Case Study

Modeling Test and Bug Sequencing Support

Christian M. Adriano – [adrianoc@uci.edu](mailto:adrianoc@uci.edu)

Xinlu Tong – [xinlut@uci.edu](mailto:xinlut@uci.edu)

1. **Motivation**

System stabilization during testing phases is very chaotic and uncertain. The charts in figure-1 illustrate different testing and correction efforts undertaken. It is possible to recognize that some systems did not stabilize (e.g., the first and the last little charts). In such cases the test phase needed to be aborted and all the setup and working effort was wasted.



**Figure-1.** Each chart is a separate test phase from a different project. Red lines are the bugs opened in each day. Blue solid lines are the bugs closed in each day. Dotted blue line represents the accumulated balance of bugs opened minus bugs closed. Source – Projects managed by the one of the authors for a major European Financial institution.

This situation is especially true for tailor made software, for which it is very difficult to predict how many and where the bugs reside after all the coding was done. Since effort and time is limited, it is paramount to wisely decide where to invest effort in bug finding. Besides that, system stabilization could be at risk when team starts modifying the code inappropriately. For instance, the failure of understanding the origination of a bug might imply on modifications resulting from unapproved change requests, invalid defects or ambiguous specifications. Moreover, the failure to recognize a proper sequence of fixing can be particularly problematic in layered or highly componentized architectures. Changing the code without being aware of code dependencies might implicate in system brittleness and code smells. Therefore, system stabilization is a real issue faced by teams in the heat of the software testing phases and sprints (for Agile settings)

1. **Objective of the Case Study – “More than Justifying a Tool”**

Nonetheless we aim at specifying a set of features to resolve a practical problem, we are no completely sure about all the variables in the problem. Therefore, we need to build a supporting material that enables us to revise the problem and the solution iteratively. We need a model that could work as a support for reflection and communication with our research peers. The model we are aiming at is not to be only an abstraction of the tool requirements but also an expression of the assumptions, premises and decisions that lead us to a final set of requirements. The model is supposed to be used in conjunction with the tool in order to educate and clarify users about the results offered by the future tool. Our inspiration is similar to the work of presented in [14].

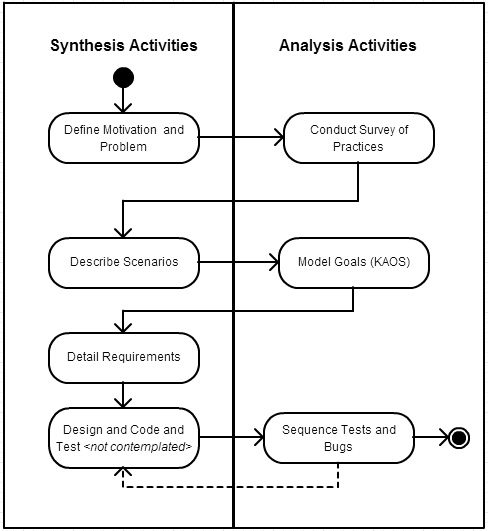
Therefore, the model should be already instantiated during requirements engineering activities in order to influence decision making during the test phase, in regards to:

* Test selection
* Bug sequencing

1. **Methodology – “Requirements Distillation Cycles”**

The stakeholders involved in system stabilization comprise practically many professional roles in the software development process – requirements engineer, developer, tester and manager. Therefore we found inevitable to conduct a survey [1] of the current practices for system stabilization from the point of view of all the different roles involved, but still some previous experience in software testing. This latter observation is necessary, because some organizations outsource software testing, hence in such particular case the visibility of testing activities is very small. We aimed at people working as a unique team from software requirements engineering through coding and software testing.

In order to interpret the results of the survey and express it in format, we choose scenarios-based technique [2]. With the scenarios we were able to identify stakeholder responsibilities, actions and results. Scenario descriptions provided synthetic unified representation to the unstructured and multi-faceted reality emerged from the surveys. Since we chose the KAOS [15] technique to formalize goals, requirements and responsibilities in an analytical format, the scenarios were a very fortunate modeling decision to bridge from the unstructured survey results to the extremely structured representation of KAOS [15]. The chart in figure-2 demonstrates our modeling approach as cycles do analysis and synthesis to distillate the requirements.



**Figure-2** Distillation cycles of Analysis and Synthesis Applied in this Case Study

1. **Summary of Activities Performed in the Case Study**

* Brief review of the literature concerning the problem of prioritizing defects and bugs
  + Such review will be largely expanded in the research project to be delivered in this class.
* Survey of current practices for sequencing tests and bugs
* KAOS model to express user goals, responsibilities and operations
* Detailed Requirements as extensions to an existing bug triage tool [3], which is already integrated to major bug tracking (Bugzilla and Jira) and code versioning (SVN) tools.
* Described the requirements and resources necessary such as
  + Decision Criteria to Sequence Tests and Bugs
  + Dependency matrices to represent code and documents traceability.

