Design of Software Systems

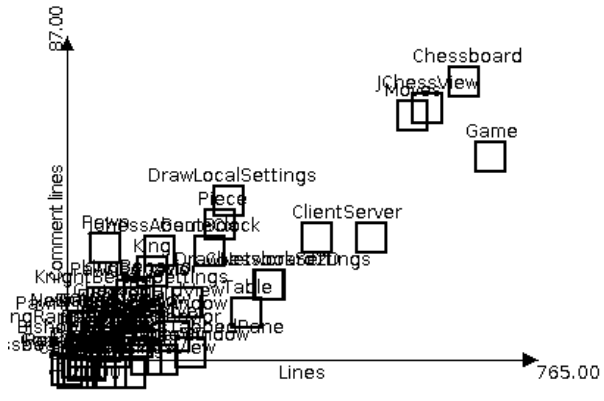
Project Assignment

1 Introduction

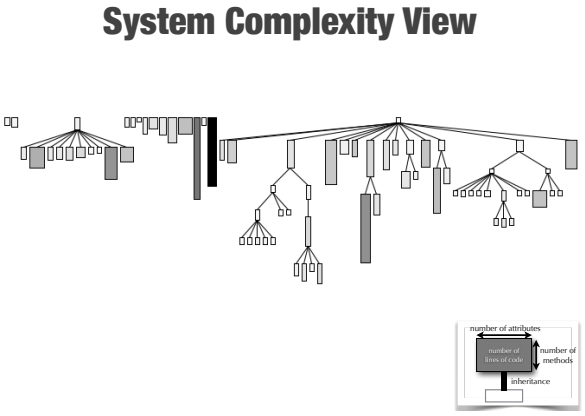
The project for the Design of Software Systems course consists of designing and implementing an extension to SonarQube[1]. SonarQube (previously known as "Sonar") is an open source software project for managing the code quality of your software projects. It can analyze the source code of your project (supporting more than 20 different programming languages, including Java) and create reports and other visualizations about various code-quality aspects. For instance, SonarQube can detect whether source code adheres to certain user-specified coding standards, it can analyze the dependencies between different components, can detect duplicated code and can even automatically find some types of source code bugs. In order to perform such analyses, SonarQube calculates various *metrics* on the source code, such as for instance the cyclomatic complexity and number of lines of code for each method.

One of the interesting aspects of SonarQube is that it is extensible. You can write a plugin to support new programming languages, new metrics or new kinds of code quality visualizations. The goal of this project is for you to design and implement a flexible new visualization plugin named *Polymorphic Views*. A polymorphic view is a visualization of some collection of resources according to one or more user-configurable metrics.

A concrete example of a polymorphic view is a scatter plot of all classes of a certain project, where the X-axis represents the number of lines of code of the classes and the Y-axis represents the number of lines of comments of the classes. Another example is a system complexity view of all classes in a certain Java package, where each class is represented by a box of which the width is relative to the number of attributes of the class, the height is relative to the number of methods and the color depends on the number of lines of code. These two examples are shown in Figure 1 below. Also see the slides of lecture 4 at `http://roelwuyts.be/OSS-1415/OSS-1415-3.Metrics_SoftwareVisualization.pdf`.



(a) Scatter plot



(b) System Complexity View

Figure 1: Example polymorphic views

[1] `http://www.sonarqube.org/`

2 Project Organization and Timeline

The project consists of three iterations. In Iteration 1, you will have to perform an analysis of SonarQube, in order to get acquainted with its design and architecture. This is an essential process to undertake, in order to start the next iteration. In Iteration 2 you will design and implement the polymorphic view extension, and in Iteration 3 you will refactor and/or further extend the extension. Table 1 shows the start and end dates for each iteration.

We expect you to team up into groups of 4 or 5 students. The group composition is up to you, but you must let us know which group you are in **before October 17**, by sending an e-mail to **oss@cs.kuleuven.be**. You will remain in the same group for all three iterations. If you have trouble finding a group, let us know as soon as possible by mailing to the same address.

After you have sent your group composition, your group will be assigned a supervisor. This supervisor is part of the course staff and is there to help you with the project. You are allowed to schedule a 1 hour meeting with your supervisor at most **once per week**. When you schedule a meeting, make sure you prepare **concrete questions** and have **appropriate design artifacts** available to support your questions (e.g. you can show class diagrams for two alternative designs that you propose, and the supervisor can help you decide which design is more appropriate). Note that the supervisor will not do the work for you: if you give insufficient input, the supervisor will not give you an answer.

