University of Aberystwyth

CS12420 Group Project

Designing a Bluej-esque UML Designer

Authors:
Daniel Malý (dam36),
Samuel Sherar (sbs1),
Lee Smith (ljs5)

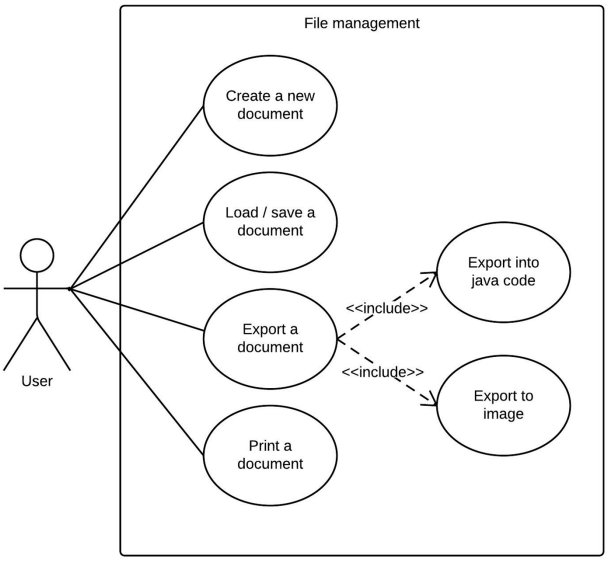
Note:
The User documentation and a HTML version of the Javadoc can be accessed from the Help menu in the program.
Testing screenshots can be found in the User Documentation.
To run from the command line. please type `java -jar uml2java_v1.0.0.jar`

Contents

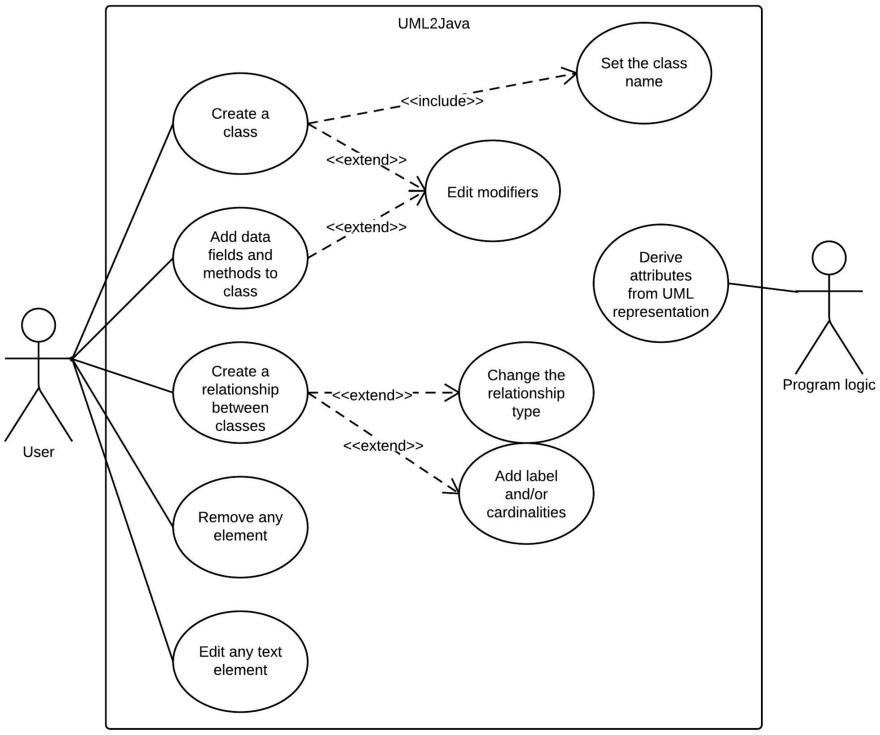
1 Analysis

For our group project, we are required to design and implement a program that can create and convert Unified Modelling Language (UML) class diagrams to java stub code. For this we will create a graphical user interface, in which the user will be able to create, delete and modify classes, add relationships between classes and add text labels in a UML class diagram.

