind that use case modeling aims at modeling what the user wants to achieve. Actor - User of the system. It can be a system that interacts with our system to fulfill certain business objective. Each communication link implies a sequence of transactions between actor and system. Usually, we develop software to automate or optimize certain work flows of business processes. With BPD, you can understand how participants work together and who is responsible for what, we can identify what functions they need the system to support. Those system functions workflow or business process user wanted could be modeled with a use cases and subsequently developed by the team. As a result, we can say that BPD helps you identify use cases for a system under developed. Visual Paradigm is a visual modeling tool that supports from performing business process to use case modelling from business requirement to application requirement by establishing the traceability linkages between the two models through model transitor feature. We need the traceability because of the following reasons: We can make sure the system can fit into real world usage by studying the part of the process flow a use case involves. To answer questions like "Why do we need this system function? To answer questions like "Has a specific operation been implemented already? The following table shows you the characteristics of pool, lane, actor, task, sub-process and use case, in terms of model transition. Anyone who has an activity to perform, relevant to the process, is said to be a participant. In use case diagram, an actor represents a user of system. Keep in mind that any person or role that is not a user of system should not be considered as actor. In use case diagram, a use case presents a goal user wants to achieve by using the system. Keep in mind that an activity need not to be relevant to any system function, and one use case may satisfy multiple activities. Some people may think that a use case diagram is similar to a BPD but quite different in notations and purposes. Do remember the fact that BPMN is designed for business people while use case diagram is for system analysts or system developers. They serve different purposes and reads a business in two distinct perspectives. BPD can

only give you hints when identifying use cases. There has no rule stating that every task existing in a BPD is equivalent to a use case. But we could elaborate a business process by a use case for automation of a feature by the target system. In the case study, I will give you some idea about what you should pay attention to when transiting a BPD to a use case diagram. You will then understand how different they are. They sell distilled water for business and home use. The following is a textual description of their water delivery process. To order distilled water, customer either calls the ordering hotline or send us Email. The customer service assistant who receives the order will check whether the customer is an existing customer or a new one. If the customer has never ordered before, the customer service assistant will create a customer account for him or her before proceeding to water delivery. The delivery of distilled water is carried out once a week on every Wednesday. So on every Wednesday morning, the customer service assistant will forward orders to the Logistics Department for delivery. Once the manager in the Logistics Department has received the orders, he will arrange the delivery by assigning workers for different orders, printing and posting the schedule. The workers receive the calls and deliver water to the customer accordingly. A business process model has been created based on the description. Now, you are requested to develop a computer system to optimize the whole process. The first thing you need to do is to develop a use case model. With the help of the BPD, try to develop a use case diagram. Download Distilled Water Delivery. You can also find this file at the bottom of this tutorial. Study the process flow carefully. The process starts when a customer places an order. Here we can think of a use case - Place Order. The use case will help automate the process by providing an interface for customer to place order without the assistance of customer service assistant, help verify customer identity and create account if customer does not exist. This prompts the Transit Model Element window, where you can select the model to place the use case and actor, and rename them. In this case we are pleased with the names of the use case and the actor. This forms a new use case diagram in UeXceler. Go back to the BPD. In the business process, the customer service assistant needs to create account for every new customer. In the new system, this can either be a part of the Place Order use case, or be a separate use case triggered by customer service assistant manually. Again, we are pleased with the name of the use case and actor. Keep everything in the Transit Model Element window unchanged. The use case diagram is updated with a new use case and actor. The manager of the Logistics Department can use the system to perform scheduling and notify workers to deliver water. Therefore, this is also a use case of the system. Check the actor Manager in the Transit Model Element window. If we keep the name of actor as Manager, this is ambiguous in the use case model because there may be many departments with many different managers in the company. Therefore, rename the actor to Logistic Department Manager. The use case diagram is updated. Therefore, we do not need to create a use case for it. Suppose the regional manager wants the system to support a new function that can generate a report to show the statistics on orders. This function can help him review and refine the marketing strategy. Although this function had not been modeled in the business process model, we can draw it directly in the use case diagram. Open the use case diagram. Draw an actor Regional Manager. Create a use case Generate Statistic Report from it with association in between. Besides, the client want to allow the logistic department manager to print logistic report. Draw the use cases respectively. Tidy up the diagram. The transition relationship enables you to trace the business process model from use case model and vice versa. Place the mouse pointer over the Place Order use case. Click on the Model Transitor resource at bottom right corner of shape. Place Order from the popup menu.