After each iteration there will be a short evaluation session where you must present and defend your work to the team of supervisors. For the first and last iterations, this defense will be private, i.e., only your group members and the team of supervisors will attend. For the second iteration, the defense will be public, i.e., the other groups will also attend.

Date	Event
10 Oct 2014	Start of iteration 1
31 Oct 2014	Defense of Iteration 1 (private) & start of Iteration 2
28 Nov 2014	Defense of Iteration 2 (public) & start of Iteration 3
19 Dec 2014	Defense of Iteration 3 (private)

Table 1: Project timeline

3 Iteration 1

3.1 Assignment

The goal of this iteration is to analyze the architecture and design of SonarQube. A thorough analysis is essential in order to start the next iteration. Following the steps below will get you started on the analysis.

Running SonarQube:

1. Go to the SonarQube web site at `http://www.sonarqube.com` and download SonarQube 3.7.4 (LTS) and SonarQube Runner (version 2.4).
2. Follow the installation guide at `http://docs.codehaus.org/display/SONAR/Installing` to set up a database server on your machine (we recommend MySQL), create a new database schema and user for SonarQube and configure the SonarQube server and Runner to access this new schema.
3. Once you have the SonarQube server up and running, choose a Java open-source project at will and analyze its source code using the Runner. You can easily find open source projects in a programming language of choice on `http://www.sourceforge.net`. See `http://docs.codehaus.org/display/SONAR/Analyzing+with+SonarQube+Runner` for instructions on how to analyze projects with SonarQube Runner. Note that you may have to add the line `sonar.java.source=1.7` to the `sonar-project.properties` file, if the project uses Java 6 or 7.
4. Experiment with the SonarQube Runner and Server. See what metrics are available by default and check out the available widgets by logging in on the SonarQube web server (the default username/password are `admin`/`admin`) and clicking 'Configure widgets'.
5. Some metrics for Java projects are based on the Java source code, while others are based on the Java bytecode. Compile the Java source code of the open-source project you selected using whatever build system the project uses and then configure SonarQube Runner to analyze the resulting bytecode by adding the line `sonar.binaries=<path to bytecode>` to the project's `sonar-project.properties` file. Run the SonarQube Runner again and see what extra metrics now are available on the server.

Getting to know the SonarQube source code:

1. SonarQube is a collection of several subcomponents, some developed by the SonarQube core developers, others developed by the community. All basic components are open source and are hosted on GitHub. The components that we are interested in are the SonarQube core components, the SonarQube Java support component and the SonarQube Runner. Table 2 shows the GitHub repository URL, the right version and the corresponding commit hash for these three components. You should clone these components to a local repository using git[2] and check out the appropriate

[2]See `http://git-scm.com/documentation` for instructions on how to use git.

Component	Repository	Version	Commit
Core	`https://github.com/SonarSource/sonarqube`	3.7.4	d25bc0e
Java	`https://github.com/SonarSource/sonar-java`	2.4	7e7e633
Runner	`https://github.com/SonarSource/sonar-runner`	2.4	578fc62

Table 2: SonarQube components

commit.

2. SonarQube uses Maven as a build system. Although you will not need to build SonarQube in order to analyze it, you will need to build your extension using Maven in the next iteration, so we recommend that you already install Maven 3.0.5 or newer. In order to browse the SonarQube source code, we recommend importing the source code into Eclipse Luna (4.4.1 or newer) using `File -> Import... -> Existing Maven Projects`. It's OK if Eclipse complains about build errors, because we will not be building SonarQube anyway. When you try to import the SonarQube components, you will see that each component consists out of several subprojects, it's OK to import all of them.

3. Try to figure out what happens what happens when you run the SonarQube Runner. What code gets executed? What is the output of the Runner and where is this output stored. Which component performs the code quality analysis? How does the SonarQube server get access to the code quality results? What is the `sonar-ws-client` project and how does it work? You should try to answer all these questions by experimenting with SonarQube and investigating the source code. You should focus on the `sonar-core`, `sonar-batch`, `sonar-plugin-api` and `sonar-ws-client` projects.