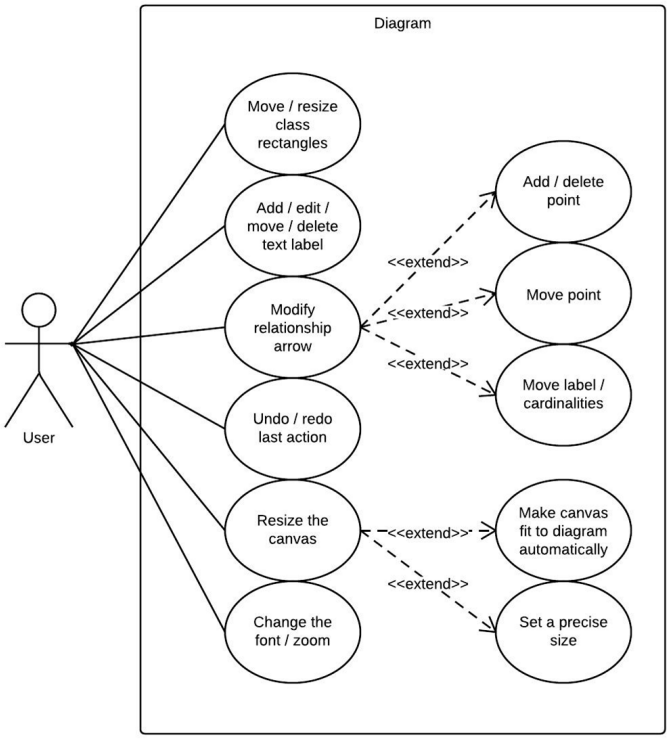
The functionality of the program that we will implement can be split into three general groups: file management, software design and diagram editing. A use case diagram for each of the above groups is provided is provided below.



The file management functionality of the program will include the following features: the user will be able to open a new document or load an existing one from a file, as well as save the current document. The program will also be able to export the currently opened class diagram into corresponding .java files containing the user-entered data fields and method stubs. It will also be possible to export a .png, .jpg or .gif image of the diagram canvas that can be easily added to any design documentation. The choice of file or directory relevant to each of these features will be up to the user and will be facilitated by the GUI. An option to print the current diagram will also be implemented.



The software design features will include the following: creating a new class, adding data fields and methods to it using standard UML notation from which the program will automatically deduce certain parameters like visibility, type and so on. Modifiers that are not supported by direct UML notation will be accessible through a right-click menu, both for classes and for attributes. The user will be able to add relationships between classes, change their type and add labels and/or cardinalities to them. Any element created this way will be easily editable and removable.



As we are aiming to let the user produce complete UML class diagrams, additional features will be necessary. Class rectangles will be movable and resizable and the user will be able to add text labels to any point on the diagram. Relationships will be highly controllable: moving around points will be possible, as well as adding new points or removing existing ones. The canvas size will be configurable to allow facilitate image export and printing. The user will be able to change the global font used in the diagram and also view the diagram under zoom. We will also attempt to implement an undo / redo facility for user actions.

