Technical specification

PMoC
14/04/2003
Version 1.16

T-76.115 Technical specification
PMoC

Version	Date	Author	Description
1.16	14.4.2003	Jan Lönnberg	Rotation and flipping of symbols added.
1.15	21.3.2003	Jan Lönnberg	Added dependency between coordinator and GML packages. Added extra mapping methods to GML section specs.
1.14	17.2.2003	Jan Lönnberg	Revised terminal toolbar. Added some items to canvas right-click menu. Added discussion of save and forget policies. Specified canvas tab naming. Added symbol id to component type file. Revised symbol toolbar requirements. Added refresh operation.
1.13	7.2.2003	Jan Lönnberg	Corrected references to unresolved issues in IO and extent of GML functionality.
1.12	5.2.2003	Jan Lönnberg	Corrected obsolete communication description in chapter 3. Revised inter-model component instantiation protocol. Fixed some typos. Added class to store symbol data to GML package.
1.11	29.1.2003	Jan Lönnberg	Major rewrite of communication between parts of system (deleted chapter 4.5 in process). Added categories to children of categories. Specified terminal toolbar.
1.10	24.1.2003	Jan Lönnberg	Component in net popup menu added, added symbol toolbar, added symbol conversion and terminal creation functions. Relaxed editing requirements on component types and their child elements. Minor changes to IO interface to match customer's network code.
1.9	21.1.2003	Jan Lönnberg	Required communication between models described.
1.8	16.1.2003	Jan Lönnberg	Various fixes to tree menus requested by customer.
1.7	14.1.2003	Jan Lönnberg	Corrected size ranges in gml package description. Various corrections to gml package class diagram.
1.6	13.1.2003	Jan Lönnberg	Added right-click menu items for tree view.
1.5	13.1.2003	Jan Lönnberg	Added GML local file system implementation.
1.4	10.1.2003	Jan Lönnberg	Added GML category and department objects. Specified GML loading mechanisms.
1.3	8.1.2003	Jan Lönnberg	Rewrote I/O package again. Added GUI description.
1.2	12.12.2002	Jan Lönnberg	Rewrote I/O specification to match customer demands and existing software
1.1	2.12.2002	Jan Lönnberg	Revised based on customer feedback
1.0	28.11.2002	Jan Lönnberg, Björn Forss	Initial version

Table of Contents

1. Introduction

1.1 Purpose of the document

This document describes how the ProConf system is to be implemented. The major technical design decisions and architecture of the system are explained here.

This document is primarily intended for developers (current and future) of the system and other parties with an interest in the technical aspects of the ProConf system such as the customer.

1.2 Description of the system

The ProConf system is a software product that can be used for creating process and automation flow diagrams that could be used with the simulation software that the customer, VTT, uses. The primary purpose of the system is to verify that the GML (Gallery Markup Language) specification together with graphics defined through SVG is a good solution for creating, storing and managing the configuration data related to simulation models.

2. Overview of the system

2.1 Purpose of the system

This is described in the Project Plan[1], section 1.2.1 and in the User Requirements Document[2].

2.2 Hardware environment

The system is done for normal PCs and requires no special hardware. A normal modern PC should be able to run the system. Further details are in the Project Plan and User Requirements Document.

2.3 Software environment

The system is mainly going to run on Microsoft Win32 platforms. Thanks to the implementation language, there should not be any big obstacles to getting it to run on other platforms as well. To be able to run on other platforms than Win32 platforms is not considered important and is not a requirement.

2.3.1 Java 2 Platform, version 1.4

The implementation is done using the Java 2 platform version 1.4[3]. The system will therefore require Java 2 Runtime Environment[4] installed on the computer to be able run. Java 2 Runtime Environment can be downloaded from Sun Microsystems Java Technology pages free of charge and is available for most platforms.

2.3.2 Batik

The system uses Batik SVG library and requires Batik version 1.5[5]. Only a beta version is available at this point, but the ready version is likely to be available when this project is finished. The beta version seems to work well enough at this point of the project. The system will require Batik version 1.5 installed to be able to run. Batik

1.5 can be downloaded free of charge form Apache Software Foundation's Batik page.

2.3.3 XML parser

The system uses the Xerces XML parser as supplied with Batik (version 1.3.1 at this date).

2.4 Limitations on implementation

The system should run on Microsoft Win32 platforms as described in 2.3. The implementation language was decided to be Java 2. The system must use GML and SVG. The most important goal of the project is to verify the GML specification.

2.5 Standards, specifications and agreements

The relevant standards are discussed in the Project Plan, User Requirements Document, GML - Gallery Markup Language Specification[6] and Coding Convention document[7]. Legal aspects of the project are discussed in the contract between the development group and the customer.

3. Architecture

This section describes the architecture of the system: the principles behind the design, how data is stored and how the code is structured.

3.1 Design philosophy

Our basic design philosophy is more or less object-oriented for several reasons:

- Graphical user interfaces are easily expressed in terms of objects and classes.
- Modularity is easier to design with object-oriented methods.
- The customer's specifications are quite data-centric and object-oriented.
- The chosen implementation language, Java, support object orientation naturally.
- Object-oriented design is familiar to the development team.

One of the central concepts of object-oriented design is to find the object classes in the desired system and design the code to reflect this. As the system should have a graphical user interface, it makes sense to consider the primary elements of the user interface and design accordingly. Similarly, the data structures should reflect the user's way of looking at things, which in our case means the customer's data model specification.

The customer made it quite clear that the system should be capable of manipulating SVG and GML data separately, with separate data views. Thus, it seems natural to break the system into separate modules for graphical data in SVG and for topological data in GML.

We have decided not to define complete class divisions for the entire system in the technical design for the following reasons:

- We wish to avoid analysis paralysis.
- Designing a system in minute detail without implementing anything easily leads to unrealistic designs.

- Nobody reads 100-page design documents anyway.

