System Design

CENT
28.05.2013

Table of Contents

1. Introduction

This section provides an overview of the system and lays out the design goals. Any references or definitions relevant to the system definition or design are also included.

The system defined in this document is a java application for displaying graph class hierarchies and information regarding the complexity of certain computational problems with respect to the different graph classes.

1.1 Purpose of the System

This section summarizes the main functions of the software in order to give an overview of normal usage.

The system shall enable the user to browse through a database of graph classes. The database also contains select computational problems and the complexity classes associated with those problems for each graph class. This information shall be made available to the end user as well as the hierarchal information for each graph class (sub- and super-classes). The complexity classes for each problem shall be shown when a graph is selected while browsing the database. There shall also be a function for showing the P/NP boundary classes and as yet open classes for a selected problem.

The system shall provide a canvas for displaying graph class hierarchies to the end user. The user shall be able to pan and zoom to different areas of the canvas (a map-like overview of the entire canvas should be shown for orientation if the canvas is zoomed). The user shall be able to select a computational problem and each graph class currently on the canvas will be coloured with respect to the complexity class for the selected problem.

The system shall also provide a mechanism for showing the inclusion relationship between two graph classes selected by the user.

1.2 Design Goals

This section lists all design goals for the system. These have been broken down into sections to enable easier mapping to the proposed architecture.

1.2.1 Performance Criteria

Response Time: The user interface should remain as responsive as the existing system while drawing graph hierarchies. A small improvement is desirable but not necessary. Should the application be implemented as a Java applet, all server requests should require a maximum of 2000ms (assuming a stable connection with at least 1Mbit is available).

1.2.2 Dependability Criteria

Robustness: To avoid errors, preconditions should be monitored for all commands and if they are not satisfied the command should be disabled. Any errors originating from the yFiles drawing library should be caught and handled appropriately. This requires that all calls to yFiles objects/methods be appropriately wrapped in error handling code. This error handling code should log the error to allow the developers to diagnose the problem.

The backing data is stored in xml files. In the event of invalid XML or an error accessing the file itself, the user should be notified and the application should be terminated.

Availability: If the application is implemented as a Java applet, availability is subject to the availability of the hosting server. Otherwise the application should be made available for offline use after the first run.

1.2.3 Maintenance Criteria

Extensibility: It should be possible to add a new language without modifying existing code. Existing code does not need to be updated unless it is changed. Any new code should be written with localisation in mind.

Adding more graph classes or problems shall only be a matter of editing the related xml files. It must not require any new code.

1.2.4 End User Criteria

Usability: The user interface shall be kept as simple as possible. Workflows that require repeated actions should be updated to avoid this (e.g. drawing a graph from the browse database dialog currently opens up a dialog to select a graph class despite a class already being selected).

Panning and zooming in a graph hierarchy shall be fast and smooth. It shall be possible using either the mouse or the keyboard (wherever possible, keyboard shortcuts should stick to the conventional keys e.g. the arrow keys for panning).

1.3 References

- Requirements document, CENT, 27.05.2013
- Object Oriented Software Engineering, 3rd Edition, Bernd Bruegge und Allen H. Dutoit

2. Current Software Architecture

The current system is an independent Java application. The entire application, including the xml data files, is downloaded and run from the user's computer.

The application implements all of the desired functionality, but the layout of graph hierarchies is very rudimentary and the algorithm for placing the connectors between graph classes does not scale well. There is no zoom function in the current version and the canvas can only be panned using the scrollbars at the bottom and right of the window. There are also some functions that show unnecessary dialogs (see *1.2.4 End User Criteria*).

The code base does not separate data access from the user interface, making it hard to maintain and error prone. Most of the functions can be reused, but will require refactoring to separate into data access, user interface and application logic components.

3. Proposed Software Architecture

3.1 Overview

The proposed system uses a three tier architecture (see *Figure 1*). The presentation layer is responsible for interpreting and relaying all user input to the logic layer, and for displaying the response to the end user. The logic layer processes the user input and retrieves the necessary information from the data layer. All data processing is performed by the data layer, removing any dependency on a certain file format or a certain database from the rest of the system.