1. **Related Work**

Finding and reproducing a bug is a very costly task that normally surpasses the cost of correcting the bug when it is found. Finding bugs involve understanding the code and design decisions made, which are very time consuming even for the case of reviewing our own legacy code, one could imagine how it is learning someone’s else code. We know a lot about bug localization [3], communication through bug reporting [4], bug triage [5][6], and bug delegation [7][8].

We although know little about how teams optimally work on filtering and sequencing tests and bugs [9]. Decisions are mostly tacit and ad hoc. For instance, who better knows the code or feature takes the burden of planning the sequence of testing and correcting actions. The performance of a team to efficiently stabilize a system is directly related to such atomic decisions. In order to multiply such knowledge and skill it is necessary to understand, make it explicit and provide some tool support. Some research made important steps in that direction [10][11][12][13][14], which we might reuse as functionalities.

1. **Standard Vocabulary**

In order to guarantee mutual understanding among the different stakeholders, we provided just in the first project document an equalization of concepts.

* 1. **Accuracy**

The percentage of correct predictions in the test

* 1. **Artifact**

Tangible by-product produced during the development of software

* 1. **Bug delegation**

To assign bugs to developers to fix

* 1. **Bug Origination**

Bug origination concerns with identifying during triage the assumptions and facts which implied in the bug to be exposed (or denounced).

* 1. **Bug triaging**

To determine and assign a priority to bugs according to certain criteria

* 1. **Code**

Specific classes or methods in implementation

* 1. **Concurrence**

Potential conflicts will occur if two developers fix bugs related to the same component

* 1. **Criteria**

Standard of test case or bug sequencing from one aspect

* 1. **Dependency matrix**

Representation of the number of dependencies between different or same artifacts in the form of matrix

* 1. **Expectation**

Requirements need to be achieved or maintained by agents

* 1. **Features**

Functionalities accessible by the user, interchangeable with the term “functionality”

* 1. **Goal**

An objective a user wish to attain with the system

* 1. **Leaf nodes**

The in-degree of components, indicating the degree to which a component is depended by others

* 1. **Learnability**

Understandability of code for testers, achieved by tagging in our system

* 1. **Ownership**

The ownership of the code in terms of the developer who was responsible to develop it

* 1. **Precision**

The percentage of positive predictions that is correct in the test

* 1. **Recall**

The percentage of positive labeled instances that were predicted as positive in the test

* 1. **Responsibility**

One-to-one assignment from operation to agent in KAOS model

* 1. **Scenario**

Description of procedure and conditions of actions, responsibilities, and actors

* 1. **Specificity**

The percentage of negative labeled instances that were predicted as negative in the test

* 1. **Traceability**

The ability to link artifacts like code, bugs and goals

1. **Survey of Practices**

Our objective was to get a sense of the current practice in terms of test and bug fix sequencing. We are departing from an intuition that, in experienced teams, optimal sequences are consistently chosen. Since there is still little tool support for recommending/defining test and bug sequencing (basically, only categorizing their priority), such could be an exploration space for unfolding trade-offs such as concurrency x ownership, efficiency x learnability, changeability x traceability/accountability. In order to evaluate each practitioner experience, we also provided questions about tools and techniques adopted. The initial population we intend to target consists of people working in software consultancy companies in US, China, India, Brazil and have between 5 to 10 years of working experience. In our first run, we interviewed 10 people. Below is an excerpt of the questionnaire.

|  |  |
| --- | --- |
| **1. In how many projects did you performed test planning and running?** | **Results** |
| **Less than 6 projects** | **2** |
| **From 6 to 12 projects** | **5** |
| **More than 12 projects** | **8** |

|  |  |
| --- | --- |
| **2. How many years did you work in planning and running tests?** | **Results** |
| **None** | **0** |
| **One year** | **1** |
| **Between 4 and 8 years** | **5** |
| **Above 8 years** | **4** |

|  |  |
| --- | --- |
| **3. How would your rank you coding skills?** | **Results** |
| **I can fairly read and understand code** | **5** |
| **I can understand high level architecture, lybraries and components** | **2** |
| **I can program and debug code** | **3** |
| **I have no coding skills** | **0** |

|  |  |
| --- | --- |
| **4. Your Gender** | **Results** |
| **Male** | **6** |
| **Female** | **3** |

|  |  |
| --- | --- |
| **5. Most of your experience was in which type of organization?** | **Results** |
| **Software Consultancy, Software Factory (Contractor)** | **9** |
| **Industry, Market (Cliente)** | **1** |
| **Academia, Research Institute** | **0** |
| **Open Source** | **0** |
| **Government** | **0** |

|  |  |
| --- | --- |
| **6. In your projects, when do you and your team start writing test scripts?** | **Results** |
| **During requirements analysis** | **2** |
| **After finishing requirements analysis** | **5** |
| **During construction** | **3** |
| **Open Source** | **0** |
| **Government** | **0** |

|  |  |
| --- | --- |
| **7. Have you ever applied any traceability data in your projects?** | **Results** |
| **Scenarios x Functionalities** | **9** |
| **Functionalities x Code Components** | **4** |
| **Code x Code** | **0** |
| **None** | **1** |