3.2 Data model

The customer provided quite explicit specifications for the data model. However, these specifications only describe the format of the GML and SVG data as XML text; it does not describe its representation in memory when the system manipulates it, nor does it specify how the data is actually divided into units and stored.

We decided to divide the XML data structures into separate files for each component, component type, enumeration, category and department. This decision was based on the customer's example data and existing implementation and the fact that this approach is easy to understand and implement. Efficiency is also a factor; if several components are stored in one file we are forced to parse all of them at once, which conflicts with the customer's desire for scalability if the amount of components in a file is large. Some form of index files to map between UUIDs and file names may also become necessary.

For efficiency reasons and to improve error checking, we manipulate GML using our own data structures instead of using the DOM interface provided by Xerces to manipulate its XML data structure. As GML is a specific instance of XML, manipulating GML data using generic XML routines is somewhat inconvenient, as the DOM model works with generic XML elements. Mapping an element type in GML to a Java class is simple and intuitive, so we use this method. Similarly, attributes are mapped to the closest possible Java attribute type (e.g. floating point values to double or Double, depending on whether an object is required or not), in order to provide proper error checking.

As the SVG data must be rendered by Batik, it makes sense to allow Batik to maintain the SVG data structures (maintaining our own data structures would involve duplicated data structures and work). This means that our SVG manipulation must be done using the DOM interface provided by Batik. We extend the GML architecture somewhat to include four types of SVG symbol: *free*, *terminal*, *component* and *connection* symbols. Connection symbols only consist of a single line (or polyline). Any SVG graphic can be converted into a free symbol, which can be made (through linking to a component type) into a component symbol, or into a terminal symbol (by defining a terminal type). Connection symbols can be created by editing the attributes of a plain line.

3.3 Program architecture

Based on the data model described above, we decided to divide the program into packages responsible for separate aspects of the system. The tree and graphical views have a corresponding package each (`browser` and `graphics` respectively). The GML data structure is stored in memory and manipulated using the `gml` package. The communication between the `browser` and the `gml` packages (coordinating changes between the packages that affect both the topological and graphical models) is handled by the `coordinator` package. The changes are communicated between the packages as method calls and using the Java drag-and-drop API. Graphical user interface functions common to several of the aforementioned packages and top-level graphical functions (such as a main window for the system) are contained in the `gui` package. The relationships between these packages are shown in figure 1.

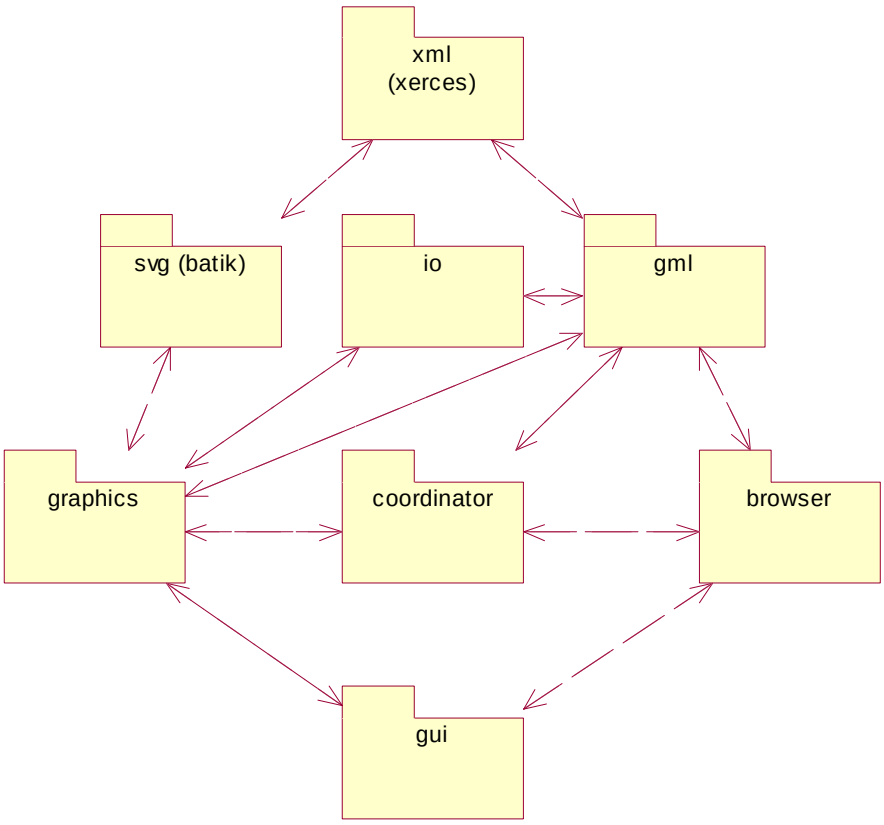

Figure 1: Packages of the system.

3.4 Communication between packages

Packages can interact in the following ways:

- Packages can instantiate and manipulate (through method calls) each other's classes. This is used as follows:
  - The `browser` package instantiates and manipulates `ComponentType`, `Component` and other GML objects from the `gml` package.
  - The `graphics` package uses the `JSVGCanvas` from Batik.
  - Both Batik and the `gml` package use Xerces through the W3C DOM API (which makes the library easier to replace).
  - Instances of the `ProConfCanvas` in `graphics` are created by the `coordinator.CanvasManager` class. Each instance of the canvas reports changes that also affect the topological model to the `coordinator.Propagator`.
  - The top level GUI spawns one instance of the `Tree` class in `browser`.
  - The top level GUI spawns one instance each of the propagator and canvas manager classes in `coordinator`.
  - Swing components are created and manipulated by the packages that provide graphical user interfaces.
  - The `io` package provides an abstract I/O class for components, component types, graphics and such identified by UUID. This will initially be implemented using the local file system and some index files, and provides the possibility to later add data storage on remote servers.