3.2 Subsystem Decomposition

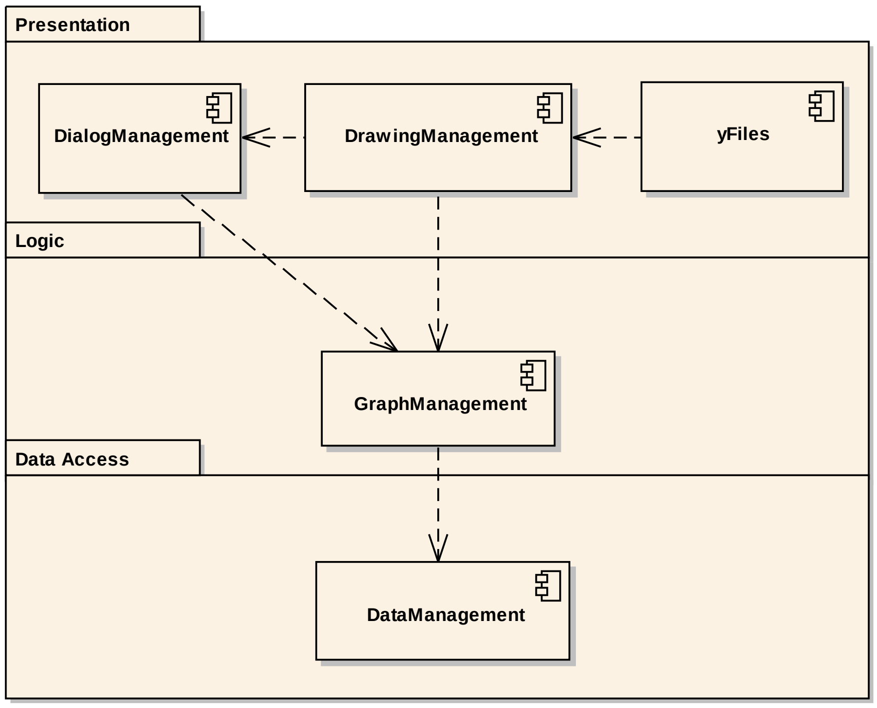

Figure 1: Subsystem decomposition of the proposed system

The decomposition of the data access layer is relatively simple, containing only a single component. This component is responsible for retrieving the data from storage (currently the xml files) and passing it to the application logic layer.

The presentation layer, however, contains three components. The yFiles component is used solely by the DrawingManagement component. This keeps this third party dependency isolated and makes it easier to swap out the drawing library in the future. The DrawingManagement component is responsible for all interaction with the canvas. It controls the drawing and colouring of graphs and any other interaction with yFiles (e.g. retrieving the graphML of the current display for exporting). The DialogManagement component controls all the dialogs in the application. It then relays any drawing commands that may originate from these dialogs to the DrawingManagement component.

Finally, the logic layer contains a single component: GraphManagement. This component controls the entire application flow. User input is relayed to the GraphManagement component, which in turn retrieves the necessary data and invokes any dialogs or drawing methods required.

3.3 Hardware/Software Mapping

If the application is not implemented as a Java applet, then all components will run on the same hardware: the end users PC. In the case that an applet is feasible, then the data access layer will be moved to the server. The applet will then retrieve the necessary information from the server through the interface defined in *4. Subsystem Services*.

