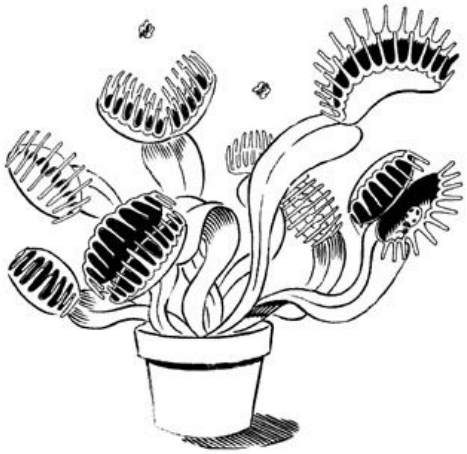

BugTrap:
A bug tracking system

Software-ontwerp
2015-2016

Iteration 2

Contents

1 Introduction

For the course *Software-ontwerp*, you will design and develop *BugTrap*, a bug tracking system. In Section 2, we explain how the project is organized, discuss the quality requirements for the software you will design and develop, and describe how we evaluate the solutions. In Section 3, we show a diagram of the domain model and explain the problem domain of the application. The use cases are described in detail in Section 4.

2 General Information

In this section, we explain how the project is organized, what is expected of the software you will develop and the deliverables you will hand in.

2.1 Team Work

For this project, you will work in groups of four. Each group is assigned an advisor from the educational staff. If you have any questions regarding the project, you can contact your advisor and schedule a meeting. When you come to the meeting, you are expected to prepare specific questions and have sufficient design documentation available. *If the design documentation is not of sufficient quality, the corresponding question will not be answered.* It is your own responsibility to organize meetings with your advisor and we advise to do this regularly. Experience from previous years shows that groups that regularly meet with their advisors produce a higher quality design. If there are problems within the group, you should immediately notify your advisor. Do not wait until right before the deadline or the exam!

To ensure that every team member practices all topics of the course, a number of roles are assigned by the team itself to the different members at the start of each iteration (or shortly thereafter in case of the first iteration). A team member that is assigned a certain role will give the presentation or demo corresponding to that role at the end of the iteration. That team member is *not supposed to do all of the work concerning his task*! He must, however, take a coordinating role in that activity (dividing the work, sending reminders about tasks to be done, make sure everything comes together, etc.), and be able to answer most questions on that topic during the evaluation. The following roles will be assigned round-robin:

Design Coordinator The design coordinator has an active role in making the design of your software. In addition to knowing the design itself, he knows *why* these design decisions were taken, and what the alternatives are.

Testing Coordinator The testing coordinator has an active role in planning, designing, and writing the tests for the software. He knows which kinds of tests are needed (scenario, unit) for testing various parts of the software, which concrete tests are chosen (success cases, failure cases, corner cases, ...), and which techniques are used (mocking, stubbing, ...).

Domain Coordinator The domain coordinator has an active role in interpreting and extending the domain model. He knows which parts of the domain are relevant for the application.

As already mentioned, the goal of these roles is to make every team member participate in all aspects of the development of your system. **During each presentation or demo, every team member must be able to explain the used domain model, the design of the system, and the functioning of your test suite.**

2.2 Iterations

The project is divided into 3 iterations. In the first iteration, you will implement the base functionality of the software. In subsequent iterations, new functionality will be added and/or existing functionality will be changed.

2.3 The Software

The focus of this course is on the *quality* (maintainability, extensibility, stability, readability,...) of the software you write. We expect you to use the development process and the techniques that are taught in this course. One of the most important concepts are the General Responsibility Assignment Software Principles (GRASP). These allow you to talk and reason about an object oriented design. **You should be able to explain all your design decisions in terms of GRASP**.

You are required to provide class and method documentation as taught in previous courses (e.g. the OGP course). When designing and implementing your system, you should use a *defensive programming style*. This means that the *client* of the public interface of a class cannot bring the objects of that class, or objects of connected classes, into an inconsistent state.

Unless explicitly stated in the assignment, you do not have to take into account persistent storage, security, multi-threading, and networking. If you have doubts about other non-functional concerns, please ask your advisor.

2.4 User Interface

You are expected to provide a functional user interface for the specified use cases, with understandable input questions and output statements. This can be as simple as a text-based command-line user interface. Although a user interface is obligatory, its appearance will not be considered for grading, i.e. you will not gain any points for making a nicer user interface. You will, however, lose many points if the quality of your software is lacking because you spent too much time on the user interface or if there is too much coupling between the user interface and your domain layer. This coupling should be kept low since "interaction with the user" is just another responsibility and like all responsibilities it should be assigned according to GRASP.

2.5 Testing

All functionality of the software should be tested. **For each use case, there should be a dedicated scenario test class.** For each use case flow, there should be at least one test method that tests the flow. Make sure you group your test code per step in the use case flow, indicating the step in comments (e.g.

`// Step 4b`). Scenario tests should not only cover success scenarios, but also negative scenarios, i.e., whether illegal input is handled defensively and exceptions are thrown as documented. You determine to which extent you use unit testing. The testing coordinator briefly motivates the choice during the evaluation of the iteration.