- The `propagator` class in `coordinator` is used to communicate between the GML and SVG data models. GML data may be read directly by the SVG manipulation packages if necessary to e.g. determine UUIDs for objects.

3.5 Error Handling

The following types of error can occur:

- Invalid user input.
- Invalid file input.
- Device or OS failure (e.g. disk read error, memory full).
- Internal software error.
- Library (e.g. graphics library) error.

All inputs are validated to be of right kind to avoid unforeseen situations. If errors occur, error messages are given to the user. Mostly usual dialog windows are used for this messaging.

In the case of user error, the most reasonable course of action is to report an error to the user and allow the user to correct his mistake. Similarly, in the case of invalid file input, the most sensible course of action is to report the problem and abort the file read operation.

In the case of device or operating system level errors, there is little we can do except report an error and abort the operation that has failed. Similarly, if a library routine reports an error, we can only report this and abort the operation.

If our own code does something wrong (as if that could happen) one of three things can happen. Either our code detects this (through sanity checks built into the code), crashes violently or continues to run but with incorrect results.

Error handling is mainly done by passing exceptions using Java's exception handling features. For exceptions thrown by our own code, we will create suitable exception subclasses if it is necessary to distinguish between different types of exception in code.

3.6 User interface

The user interface of the system should consist of a main window comprised of four main parts:

- Pull-down menus containing global commands (and possibly common actions from the following two parts).
- A tree view based on the topological model.
- A graphical view.
- Separate property dialogue boxes.

3.6.1 Menus

These have not yet been specified. As all required features will be accessible through other means, the importance of these menus is at most desirable.

3.6.2 Tree view

The tree view shows the objects of the topological model as a tree rooted at "The World" (or "Hosts", "Network" or suchlike if a less ostentatious term is desired). The root of the tree contains the known hosts running accessible model servers (for our purposes, this is only one machine: "Local host" or "This computer" (or "My Computer" for consistency with Windows)). For each host, its children are a list of servers on the host (again, for our purposes, one server is enough: "Local files"). For each server, its children are departments and so on according to the topological model. A detailed list of what the tree contains is in Table 1 (enumerations and terminal rules have not been specified yet and are considered optional by the customer).

Item	**Children**
Root	Hosts
Host	Servers
Server	Departments
Department	Categories
Category	Components, symbols, component types (enumerations, terminal rules), categories.
Component	Components, properties
Property	Vector
Vector	Vectors, values
Component type	Property types
Property types	Vector info, constraints
Vector info	Defaults, size ranges, min/max values
Default	Value

Table 1: Structure of topological tree view

The tree view should have options to choose which types of object are visible and what the root of the tree is (see the requirements document). The user should be able to store different views of the tree (position in tree, expanded objects, visible types, root) and switch between these using tabs representing different views. The topological model is manipulated primarily using pop-up menus accessible by right-clicking objects in the tree. These menus contain the following:

- For all objects (except hosts and servers):
  - New child element (see Table 1 for possible child elements) (optionally also create graphical equivalent where applicable).
  - Delete element (optionally also delete graphical equivalent where applicable).
  - Rename element (where applicable) (in-place editing).
  - Cut/copy/paste element (not required) (optionally also affecting graphical equivalent).
- For categories (category type shown in icon):
  - Toggle for supported attribute (only in type library).
  - Import SVG as (free) symbol.
- For components:
  - Attributes dialogue box, containing:
    - Toggle for tracked attribute.
    - Display of runnable and loadable attributes.

        - Properties represented as a tree of vectors with values for current vector shown as a table.
            - Vectors in this tree have the same right-click menu as in the main tree, with the addition of lifted attribute for the main vector in each property.
    - Open graphical representation of net.
    - Select component type.
- For values:
    - Change value (handled like rename).
- For component types:
    - Toggles for locked (can only be toggled once, and doing this locks the type and its children), loadable, runnable, liftValues, liftTerminals and composed attributes.
    - Open symbol.
- For property types:
    - Toggle for trackable attribute.
    - Submenu of alternatives for necessity.
- For vector infos:
    - Change label (handled like rename).
    - Attribute box:
        - List of alternatives for type (drop-down list).
        - Unit.
        - Minimum/maximum value.
        - Add/remove size range.
- For size ranges:
    - Attribute box: change min/max.
- For defaults:
    - Attribute box: index, count and value.
- For all symbols:
    - Open symbol.

Double clicking an element opens the corresponding attribute box or graphical object. In the case of components, we show the graphical net.

Creating a department also creates a type library, which may no be deleted. This ensures that each department has exactly one type library.

Creating a new component automatically opens the attribute box if there are required properties. These are initially given an empty value that must be set by the user. Implicit properties are filled in by the system when the component is created. Optional properties must be added by the user. For some properties, special editing tools may be provided (e.g. editing a start terminal property allows you to select an end terminal property).

Component types are instantiated by dragging a type to the place in which you wish to instantiate.

The editing commands described above are optional for component types and their children.

The tree also contains a “refresh” operation, which updates the tree structure using a GQL `GetTree` operation. It does not reload the contents of components or component types.

3.6.3 Graphics view

From the tree view, a graphical canvas can be opened for a component (the canvas showing a net of the component’s subcomponents and such) and a symbol (showing the symbol itself). Tabs can be used to switch between open canvases (the tabs are labelled with an icon indicating the type of object shown on the canvas and the name of the object (for graphical objects attached to a GML object, this is the GML object’s name, otherwise it is the symbol’s name)). The graphics view contains a toolbar with tools to manipulate the current symbol or net. For nets, and free, component or terminal symbols, the tools include (at least) the usual graphical primitives (lines, polylines, rectangles, ellipses, polygons), some way to specify whether they are filled or not and both fill and outline colour settings. For connection symbols, the only editing operations are attribute changes on the single line. It should be possible to flip symbols horizontally and vertically or rotate them in 90 degree increments.