4. Analyze the SonarQube source code with some the source code analysis tools you've seen in the class lectures. Focus on the `sonar-core`, `sonar-batch` and `sonar-ws-client` projects. What do you think about the source code quality?

3.2 Deliverables and Evaluation

At the end of the first iteration, your group is expected to present its analysis to the team of supervisors. The presentation should take no more than **10 minutes**. The purpose of this presentation is for you to show us that you understand the architecture and design of SonarQube at an abstract level. You should bring any presentation material that you consider useful to achieve this goal. You do not have to make slides, but if you want to, *you* must bring the necessary hardware (e.g., a laptop) for showing them to us.

The presentation should include at least the following elements:

- A high-level overview of the SonarQube architecture. This should answer questions such as
  - What happens when you execute the SonarQube Runner?
  - What is the output of the SonarQube Runner and where is it stored?

  - When is the code quality analysis performed and which component is responsible for this?
  - How can the SonarQube server access the code quality analysis results?
  - What is the `sonar-ws-client` project and how does it work?
- A domain model illustrating at least the following key SonarQube concepts and the relations between them: Resources, Metrics, Measures and Dependencies.
- A discussion about the design of SonarQube. What are the strong and weak points of the design? What tools did you use to evaluate the design?
- A discussion about the testing approach of SonarQube. Do they use scenario and/or unit tests?
- A high-level plan on how you will develop the Polymorphic Views extension.
  - Do you think SonarQube provides a clean way of developing this extension?
  - When will the polymorphic view output be generated? When running the SonarQube Runner, when visiting the SonarQube web page, or some other time?
  - Where is the necessary data for generating the polymorphic view stored? How will this data be retrieved?

4 Iteration 2

4.1 Assignment

The goal of iteration 2 is to design and implement the Polymorphic Views extension as a SonarQube plugin. For this iteration, the extension must support a scatter plot view and a system complexity view.

A scatter plot view displays two metrics for a set of SonarQube resources, in a 2D Cartesian coordinate plane (see Figure 1a). Each resource should be represented by a rectangle. The position of each rectangle depends on the values of the metrics for the corresponding resource. The height, width and background color of these rectangles can either be fixed or can each depend on a different SonarQube metric. The X and Y axes should be drawn as part of the graph and both axes should have a label indicating the the metric displayed on that axis, and the minimum and maximum value of the metric over all resources in the view should be displayed at the start and end of the axis respectively.

A system complexity view displays the inheritance structure of a set of classes (see Figure 1b). Each class is represented by a rectangle. The class(es) at the root of the hierarchy are at the top of the view and each subclass is displayed at the level below its parent. A line should be drawn between each subclass and its parent. The name of each class is displayed above its rectangle (not shown in Figure 1b). As in the scatter plot view, the height, width and background color of the rectangles can either be fixed or can each depend on a different SonarQube metric. The height of a level is determined by the tallest rectangle in that level. The horizontal ordering of classes within each vertical level of a hierarchy should be alphabetical.

Both types of views should be customizable by the end user, based on the parameters described below.

Name	Description
`resources`	This parameter determines what type of resources are to be displayed in the view. If the parameter is equal to "`classes`", then only classes should be displayed in the view. If the parameter is equal to "`packages`", then only packages should be displayed in the view. Other values are invalid.
`parent`	This parameter determines which resources are to be displayed in the view. Its value should be the key of a parent resource, and all resources that are hierarchically below this parent resource should be displayed in the view. For instance, if the specified value is the key of a certain project, than all resources that are part of this project are to be displayed in the view (subject to the constraint set by the `resources` parameter). Each resource should be represented by a rectangle.
`type`	This parameter determines the type of plot to display. If the parameter is equal to "`scatter`", a scatter plot should be displayed. If the parameter is equal to "`syscomp`", a system complexity view should be displayed. If a system complexity view is chosen, the only valid value for the `resources` parameter is "`classes`"