Tests should have good coverage, i.e. a testing strategy that leaves large portions of a software system untested is of low value. Several tools exist to give a rough estimate of how much code is tested. One such tool is Eclemma[1]. If this tool reports that only 60% of your code is covered by tests, this indicates there may be a serious problem with (the execution of) your testing strategy. However, be careful when drawing conclusions from both reported high coverage and reported low coverage (and understand why you should be careful). The testing coordinator is expected to use a coverage tool and briefly report the results during the evaluation of the iteration.

2.6 UML Tools

There are many tools available to create UML diagrams depicting your design. You are free to use any of these as long as it produces correct UML. One of these UML tools is Visual Paradigm. Instructions to run Visual Paradigm in the computer labs is described in the following file:

`/localhost/packages/visual_paradigm/README.CS.KULEUVEN.BE`

This file also contains the location of the license key that you can use on your own computer.

2.7 What You Should Hand In

Exactly *one* person of the team hands in a ZIP-archive via Toledo. The archive contains the items below and follows the structure defined below. **Make sure that you use the prescribed directory names.**

- directory `groupXX` (where XX is your group number (e.g. 01, 12, ...))
  - `doc`: a folder containing the Javadoc documentation of your entire system
  - `diagrams`: a folder containing UML diagrams that describe your system (at least one structural overview of your entire design, and sufficient detailed structural and behavioural diagrams to illustrate every use case)
  - `src`: a folder containing your source code
  - `system.jar`: an executable JAR file of your system

When including your source code into the archive, make sure to *not include files from your version control system.* If you use subversion, you can do this with the the `svn export` command, which omits unnecessary repository folders from the source tree. Make sure you choose relevant file names for your analysis and

[1] http://www.eclemma.org

design diagrams (e.g. `SSDsomeOperation.png`). You do **not** have to include the project file of your UML tool, only the exported diagrams. We should be able to start your system by executing the JAR file with the following command: `java -jar system.jar`.

Needless to say, the general rule that anything submitted by a student or group of students must have been authored exclusively by that student or group of students, and that accepting help from third parties constitutes exam fraud, applies here.

2.7.1 Late Submission Policy

If the zip file is submitted N minutes late, with $0 \leq N \leq 240$, the score for all team members is scaled by $(240 - N)/240$ for that iteration. For example, if your solution is submitted 30 minutes late, the score is scaled by 87.5%. So the maximum score for an iteration for which you can earn 4 points is reduced to 3.5. If the zip file is submitted more than 4 hours late, the score for all team members is 0 for that iteration.

2.7.2 When Toledo Fails

If the Toledo website is down – and *only* if Toledo is down – at the time of the deadline, submit your solution by e-mailing the ZIP-archive to your advisor. The timestamp of the departmental e-mail server counts as your submission time.

2.8 Evaluation

After iteration 1, and again after iteration 2, there will be an intermediate evaluation of your solution. An intermediate evaluation lasts 15 minutes and consists of: a presentation about the design and the testing approach, accompanied by a demo of the tests.

The intermediate evaluation of an iteration will cover only the part of the software that was developed during that iteration. Before the final exam, the *entire* project will be evaluated. It is your own responsibility to process the feedback, and discuss the results with your advisor.

The evaluation of an iteration is planned in the week after that iteration. Immediately after the evaluation is done, you mail the PDF file of your presentation to Prof. Bart Jacobs & Tom Holvoet and to your advisor.

2.8.1 Presentation Of The Current Iteration

The main part of the presentation should cover the design. The motivation of your design decisions *must* be explained in terms of GRASP principles. Use the appropriate design diagrams to illustrate how the most important parts of your software work. Your presentation should cover the following elements. Note that these are not necessarily all separate sections in the presentation.

1. A discussion of the high level design of the software (use GRASP patterns). Give a rationale for all the important design decisions your team has made.

2. A more detailed discussion of the parts that you think are the most interesting in terms of design (use GRASP patterns). Again we expect a rationale here for the important design decisions.

3. A discussion of the extensibility of the system. Briefly discuss how your system can deal with a number of change scenarios (e.g. extra constraints, additional domain concepts,...).

4. A discussion of the testing approach used in the current iteration.

5. An overview of the project management. Give an approximation of how many hours each team member worked. Use the following categories: group work, individual work, and study (excluding the classes and exercise sessions). In addition, insert a slide that describes the roles of the team members of the current iteration, and the roles for the next iteration. Note that these slides do not have to be presented, but we need the information.

Your presentation should not consist of slides filled with text, but of slides with clear design diagrams and keywords or a few short sentences. The goal of giving a presentation is to communicate a message, not to write a novel. All design diagrams should be *clearly readable* and use the correct UML notation. It is therefore typically a bad idea to create a single class diagram with all information. Instead, you can for example use an overview class diagram with only the most important classes, and use more detailed class diagrams to document specific parts of the system. Similarly, use appropriate interaction diagrams to illustrate the working of the most important (or complex) parts of the system.

2.9 Peer/Self-assessment

In order for you to critically reflect upon the contribution of each team member, you are asked to perform a peer/self-assessment within your team. For each team member (including yourself) and for each of the criteria below, you must give a score on the following scale: *poor/lacking/adequate/good/excellent*. The criteria to be used are:

- Design skills (use of GRASP and DESIGN patterns, ...)
- Coding skills (correctness, defensive programming, documentation,...)
- Testing skills (approach, test suite, coverage, ...)
- Collaboration (teamwork, communication, commitment)

In addition to the scores themselves, we expect you to briefly explain for each of the criteria why you have given these particular scores to each of the team members. The total length of your evaluation should not exceed 1 page.