A toolbar is also shown that contains component type symbols with an open type, free symbols and connection symbols as well as subcategories (and the parent category, if any). This *symbol toolbar* can be used to browse through the type library of the department of the currently shown net (if any; hidden if none), and symbols are selected from here for graphical instantiation (the position to place the symbol on the net is then chosen, by clicking the position (or positions, the first of which is a start terminal and the last an end terminal, for connection symbols)). Symbols are identified by name and optionally by thumbnail. The symbol toolbar is optional if the tree supports tabs with different, selectable roots.

If a component type symbol is being edited, a *terminal toolbar* is shown containing properties (specifically, the terminal property types of the component type to which the symbol is attached) that can be attached to terminals. These properties are attached to terminal symbols by selecting from a menu that drops down from the property the desired terminal symbol (identified by name, optionally by a thumbnail). This terminal symbol (attached to a property) can then be placed on the component type symbol by selecting the property from the toolbar and dragging it onto the canvas. The drop-down menu for each property contains the categories and suitable terminal symbols in the type library arranged as a tree of nested menus (every category is represented as a submenu, every symbol as a menu item). For speed, submenus should only be constructed when selected; categories should be loaded on demand as described in subsection 4.7. Note that until the terminal symbol is added to the canvas, there is no connection between the property and the symbol except in the state of the terminal toolbar.

If a free symbol is shown in the canvas, options should exist to link it to a component type or add a terminal definition. This converts the symbol into a component symbol or terminal symbol respectively. To link to a component type, drag the symbol’s tab (or the symbol in the tree) onto the component type in the tree. To add a terminal definition, right-click the symbol tab or the symbol in the tree and choose to convert it

to a terminal symbol. The canvas tab's right-click menu should also contain options to open the parent net and save the current canvas.

Right-clicking a component should produce a menu with the following:

- Select in tree.
- Attributes.
- Open subcomponent net.

4. Package descriptions

The system is divided into packages with different functionality. The root package for the system is `fi.vtt.proconf`. For clarity, attribute get/set/iterator methods have been left out, as have constructors in cases where classes only have one constructor with no arguments (that creates an "empty" instance of the class).

4.1 gml

The `fi.vtt.proconf.gml` package takes care of GML handling. This package includes all functions for manipulation of the GML data structure. The GML classes are represented using similar Java classes. The attributes and referenced objects of the GML classes are represented as attributes in the Java classes. The GML attributes and referenced objects are private and can be manipulated using the usual public get/set methods for each attribute. These methods perform validation of changes made to ensure data consistency. The collections of referenced objects are manipulated using public add/remove/iterator methods that add or delete objects or return an iterator for the collection respectively.

The GML tree is loaded as follows to ensure scalability:

- Components, component types and categories are *fully loaded*, *partially loaded* or *unloaded*.
- A component is *partially loaded* if only its `Component` object is loaded. This can easily be generated from the result of the GQL `GetTree` operation. Partially loaded components are represented as `Component` objects with null `properties`, `subcomponents` and `type`.
- A component is *fully loaded* if its subtree is loaded except for its subcomponents, which are partially or fully loaded (in other words, when a component is loaded, it and all its properties and vectors are loaded, but its subcomponents need only have their main `Component` object loaded). Fully loaded components are represented as full `Component` objects with meaningful child collections.
- Component types and categories are *partially loaded* if only their `ComponentType` or `Category` object is loaded and *fully loaded* if their child elements are partially or fully loaded. Null collections are used in partially loaded component types and categories.

The general idea is that items in the tree view that are not visible are unloaded, items that are visible and are leaf nodes in the tree view are partially loaded and items that are visible and are not leaf nodes are fully loaded. In other words, when a user expands a category node in the tree, this category is loaded fully, and all of its subcategories, components and component types must be loaded partially if they are not already loaded.

Partial loading is represented as a constructor that creates an object with a specified id. The other properties must then be set. This object can then be fully loaded through its `load` method. The constructor without an id creates a new empty object of the specified class. Only fully loaded elements may be saved.

To ensure data synchronisation, all affected objects (including graphics) are saved every time a change is made to the topological model. To improve efficiency in a common case, when editing properties of an object from a dialogue box, changes are not saved until the box is closed. Graphical objects can also be saved be right-clicking on their corresponding tab and selecting “save”. Graphical objects are also saved whenever their canvas is closed.

To allow the program to free unused data structures (and thus avoid eventually using all available memory), components can be changed from fully loaded to partially loaded whenever none of their descendants are visible or have their nets open. In this process, subcomponents may become completely unloaded. Component types can be changed from fully loaded to partially loaded if no instances of them are fully loaded, none of their descendants are visible and their symbol is not open. Categories and departments may be changed from fully loaded to partially loaded whenever none of their descendants are visible in the tree or in a canvas (this may fully unload their children).

The GML specification does not contain a reference from a component type to its symbol(s). However, for efficiency reasons, we require component type files to contain as child elements `SymbolReference` objects (abbreviated `SR`), which have one attribute: `sid`, the UUID of the symbol. Only one `SymbolReference` is used, the rest are ignored (space for future expansion).

All GML classes that can be referred to by UUID have additional methods called `getObject` (takes an UUID and returns the corresponding object), `getDepartment` (takes an object and returns its department), `getDepId` (takes an object and returns its department UUID) and `getParent` (takes an object and returns its parent). `PropertyInfo` may also contain additional methods to identify specific groups of property types. For clarity, these are not shown on the diagram.