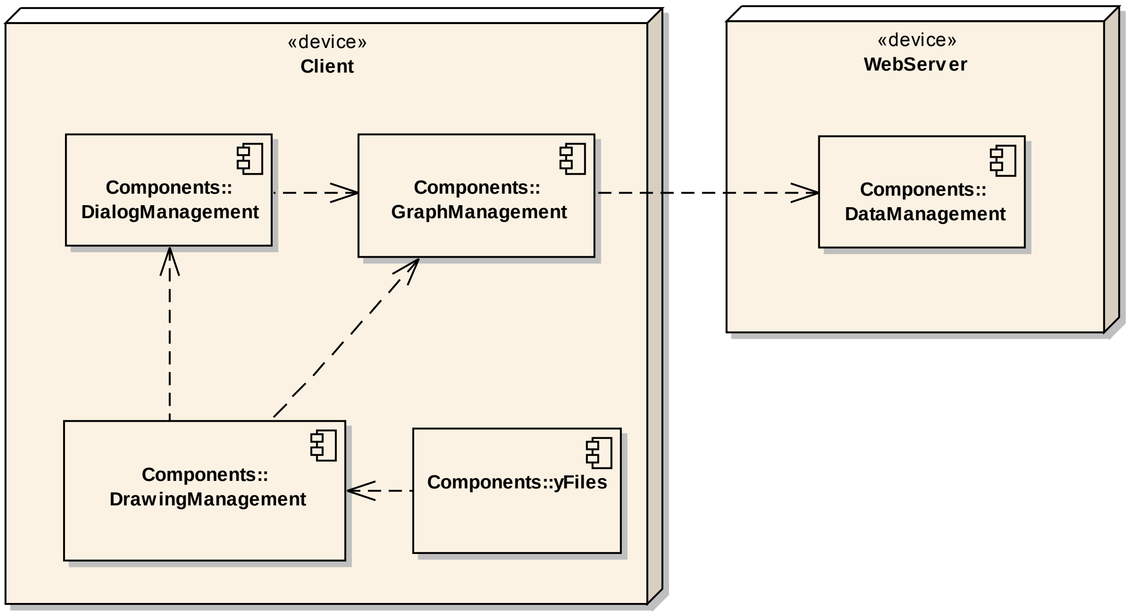
«device»
Client
Components::DialogManagement
Components::GraphManagement
Components::DrawingManagement
Components::yFiles
«device»
WebServer
Components::DataManagement

Figure 2: Hardware/software mapping for the proposed system in the case that it is implemented as a Java applet

3.4 Persistent Data Management

As the data storage is in the form of read-only xml files we do not need to be concerned with corrupting the data or concurrency control (although, as stated in *1.2.2 Dependability Criteria* we do need to be prepared for invalid xml or corrupt files). Even if the implementation is in the form of an applet the server-side access to the files will still be read-only, causing no concurrency problems.

The only files generated by the application are the exported graphs. As this is a user-initiated action we do not need to be concerned with multiple simultaneous writes to the same file.

3.5 Global Software Control

The end-user interface for the system will use procedure-driven control. No action will be taken without user input.

No multithreading will be used (with the exception of internal threading implementations in the Java Virtual Machine).

3.6 Boundary Conditions

When the system is started the data files should be checked for consistency and the data loaded into the DataManagement component (enabling faster access at the cost of a little memory).

If an unexpected exception occurs it should be logged and the application should exit gracefully. An option to send the log file or error information via email should be available to the user.

If the user exceeds a (yet to be determined) maximum number of graph classes on the application canvas, a warning should be shown and the graph classes added in the last action should be cleared from the canvas (to avoid showing incomplete information).

After use all objects should be released (nullified) to allow for garbage collection. This is especially important after the xml files have been read into memory.