Please be fair and to the point. Your team members will not have access to your evaluation report. If the reports reveal significant problems, the project advisor may discuss these issues with you and/or your team. Please note that your score for this course will be based on the quality of the work that has been delivered, and not on how you are rated by your other team members.

Submit your peer/self-assessment by e-mail to both Prof. Bart Jacobs & Tom Holvoet and your project advisor, using the following subject: **[SWOP] peer-/self-assessment of group $groupnumber$ by $firstname$ $lastname$**.

2.10 Deadlines

- The deadline for emailing an updated domain model to your advisor is **25 March, 2016, 6pm** (see Subsection 3.1).
- The deadline for handing in the ZIP-archive on Toledo is **15 April, 2016, 6pm**.
- The deadline for submitting your peer/self-assessment is **17 April, 2016, 6pm**, by e-mail to both your project advisor and Prof. Bart Jacobs & Tom Holvoet.

3 Problem Domain Analysis

3.1 Domain Model

For this iteration you will have to draw the domain model of BugTrap yourselves. All the changes compared to the first iteration are marked in red and it's up to you to decide how the domain model should be extended. Remember that a domain model is not the same thing as a class diagram of your system. A domain model simply expresses the client's view of the problem domain.
Since domain analysis necessarily has to be performed before system design, we expect you to complete the domain model before you start designing. The deadline for handing in a completed domain model is 25 March, 2015, 6pm. You hand in this domain model by emailing it to your adviser.

3.2 Domain Concepts

While the domain model illustrates the different concepts from the problem domain, these deserve some more explanation. Below, each of the concepts from the domain model is explained in more detail.

User : Persons that interact with BugTrap are called users. Each user has a first name, an optional middle name, and a last name. Besides these, each user must have a unique user name.

Administrator : An administrator is a user that can set up the configuration of BugTrap. He can create projects and the subsystems they are composed of. An administrator must also assign a lead developer to each project. Once a project has been properly set up, the lead developer can start assigning other developers and issuers can start submitting bug reports.

Issuer : An issuer is a user that can publish new bug reports. An issuer can be a software developer that works for the client or simply a user of some software product produced by the client. In BugTrap there is no restriction on which subsystem an issuer can submit a bug report for.

Developer Developers are issuers that are employed by the client for which BugTrap is being developed. They can work on multiple projects at the same time and for each project a developer participates in, he assumes at least one specific role. Developers can also be assigned to bug reports. Such an assignment gives them partial responsibility in the real world for solving the issue. Once the issue is solved, they can report that to BugTrap (see the use case "Update Bug Report").

Project : A project represents an independent software entity. It has a name and a description that expresses its purpose. A project also has two dates associated with it: the date the project was inserted into BugTrap and the date on which the project starts or has started. Upon creation, each project also receives a budget estimate that can be used for statistical analysis. Finally, each project is assigned a version identifier (e.g. 1.0 or 2.3) to differentiate different versions of the same software project and has an achieved milestone.

Subsystem : A project can consist of multiple subsystems and a subsystem itself can also be composed out of several subsystems. A subsystem has a name, a description and an achieved milestone. Of course a subsystem cannot be recursively part of itself. Projects cannot share subsystems.

Role : Each developer can be assigned to a project to fulfill a specific role. Each project has exactly one project leader and can have multiple programmers and testers. Some operations within BugTrap are allowed or not depending on the role a developer has within a specific project (see further). A developer can be assigned multiple roles within a project.

Bug Report : A bug report signals that something is wrong with a specific subsystem. When creating a bug report, a title and a description have to be supplied. The title is a very short statement of what is wrong and the description is a verbose version. The date of when the bug report was inserted into BugTrap is also recorded. At any point in time, a bug report has a single specific tag that describes its state: *New*, *Assigned*, *NotABug*, *Under Review*, *Resolved*, *Closed* and *Duplicate*. The exact meaning of these tags is discussed further on.
When a bug report is created, it can be marked as private. Initially, a private bug report is only visible for the creator and for the developers assigned to the project. Once the first patch is submitted for the bug report, it is visible for all the users of BugTrap. Only if a bug report is visible is it considered by the use case "Select Bug Report" for all users. This mechanism is important for security-sensitive bug reports that cannot be made public before a solution is available.
The creator of a bug report can also provide additional information if he wants to: a target milestone, a reproducible procedure to trigger the bug, a stack trace or the error message shown. All of these attributes are optional and the last three are simply textual information.

Comment To track the evolution of reported bugs, issuers can comment on them. A comment consists of some text together with the date that the comment was inserted in BugTrap. A comment is always about a single bug report. Comments can however also be replies to other comments.

Milestones A milestone is a label given to a bug report, project or subsystem. For a project or subsystem such a milestone represents the progress achieved so far. For a bug report such a milestone represents the target before which the bug report must be solved. Further on, the mechanism of milestones is explained in more detail.

Notification If an issuer indicates that he is interested in some project, subsystem or specific bug report, he will start receiving notifications in his mailbox. These notifications are messages that are sent each time new information about the bug reports of interest is inserted into BugTrap (see further).