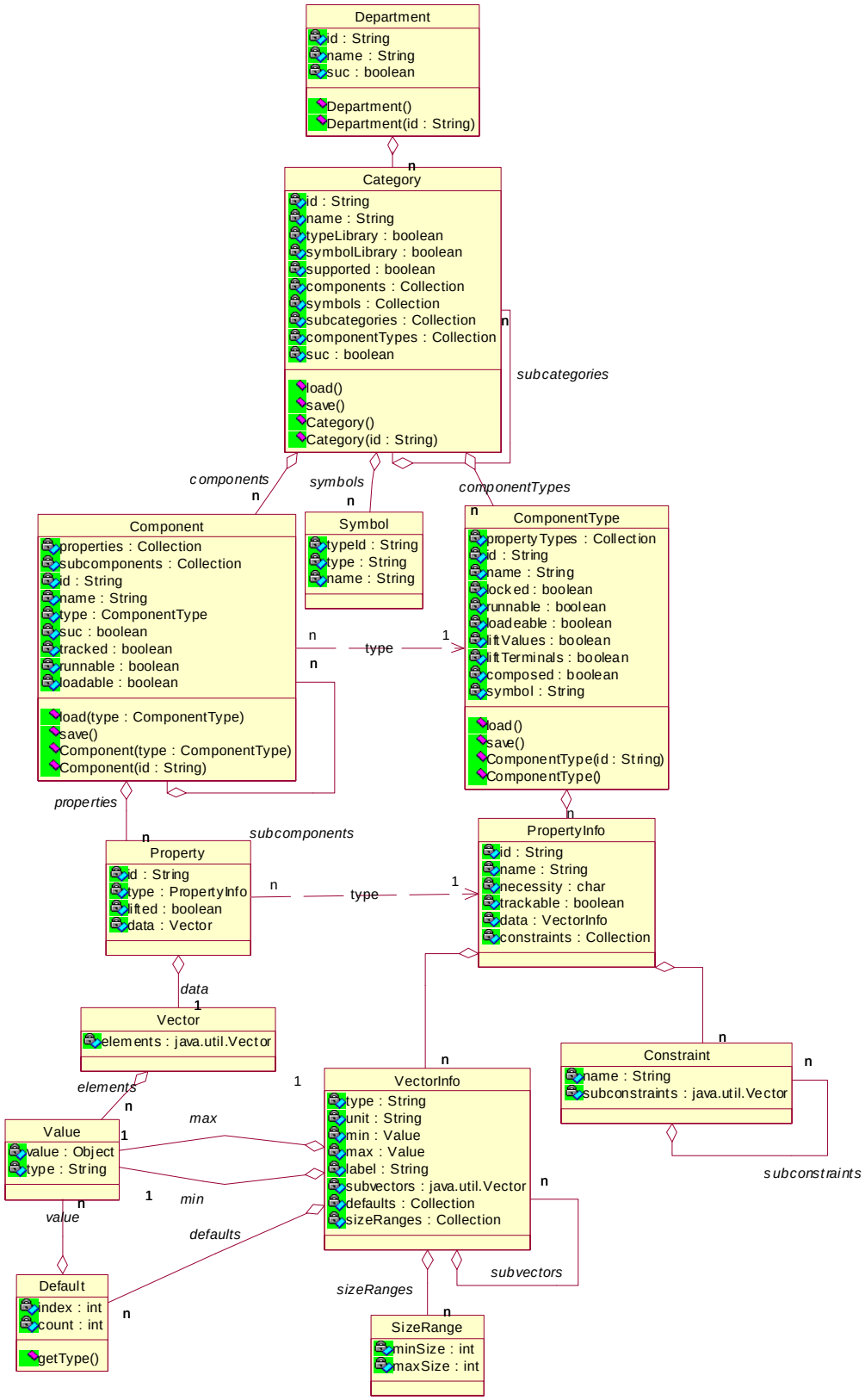

Figure 2: gml package.

4.2 browser

The `fi.vtt.proconf.browser` package provides a tree view of the topological model, which the user can manipulate. The `Tree` class handles this tree view and provides methods for the `neteditor` and `symboleditor` classes to modify the topological model based on changes to the graphical model.

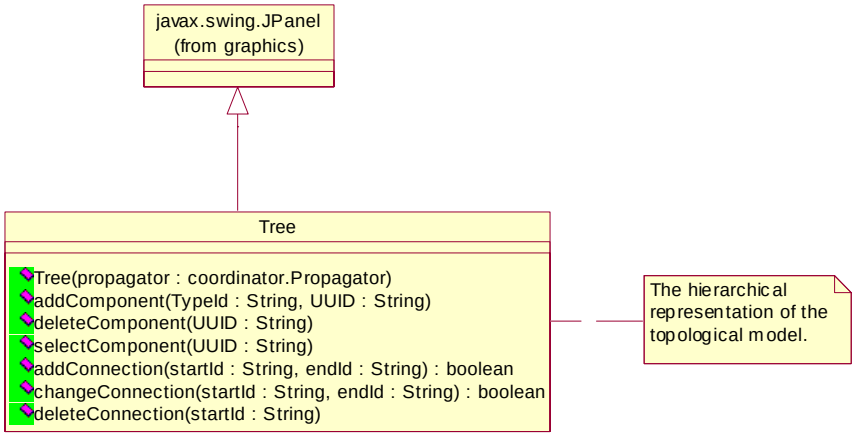

Figure 3: browser package.

4.3 graphics

The `fi.vtt.proconf.graphics` package takes care of SVG drawing. This package includes classes for a canvas (`ProConfCanvas`), drawing tools and SVG manipulation. The canvas has methods that the `neteditor` and `symboleditor` classes can use to modify the topological model based on changes to the graphical model.