|  |  |
| --- | --- |
| **8. How often do you work in projects with test automation?** | **Results** |
| **More than 75% of projects** | **0** |
| **Between 75% and 50%** | **4** |
| **Less than 50%** | **4** |
| **Never** | **2** |

|  |  |
| --- | --- |
| **9. How often do you work in projects with bug tracking tools (e.g., Bugzilla e Jira)?** | **Results** |
| **More than 75% of projects** | **6** |
| **Between 75% and 50%** | **1** |
| **Less than 50%** | **2** |
| **Never** | **1** |

|  |  |
| --- | --- |
| **10. How often do you work in Agile projects?** | **Results** |
| **More than 75% of projects** | **3** |
| **Between 75% and 50%** | **2** |
| **Less than 50%** | **4** |
| **Never** | **1** |

|  |  |
| --- | --- |
| **11. How much do you know about Test Driven Development?** | **Results** |
| **A lot, since it is part of our teams practice** | **2** |
| **Some of my teams adopted it** | **3** |
| **I only read about it** | **3** |
| **Never heard about it** | **2** |

|  |  |
| --- | --- |
| **12. Choose all the options that apply. How do you and your team define the order (sequence) of tests to run (in an integration test environment)?** | **Results** |
| **Based on dependencies among funcionalities** | **8** |
| **Based on code dependencies** | **4** |
| **Based on criteria for quality risk** | **3** |
| **Based on importance to the final user** | **7** |

|  |  |
| --- | --- |
| **13. After having the bugs delegated or chosen by developers, how serious are eventual code and feature concurrences?** | **Results** |
| **Very serious when it happens** | **3** |
| **Not so serious when it happens** | **6** |
| **Never saw happen** | **1** |

|  |  |
| --- | --- |
| **14. Choose all the options that apply. How do you and your team define the order (sequence) of bugs to fix during integration test?** | **Results** |
| **Based on dependencies among funcionalities** | **6** |
| **Based on code dependencies** | **3** |
| **Based on criteria for quality risk** | **8** |
| **Based on importance to the final user** | **4** |
|  | **0** |

|  |  |
| --- | --- |
| **15. How often your projects adopt continuous build integration?** | **Results** |
| **More than 75% of projects** | **2** |
| **Between 75% and 50%** | **5** |
| **Less than 50%** | **3** |
| **Never** | **0** |

|  |  |
| --- | --- |
| **16. How often do you have a developer fixing a code she did not implemented?** | **Results** |
| **More than 75% of projects** | **3** |
| **Between 75% and 50%** | **3** |
| **Less than 50%** | **4** |
| **Never** | **0** |

|  |  |
| --- | --- |
| **17. Remember the most difficult bug you and your team had to fix. Choose all the options that apply.** | **Results** |
| **It was in a code nobody had knowledge about** | **2** |
| **It was hidden in a third party component (COTS)** | **2** |
| **It was intermitent, hence difficult to spot** | **8** |
| **It was costly to reproduce (a lot of setup needed)** | **4** |
| **Implied in large changes to the system** | **4** |
| **Implied in major changes to requirements** | **4** |
| **It was a non-functional problem (e.g., performance, security, scaleability)** | **4** |

**Preliminary Discussion of Survey Results**

Since we are proposing an improvement for the existing set of tools and techniques, it is paramount to look for the need of it among professionals already using them. That is reason to have questions concerning test driven development, continuous integration, and use of bug tracking tools.

The results demonstrated that most of the interviewees work on very professional settings in which the core business is software development. Half of the organizations adopt Agile methods frequently and the other half not. Since Agile demands a constant prioritization of features and defects in the backlog, this is also an indication for the need to more automated tools for sequencing. Most of interviewees reported (60%) the use of bug tracking tools.

We are still waiting for more people to answer the survey and we will come with deeper analyses in the Research Project. By now the results received are enough to extract the scenarios as follow in the next section.

1. **Scenarios**

**Test Sequencing Scenario**

The survey demonstrated in questions 12 that users rely more heavily on functionalities dependencies and user satisfaction criteria. Moreover,

A scenario supporting such finding would like:

1. During interview the Requirement Engineer categorizes each requirement in level of importance.
2. Besides that, the Requirement Engineer annotates all the dependencies held by every functionality to other functionalities. Such dependencies may be related to sequence of execution or pre-conditions and post-conditions.
3. Later on, when the team has had specified which code components will support each functionality, the Requirement Engineer may revisit the dependencies to see whether some new was included among functionalities.
4. The Tester will consult such dependency information in order to discover which functionalities must be testes previous to some target functionality.
5. The mechanics of how the Tester will extract this information is to be defined as requirement.

**Bug Origination Scenario**

Origination means to “come into being”. Bug origination concerns with identifying during triage the assumptions and facts which implied in the bug to be exposed (or denounced). Bug origination does not mean finding the root-cause of the bug, which in such case is related to the piece of code associated with failures. Bug origination happens before debugging. As we described in the motivation, bug origination is also a threat to system stabilization. Below we describe the scenario of it.