Mailbox All the notifications that an issuer receives are kept in his mailbox. They are listed in chronological order and are kept there indefinitely.

Test A test is a piece of code that triggers a bug. Projects tracked by BugTrap can be implemented in different programming languages and to have maximal flexibility, tests are represented as text in BugTrap. A developer who is assigned as a tester to some project, can submit tests for bug reports of that project. It is up to the lead developer to accept or reject tests for some bug report (see further).

Patch Most bugs that are reported to BugTrap can simply be solved by correcting some piece of code of the corresponding project. Proposed corrections that can solve a bug are called patches. As for tests, patches can be submitted for bug reports and are represented as text in BugTrap. It is the task of programmers to propose patches and the lead developer decides if the patches indeed fix the problem.

3.3 Domain Requirements

3.3.1 Bug report life cycle

During its lifetime a bug report can have different tags associated with it, but at any point in time it has exactly one tag. Initially each bug report has the *New* tag meaning that it is a freshly created bug report. Such a bug report does not yet have any responsible developer associated with it. Once such a developer is assigned the bug report gets the tag *Assigned*. Note that it is possible to assign multiple developers, but after the first assignment the bug report at stake gets the tag *Assigned*.

When a developer that is assigned as a tester to the project of some bug report thinks he has isolated the problem described in that bug report, he can submit a test. This test should test for expected behavior, but trigger the bug. Only when a bug is *Assigned*, can such a test be submitted and it is possible to submit multiple tests.

Once at least one test is submitted for some bug report, a developer that is assigned as a programmer to the corresponding project can submit a patch. If no tests are submitted yet for a bug report, a patch is not accepted by BugTrap. A patch is a proposed modification of the program code (which is not represented in BugTrap) that possibly solves the problem. Multiple patches can be submitted for a bug report, but already after the first patch, the tag of the bug report at stake transitions from *Assigned* to *UnderReview*.

The tag *UnderReview* indicates that the lead developer will have to review the bug report and decide whether to mark it as resolved or not. When the lead developer is satisfied with the tests and one of the proposed patches, he selects that one patch and the bug report gets the tag *Resolved*. Of course BugTrap remembers which patch was selected. From then on no new tests or patches can be submitted for the bug report. If the lead developer is not satisfied with the proposed tests and patches he reverts the bug report to *Assigned* and all the tests and patches will be removed automatically. Only the lead developer can mark a bug report as *Resolved* or can revert it to *Assigned*.

As a final step, the creator of a resolved bug report can close it (i.e. assign the tag *Closed*) by specifying how satisfied he is with the solution. He does this

by giving a score on a scale from 1 to 5, where 1 means not satisfied at all and 5 means very satisfied.

A bug report that has dependencies can only have the tag *New* or *Assigned* until all of its dependencies are *Resolved* or *Closed*.

Sometimes it turns out that a bug report is no bug report at all. In such a case the lead developer of the project can decide to flag the bug report as such by giving it the tag *NotABug*. It can also occur that the same bug is reported twice. In this case the lead developer can assign the tag *Duplicate* to one of the corresponding bug reports. Only the lead developer is allowed to perform these two actions. Once a bug report is tagged as *NotABug*, *Duplicate* or *Closed*, it indefinitely remains as such.

3.3.2 Developer roles

A developer that participates in some project always assumes at least one role in that project. The role does not only describe the responsibilities of that developer in the real world, it also determines what actions he is authorized to perform within BugTrap[2].

When a project is created by an administrator, a lead developer is assigned to that project. From then on, this developer has the permission to assign other developers to that project to fulfill a specific role. Not the role of lead developer of course, since it is already taken. Only the lead developer has the permission to assign other developers to a project.

The other operations in BugTrap that are authorized depending on the role a developer has within a project, are all concerned with assigning and updating bug reports of that project. The list below determines for each role which operations in BugTrap are allowed:

Lead : The lead developer can assign bug reports to developers. He can mark bug reports as *Duplicate*, *NotABug* or *Resolved*.

Tester : A tester reviews the code of the projects he is assigned to. If he finds a bug in the software he is reviewing, he files a bug report. As the tester has good knowledge about the failing code and who wrote it, he is allowed to assign developers to the bug report (including himself). It is his responsibility to submit tests for the bug reports assigned to him.

Programmer : A programmer writes code for the projects he is assigned to. If the lead developer of a project assigned a bug report to a programmer, it is the programmer's responsibility to submit patches for the bug report.

3.3.3 Notification system

The notification system allows users to receive personally tailored notifications informing them about changes in the BugTrap system. Users can indicate whether

[2]This does not mean that you need to implement a full fledged authentication system. A user can trivially log in by specifying their user name (see the "Log in" use case). It is then sufficient to keep track of the "current user" and make authorization decisions based on that user.

they want to receive updates about specific objects of interest: projects, subsystems or individual bug reports. Users can also indicate for which kind of changes they want to receive updates:

- The creation of a new bug report
- A change in tag of some bug report
- A bug report receiving a specific tag
- A new comment for some bug report

For example, a developer can indicate that he wants to be notified about all new bugs that are submitted for a specific subsystem. Similarly, a project lead can indicate that he wants to be notified whenever a bug for his project is marked as *UnderReview*.