`xmetric`	This parameter is only valid for scatter plots, and determines what SonarQube metric is used for the x-axis of the scatter plot. The value should be the key of a valid SonarQube metric.
`ymetric`	This parameter is only valid for scatter plots, and determines what SonarQube metric is used for the y-axis of the scatter plot. The value should be the key of a valid SonarQube metric.
`size`	The format of this parameter is `<int>x<int>`. This parameter is only valid for scatter plots, and determines the size (width x height) in pixels of the output image. For system complexity views, the image size should automatically be determined to match its contents' size.
`boxwidth`	This parameter determines the width of the rectangles representing resources. If the value is an integer X, the rectangle width should be equal to X pixels for each resource. If the value is a string K, the rectangle width should be equal to the value of the SonarQube metric with key K for the resource represented by the rectangle.
`boxheight`	This parameter determines the height of the rectangles representing resources. If the value is an integer X, the rectangle height should be equal to X pixels for each resource. If the value is a string K, the rectangle height should be equal to the value of the SonarQube metric with key K for the resource represented by the rectangle.
`boxcolor`	This parameter determines the background color of the rectangles representing resources. If the value has the format `r<int>g<int>b<int>`, then each rectangle should have the corresponding RGB background color (255 is the maximum value for each color component). If the value has the format `min<float>max<float>key<string>` then the rectangle background color should be the grayscale color resulting from linearly interpolating the specified metric value between the given minimum and maximum values for the resource represented by the rectangle (the min value should result in white, and the max value should result in black).

You should choose a sensible default value for each parameter, to use when the parameter is unspecified or invalid.

4.2 Extension points

When designing your extension, you should take into account the following critical extension points:

Layout It should be easy to add more layout types besides the scatter plot and system complexity view. For instance, it should be easy to add a "Coffee Flavor Wheel" type layout[3].

Resource representation It should be easy to add a parameter to let users choose

[3]See `http://livingarchive.inn.ac/visualisations/show/511a7c2fdefba03521000001`

the representation used for resources. For instance, users could choose to represent resources by circles instead of rectangles.

Resource visualization properties Currently, there are two kinds of resource visualization properties: (1) fixed properties and (2) SonarQube based properties. For instance, if the boxwidth parameter is set to an integer X, the width of each rectangle in the view is fixed to X pixels. If the boxwidth parameter is set to a string K, the width of each rectangle in the view depends on the value of the metric with key K for that resource. It should be easy to add new kinds of resource visualization properties. For instance, a user might want to aggregate two SonarQube properties into one (e.g. by averaging them or summing them up).

Lines Currently, the lines connecting rectangles in the system complexity view are straight lines. It should be easy to add a parameter to let users choose between different kinds of lines. For instance, users might want to use orthogonal lines instead of straight lines.

Drawing backend It should be easy to change the backend technology used to draw the view. For instance, you might for the moment choose to draw the view using Java2D[4], but it should be easy to switch to a different drawing backend such as JFreeChart[5].

4.3 Implementation, Documentation and Testing

Although the main emphasis of this course is on the software *design* of the extension, we expect you to properly document and test your code. At the minimum, we expect a JavaDoc description of each class and interface you create. We also expect you to apply defensive programming and to create appropriate tests, as you have been taught in this course and the previous programming courses you took. It is important to have a proper methodological programming approach for this course. For instance, decide what you will test, how you will test it and when to write the tests, and stick to your methodology throughout the project.

4.4 Source code repository

We expect you to use Git to manage your source code. You can apply for a free student account at `https://education.github.com/`, so you can create a private source code repository on Github that you can share with your team members. It is **not** allowed to share your repository or any part of the project with other groups. You must give your assigned supervisor access to the repository (this will be discussed during one of the first meetings you schedule with your supervisor).

[4] `http://docs.oracle.com/javase/tutorial/2d/`

[5] `http://www.jfree.org/jfreechart/`

4.5 Getting started

To help you get started on the project, a zip-file has been uploaded to Toledo. This zip-file contains two folders: `sonar-3.7.4` and `sonar-polymorphic-views`.

The `sonar-3.7.4` folder contains a binary distribution of SonarQube, similar to the one you can download from the SonarQube website. We have, however, made a few important changes to the SonarQube code to facilitate the development of the extension. For instance, the default SonarQube distribution does not keep track of inheritance dependencies between classes, while our version does. Hence, when trying out your extension, it is important that you are running our version of SonarQube, instead of the standard distribution that you can download from the website. Don't forget to configure the SonarQube installation to access your database, as described at `http://docs.codehaus.org/display/SONAR/Installing#Installing-installingWebServerInstallingtheWebServer`.