2 Design

2.1 Class Descriptions and Diagrams

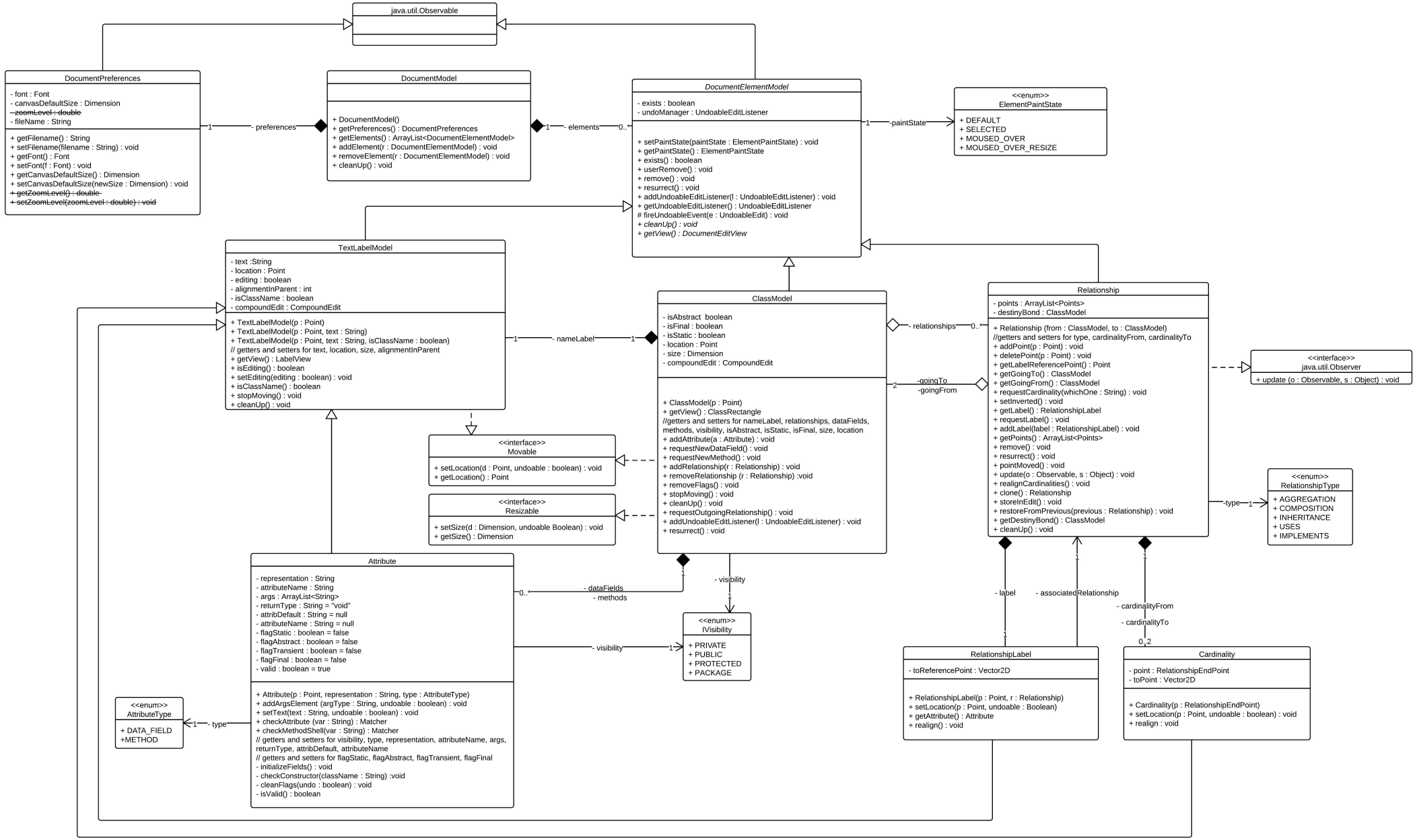
The application is built according to the model-view-controller architecture. The actual class diagram the user is making in the application window consists of individual MVC triplets of classes. The source code packages are split up into model, view and controller classes with three more packages for classes the gui, assuring the undo functionality and the export functionality. Below are brief descriptions of all the packages and their class diagrams. Making a global class diagram for the whole application proved to be an impossible task (we would need a few more dimensions than paper could provide us with), so we did our best to make a class diagram for each individual package while showing most of the relationships linking packages together. Classes that do not belong to the package in question are denoted by their full package name.

2.1.1 Model

The classes in the model package are designed to hold all the relevant data about a document. When saving files, the model is all that is saved and it is all that is necessary to then completely reconstruct the whole document. A document, that is a user-created class diagram, is represented by the DocumentModel class.

DocumentModel and DocumentPreferences
This class acts mainly as a wrapper for all the data about a document. By passing a DocumentModel object around, we can pass the whole of the model. The class holds a DocumentPreferences object that stores persistent global data about a document like font, canvas size, associated filename and zoom. The DocumentPreferences class extends Observable which lets individual model elements react to changes the user makes to it by acting as Observers. The second principal data field in DocumentModel is an ArrayList of DocumentElementModels.

uk.ac.aber.dcs.cs124group.model



DocumentElementModel and ElementPaintState
An 'element' is understood to be a discrete, controllable and viewable entity in the class diagram. All elements the application supports share a certain amount of characteristics. These are represented in the abstract DocumentElementModel class. The most important characteristic is existence (this includes the remove() and resurrect() methods). More information about what this means and how existence is handled within the application, please refer to section **2.1.6** on undos. Other characteristics shared by elements are, font, zoom and the paint state. The enum ElementPaintState defines four possible states an element may be in based on mouse actions. This is used not only to work out exactly how an element is to be painted but can also be used to find out when an element has been clicked on from outside its own MVC triad. The other fields and methods in DocumentElementModel relate to the undo functionality, described in section **2.1.6**.