Every user has a mailbox that keeps track of his received notifications in chronological order. Users can also terminate their notification registrations. Users will only start receiving notifications about an object of interest after he has registered and he will stop receiving these notifications after he has unregistered. So a user will never see notifications in his mailbox about some object of interest that were sent while he was not registered.

Whenever a user is receiving notifications about some bug report of a specific project (for which he is not a developer) that has been marked as private, he will still receive, but temporarily not be able to see, notifications for that bug report. Only when a patch has been submitted for that private bug report, will the notifications become visible to him (including those that were sent while the bug report was private). Mailboxes always use the time that a notification was sent for chronologically ordering the notifications.

3.3.4 Milestones

Projects and subsystems all can be tagged with any number of *achieved milestones*, which represents their progress made so far. These milestones are unrelated to version identifiers. Milestones take the form of dot-separated numbers prefixed with the letter M (e.g. $M0.5$ or $M1.2.1$) and they are lexicographically ordered (e.g. $M0.5 < M1.2.1$ and $M1.2.1 < M1.3.0$). Initially all projects and subsystems have the default achieved milestone: $M0$.

A bug report can also have a milestone and then it is called a *target milestone*. Target milestones indicate that the problem of the bug report should be resolved before the subsystem can achieve that milestone. This also means that a new bug report must have a strictly higher target milestone than the highest currently achieved milestone of the relevant subsystem.

Achieved milestones of projects and subsystems can be incremented via the use case “Declare Achieved Milestone”, but there are some restrictions:

- A project’s or subsystem’s achieved milestone should at all times be less than or equal to the highest achieved milestone of all the subsystems it (recursively) contains.

- If a project or subsystem has a bug report that is not *NotABug*, *Duplicate* or *Closed* and this bug report has a target milestone that is less than or equal to the newly proposed achieved milestone for the project or subsystem, the increment is rejected.

3.3.5 Initialization

Your implementation of BugTrap should be able to initialize itself to the state discussed here. You are free to choose the mechanism by which you implement this. However, it is advised to discuss this with your adviser. This initialized state will be convenient for demonstrating your system. Your system does not have to (and probably should best not) be designed based on the structure of the initialization data presented here.

Users

- Administrator: Frederick Sam Curtis, user name curt
- Issuer: John Doctor, user name doc
- Issuer: Charles Arnold Berg, user name charlie
- Developer: Joseph Mays, user name major
- Developer: Maria Carney, user name maria

Projects and subsystems Figure 1 shows the initial state of projects and subsystems. It is also shown which developers are assigned to the projects with the corresponding roles.

Bug Reports

- BugReport 1:
  - title = "The function parse_ewd returns unexpected results"
  - description = "If the function parse_ewd is invoked while..."
  - creationDate = 03/01/2016
  - subsystem = SubsystemB1
  - tag = Closed
  - assignees = [maria]
  - targetMilestone = M1.1
  - tests = ["bool test_invalid_args1(){...}", ...]
  - patches = ["e3109fcc9...", ...]
  - indexSelectedPatch = 0
  - private = false
- BugReport 2:

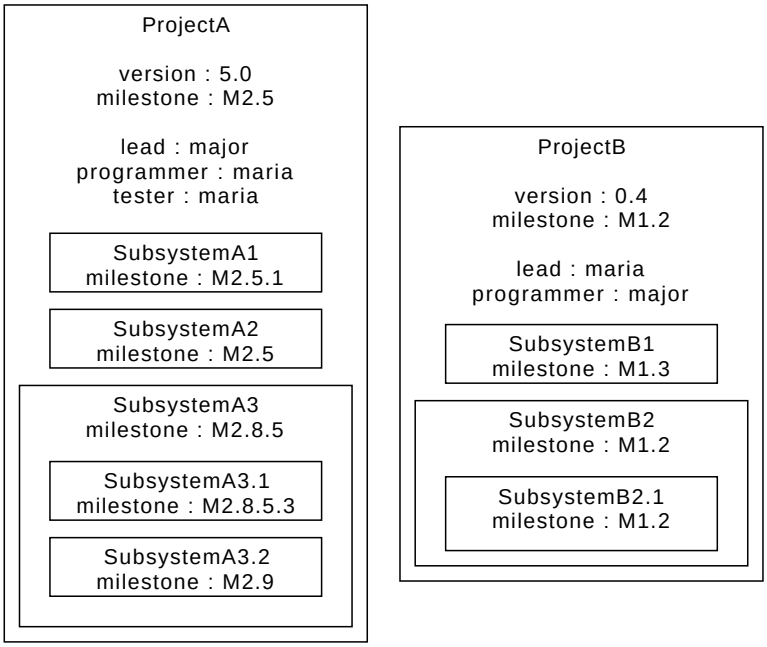

Figure 1: An initial configuration for BugTrap

  - title = "Crash while processing user input"
  - description = "If incorrect user input is entered into the system..."
  - creationDate = 15/01/2016
  - subsystem = SubsystemA3.1
  - tag = Assigned
  - assignees = [major, maria]
  - errorMessage = "Internal Error 45: The..."
  - private = false

- BugReport 3:
  - title = "SubsystemA2 feezes"
  - description = "If the function process_dfe is invoked with..."
  - creationDate = 04/02/2016
  - subsystem = SubsystemA2
  - tag = New
  - assignees = []
  - targetMilestone = M3.2
  - reproduceError = "Launch with command line invocation:..."
  - stackTrace = "Exception in thread "main" java.lang..."
  - private = true

4 Use Cases

Figure 2 shows the use case diagram for BugTrap. Use cases are used to describe the functional requirements of a system and the following sections describe the required use cases for BugTrap at a high level of abstraction. Implementation details like allowing the user to cancel a use case or handling incorrect user input are not incorporated in the use case descriptions.