1. Tester runs test cases for functionalities “Import File”, “Calculate Forecast”, “Generate Report”. For each of these functionalities Tester registers the following bugs for the previous respective functionalities: Bug-01, Bug-02 and Bug-03.
2. The Tester uses Jira for bug tracking and Porchlight for classifying bugs by means of tags.
3. During triage meeting, the following situations took place:
   1. Bug-01 is tagged as *Invalid*. The team-agreed reasoning is that before importing, some data types must present in the test database. Since one of the data types were missing, one of the elements of the file could not be imported and generated an error. For such situation, there is nothing to be coded; hence it is just the case of the tester properly setting up the test database.
   2. Bug-02 is tagged as *Change Request*. The reason was that the test script involved running “Process Forecast” functionality with incomplete data from the current month. The system was specified to solely rely on data from closed months. For such situation, there is nothing to be fixed; hence it is just the case of the Tester properly adjusting the test script.
   3. Bug-03 is tagged as *Specification Failure*. The reason was that the “Generate Report” use case specified the calculation to contemplate 252 days year, while the financial API required by the client architecture is based on a year calendar (365 days). The Tester wrote the Test Case following the 252 days year, while the Programmer was instructed by the Software Architecture to reuse the API (365 day year). Hence, a decision must be made in terms of reuse versus business rule. Such situation could not be resolved in triage meeting time, but must be brought back for final decision making.
4. Having the tags for each Bug, the Team Leader makes the following decisions:
   1. Bug-01 (*Invalid*):
      1. Tester corrects and reviews the database for more situations of setup failure in constant types.
      2. Closes Bug-01
   2. Bug-02 (*Change Request*):
      1. Tester adjusts the test scripts to reflect the data range consistent with the specification. The Team Leader may also demand the Requirement Engineer to update the specifications to make the rule explicit.
      2. Closes Bug-02
   3. Bug-03 (*Specification Failure*):
      1. Demands the Requirements Engineer to revisit the decision of library reuse and discuss the unfolding consequences for the “Process Forecast” functionality.
      2. After that, Requirements Engineer produces a revised specification for which the Test Cases must both be adjusted and rerun.
      3. Closes Bug-03

**Bug Sequencing Scenario**

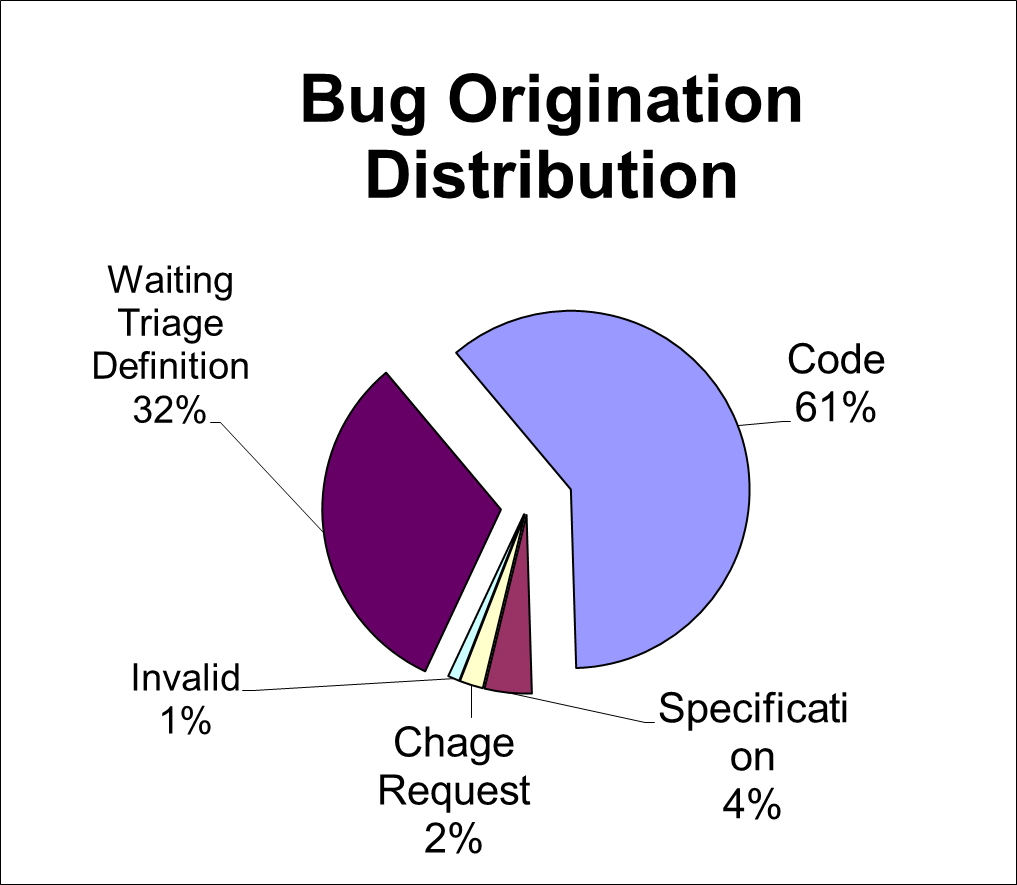
The questions 14 to 17 are crucial to understand the reality of the problem of bug sequencing. Starting with question 15 demonstrates that teams are already using continuous integration tools (70% of time using form more than 50% of projects). This means that we may infer that code is being built consistently until it reaches the testing phases. Question 16 demonstrates that most of time, programmers are fixing code from other programmers. We can infer that before starting to fix bugs, programmers have superficial knowledge about the code where the bugs resides. It is an even more sound argument to support them in defining the sequence of bug fixing. Question 17 demonstrates that hard bugs have diversified root-causes without any concentration in any group. Question 14 also demonstrates that no preferential method for prioritization is possibly adopted. Therefore, we may infer that in the absence of proper support, developers and testers decide tacitly and subjectively the sequence of bugs to fix.