Chapter 4 : How to Find Use Cases from Business Process (BPMN)?

Before we discuss the points to be remembered for identifying test scenarios and designing test cases, let me first explain the meaning behind `test scenario' and `test case'. A `test scenario' is the summary of a product's functionality, i.e. what will be tested.

Introduction A use case shows the behaviour or functionality of a system. It consists of a set of possible sequences of interactions between a system and a user in a particular environment that are related to a particular goal. In Software Engineering,a use case is used to describe the steps between an user and a software system which guides the user to useful output. The user also called an actor could be a human user, an external hardware,software sysytems,or other subjects. Technically speaking use cases are software modeling technique which helps Architects and Analysts to put together the features to implement and how to fix the errors in simple pictures. Due to its simplicity in conveying its ideas to customers, technical gurus and executives a like it is very popular and often considered powerful. A use case can identify,clarify and organize system requirements in simple steps and can help avoid scope creep. Systems Analysts tries to put a set of all possible sequences of interactions between systems and actors in a particular environment related to a specific goal. It consists of a group of elements that can be used together in such a way that will have an effect larger than the sum of the separate elements combined. A use case can be thought of as a collection of all possible scenarios related to a specific goal. No wonder why use cases are so popular but use case modeling, when used in isolation and performed incorrectly, may lead to certain types of problems. People may mistake that easy-to-read also means easy-to write, but that is a mistake. It could be terribly hard to write easy-to-read stories. Use cases are stories, prose essays, and so bring along all the associated difficulties of story writing in general. When Ivar Jacobson came up with this concept in it was based upon many years of work in component based system development. There were many different techniques then but they were overlapping or had gaps. Tweny five years after though although Use cases are still relevant but it has evolved and being used with a lot of help from web based technology. This paper dives deep and tries to find out how effective use-cases are in system Analysis and software development today. Effectiveness of use-case in concurrent development processes The software development process used by a company today would highly be dependent upon the development methodology being used in the company. Similarly the degree to which Use case is exploited would also depend upon the "the belief level" of the Systems Analyst and the Architect in use cases technology itself. In certain development methodology, only a brief use case usage could be limited to a simple survey for requirements gathering whereas in other development methodologies,use cases evolve in complexity and remarkable change in character as the development process moves forward. Moreover,there are some methodologies that may begin as brief business use cases but that could evolve into more detailed system use cases,and then finally develop into highly detailed and exhaustive test cases. It is the ability of use cases to unite the development activities that makes them such a powerful tool for the planning and tracking of software development projects. In order to understand the power of use cases, it is worth considering the life cycle in detail. Use cases can play a part in the majority of the stages directly associated with systems and software development life cycle. Let us have a close glance on some of those cycles: A requirement is something that a computer application must do for its users. Requirements are generally categorised into functional and non-functional requirements. The functional requirements specify the input and output behaviour of a system. Nonfunctiona requirements specify the other qualities that the system must have, such as those related to the usability, reliablity, performance, and supportability of the system. Use-case model is widely used in requirements gathering and is very popular. In fact the use-case model is the result of the requirements discipline. Requirements gathering utilizes this model from Identified to Agreed upon and it also works out the domain model which defines the terminology used by the use cases. But let us take a pause and focus on other techniques of gathering functional requirements because no technique is foolproof. There is a