4.1 Use Case: Login

Primary Actor: User.

Success End Condition: The user is successfully logged in into BugTrap.

Main Success Scenario:

1. The user indicates if he wants to log in as an administrator, issuer or developer.
2. The system shows an overview of the users of the selected category.
3. The user selects one of the shown users[3].
4. The system greets the user.

4.2 Use Case: Create Project

Primary Actor: Administrator.

Precondition: The administrator is logged in.

Success End Condition: A new project has been created.

Main Success Scenario:

1. The administrator indicates he wants to create a new project.
2. The system shows a form to enter the project details: name, description, starting date and budget estimate.
3. The administrator enters all the project details.
4. The system shows a list of possible lead developers.
5. The administrator selects a lead developer.
6. The system creates the project and shows an overview.

Extensions:

1a. The administrator indicates he wants to create a next version of an existing project. Forked projects are independent and start with achieved milestone M0 and without any bug reports.

1. The system shows a list of existing projects.
2. The administrator selects an existing project.

[3] A real world application would enforce some form of authentication here. This is out of scope for this assignment

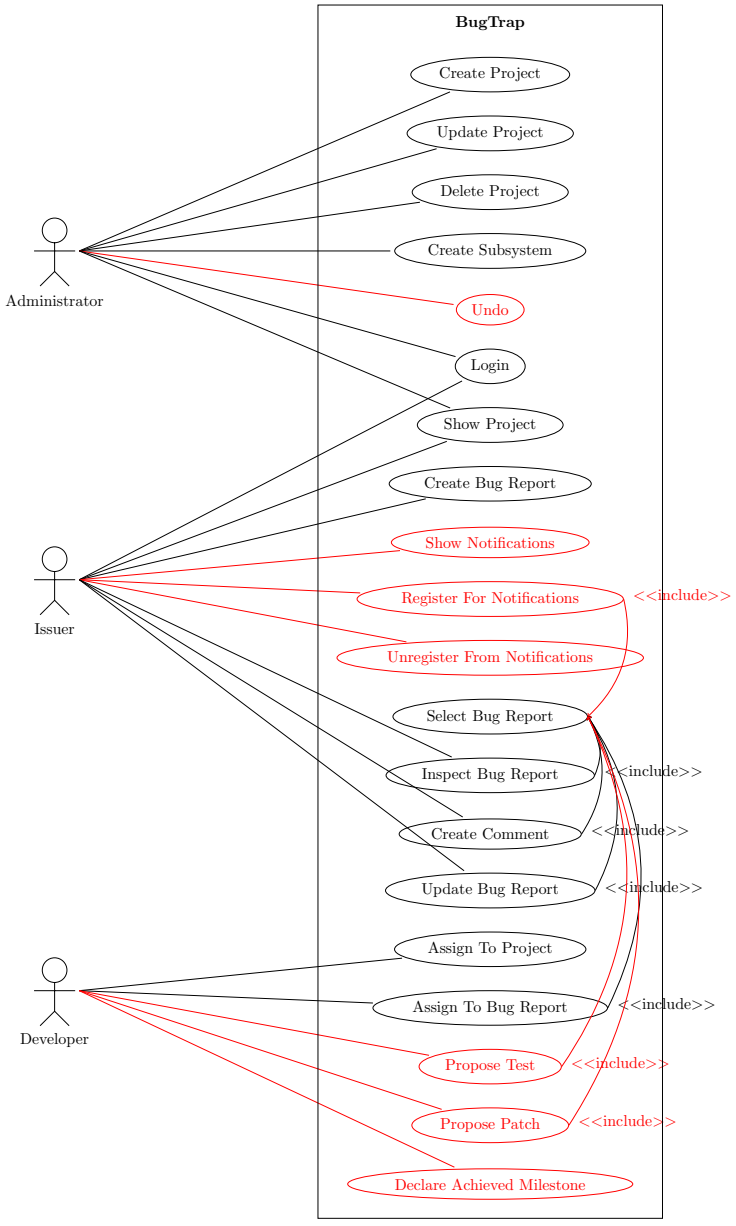

Figure 2: Use case diagram for BugTrap.

3. The system shows a form to enter the missing project details: new incremented version identifier, starting date and budget estimate.
4. The administrator enters all the missing project details.
5. The use case returns to step 4 of the normal flow.

4.3 Use Case: Update Project

Primary Actor: Administrator.

Precondition: The administrator is logged in.

Success End Condition: A project has been updated.

Main Success Scenario:

1. The administrator indicates he wants to update a project.
2. The system shows a list of all projects.
3. The administrator selects a project.
4. The system shows a form to update the project details: name, description, starting date and budget estimate.
5. The administrator modifies the details as he sees fit.
6. The system updates the project.

