Requirements Specification

CENT
27.05.2013

Table of Contents

1. Introduction

1.1. Purpose

The purpose of this document is to present a detailed description of the proposed java application for displaying graph classes and their inclusions (henceforth referred to as *the system*). This document will be used as a contractual agreement between the client and developer. This document complies with the IEEE.830-1998 Software Requirements Specification.

1.2. Scope of Project

The software to be developed is an upgrade of the current java application found on graphclasses.org. All current functions should be present in the upgraded system but the user interface should be improved. The new system should use the yFiles library (http://www.yworks.com/en/index.html) for drawing the hierarchal view of the graph classes. The current code base should be refactored to allow for easier maintenance. The scope of the refactoring should cover, at a minimum, the separation of data access and user interface.

1.3. References

IEEE 830-1998 Recommended Practice for Software Requirements Specifications.

1.4. Overview of Document

The next chapter will contain a general description of the proposed functionality of the software. This is followed by more specific requirements in chapter 3. This includes details of the required functions for the end user as well as infrastructural requirements.

2. Overall Description

2.1 Product Perspective

The system provides a user interface for a database of the different graph classes and their sub- and super-classes. The existing system has only a rudimentary user interface that does not include basic functions such as zooming in and out of a graph class hierarchy. It also uses a very rudimentary algorithm for positioning the arrows used to represent a hierarchal relationship. The new system should provide a better user experience by utilising the yFiles library to display the graph hierarchy.

2.2 Product Functions

The current system allows the end user to browse the database and select a graph class to show the hierarchal structure of the sub- and super-classes. The database includes a set of problems and the relative complexity of these problems for each graph class. The user can select a graph type to draw and include the sub-classes, super-classes or both. The graph can then be coloured to show the related complexity when solving a problem for the graph. The following colouring system is currently in place:

Green: Problem is solvable in linear time.
Dark Green: Problem is solvable in polynomial time.
Red: Problem is NP-Complete.
White: Problem is open.

The user can also select two graph classes in order to show the inclusion relationship between the two classes. If none is found then the common sub- and super-classes are shown.

A computational problem can also be selected in order to show the P/NP boundary classes. These are shown in lists along with a list of the classes for which the problems are still open.

All functions must be maintained in the upgraded system. The colouring system should be implemented so that adding a new colour for a complexity class does not require a redesign.

2.3 User Characteristics

The system will be publicly available meaning the user will be a large range of different skill levels. Hence the system should be usable without any training. It is expected that the user has knowledge of graph classes and the related terminology. The system owner is expected to be computer and internet literate in order to maintain the system.

2.4 Constraints

The following constraints have been placed on the development project

- yFiles library should be used for drawing graphs
- The program should run as a java applet if performance is acceptable (the client will decide what is acceptable after seeing initial test results).

2.5 Assumptions and Dependencies

No assumptions have been made about the infrastructure. The system has no external dependencies with the exception of the Java runtime (version 1.6 or above).

2.6 Apportioning of Requirements

All requirements must be implemented in the first release of the product.

3. Requirements Specification

This section provides a detailed description of the system requirements, including end-user and infrastructural requirements.

3.1 External Interface Requirements

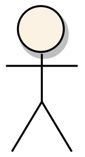
The system has no dependencies to external components but must be hosted on the graphclasses.org website (as a download or an applet, depending on the decided implementation).

3.2 Functional Requirements

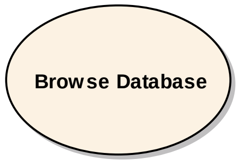
This section provides a description of all functional requirements. This refers to the functions provided to the end user by the system.
The required functionality is described using UML use case diagrams. To enable easier interpretation these diagrams are accompanied by a detailed description where necessary. Use case diagrams depict the individual scenarios that can be invoked by the end user.

3.2.1 Diagram Legend

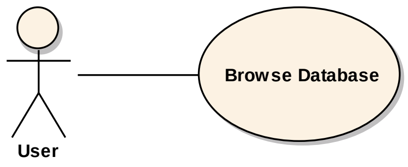
The following describes the symbols used in the use case diagrams. This is not an exhaustive list but contains all the symbols used in this document.



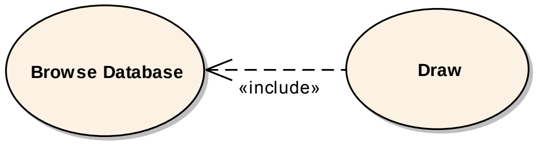
Actor: An actor represents an entity that can start a process. This can be a user, administrator, or even another system. For the proposed system the only actor is the end user.