section of people who would like the "visual approach" while others might prefer abstract. Yet another section of people would have faith only in concrete and they need somthing which would engage them. They would get distracted by simple use cases which is partly visual. Therefore some Systems Analysts would uses as detailed model as Storyboard is. This gives the analysts and the architect an oppurtunity to visualize and identify the problems before significant investment is undertaken in developing the system. This is almost like developing sketches of major events as done in TV and film industry. This also provides an oppurtunity for customer experience by visualizing the "whole" picture. This is something use case model can not provide because use case model focuses on specific user interfaces and are descrete. Where as storyboards can describe whole business process. On the other hand there are certain industry in which working model would be necessary as a protytypes of the actual system to make sure that system we are targetting would fulfill our requirements. Again our use case model may not compete there as well. But of course developing a prototype would incur some cost and this could be challenging in the current economic situations. Use cases are realized in analysis and design models. Use-case realizations are created that describe how the use cases are performed in terms of interacting objects in the model. This model describes the different parts of the implemented system and how the parts need to interact to perform the use cases. Analysis and design of the use cases matures them through the states of Analyzed and Designed. These states do not change the description of the use cases, but indicate that the use cases have been realized in the analysis and design of the system. The use-case suggest large-scale partitioning of the problem domain. It also provides structuring of analysis objects i. During implementation, the design model is the implementation speci-fication. Because use cases are the basis for the design model, they are implemented in terms of design classes. Once the code has been written to enable a use case to be executed, it can be considered to be in the Implemented state. It suggest iterative prototypes for spiral development. During testing, the use cases constitute the basis for identifying test cases and test procedures; that is, the system is verified by performing each use case. When the tests related to a use case have been successfully passed by the system, the use case can be considered to be in the Verified state. The Accepted state is reached when a version of the system that implements the use case passes independent user-acceptance testing. Most popular model today UML provides generalization, include and extends as three possible relationships. A generalization relationship between use cases implies that the child use case contains all the attributes, sequences of behavior, and extension points defined in the parent use case, and participates in all relationships of the parent use case. The child use case may define new behavior sequences, as well as add behavior into and specialize existing behavior of the parent use case. An include relationship between two use cases means that the sequence of behavior described in the included or sub use case is included in the sequence of the base including use case. The extends relationship provides a way of capturing a variant to a use case. Typically extensions are used to specify the changes in steps that occur in order to accommodate an assumption that is false Coleman, The extends relationship includes the condition that must be satisfied if the extension is to take place, and references to the extension points which define the locations in the base extended use case where the additions are to be made. Use-case extensions permit us to describe additional behaviors that can be inserted into an existing use-case.

Chapter 5 : How to write a good test case - Apache OpenOffice Wiki

A test case is a set of test inputs, execution conditions, and expected results developed for a particular objective: to exercise a particular program path or verify compliance with a specific requirement, for.

May 04, Overview of the requirements types A requirement is defined as "a condition or capability to which a system must conform". A capability needed by a customer or user to solve a problem or achieve an objective A capability that must be met or possessed by a system to satisfy a contract, standard, specification, regulation, or other formally imposed document A restriction imposed by a stakeholder Figure 1 shows the requirements pyramid with the different levels of requirements. The requirements pyramid On the top level are stakeholder needs. Usually, a project contains five to fifteen of these high-level needs. On the lower levels are features, use cases, and supplementary specifications. On different levels of these requirements are different details. The lower the level, the more detailed the requirement is. For example, a need can be: The feature can refine this requirement to be: On the supplementary specification level, the requirement will be even more specific: The further down, the more detailed the requirement. Traceability between requirements Traceability is a technique that provides a relationship between different levels of requirements in the system. This technique helps you determine the origin of any requirement. Figure 2 illustrates how requirements are traced from the top level down. Every need usually maps to a couple of features, and then features map to use cases and supplementary requirements. Traceability requirements pyramid View image at full size Use cases describe functional requirements, and supplementary specifications describe non-functional items. In addition, every use case maps to many scenarios. Mapping use cases to scenarios, then, is a one to many relationship. Scenarios map to test cases also in a one to many relationship. Between needs and features, on the other hand, there is many to many mapping. Traceability plays several important roles: Verify that an implementation fulfills all requirements: Everything that the customer requested was implemented Verify that the application does only what was requested: When some requirements change, we want to know which test cases should be redone to test this change A traceability item is a project element that needs to be traced from another element. Some examples of requirement types in RequisitePro are stakeholder needs, features, use cases, actors, and glossary terms. In RequisitePro there is a convenient way of showing traceability in special views. Figure 3 shows an example of mapping features to use cases. Traceability in RequisitePro View image at full size There is some question as to which direction the arrows should go: Even the two examples in RequisitePro use different guidelines. Actors and use cases An actor is someone or something that interacts with the system. A use case is a description of a system in terms of a sequence of actions. It should yield an observable result or value for the actor. Following are some characteristics of use cases, which: Are initiated by an actor Model an interaction between an actor and the system Describe a sequence of actions Capture functional requirements Should provide some value to an actor Represent a complete and meaningful flow of events The purpose of a use case is to facilitate agreement between developers, customers, and users about what the system should do. A use case becomes sort of a contract between developers and customers. In addition, you can produce sequence diagrams, collaboration diagrams, and class diagrams from use cases. Furthermore, you can derive user documentation from use cases. Use cases may also be useful in planning the technical content of iterations, and give system developers a better understanding of the purpose of the system. Finally, you can use them as an input for test cases. Use case diagrams present relationships between actors and use cases. In this article we will use an online bookstore as an example of a project. Figure 4 shows a use case diagram for this project. Use Case Diagram The general format of a use case is:

Chapter 6 : What is Use case testing in software testing?

A test case in software engineering is a set of conditions or variables under which a tester will determine whether an application or software system is working correctly or not.

User should not Login into application As Expected While drafting a test case do include the following information The description of what requirement is being tested The explanation of how the system will be tested The test setup like: Test Cases need to be simple and transparent: Create test cases that are as simple as possible. They must be clear and concise as the author of test case may not execute them. Use assertive language like go to home page, enter data, click on this and so on. This makes the understanding the test steps easy and test execution faster. Create Test Case with End User in Mind Ultimate goal of any software project is to create test cases that meets customer requirements and is easy to use and operate. A tester must create test cases keeping in mind the end user perspective 3. Avoid test case repetition. Do not repeat test cases. If a test case is needed for executing some other test case, call the test case by its test case id in the pre-condition column 4. Do not Assume Do not assume functionality and features of your software application while preparing test case. Stick to the Specification Documents. Test Cases must be identifiable. Name the test case id such that they are identified easily while tracking defects or identifying a software requirement at a later stage. Testing techniques help you select a few test cases with the maximum possibility of finding a defect. This method is used when software behavior changes from one state to another following particular action. Self cleaning The test case you create must return the Test Environment to the pre-test state and should not render the test environment unusable. This is especially true for configuration testing. Repeatable and self-standing The test case should generate the same results every time no matter who tests it After creating test cases, get them reviewed by your colleagues. Your peers can uncover defects in your test case design, which you may easily miss. Test Case Management Tools Test management tools are the automation tools that help to manage and maintain the Test Cases. Main Features of a test case management tool are For documenting Test Cases: Test Case can be executed through the tools and results obtained can be easily recorded. Automate the Defect Tracking: Failed tests are automatically linked to the bug tracker , which in turn can be assigned to the developers and can be tracked by email notifications. Requirements, Test cases, Execution of Test cases are all interlinked through the tools, and each case can be traced to each other to check test coverage. Test cases should be reusable and should be protected from being lost or corrupted due to poor version control. Test Case Management Tools offer features like Naming and numbering conventions.

Chapter 7 : Identify use case scenarios | Microsoft Docs

Test cases 3 through 7 test at the boundaries of the quadrants. Test cases 8 through 11 test values within each of the quadrants. Test case 12 tests that the "wrapping around" of direction, where it becomes greater than degrees.