4.4 Use Case: Delete Project

Primary Actor: Administrator.

Precondition: The administrator is logged in.

Success End Condition: A project was deleted.

Main Success Scenario:

1. The administrator indicates he wants to delete a project.
2. The system shows a list of all projects.
3. The administrator selects a project.
4. The system deletes a project and recursively all subsystems that are part of the project. All bug reports fore those subsystem are also removed from BugTrap.

4.5 Use Case: Show Project

Primary Actor: User.

Precondition: The user is logged in.

Success End Condition: The system has produced a project overview.

Main Success Scenario:

1. The user indicates he wants to take a look at some project.

2. The system shows a list of all projects.
3. The user selects a project.
4. The system shows a detailed overview of the selected project and all its subsystems.

4.6 Use Case: Create Subsystem

Primary Actor: Administrator.

Precondition: The administrator is logged in.

Success End Condition: A new subsystem has been created.

Main Success Scenario:

1. The administrator indicates he wants to create a new subsystem.
2. The system shows a list of projects and subsystems.
3. The administrator selects the project or subsystem that the new subsystem will be part of.
4. The system shows the subsystem creation form.
5. The administrator enters the subsystem details: name and description.
6. The system creates the subsystem.

4.7 Use Case: Undo

Primary Actor: Administrator.

Precondition: The administrator is logged in.

Success End Condition: One or more use cases are reverted.

Main Success Scenario:

1. The administrator indicates he wants to undo some successfully completed use cases that changed the state of BugTrap[4].
2. The system shows a list of the last 10 completed use case instances that modified the state of BugTrap.
3. The administrator indicates how many use cases he wants to revert starting with the last.
4. The system reverts the selected use cases starting with the last completed one and, if necessary, sends the required notifications if some object of interest is modified by the undoing of a use case.

[4]Identify these use cases yourselves. The use cases "Register For Notifications" and "Unregister From Notifications" definitely change the state of BugTrap. So it should be possible to undo them. Notifications sent in the mean time however, are never revoked; i.e. they remain in mailboxes even if they were sent because of a registration that is undone. Likewise, notifications that are missed due to a canceled registration that was later undone, are never received.

4.8 Use Case: Create Bug Report

Primary Actor: Issuer.

Precondition: The issuer is logged in.

Success End Condition: A bug report is created in the system.

Main Success Scenario:

1. The issuer indicates he wants to file a bug report.
2. The system shows a list of projects.
3. The issuer selects a project.
4. The system shows a list of subsystems of the selected project.
5. The issuer selects a subsystem.
6. The system shows the bug report creation form.
7. The issuer enters the bug report details: title and description.
8. The system asks which of the optional attributes of a bug report the issuer wants to add: how to reproduce the bug, a stack trace or an error message.
9. The issuer enters the selected optional attributes as text.
10. The system asks if the bug report is private.
11. The issuer indicates if the bug report is private or not.
12. The system shows a list of possible dependencies of this bug report. These are the bug reports of the same project.
13. The issuer selects the dependencies.
14. The system creates the bug report.

4.9 Use Case: Show Notifications

Primary Actor: Issuer.

Precondition: The issuer is logged in.

Success End Condition: The system has shown the requested notifications for the issuer in chronological order.

Main Success Scenario:

1. The issuer indicates he wants to view his notifications.
2. The system asks how many notifications the issuer wants to see.
3. The issuer specifies the number of notifications.
4. The system shows the requested number of received notifications in chronological order with the most recent notification first.

4.10 Use Case: Register For Notifications

Primary Actor: Issuer.

Precondition: The issuer is logged in.

Success End Condition: From now on, the issuer will start receiving specific notifications for some selected object of interest.

Main Success Scenario:

1. The issuer indicates that he wants to register for receiving notifications.
2. The system asks if he wants to register for a project, subsystem or bug report.
3. The issuer indicates he wants to register for a project.
4. The system shows a list of projects.
5. The issuer selects a project.
6. The system presents a form describing the specific system changes that can be subscribed to for the selected object of interest:
   - The creation of a new bug report (only applicable if the object of interest is a project or subsystem)
   - A bug report receiving a new tag
   - A bug report receiving a specific tag
   - A new comment for a bug report
7. The issuer selects the system change he wants to be notified of.
8. The system registers this issuer to receive notifications about the selected object of interest for the specified changes.

Extensions:

3a. The issuer indicates he wants to register for a subsystem.
   1. The system shows a list of projects.
   2. The user selects a project.
   3. The system shows all the subsystems of the selected project.
   4. The user selects a subsystem.
   5. The use case continues with step 6.

3b. The issuer indicates he wants to register for a bug report.
   1. Include use case **Select Bug Report**.
   2. The use case continues with step 6.

4.11 Use Case: Unregister From Notifications

Primary Actor: Issuer.

Precondition: The issuer is logged in.

Success End Condition: From now on, the issuer will not receive any notifications for the specified BugTrap component anymore.

Main Success Scenario:

1. The issuer indicates that he wants to unregister from receiving specific notifications.
2. The system shows all active registrations for notifications.
3. The issuer selects a specific registration.
4. The system deactivates the specified registration for notifications.

4.12 Included Use Case: Select Bug Report

Primary Actor: Issuer.