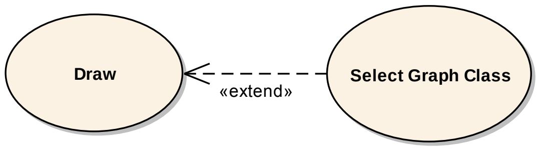
Use Case: An oval represents a use case. The name of the use case is shown in the oval.



Usage line: A line between an actor and a use case indicates that the actor can initiate the use case.

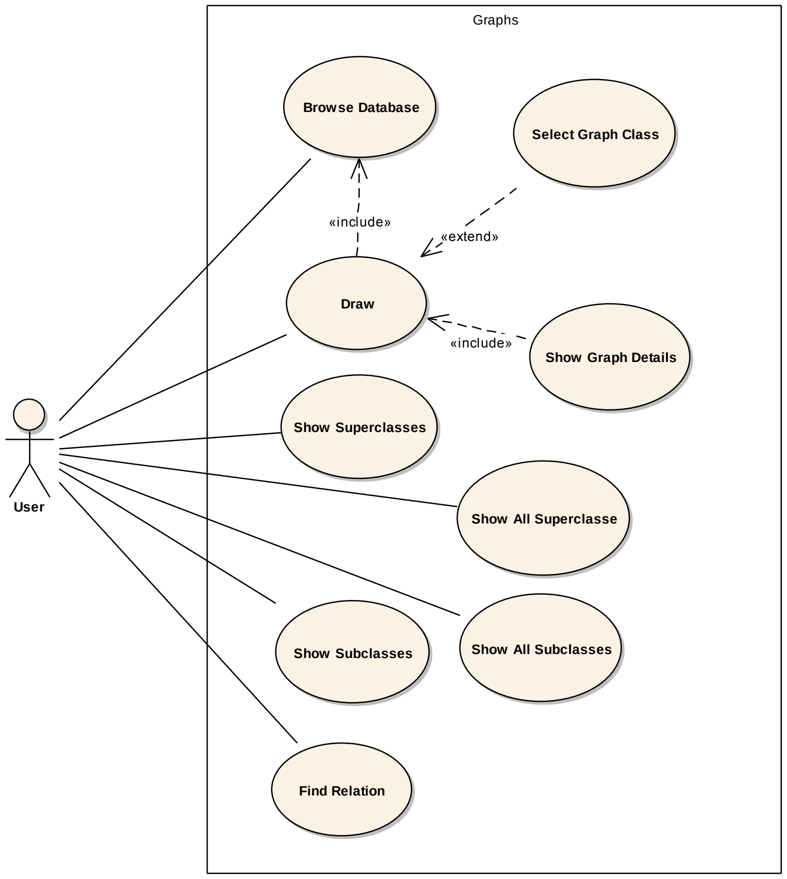


Include arrow: An include arrow between two use cases indicates that the use case at the arrows origin can optionally be invoked when the other use case is invoked.



Extend arrow: This indicates a 'special case' scenario. In the image to the left the use case ‘Select Graph Class’ is only invoked if no graph type has already been selected.

3.2.2 Graph Use Cases



3.2.2.1 Browse Database

Description:	A dialog is shown in which the user can browse through all graph types available in the database. Upon selection of a graph class a list of the complexity of each of the problems in the database, with respect to the graph class, is shown, as well as lists of the sub- and super-classes, and the equivalent classes.
Trigger:	Execution of the *browse database* user interface command.
Precondition:	None
Postcondition:	None.
Normal flow:	• User executes the *browse database* command. • A request is made to the data layer to get all the available graph types. • The graph classes are presented to the user as a list ordered alphabetically. • A graph class is selected by the user. • The details (problem complexity, sub-/super-classes etc.) are displayed to the user.

3.2.2.2 Draw

Description:	The selected graph is drawn on the canvas. The user can select to draw the sub-classes, super-classes or both. A graph type must have been selected in the browse database dialog, or the select graph dialog (see 3.2.2.3) is shown and a graph type is selected there.
Trigger:	Execution of the *draw* user interface command.
Precondition:	A graph type has been selected.
Postcondition:	None.
Normal flow:	• User executes the *draw* command. • A check is made to see if a graph class has been selected in the browse database dialog. If not then the select graph dialog is shown (use case 3.2.2.3). • The user selects whether to draw sub-classes, super-classes or both. • The graph hierarchy is rendered to the application canvas.

3.2.2.3 Select Graph Class

Description:	This is required if the user directly selects the draw command without first browsing the database and selecting a graph class from there.
Trigger:	Execution of the *draw* user interface command when no graph class is pre-selected.
Precondition:	No graph class has been selected.
Postcondition:	A graph class has been selected or the command has been cancelled.
Normal flow:	• A request is made to the data layer to get all the available graph classes. • The graph classes are presented to the user as a list ordered alphabetically. • A graph class is selected by the user. • The user confirms selection and control is returned to the invoking command.