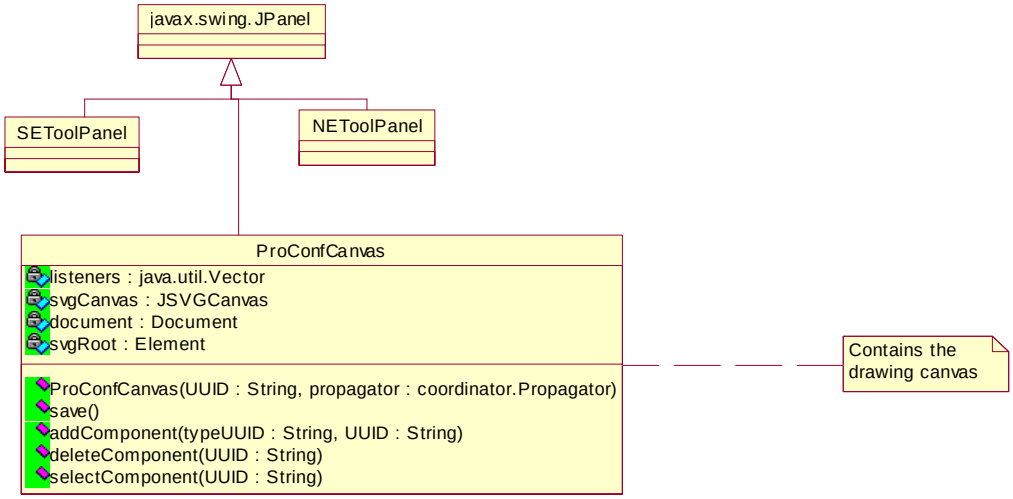

Figure 4: graphics package.

4.4 coordinator

The package `fi.vtt.proconf.coordinator` is responsible for maintaining consistency between the GML and SVG models. Changes made in one model that should be propagated to the other are sent through this package. Each package contains event listener classes for both messages from the tree and from a canvas.

When a deletion is made, the other model's class is notified of it before the deletion is made. When an insertion or change is made, the other model's class is notified afterwards. Changes made in the tree that change a graphical object cause the corresponding canvas to be opened and the corresponding graphical change initiated.

The `graphics` package is responsible for checking for an object's topological equivalent.

The package must handle the following actions:

- Actions initiated by the tree:
  - Deleting a component (with a graphical equivalent) involves:
    - Closing the window containing its net.
    - Deleting the net (we assume that only components without subcomponents may be deleted; components with subcomponents can (optionally) be deleted using a recursive delete handled by the tree).
    - Removing the component from its parent's net.
  - Opening a net or symbol.
    - Creates a new canvas if necessary.
    - Activates the canvas.
  - **(Optional)** Adding, changing or deleting a connection (by changing the value of the start terminal property). The corresponding change must be made to the net.
    - Deletes the connection in the graphical model and creates a new one.
  - Subcomponent creation is handled by dragging a component type from the tree into a net.
    - This creates the new component in both the tree and the net.
    - Dragging is handled using the Java Drag and Drop API, transferring the component type id as a string.
    - The id of the component whose net the instantiation should be made in and the id of the instantiated component type is sent to the tree using the `componentInstantiateStarted` method, which returns the id of the new component.
    - The new component's graphical instance is created.
  - Adding/removing a symbol or category to/from a category in the type library (that may be the category shown in the symbol toolbar).
    - The symbol or category is specified by id.
  - Selecting a component on the canvas from the tree.
    - The component (specified by id) is selected on the canvas.
  - **(Optional)** Selecting a terminal on the canvas from the tree.
    - The terminal (specified by property id) is selected on the canvas.
  - Converting a free symbol into a component type symbol by dragging it onto a component type.
    - The id of the free symbol is specified in the drag message (see subcomponent creation for details on data transfer).

        - The id of the symbol and type is sent back to the graphics system for SVG modification using the `attachSymbolToType` method.
    - **(Optional)** Converting a free symbol into a terminal symbol.
        - The canvas or canvas maintainer is notified so that it may make the requisite changes to the SVG code.
    - Adding/removing a terminal property adds/removes the property to/from the terminal toolbar (if visible for this component type symbol).
        - Identified by property id.
    - Deleting a symbol closes the canvas showing it.
        - Symbol specified by id.
    - Drag free symbol into canvas to instantiate.
        - Symbol specified by id, sent in drag message.
- Actions initiated by the canvas:
    - The tree is notified of deletions and selections (i.e. "select in tree" operations, not just clicking a component) of subcomponents.
        - Specified by component id.
    - When a connection is added, changed or deleted, the corresponding change is requested from the tree, which (for additions and changes) replies by acknowledging or rejecting the change (boolean value indicating success).
        - Specified by id of start and end properties.
    - **(Optional)** The user can request that the attributes box for a specified component be opened.
        - Component specified by id.
    - **(Optional)** Selecting a terminal in the tree from the canvas.
        - Specified by property id.
    - Open a net by double-clicking its component's symbol in the parent net.
        - Component id sent to canvas manager.
- Action initiated by canvas manager:
    - Connection of free symbol to component type.
        - Drag to tree (drag data is symbol id).
        - Update type attribute in SVG (done by canvas or canvas manager).
    - Conversion of free symbol to terminal symbol.
        - Update type attribute in SVG (done by canvas or canvas manager).

The new design involves a singleton class, with a method for each of the events described above that modifies the other model as required. This will simplify implementation somewhat (the only internal state required in the coordinating class would be information on which canvases are open). As static classes are undesirable, the GUI will create one instance and then pass it to the tree.

The `Propagator` class has a method for every communication event described above that propagates changes. The `CanvasManager` class keeps track of canvases, provides

the tabs to select the canvases and provides operations that ensure that canvases are open or closed.

For clarity, optional operations have been left out from the following UML diagram.