Precondition: The system needs the issuer to select a bug report.

Success End Condition: The issuer has selected a bug report.

Main Success Scenario:

1. The system shows a list of possible searching modes:
   - Search for bug reports with a specific string in the title or description
   - Search for bug reports filed by some specific user
   - Search for bug reports assigned to specific user
   - ...
2. The issuer selects a searching mode and provides the required search parameters.
3. The system shows an ordered list of bug reports that matched the search query.
4. The issuer selects a bug report from the ordered list.

4.13 Use Case: Inspect Bug Report

Primary Actor: Issuer.

Precondition: The issuer is logged in.

Success End Condition: A detailed overview of the selected report is shown.

Main Success Scenario:

1. The issuer indicates he wants to inspect some bug report.
2. Include use case **Select Bug Report**.
3. The system shows a detailed overview of the selected bug report and all its comments.

4.14 Use Case: Create Comment

Primary Actor: Issuer.

Precondition: The issuer is logged in.

Success End Condition: The issuer has submitted a comment.

Main Success Scenario:

1. The issuer indicates he wants to create a comment.
2. Include use case **Select Bug Report**.
3. The system shows a list of all comments of the selected bug report.
4. The issuer indicates if he wants to comment directly on the bug report or on some other comment.
5. The system asks for the text of the comment.
6. The issuer writes his comment.
7. The system adds the comment to the selected use case.

4.15 Use Case: Assign To Project

Primary Actor: Developer.

Precondition: The developer is logged in.

Success End Condition: Another developer is assigned to a project.

Main Success Scenario:

1. The developer indicates he wants to assign another developer.
2. The system shows a list of the projects in which the logged in user is assigned as lead developer.
3. The lead developer selects one of his projects.
4. The system shows a list of other developers to assign.
5. The lead developer selects one of these other developers.
6. The system shows a list of possible (i.e. not yet assigned) roles for the selected developer.
7. The lead developer selects a role.
8. The systems assigns the selected role to the selected developer.

Extensions:

2a. The logged in developer is not assigned a lead role in any project:

1. The use case ends here.

4.16 Use Case: Assign To Bug Report

Primary Actor: Developer.

Precondition: The developer is logged in.

Success End Condition: A developer is assigned to a bug report.

Main Success Scenario:

1. The developer indicates he wants to assign a developer to a bug report.
2. Include use case **Select Bug Report**.
3. The system shows a list of developers that are involved in the project.
4. The logged in developer selects one or more of the developers to assign to the selected bug report on top of those already assigned.
5. The systems assigns the selected developers to the selected bug report.

Extensions:

3a. The selected bug report is of a project that the logged in developer is not involved in as lead or tester.

1. The use case returns to step 2.

4.17 Use Case: Update Bug Report

Primary Actor: Issuer.

Precondition: The issuer has updated a bug report.

Success End Condition:

Main Success Scenario:

1. The issuer indicates he wants to update a bug report.
2. Include use case **Select Bug Report**.
3. The issuer suggests a new tag for the bug report.
4. The system asks for the corresponding information for that tag.
5. The issuer provides the requested information.
6. The system gives the selected bug report the new tag.

Extensions:

4a. The issuer does not have the permission to assign the tag:

1. The use case ends here.

4.18 Use Case: Propose Test

Primary Actor: Developer.

Precondition: The developer is logged in.

Success End Condition: A test case that tests for expected functionality, but triggers a bug is added to the corresponding bug report.

Main Success Scenario:

1. The developer indicates that he wants to submit a test for some bug report.
2. Include use case **Select Bug Report**.
3. The system shows the form for uploading the test code.
4. The developer provides the details for uploading the test code.
5. The system attaches the test to the bug report.

Extensions:

3a. The developer is not assigned as a tester to the corresponding project.

1. The use case ends here.

4.19 Use Case: Propose Patch

Primary Actor: Developer.

Precondition: The developer is logged in.

Success End Condition: A patch that solves a bug is added to the corresponding bug report.

Main Success Scenario:

1. The developer indicates that he wants to submit a patch for some bug report.
2. Include use case **Select Bug Report**.
3. The system shows the form for uploading the patch.
4. The developer provides the details for uploading the patch.
5. The system attaches the patch to the bug report.

Extensions:

3a. The developer is not assigned as a developer to the project.

1. The use case ends here.

4.20 Use Case: Declare Achieved Milestone

Primary Actor: Developer.

Precondition: The developer is logged in.

Success End Condition: The new milestone is added to the selected project or subsystem.

Main Success Scenario:

1. The developer indicates that he wants to declare an achieved milestone.
2. The system shows a list of projects.
3. The developer selects a project.
4. The system shows a list of subsystems of the selected project.
5. The developer selects a subsystem.
6. The system shows the currently achieved milestones and asks for a new one.
7. The developer proposes a new achieved milestone.
8. The system updates the achieved milestone of the selected component. If necessary, the system first recursively updates the achieved milestone of all the subsystems that the component contains.

Extensions:

5a. The developer indicates he wants to change the achieved milestone of the entire project.

1. The use case continues with step 6.

8a. The new achieved milestone could not be assigned due to some constraint (see in 3.3.4).

1. The system is restored and the use case has no effect.
2. The use case ends here.

Good luck!
The SWOP Team members