3.2.2.4 Show Superclasses

Description:	This command can be invoked after a graph hierarchy has been drawn. The command adds the direct superclasses of the selected graph class to the canvas (the superclasses of the graph class selected on the canvas, not necessarily the class originally selected when the hierarchy was drawn).
Trigger:	Execution of the *show superclasses* user interface command.
Precondition:	A graph class is selected on the application canvas.
Postcondition:	None.
Normal flow:	• The user selects a graph class on the canvas by clicking. • User executes the *show superclasses* command. • The superclasses are added to the canvas.

3.2.2.5 Show All Superclasses

Description:	This command can be invoked after a graph hierarchy has been drawn. The command adds the complete superclass hierarchy of the selected graph class to the canvas (the superclass hierarchy of the graph class selected on the canvas, not necessarily the class originally selected when the hierarchy was drawn).
Trigger:	Execution of the *show all superclasses* user interface command.
Precondition:	A graph class is selected on the application canvas.
Postcondition:	None.
Normal flow:	• The user selects a graph class on the canvas by clicking. • User executes the *show all superclasses* command. • The entire hierarchy of superclasses is added to the canvas.

3.2.2.6 Show Subclasses

Description:	This command can be invoked after a graph hierarchy has been drawn. The command adds the direct subclasses of the selected graph class to the canvas (the subclasses of the graph class selected on the canvas, not necessarily the class originally selected when the hierarchy was drawn).
Trigger:	Execution of the *show subclasses* user interface command.
Precondition:	A graph class is selected on the application canvas.
Postcondition:	None.
Normal flow:	• The user selects a graph class on the canvas by clicking. • User executes the *show subclasses* command. • The subclasses are added to the canvas.

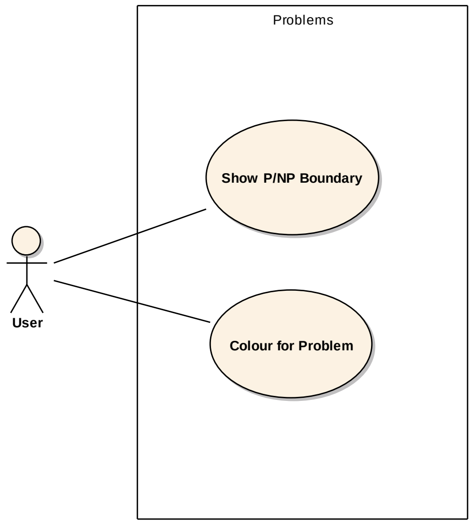
3.2.2.7 Show All Subclasses

Description:	This command can be invoked after a graph hierarchy has been drawn. The command adds the complete subclass hierarchy of the selected graph class to the canvas (the subclass hierarchy of the graph class selected on the canvas, not necessarily the class originally selected when the hierarchy was drawn).
Trigger:	Execution of the *show all subclasses* user interface command.
Precondition:	A graph class is selected on the application canvas.
Postcondition:	None.
Normal flow:	• The user selects a graph class on the canvas by clicking. • User executes the *show all subclasses* command. • The entire hierarchy of subclasses is added to the canvas.

3.2.2.8 Find Relation

Description:	A dialog is shown to the user with two lists, each containing all of the available graph classes. The user can select a graph class from each list and then show the inclusion relationship between the two graph classes (including the inclusion path from one class to the other). If no inclusion relationship is found then the common sub- and super-classes are shown.
Trigger:	Execution of the *find relation* user interface command.
Precondition:	None.
Postcondition:	None.
Normal flow:	• The user invokes the *find relation* command • The user selects two graph types and confirms • The relationship information between the two classes is shown to the user.

3.2.3 Computational Problem Use Cases



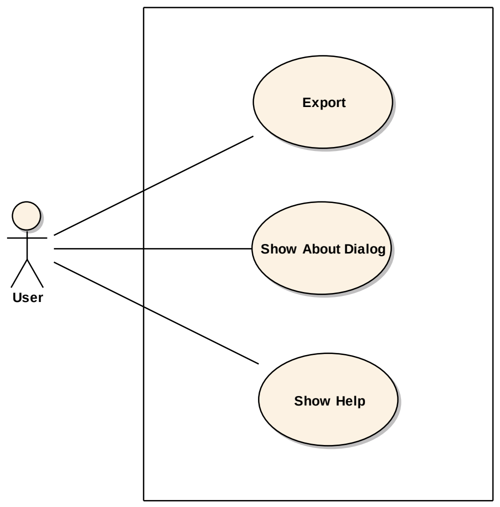
3.2.3.1 Show P/NP Boundary

Description:	Shows the boundary classes for a selected problem (NP-Complete, Open, Max. P) in three lists.
Trigger:	Selection of a problem from the *boundary/open classes* flyout menu.
Precondition:	None.
Postcondition:	None.
Normal flow:	• The user selects a computational problem from *boundary/open classes* flyout menu. • A dialog is shown with three lists of graph classes for NP-Complete, Open and Maximal P.