The scenario of bug sequence involves the following:

* Tester updates the dependency information built upon the traceability of various artifacts
* The Tester and Programmers decide the a set of criteria they should use to prioritize bug fixing
* The tool recommends a sequence of bugs to be fixed based on the selected criteria

**Monitoring Charts**

**Distribution for Origination Categories**

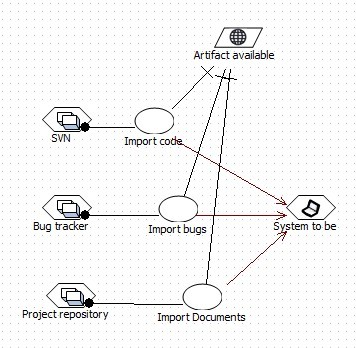


**Convergence of Testing and Fixing**

**Red = Bugs opened; Blue=Bugs closed; Dotted line = balance**

* + 1. **Application Frontier**

Since there are many existing bug tracking tools and version control tools, we don’t need to integrate them in our system. Therefore, our to-be system will be responsible for test case and bug sequencing. But we need to import artifacts like dependency matrices, bug reports and documents from other third-party systems. In this way, we can make artifacts available to support criteria of sequencing and other functions of our system.



Here in this system frontier diagram we can see that there are some operations that import certain resources, which we assigned to third party software agents like SVN, bug trackers and the project repository, and these operations come from the expectation of maintaining artifacts available. Also, this assignment is one-to-one assignment, so it can be called responsibility in KAOS model [15].

1. **Model Description**

**Process of Modeling**

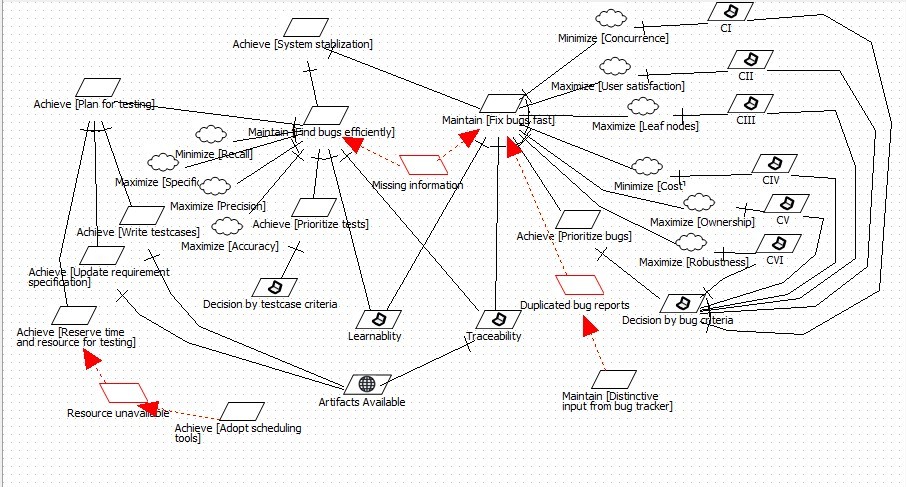
Refining goal-based model can be intuitive and convenient. At first we thought of two goals: to find bugs efficiently and to fix bugs faster. Then we asked why we need to achieve these two goals. By trying to answer this question, we realized that the ultimate goal of our system is to support system stabilization. Also, by asking how can we better find and fix bugs, we generated two subgoals respectively from those goals, one is to prioritize tests and another is to prioritize bugs. In the same way, we generated many specific goals and even requirements as well.

However, we felt that the word “efficiently” was too vague and implied some nonfunctional goals. So, after some research, we generated four nonfunctional subgoals called Minimize Recall, Maximize Specificity, Maximize Precision and Maximize Accuracy for Find bugs efficiently. These four goals are about if test cases are effective and if test cases will involve some potential defects or some seemingly problematic non-defects.

In addition, we consider the possibility that to find bugs efficiently, we can also make some plan for testing. We can achieve this by writing test cases, updating requirement specification and reserve time and resource for testing. We notice that although some of these goals have nothing to do with the system requirements, we can still benefit from them. This is one of the virtues of KAOS model. It can present the whole picture, involving behaviors in the environment.