Textual elements: TextLabelModel, Attribute, Cardinality, RelationshipLabel
Any textual element in the class diagram is actually a text label, represented by a TextLabelModel object. It can either be a plain label sitting somewhere outside any other element or it can be a class name, an attribute, a cardinality or a relationship label. The latter three all have their own model classes that extend TextLabelModel which defines basic operations like setting the text, location and size in the class diagram and enabling editing. Subclasses of TextLabelModel will usually restrict their own movability.

Attribute, again
This class merits its own section as understanding how it works is crucial to understanding the export functionality. Attributes are split into two types — data fields and methods. They have a representation — a String that the user enters into the class diagram. From this String, information is extracted about the visibility, type, default value, return type and arguments, where applicable, and attributes of this class are initialised. Modifiers can be added and edited by right-clicking. When exporting Attributes into java code, the Exporter class only looks at these initialised fields. If the representation does not conform to UML standards, the Attribute is flagged as invalid and is ignored by the Exporter. Exportable Attributes may also be obtained by calling the getAttribute() method on a RelationshipLabel object.

ClassModel
As the name suggests, ClassModels hold data about a class rectangle in the UML diagram. They all contain a TextLabelModel that is the name of the class shown on the top of the rectangle and may also contain user-defined Attributes and Relationships. Also provided are modifier and visibility flags. Methods for manipulating all of this data are defined in the class.

Relationship
Relationships represent data about 'arrows' on the class diagram. They have a type, a 'from' ClassModel, which is always the class at which the arrowhead or diamond is shown, regardless of the type, a 'to' ClassModel and an ArrayList of user-defined points. They may also contain two Cardinalities and a RelationshipLabel. Methods manipulating all of this data are provided, as well as cloning and restoring facilities that are used by the undo functionality.

Movable and Resizable
These are two interfaces implemented by user-movable and / or user-resizable elements. Currently, ClassModel implements both and TextLabelModel implements Movable. These classes are mainly defined to facilitate undo functionality.

The enums: IVisibility, RelationshipType, AttributeType
These are used in place of static constants and are very much self-explanatory.

2.1.2 View

Each top-level model element (ClassModel, Relationship, TextLabelModel) has an associated view class: **ClassRectangle**, **RelationshipArrow** or **LabelView**, respectively. These all act as Observers to their associated models. They all contain methods that relate to painting and laying out the elements on the canvas. They also perform some controller duties, mainly while loading an existing document from a file and when adding sub-elements to their model as model classes cannot do this themselves. All three inherit from the abstract class **DocumentElementView** which extends JPanel. This lets views be added to the diagram canvas as regular components. More information about what these classes do when the program runs can be found in descriptions of the implementation of individual use cases.