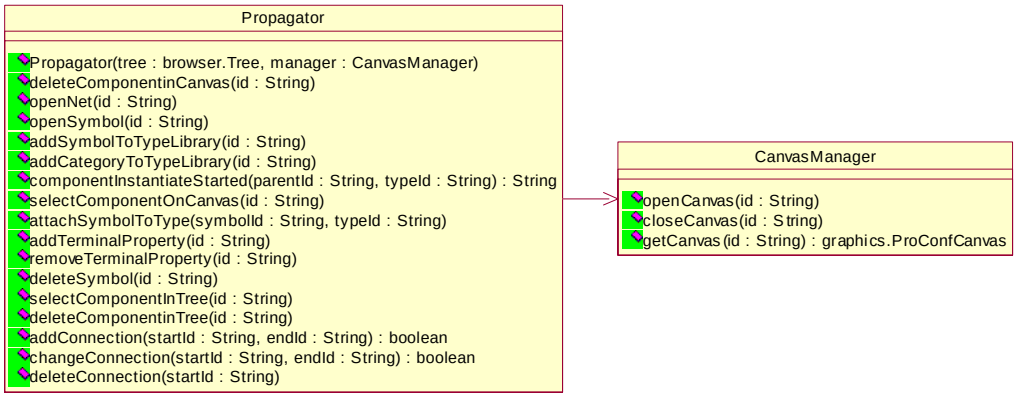

Figure 5: coordinator package.

4.6 gui

Graphical user interface functions common to several of the aforementioned packages and top-level graphical functions (such as a main window for the system) are contained in the `gui` package.

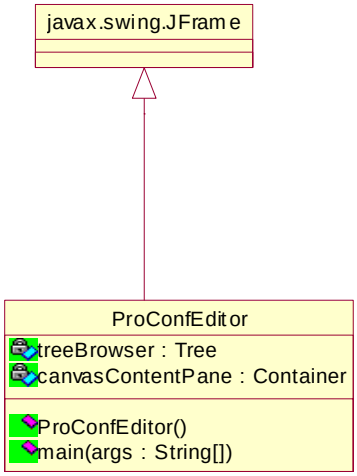

Figure 6: gui package.

4.7 io

This package provides abstract file I/O based on UUIDs: loading and saving of components, component types, and so on. The I/O functions work with strings containing XML data. Each IO handler works with a connection to a server, which is a process running on a host. Each server can provide access to multiple departments.

The `getElement` method takes the UUID and type of a component, component type, category, department or graphics file and returns the XML data for this object as a string (equivalent to the `get*` call in GQL for the type of the object). When getting a component, subcomponents (in GQL terms, a level of 0 is used), property types

(extended property info is false in GQL) and GQL extended vertical or horizontal information are not returned.

The `setElement` method takes a component, component type, category, department or graphics file as an XML string identified by the specified UUID and type, and saves it as a child of the specified parent (equivalent to the `set*` call in GQL for the type of the object).

The `removeElement` method removes the specified element (of the specified type) (equivalent to the corresponding `remove*` call in GQL for the type of the object).

The `getTree` method returns the children of the specified nodes (using XML request and reply formats according to the GQL specification) (behaving like its GQL counterpart). The department in which the element to be manipulated is must be specified for GQL compatibility even though it may appear redundant.

We will implement this package using calls to the local file system and tree structure files (for these purposes, we can assume that there is only one possible connection; to the local file system).

Each component, component type, and graphics file is stored as a GML file (extension `.gml`) and a tree structure file (extension `.gts`). Categories and departments are stored (as yet) using only tree structure files. The tree structure file consists of one line containing the parent's UUID followed by GML data for the element and its children (i.e. the GML elements for the specified element and its children, as in the output of `getTree`). The filenames for an element are constructed by taking the UUID and adding the relevant extension.

The reason for two separate files is to map the `getTree` and `getElement` methods to reading one file each. The `setElement` and `removeElement` methods must update both files for a specified element and the parent element's child list.

The IO specification is described in more detail in a separate document by Jyrki Peltoniemi[8].

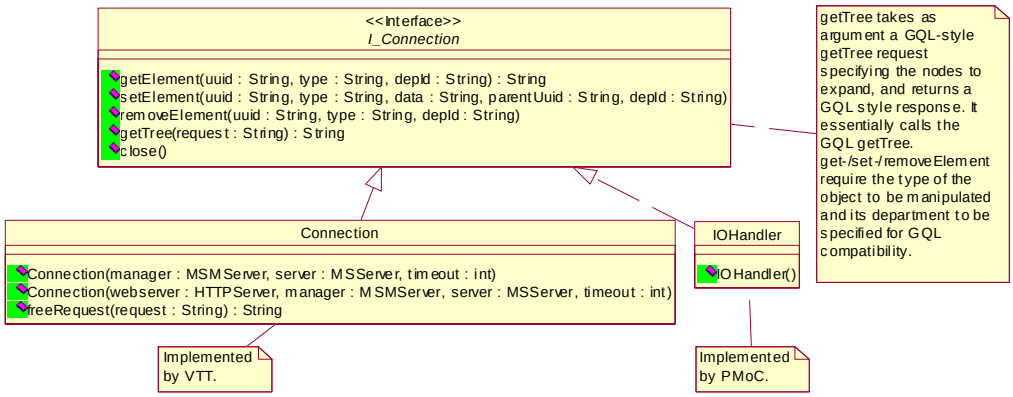

Figure 7: io package.

5. Technical decisions

This section documents the largest and most pivotal technical decisions made for the system and the reasoning behind them.

5.1 SVG library

As SVG plays an important role in the system, the way in which SVG is implemented plays a critical role in the project. The system needs to be able to parse and output SVG files, render SVG images, modify SVG content and accept graphical user input through the SVG image.

Due to the extent of the SVG specification and our limited time budget, we ruled out implementing SVG rendering ourselves. This forced us to look for libraries with the required functionality.

The W3C's SVG site[9] provided us with a convenient starting point for our search. From their list of implementations, we culled libraries with inadequate SVG support (non-native SVG tools), lack of rendering support (e.g. Savage SVG toolkit), halted development (e.g. Jackaroo), expensive licenses or lack of support for our target platform (most mobile implementations). After this, we were left with only a few serious contenders.