When drawing KAOS models on the whiteboard, sometimes it is hard to create a goal without association with other goals. After we have almost done with the goal model, we found out that we still didn’t have any obstacle and we knew that it is not possible to have a perfect system at the first place. So we checked some goals concerned with resources and artifacts, and generated some obstacles about resource availability and resolutions to them.

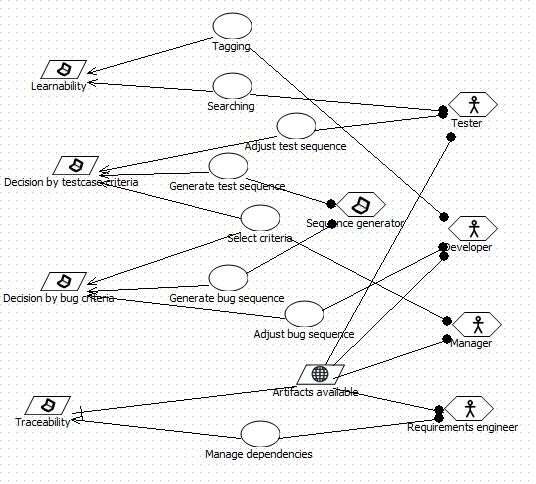
In addition, we cannot determine the relationships of goals well in the goal tree. Sometimes it seems plausible to connect a goal with a higher goal, but sometimes this connection makes less sense than the connection with lower goal. Nevertheless, we think the structure of goal tree doesn’t matter too much to our final requirements.



**Goal Model**

The goal model of our case study is rather complicated. It involves behavioral goals, requirement goals, obstacles, and expectations. The top goal of our project is Achieve[System stablization], which is to help make a robust and bug-free system. To achieve this, we have Maintain[Find bugs efficiently] and Maintain[Fix bugs fast], the first of which focuses on test process and the second focuses on bug delegation and assignment. We listed 6 criteria derived from some NFR goals that we elicited from the previous discussion and research. Criteria are represented in a form as CI, CII, etc. To achieve these criteria is also subgoal of Decision by bug criteria, which indicates sequencing and prioritizing of bugs according to those criteria, just like the requirement of Decision by testcase criteria.

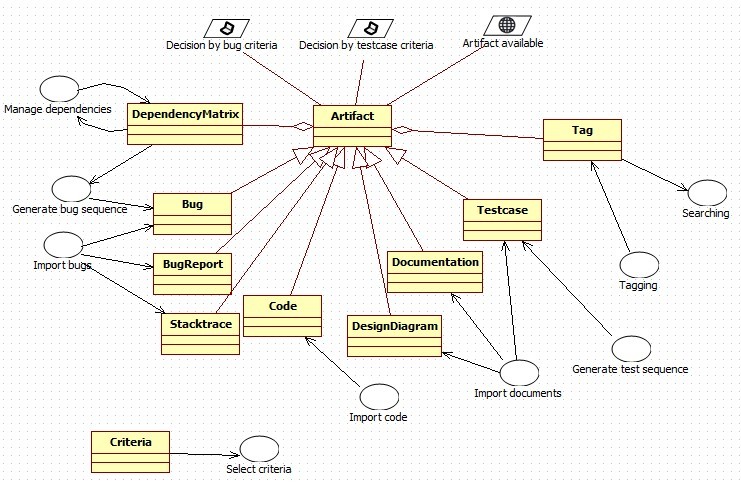
To find bugs efficiently and fix bugs fast, we need support of both Learnability and Traceability as a basis for applying criteria. We use dependency matrices that reflect dependencies of different artifacts to maintain traceability and tags added by developers to maintain learnability. To maintain traceability, we should also expect different agents work together to make artifacts available. This is represented in the form of expectation.



**Operation Model**

The operation model describes the operationalization of requirements and the assignment of operations to agents, including both human and software agents. From the model we can see that there are many operations and agents of different roles in a project. Each agent is responsible for several operations and only the expectation Artifacts available is assigned to all the human agents.

We can notice that except for sequence generating, there are two operations to adjust the sequences. This is because the automatically generated sequence may not apply to realistic situations for testers and developers, so they have the right to slightly adjust those sequences.



**Object Model**

In the object model, we have a core object called Artifact, which represents all of the by-products of the development process. There are many specific artifacts like bugs and code deriving from it. The object in KAOS model can be connected with operations with arrowed lines that indicate input and output relationship between them. We can conveniently convert this later directly to messages or functions for objects. Take “Import bugs” as an example, this operation provides input to bug, bug reports and stack traces. The line without arrow indicated concern from the goals and requirements, which provides traceability from objects directly to goals. I think this is helpful for evaluation in the design phase later.

Tags are a kind of component of artifacts. Developers add them in the process of development, and later testers can search them as a reference to better understand artifacts.

Dependency Matrix, a kind of component within Artifact, is closely related to operation “Manage dependencies”, and that operation can both provide input and receive output from the object.