uk.ac.aber.dcs.cs124group.view

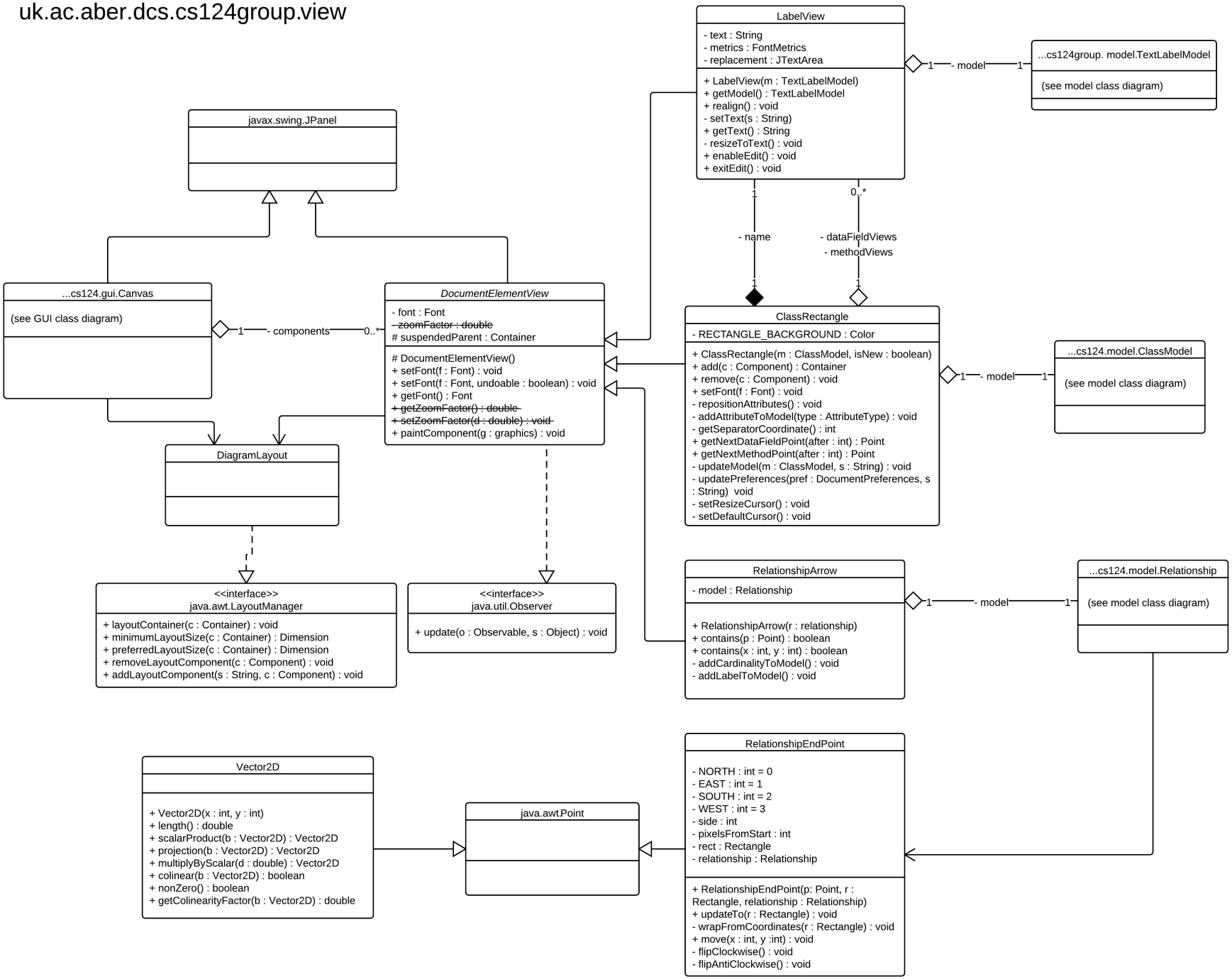


Diagram Layout
We actually had to define our own LayoutManager for the task of laying out components on the canvas. DiagramLayout works very similarly to absolute positioning — it polls the preferred size and location of the component in question and simply obeys its wishes. No other LayoutManager in the Java API was appropriate for this task, however.

Geometry aides: Vector2D and RelationshipEndPoint
The Vector2D class is used heavily where relationships are concerned — we are not dealing with rectangular shapes anymore but lines. This requires some rather complicated geometry which would be impossible without the use of vectors. The RelationshipEndPoint class restricts the movability of java.awt.Point so that the endpoints of a relationship are unable to leave their assigned class rectangle.

2.1.3 GUI

The GUI package manages the overall layout of the program, without interfering with the backend. All events triggered by the GUI's elements are handled by the Manager class.

2.1.4 Exporter

Our link with the outside world. The Exporter class handles exporting the class diagram into stub Java code or image files while the PrinterDriver handles printing. Most of the methods in the Exporter deal with extracting information from the model and then producing the contents of .java files.

2.1.5 Controller

The multi-listener adapter: DiagramListener
Due to the huge amount of methods listeners have to implement that we don't always want, we defined our own adapter class implementing KeyListener, MouseMotionListener and MouseListener. Listeners extending this class will override only those methods that are relevant to their operation.

The controller triad: ClassController, LabelController, RelationshipController
As with views, each of the three controllers has an associated model class. The controllers are added to views as listeners and when events happen on the views (such as the user clicking on a menu item in the right-click popup menu), they call the appropriate model methods. The controllers cannot access views directly — every single user action has to go straight through the model. Views are then updated in the usual Observable-Observer way. This ensures no user actions are "lost" in the views but it does require some additional overhead when the user requests the addition of a sub-element that only the view can perform. Methods in the model starting with "request..." are parts of this overhead.