4. Subsystem Services

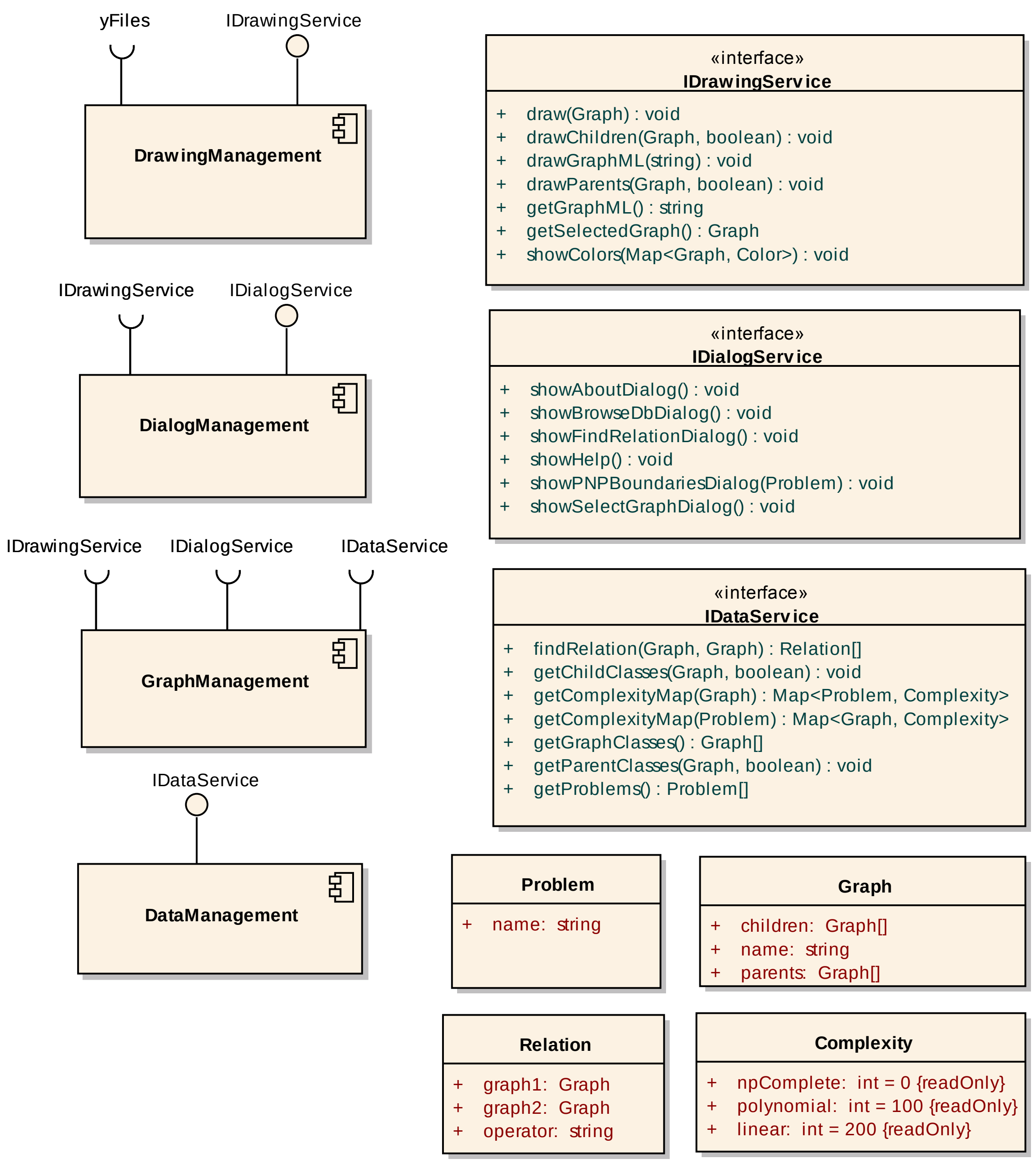
yFiles
IDrawingService
DrawingManagement
IDrawingService
IDialogService
DialogManagement
IDrawingService
IDialogService
IDataService
GraphManagement
IDataService
DataManagement
«interface»
IDrawingService
+ draw(Graph) : void
+ drawChildren(Graph, boolean) : void
+ drawGraphML(string) : void
+ drawParents(Graph, boolean) : void
+ getGraphML() : string
+ getSelectedGraph() : Graph
+ showColors(Map<Graph, Color>) : void
«interface»
IDialogService
+ showAboutDialog() : void
+ showBrowseDbDialog() : void
+ showFindRelationDialog() : void
+ showHelp() : void
+ showPNPBoundariesDialog(Problem) : void
+ showSelectGraphDialog() : void
«interface»
IDataService
+ findRelation(Graph, Graph) : Relation[]
+ getChildClasses(Graph, boolean) : void
+ getComplexityMap(Graph) : Map<Problem, Complexity>
+ getComplexityMap(Problem) : Map<Graph, Complexity>
+ getGraphClasses() : Graph[]
+ getParentClasses(Graph, boolean) : void
+ getProblems() : Problem[]
Problem
+ name: string
Graph
+ children: Graph[]
+ name: string
+ parents: Graph[]
Relation
+ graph1: Graph
+ graph2: Graph
+ operator: string
Complexity
+ npComplete: int = 0 {readOnly}
+ polynomial: int = 100 {readOnly}
+ linear: int = 200 {readOnly}

Figure 3: Interface definitions for communication between components.

In *Figure 3* all of the required interfaces for communication between components are listed. The classes Problem, Graph, Complexity and Relation are not representative of the final implementation, they just include the fields that are important for interpretation of the interface methods.

IDataService.getChildren and IDataService.getParents make use of the graph parameter being passed by reference and fill in the children and parents fields, respectively.

IDrawingService.drawChildren and IDrawingService.drawParents are used to extend the graph hierarchy by adding the parent or child classes for the selected graph (available through the IDrawingService.getSelectedGraph).

The rest of the methods are self-explanatory when viewed in context.