* + 1. **Extracted Requirements**
  1. **Preventing Broken Tests for the Bug Origination Scenario**

Programmers are knitters. Software Architects are painters, and Requirements Engineers are sculptors. Many agree that Software Testers are detectives, but working mostly with black boxes which resemble the tools and methods of an industrial quality control era. How to better integrate detectives to our development process? How have detectives investigating the whole software process and contributing to prevent defects in the product? But deciding whether test or code will be broken is difficult to automate. Process changes are mostly prescriptive, hence neither predictive nor operative (transform how people do things). What if I requested the Knitter to annotate on the Painter’s canvas, or on the Detective’s note book, or on the Sculptor’s piecework? In other words, we could have programmers and testers cross annotating the artifacts (code, use cases and test cases) during development.

* 1. **Annotations for Traceability for the Sequencing Scenarios**

Annotations would be contextualized navigational aids in a form of comments + hyperlinks associated to any part of any artifact. We would not use Eclipse for that, but some kind of social network-like interface. The team could populate the network with logs (ex. SVN), comments, references, specifications and evidences (stack-trace, bug reports). Different visions of the network could be created. Possible outcomes:

* Early detection of wrong assumptions
* Increase collaboration during artifact creation
* More transparency and shared sense of ownership
* Smoother learning curve for new team members

Annotations could also be used for sharing, for linking, for adding content, for ranking, for categorizing. Recommend what to annotate. Request to be annotated. Groom sessions could be supported by a tool. In such sessions (maybe once a week), the team sits to review the annotations received, select some and discuss them.

An annotation repository could become a dictionary to be used during artifact composition and reporting. The user could navigate from the bug report to the stacktrace, from the specification and to the code, etc.

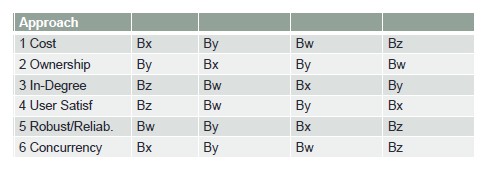
**Navegability by Auto-Completion during Bug Report -** What if the annotations provide a base to help auto-complete in the following situations: Bug reporting, Exception handling messaging (stacktrace), and topics in the documentation and code.

* 1. **Generate test case sequence**

The system should generate a sequence of test cases according to criteria of recall, specificity, precision, accuracy and others, or a combination of these criteria.

* 1. **Generate bug sequence**

The system should generate a sequence of bugs according to criteria of concurrency, user satisfaction, leaf nodes, cost, ownership and robustness (reliability), or a combination of these criteria.



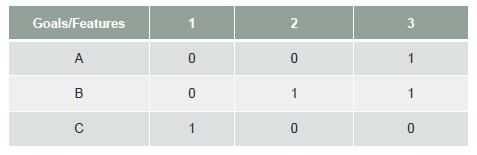
With different criteria we can obtain distinctive sequences. We can see this difference from the picture above that there is no identical sequence from 6 criteria.

* 1. **Adjust test case sequence / adjust bug sequence**

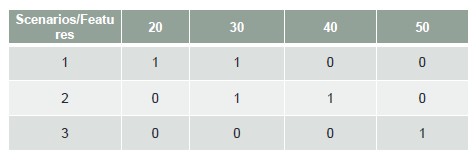
The system should enable developers and testers to adjust test case or bug sequence according to the actual situation.

* 1. **Manage dependencies**

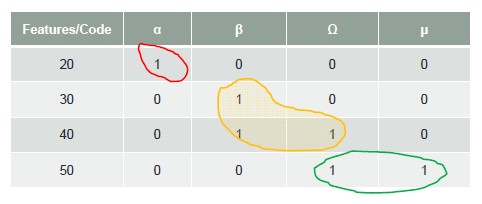
The system should enable project managers to review and modify dependency information or dependency matrices generated from artifacts. Dependency matrices represent the relation between goals, scenarios, features, code and bugs as well as that among the same artifact.



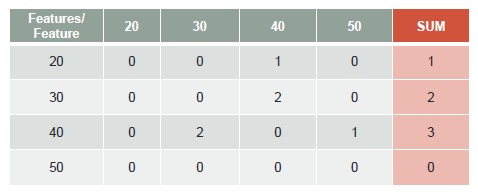
Here is an example of a dependency matrix for goals and features. From this matrix we know that Goal A depends on Feature 3, Goal B depends on Feature 2 and 3, and Goal C depends on Feature 1. Also, the number indicates the number of dependencies, which in this simple case is 1.



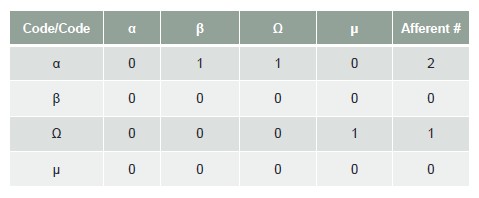
The matrix above shows dependencies between scenarios and features.