5.1.1 Adobe SVG Viewer

Initially, this viewer seemed to be the most promising, as it has the support of a major corporation, is under continuing development, supports all important parts of the SVG specification, is freely available, seems to be quite efficient and provides all the rendering and editing features we need. It uses Microsoft ActiveX and COM technology to interface with other software. However, upon closer examination and attempts to actually create software that used it, we encountered several problems. Worst of these was the complete lack of official development documentation for the use of the SVG Viewer with C, C++ or any other programming language. This made it almost impossible to produce code that manipulated the SVG image or accepted user input. Also, Adobe's licensing could cause some problems with distribution, and the SVG Viewer tended to pop up license agreements for no apparent reason. Also, the ActiveX and COM interfaces are quite complicated and would require a lot of extra work even if they were properly documented. For this reason, we rejected Adobe SVG Viewer.

5.1.2 CSIRO

The CSIRO SVG Toolkit is a freely available SVG viewer and DOM implementation for Java. However, its SVG and DOM support is somewhat limited, and its developers recommend Batik for better standards support. Apart from these minor problems, CSIRO could be a reasonable choice.

5.1.3 Mozilla SVG

A cursory examination of the Mozilla SVG code revealed that it is designed for the Mozilla browser primarily for SVG viewing purposes. Its SVG support is still quite limited, and it is not included in the main Mozilla distribution for various reasons. This suggests that the Mozilla SVG code is not quite usable yet.

5.1.4 Batik

The Apache Software Foundation's Batik SVG Toolkit is a highly extensive SVG implementation for Java, which appears to support everything we require. It is distributed under the extremely lax Apache Software License, which just about allows us to do anything we like with Batik. Many other open source SVG projects have been stopped in order to better support Batik. After rejecting Adobe SVG Viewer, Batik became our primary candidate. We have succeeded in creating applications that load, display and manipulate SVG data as well as reacting to user input made through the displayed SVG image. Batik appears therefore to be our best choice.

The greatest concern with Batik is efficiency; however, if we are careful only to open SVG images that are actually edited or shown (as opposed to loading all components at once), our tests suggest it is sufficiently fast.

5.2 XML library

We can either use a separate XML library to parse GML, or use the same library as the SVG library we use. Because our final choice was Batik, we decided to use Xerces, which is used by Batik, to parse XML. Xerces supports the relevant standard interfaces for XML parsing and manipulation in Java, so replacing it with a different equally standards-compliant library at a later date should not be a problem. Xerces is also licensed under the Apache Software License, which means that we have no library licensing problems at all.

5.3 Implementation language and environment

Although the initial plan was to use C or C++ as the implementation language, SVG library concerns forced us to change to Java. This is likely to be a good thing anyway, as many of the team members have more Java experience than C++ experience and Java has less development overhead regarding e.g. communication with libraries and memory management than C++, which speeds development and decreases the probability of mistakes.

On moving to Java, we decided to use Borland's JBuilder for interface design (this was influenced by the customer's propensity for Borland's similar environment for Pascal, Delphi). For additional flexibility, we noted that JBuilder and Sun's freely available Java SDK can exchange code quite easily, allowing us to use Sun's SDK for development of non-graphical parts of the program.

5.4 GUI library

As we are using Java, we have several GUI libraries available to us. We decided to limit ourselves to standard libraries to avoid depending on too many external libraries, which left us with AWT or Swing. AWT is quite limited and is not being developed actively, while Swing contains more or less everything we need, is actively used and developed and is familiar to most of us. Thus, Swing was the obvious choice.

5.5 UUID generation

To generate UUIDs, we have several solutions. One is to implement our own UUID generator (which will take some time, as the procedure is quite complex), use someone else's generator or call the UUID generator in Windows. Searching the WWW turned up the Java Uuid Generator (or JUG)[10], which supports Windows,

Linux and Solaris (native support is required to get the machine's MAC address). This is under the Lesser General Public License, which requires that we distribute source code for any modified versions we distribute (no great hardship).

6. Rejected solutions

6.1 SVG libraries

As documented above, we initially considered using Adobe SVG Viewer for our SVG needs. As we suspected (based on the customer's experiences) that we would encounter problems, we decided to test the viewer before designing the system. Due to the problems mentioned above, we rejected Adobe SVG Viewer.

6.2 Development language and environment

Our initial plan was to use the Microsoft Visual C++ environment supplied by SoberIT for development. However, after some attempts to produce working test programs using this environment, we were forced to concede that learning to use it would delay us a lot. Also, the user interface development facilities in Visual C++ are abysmal compared to e.g. Borland's C++ Builder or JBuilder tools with which several of the developers are already familiar. Most of this is due to the extremely low-level design of Microsoft's MFC library, which works on explicit message passing and window-level callback function techniques.

7. Future development

The customer's need to further develop the system after this project is considered an important requirement. The system should be designed so that the future development is possible and also easy to do. This requires good technical documentation, clear design and well-commented code. The customer is closely involved in the development of the system and gives feedback to the project group. Different modules of the system should be as independent as possible, so that modules can be changed without bigger rewrites of the whole system.

8. Undecided matters

The exact division of the tasks to be performed by each package into classes is left to the implementers.

---

[1] PMoC: Project Plan
[2] PMoC: User Requirements Document
[3] Sun Microsystems: Java 2 Standard Edition, http://java.sun.com/j2se/
[4] Sun Microsystems: Java 2 Runtime Environment, http://java.sun.com/j2se/1.4/download.html
[5] Apache Software Foundation: Batik SVG library, http://xml.apache.org/batik/
[6] Tommi Karhela/VTT: GML – Gallery Markup Language Specification
[7] PMoC: Coding Conventions Document
[8] Jyrki Peltoniemi/VTT: ProConf Java IO-Interface
[9] W3C: Scalable Vector Graphics (SVG): http://www.w3.org/Graphics/SVG/Overview.htm8
[10] Tatu Saloranta: Java Uuid Generator, http://www.doomdark.org/doomdark/proj/jug/