The `sonar-polymorphic-views` folder provides a project template that you can start from, to implement the extension. You should place this folder somewhere on disk and then import the Maven project inside of it into Eclipse (using `File -> Import... -> Existing Maven Projects`). You will then see that this template contains a `polymorphicviews` package and a `sonarfacade` package. The `polymorphicviews` package is a starting point for your extension. It contains three classes:

`PolymetricViewsPlugin` This class is what defines your plugin. It contains just one method, named `getExtensions()`, which lists other classes that are to be registered with the SonarQube server (for use in the IoC container). You most likely do not have to change the implementation of this method.

`PolymetricViewsChart` SonarQube contains a `ChartsServlet` that can conveniently be used to support your extension. By implementing the `org.sonar.api.charts.Chart` interface and returning an identifier for your chart in the `getKey()` method, you can implement a new kind of chart. For instance, the `PolymetricViewsChart` class returns "`polymorphic`" from its `getKey()` method, which means that the `generateImage()` method of this class will be called when users access the URL `http://localhost:9000/chart?ck=polymorphic`. Any further URL parameters added after the `ck` parameter are made available through the `ChartParameters` argument of the `generateImage()` method. The image returned by this method will be displayed in the browser. The given `PolymetricViewsChart` class provides a basic implementation of a new chart type, and you should adapt this class to call your own Polymorphic Views implementation.

`PolymetricViewsWidget` This class defines a SonarQube widget. As you have probably discovered in iteration 1, widgets are used to extend the SonarQube web server with new visualizations. The `PolymetricViewsWidget` class simply defines a name and description for your widget, and refers to a Ruby template that determines what the widget will look like. The current Ruby template simply displays the image returned by accessing `http://localhost:9000/chart?ck=polymorphic&resources=classes&parent=<current-project-name>&type=scatter&xmetric=lines&ymetric=comment_lines`. Although this template is extremely basic, you can use it to test whether your extension works. You are not expected to extend the template.

The `sonarfacade` package contains a Facade for accessing the SonarQube database from your extension. There is one implementation provided, that accesses the database through the SonarQube web service. Although it is not very common to use a web service for communication between two components running on the same server, in this case it is a good option because the web service is the only stable, documented interface to the SonarQube database that can easily be accessed from Java. You can use the Facade by instantiating the `WebServiceSonarFacade` class and subsequently using its methods to perform queries.

4.6 Installing your plugin in SonarQube

As described in Section 4.3, you should of course test your extension using a proper testing methodology including unit tests. However, at some point during development, you will want to intall and test your plugin in the SonarQube server. In order to install your plugin in SonarQube, you should follow these steps:

1. Make sure Maven 3.0.5 or newer is installed on your system and that the maven binaries are in your path environment variable.
2. Open a terminal and go to the root of your project directory (i.e., the `sonar-polymorphicviews` folder if you are using the provided template).
3. Run `mvn clean`. This will "clean" the files and directories generated by Maven during the build process.
4. Run `mvn install`. This compiles the source code of your plugin and packages the resulting bytecode into a jar-archive. The resulting jar-file will be available at `<project-root>/target/sonar-polymorphic-views-1.0-SNAPSHOT.jar`.
5. Copy the jar-file into `<sonar-3.7.4-path>/extensions/plugins/`.
6. (Re)start the SonarQube server. Your plugin will be initialized when starting the server. If there are problems, you might see an error message appear on the command line when starting the server, or in the log file at `<sonar-3.7.4-path>/logs/sonar.log`.
7. Before you start testing, make sure you have analyzed a Java project using the SonarQube runner. Also make sure you have enabled binary analysis as well as source code analysis.
8. In order to test your chart, go to `http://localhost:9000/chart?ck=polymorphic&<more-parameters>` with your browser and add the necessary parameters to test whatever you want to try out.
9. In order to test the widget, go to `http://localhost:9000/`, log in using `admin`/`admin` as the username/password, click on "Configure widgets" and add the Polymorphic Views widget to a dashboard. The view should now show up when you open the dashboard.

4.7 Deliverables and Evaluation

At the end of iteration 2, you should have a working implementation of the assignment described in Section 4.1. You must present your solution to the team of supervisors and the other students in the lecture session of **November 28**. You should prepare slides to explain your design. If you need a laptop to show the slides, let your supervisor know before November 19. Your presentation should take a maximum of **10 minutes** (we advise you to rehearse what you want to say to stay within this time limit).