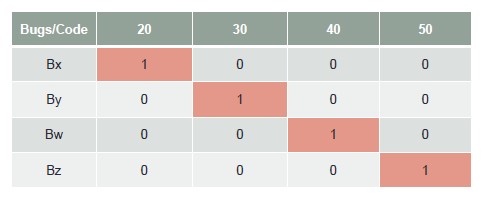
The matrix above shows dependencies between features and code.



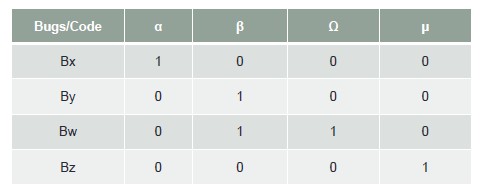
The matrix above shows dependencies between features themselves. Among them some features have multiple dependencies with each other. From the sum values in the last column of the matrix we can see that the feature which depends on others most is the one in the top of the layer structure of the system, and the one depending on others least is in the infrastructure layer or database layer. So this matrix can be used in Leaf nodes criteria.



The matrix above shows dependencies within code.



The matrix above shows dependencies between bugs and code. According to the existing assumption of our system, one bug exists in one feature. So the matrix should be a diagonal matrix.



The matrix above shows dependencies between bugs and code.

* 1. **Import code**

The system can import source code from SVN repository, which also includes information of version and ownership of change sets.

* 1. **Import bugs**

The system can import information of bugs such as bug reports and stack traces from bug tracking tools like JIRA and Bugzilla. Meanwhile, bug reports must not contain duplicated information that may lower the efficiency of the system.

* 1. **Import documents**

The system can import documents and UML diagrams from project depository.

* + 1. **Future Work in Three Horizons**

**Literature Survey - Requirements Engineering Class INF 213**

Survey the literature of techniques, models and tools related to the problem of sequencing tests and bugs. More specifically we should develop a taxonomy of solutions ranging from collaborative tools to traceability mechanisms. In such work we will try with the concepts of Dunnel to describe classes which could enable us to describe both taxonomic as well as paradigmatic classifications.

**Requirements Implementation**

Implement some of the features in Porchlight and try the tool in a realistic work setting. Gather feedback concerning both usability and usefulness to improve activities before and after bug triage.

**Model Validation through Stability Measurement and Traceability Patterns**

Record the uses practiced with the new tool and compare uses against historical data. Look for improvements in system stabilization metrics, such as performance (time to fix, time to find bugs), rework (bug reopening, bug reassignment), and quality (number of bugs).

A second step in this horizon would be to look for hints in the dependency matrices to explain improvements. Such could indicate good traceability patterns that should be suggested, replicated and enforced by tools and processes.

* + 1. **References**

[1] Goguen, J., Linde, C., Techniques for Requirements Elicitation, in proceedings of First IEEE International Symposium on Requirements Engineering (RE'93), pp.152-164, 1993.

[2] Sutcliffe, A., Scenario-based requirements engineering, in proceedings of 11th IEEE International Requirements Engineering Conference (RE'03), Monterey Bay, USA, 8-12th September 2003,pp.320-329, 2003.

[3] James A. Jones, Mary Jean Harrold, John Stasko , Visualization of Test Information to Assist Fault Localization, *In proc ICSE 2002.*

[4] Breu, S., et al., Information needs in bug reports: improving cooperation between developers and users. *In proc of CSCW 2010.*

[5]Bortis, G., van der Hoek, A., PorchLight: A Tag-based Approach to Bug Triaging,*. In proc.* of *CSCW 2011,*

[6] Bertram, D., et al., Communication, collaboration, and bugs: the social nature of issue tracking in small, collocated teams. *In proc of CSCW 2010.*

[7] Anvik, J., Hiew, L., Murphy, G.,C., 2006. Who should fix this bug? *In proc ICSE 2006.*

[8]Guo, P. J., et al., “Not My Bug!” and Other Reasons for Software Bug Report Reassignment. *In proc.* of *CSCW 2011*

[9] Xuan, J., et al., Developer Prioritization in Bug Repositories, *In proc of ICSE 2012.*

[10] Srivastava, A., Thiagarajan, J., Effectively Prioritizing Tests in Development Environment, *In proc of ISSTA 2002*

[11] Kim, S. ,Pan, K., Whitehead, Jr., E.,J., Memories of Bug Fixes, *In proc of FSE 2006.*

[12] Kim, S , Ernst M., Which Warnings Should I Fix First. *In proc of FSE 2007.*

[13] Ayewah, N., Pugh, W.,The Google FindBugs Fixit. *In proc of ISSTA 2010.*

[14] Leila Naslavsky, Hadar Ziv, Debra J. Richardson , MbSRT2: Model-based Selective Regression Testing with Traceability, *in proceedings of Third International Conference on Software Testing, Verification and Validation, pp. 89-98, 2010*

[15] Letier, E.; Kramer, J.; Magee, J.; Uchitel, S. (2008) Deriving event-based transition systems from goal-oriented requirements models, in Journal of Automated Software Engineering, vol. 15, no. 2, pp 175-206, 2008