Firesmith Introduction Over the last three years, use cases have become well established as one of the fundamental techniques of object-oriented analysis. Possibly in reaction to the previous structured methods, early object-oriented development methods overemphasized static architecture and partially ignored dynamic behavior issues during requirements analysis, especially above the individual class level where state modeling provides an important technique for dynamic behavior specification. Use cases provide a great many benefits in addition to correcting this overemphasis, and most major object-oriented development methods including my own have jumped on the band wagon and added use cases during the last few years. In the resulting hoopla and hype, however, there has been little discussion of the limitations and potential pitfalls associated with use cases. This column is an attempt to provide a more balanced presentation and to caution against the uncritical acceptance of use cases as the latest patent medicine for all software ailments. Definitions The term use case was introduced by Ivar Jacobson et al. A use case is a description of a cohesive set of possible dialogs i. An actor is a role played by a user i. A use case is thus a general way of using some part of the functionality of a system. Therefore, the description of an individual use case typically can be divided into a basic course and zero or more alternative courses. The basic course of a use case is the most common or important sequence of transactions that satisfy the use case. The basic course is therefore always developed first. The alternative courses are variants of the basic course and are often used to identify error handling. Within reason, the more alternative courses identified and described, the more complete the description of the use case and the more robust the resulting system. As a user-centered analysis technique, the purpose of a use case is to yield a result of measurable value to an actor in response to the initial request of that actor. A use case may involve multiple actors, but only a single actor initiates the use case. Because actors are beyond the scope of the system, use-case modeling ignores direct interactions between actors. A use case may either be an abstract use case or a concrete use case. An abstract use case will not be instantiated on their own, but is only meaningful when used to describe functionality that is common between other use cases. On the other hand, a concrete use case can be instantiated to create a specific scenario. According to Ivar Jacobson, use cases are related by two main associations: The extends association specifies how one use case description inserts itself into, and thus extends, a second use case description that is completely independent and unknowing of the first use case. Depending on some condition, the second use case may either be performed with or without the extending use case. These two associations are closely related and easy to confuse. The actual distinction between these two associations is unclear, and Rumbaugh [4] has thankfully combined them into a single adds association from the main concrete use case to the abstract use cases that it uses. Clearly, use cases are functional abstractions and are thus large operations, the implementation of which thread through multiple objects and classes. However, a use case need not have anything to do with objects. It is a discipline of its own, orthogonal to object modeling. Except for those requirements concerned with initialization, the functional requirements for Door Master are captured in the following nine use cases: Enter the Disabled Door. Employees and security guards enter freely through the door when Door Master is disabled. Enter the Secured Door. Employees and security guards enter through the door by 1 entering the entry code on the numeric keypad, 2 entering through the door, and 3 closing the door behind them. Change the Entry Code. Security guards change the entry code by 1 pressing the change entry code button on the control panel, 2 providing authorization by entering the security code on the numeric keypad, 3 entering the new entry code on the numeric keypad, and 4 verifying the new entry code by reentering it on the numeric keypad. Change the Security Code. Security guards change the security code by 1 pressing the change security code button on the control panel, 2 providing authorization by entering the old security code on the numeric keypad, 3 entering the new security code on the numeric

keypad, and 4 verifying the new security code by reentering it on the numeric keypad. Enable the Door Master. Security guards enable Door Master by 1 pressing the enable button on the control panel and 2 providing authorization by entering the security code on the numeric keypad. Door master then 3 turns off the disabled light, 4 turns on the enabled light, and 5 locks the door. Disable the Door Master. Security guards disable Door Master by 1 pressing the disable button on the control panel and 2 providing authorization by entering the security code on the numeric keypad. Door master then 3 turns off the enabled light, 4 turns on the disabled light, and 5 unlocks the door. The following two abstract use cases are common to, and are therefore used by, five of the concrete use cases: Enter the Entry Code. Employees and security guards enter the entry code by pressing five keys on the numeric keypad followed by the enter key. Door master beeps after each key and verifies the entry code. Enter the Security Code. Employees and security guards enter the entry code by pressing seven keys on the numeric keypad followed by the enter key. The alarm is raised if the door is left open too long or if the door is not shut when Door Master is enabled. The security guards disable the alarm by entering the security code. A Use Case Driven Approach in They have been added to numerous object-oriented development methods e. Use cases are a powerful technique for the elicitation and documentation of blackbox functional requirements. Because they are written in natural language, use cases are easy to understand and provide an excellent way for communicating with customers and users. Although computer-aided software engineering CASE tools are useful for drawing the corresponding interaction diagrams, use cases themselves require remarkably little tool support. Use cases can help manage the complexity of large projects by decomposing the problem into major functions i. Because they typically involve the collaboration of multiple objects and classes, use cases help provide the rationale for the messages that glue the objects and classes together. Use cases also provide an alternative to the overemphasis of traditional object-oriented development methods on such static architecture issues as inheritance and the identification of objects and classes. Use cases provide an objective means of project tracking in which earned value can be defined in terms of use cases implemented, tested, and delivered. Use cases can form the foundation on which to specify end-to-end timing requirements for real-time applications. The Dangers of Misusing Use Cases Because of their many important advantages and extreme popularity, use cases have become a fundamental part of object technology and have been incorporated in one form or another into most major object-oriented development methods. In the rush to jump onto the use case bandwagon, use cases have been perceived by some as either a panacea or as an end in-and-of themselves. Unfortunately, this has often led to the uncritical acceptance of use cases without any examination of their numerous limitations and ample opportunities they offer for misuse. The following provides an overview of the major risks associated with use cases: Use cases are not object-oriented. Each use case captures a major functional abstraction that can cause the numerous problems with functional decomposition that object technology was to avoid. The functional nature of use cases naturally leads to the functional decomposition of a system in terms of concrete and abstract use cases that are related by extends and uses associations. Each individual use case involves different features of multiple objects and classes, and each individual object or class is often involved in the implementation of multiple use cases. Therefore, any decomposition based on use cases scatters the features of the objects and classes among the individual use cases. On large projects, different use cases are often assigned to different teams of developers or to different builds and releases. Because the use cases do not map one-to-one to the objects and classes, these teams can easily design and code multiple, redundant, partial variants of the same classes, producing a corresponding decrease in productivity, reuse, and maintainability. The use case model and the object model belong to different paradigms i. The simple structure of the use case model does not clearly map to the network structure of the object model with its collaborating objects and classes. The requirements trace from the use cases to the objects and classes is also not one-to-one. These mappings are informal and somewhat arbitrary, providing little guidance to the designer as to the identification of objects, classes, and their interactions. The situation is clearly reminiscent of the large semantic gap that existed between the data flow diagrams network of structured analysis and the structure charts hierarchy of