3.2.3.2 Colour for Problem

Description:	Colours the current graph hierarchy according to the complexity class of a selected problem. Each graph class is coloured green if the problem can be solved for it in linear time, dark green for polynomial time and red if the problem is NP-Complete for that graph class. If the problem cannot be solved for a graph class then the class is left white.
Trigger:	Selection of a problem from the *colour for problem* flyout menu.
Precondition:	A graph hierarchy has been drawn on the application canvas.
Postcondition:	None.
Normal flow:	• The user selects a computational problem from the *colour for problem* flyout menu. • The graph classes are all coloured using the complexity class appropriate for the selected problem.

3.2.4 Miscellaneous Use Cases



Export	Exports the currently displayed hierarchy in PostScript, graphML or SVG format.
Show About Dialog	Displays the about dialog with information about the application.
Show Help	Displays the online help.

3.3 Performance Requirements

The user interface should remain responsive, rendering graph hierarchies should be at least as fast as in the existing program.
If accessing the backing data is moved to the server (this is only the case if the system is implemented as an applet), then average data access times should not exceed 2 seconds.

3.4 Logical Database Requirements

The current data storage system (xml files) will also be used in the new system.

3.5 Design Constraints

The resulting system should run on Java runtime 1.6 or above.

3.6 Software System Attributes

3.6.1 Availability

The system must be available at all times with the only exception being downtime on the hosting system. During downtime a maintenance page should be shown to any visitors to the site.

3.6.2 Security

There are no security requirements for the new system.

3.6.3 Reliability
The system should remain stable in the case of incorrect user input. Any errors that occur from the yFiles drawing library should be handled and so should not cause system instability.

3.6.4 Maintainability
The system should not require regular maintenance and should accept changes to the xml data storage files (e.g. adding a new graph type) without requiring any changes. Optionally, adding a new user interface language should require minimal effort.

3.6.5 Usability
The system should be easy to use with a simple but functional user interface. The current user interface requires duplicate input in some places (e.g. 'draw' in the browse database dialog – this opens a dialog to select a graph class although one has already been selected), this should be improved wherever possible.

3.6.6 Autonomy
The system should run indefinitely as long as the server is available. In the case that a java applet is created the server should run unattended for at least seven days, allowing for a weekly maintenance plan.

4. Appendix

This section provides a dynamic list of all features or functions that have been defined after the finalisation of the above document.

4.1 General

- Overview map
  - Always visible view of the entire graph (regardless of zoom level).
  - Display of the current view by way of an easily visible rectangle (parts of the graph that are not visible are shown with a blue glass overlay, visible parts are shown normally).
  - The user should be able to drag the rectangle showing the visible area of the graph in order to adjust the current view.
  - The current view should also be adjustable by means of arrows to pan (up, down, left, right) and to zoom (larger, smaller).
  - Changes to the current view should be immediately reflected in the overview map, and vice versa.
  - The design should be orientated on well-known applications such as Google Maps.

4.2 Vertex Design

- Vertices should have a white background with a black edge.
- The size of vertices should be constant.
- The vertices should be displayed as a rounded rectangle.
- The height should be approximately double the font height.
- The vertex labels should be a single line of text centred in the rectangle.
- Font size should be adjusted to fit the maximum label width inside the rectangle.

4.2.1 Vertex Highlighting

- When a vertex is selected, the neighbouring vertices should be selected (direct neighbours only).
- This selection can be iterated over the entire graph (selecting the neighbours of all the selected vertices iteratively).
- This should be possible for every selection of vertices, regardless of the relationships between those vertices. For example, one could select the uppermost and lowermost vertices and use this function to allow the selections to ‘grow together’.

4.3 Edge Design

- The number of edge-crossings should be minimized (and multi-crossings of more than two edges in a single point avoided completely).
- There should be no contact or overlay of nodes and edges with the exception of incident nodes and edges.
- Edges must run vertically or horizontally, meaning any changes of direction are always 90°.
- A minimum distance between nodes will be defined, as will a minimum length for edges.
- With the implementation of the previous two points any overlapping of nodes is avoided.

4.4 Layout

- A hierarchal layout will be used to allow the proper display of super- and subclasses.