structured design. The use of the single object paradigm was supposed to avoid this problem. Another potential problem with use case modeling is knowing when to stop. How many use cases are required to adequately specify a non-trivial, real-world application? As object technology is applied to ever increasingly complex projects, the simple examples and techniques of the text books often have trouble scaling up. The use of concurrency and distributed architectures often means that the order of the interactions between the system and its environment is potentially infinite. Too few use cases result in an inadequate specification, while too many use cases leads to functional decomposition and the scattering of objects and classes to the four winds. Often, systems and software engineers must often limit their analysis to the most obvious or important scenarios and hope that their analysis generalizes to all use cases. Although use cases are functional abstractions, use case modeling typically does not yet apply all of the traditional techniques that are useful for analyzing and designing functional abstractions. Most current techniques do not easily handle the existence of branches and loops in the logic of a use case. Interaction diagrams are primarily oriented towards a simple, linear sequence of interactions between the actors and the major classes of the system. The use of abstract use cases and either extends or uses associations to solve this problem only exacerbates the functional decomposition problem. Some approach similar to that of the basis paths of structured testing would clearly help determine the adequacy of the use case model, but such an approach is as yet not available to the typical developer. Most techniques do not address the issues of concurrency and the different types of messages that result. As illustrated in [4, 6], the concepts of preconditions, postconditions, invariants, and triggers should also be added to better analyze and specify use cases. Being created at the highest level of abstraction before objects and classes have been identified, use cases ignore the encapsulation of attributes and operations into objects. Use cases therefore typically ignore issues of state modeling that clearly impact the applicability of some use cases. The basic ideas and techniques of use cases should also be applied to Booch mechanisms [7] and integration testing, but adequate extensions have yet to be published. Another major problem with use case modeling is the lack of formality in the definitions of the terms use case, actor, extends, and uses. Similarly, the specification of individual use cases in natural languages such as English provides ample room for miscommunication and misunderstandings. Use cases provide a much less formal specification of their instances i. While everything may seem clear that the highest level of abstraction, the translation of use cases into design and code at lower levels of abstraction is based on informal human understanding of what must be done. This also causes problems when it comes to using use cases for the specification of acceptance tests, because the criteria for passing those tests may not be adequately defined.

Chapter 8 : Traceability from Use Cases to Test Cases

A use-case scenario is an instance of a use case, or a complete "path" through the use case. End users of the completed system can go down many paths as they execute the functionality specified in the use case.

Overview Software quality is assessed along different dimensions, including reliability, function, and performance see Concepts: The Workload Analysis Document see Artifact: The workload analysis document is used by the following roles: Test Designer uses the workload analysis document to derive test cases for the different performance tests the tester see Role: Tester uses the workload analysis document to better understand the goals of the test and execute it properly the user representative see Role: Stakeholder uses the workload analysis document to assess the appropriateness of the workload, test cases, and tests being executed to assess performance The information included in the workload analysis document focuses on characteristics and attributes of following primary variables: Use Cases see Artifact: Use Case to be executed and evaluated during the performance tests Actors see Artifact: Software Architecture Document , Deployment view Performance tests, as their name implies, are executed to measure and evaluate the performance characteristics or behaviors of the target-of-test. Successfully designing, implementing, and executing performance tests requires identifying and using values to the variables that represent realistic, and exceptional values for these variables. Two types of use cases are identified and used for performance testing: Critical use cases are those use cases that will be the focus of a performance test - that is their performance behaviors will be measured and evaluated. To identify the critical use cases, identify those use cases that meet one or more of the following criteria: As the critical use cases are being identified and listed, the use case flow of events should be reviewed. Specifically, begin to identify the specific sequence of events between the actor type and system when the use case is executed. Additionally, identify or verify the following information: The frequency of execution of the use case, such as the number of simultaneous instances of the use case or as a percent of the total load on the system. Significant Use Cases Unlike critical use cases, which are the focus of performance test, significant use cases are those use cases that may impact the performance behaviors of critical use cases. Significant use cases include those use cases that meet the one or more of the following criteria: That is, one instance of an actor may interact with the target-of-test differently take longer to respond to prompts, enter different data values, etc. Consider the simple use cases below: Actors and use cases in an ATM machine. The actor "Customer" in the first instance of the use-case execution, in an experienced ATM user, but in another instance of the actor, it is an inexperienced ATM user. The experienced ATM actor quickly navigates through the ATM user-interface and spends little time reading each prompt, instead, pressing the buttons by rote. The inexperienced ATM actor however, reads each prompt and takes extra time to interpret the information before responding. Realistic performance tests reflect this difference to ensure accurate assessment of the performance behaviors when the target-of-test is deployed. Begin by identifying the actors for each use case identified above. Then identify the different actor stereotypes that may execute each use case. In the ATM example above, we may have the following actor stereotypes: Additionally, for each actor stereotype, identify their work profile, specific all the use cases and flows they execute, and the percentage of time or proportion of effort spent by the actor executing the use cases. Identifying this information is used in identifying and creating a realistic load see Load and Load Attributes below. The specific attributes and variables of the deployment system that uniquely identify the environment must also be identified, as these attributes also impact the measurement and evaluation of performance. The physical hardware CPU speed, memory, disk caching, etc. The deployment architecture number of servers, distribution of processing, etc. The network attributes Other software and use cases that may be installed and executed simultaneously to the target-of-test Identify and list the system attributes and variables that are to be considered for inclusion in the performance tests. This information may be obtained from several sources, including:

Chapter 9 : Use Case Examples -- Effective Samples and Tips

This mistake is an indicator that the use case needs to be split into more than one use case. This mistake leads to missing requirements because the more you try to cover in a single use case, the more likely you are to overlook steps and alternate paths through it.

The mechanism for determining whether a software program or system has passed or failed such a test is known as a test oracle. In some settings, an oracle could be a requirement or use case, while in others it could be a heuristic. It may take many test cases to determine that a software program or system is considered sufficiently scrutinized to be released. Test cases are often referred to as test scripts, particularly when written. Written test cases are usually collected into test suites. What is test case? IEEE Standard defines test case as follows: Page 2 Bob Binder , p. The expected result specifies what the IUT should produce from the test inputs. This specification includes messages generated by the IUT, exceptions, returned values, and resultant state of the IUT and its environment. Test cases may also specify initial and resulting conditions for other objects that constitute the IUT and its environment. Brian Marick uses a related term to describe the lightly documented test case, the test idea: The idea is to check if the code handles an error case. Defining and providing element has proved helpful in a variety of areas during the test development process. It also helps those who run the tests to verify the conformance of their implementations. For example, metadata can be used to: The name by which the test case is formally known. A brief explanation of the reason the test case was developed. A representation in words of the nature and characteristics of the test case. An unambiguous feature or module is being test Preconditions: Conditions that must be met before this test is executed. Identification of the portion of the specification tested by this test case. A list of references to relevant materials. It can help to clarify the intent or usefulness of the test case. Which product is being test. Will this case be executed manually or automatically Requirement: What feature developing activity request this test Focus: The test will be executed in which operation system Release: The test is able to be executed from which software version. An identifier that allows one to distinguish between different revisions of test case. One of an enumerated list of values that can be used to track the state of a test at a given time. Update test case in which date. The individual or organization that contributed this test case.