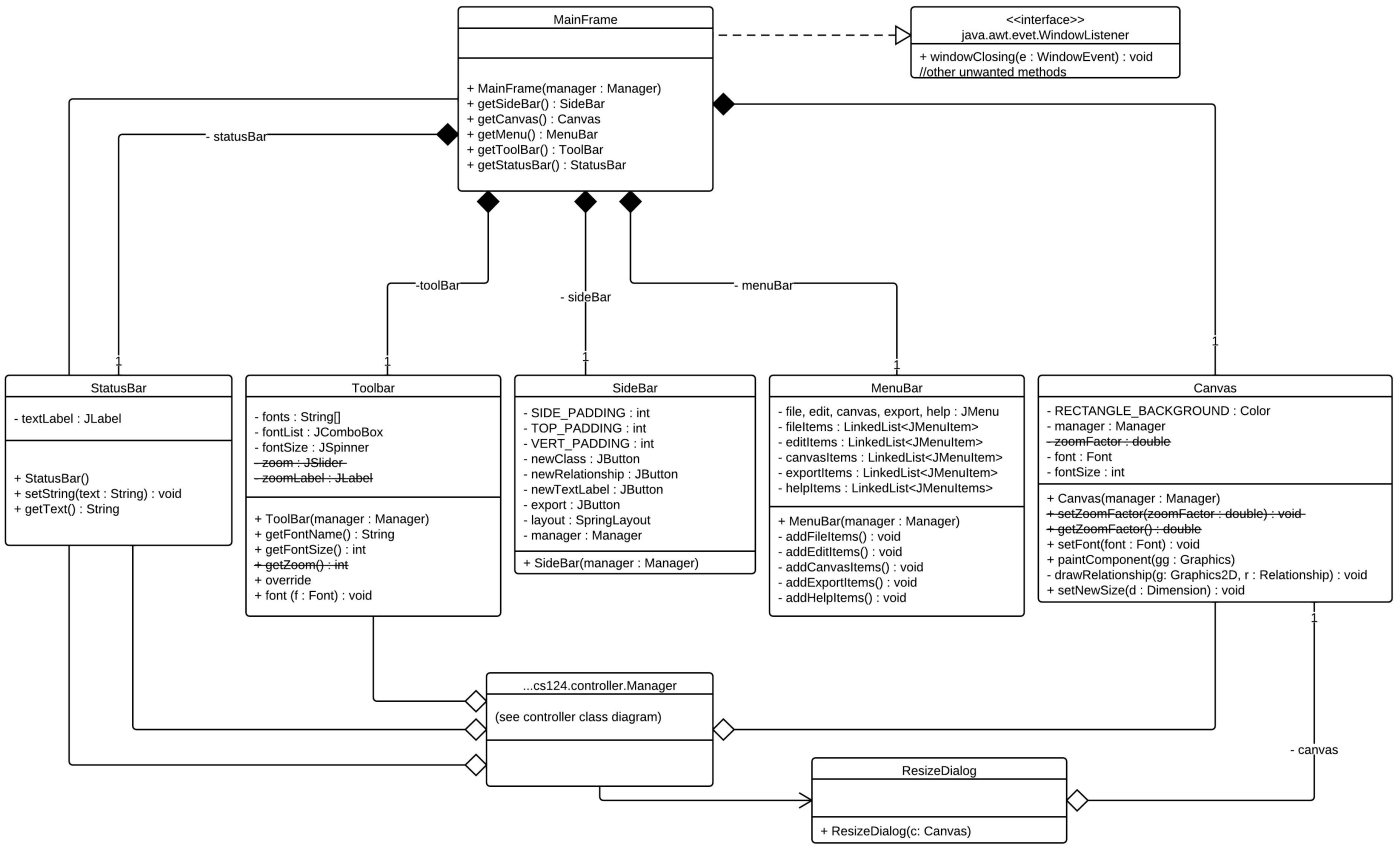
Manager
Originally, the Manager class was meant to be the only controller in the program. As the program grew, many of its duties had to be delegated to the elements themselves to avoid it growing into ridiculous size. It still retains, however, the task of listening to all of the events that occur in the GUI classes. These events are interpreted and appropriate methods are called. In general, the Manager controls anything that is global to the current document. It also manages adding top-level elements to the document. The heart of the undo functionality also resides in the Manager as undos must be global to the document.

ListeningMode: Turning a deaf ear on unwanted events
A controller will not always want to react to events in the same way. The ListeningMode enum defines possible states in which a controller might be. Events will then be reacted to according to the mode currently set on the controller.

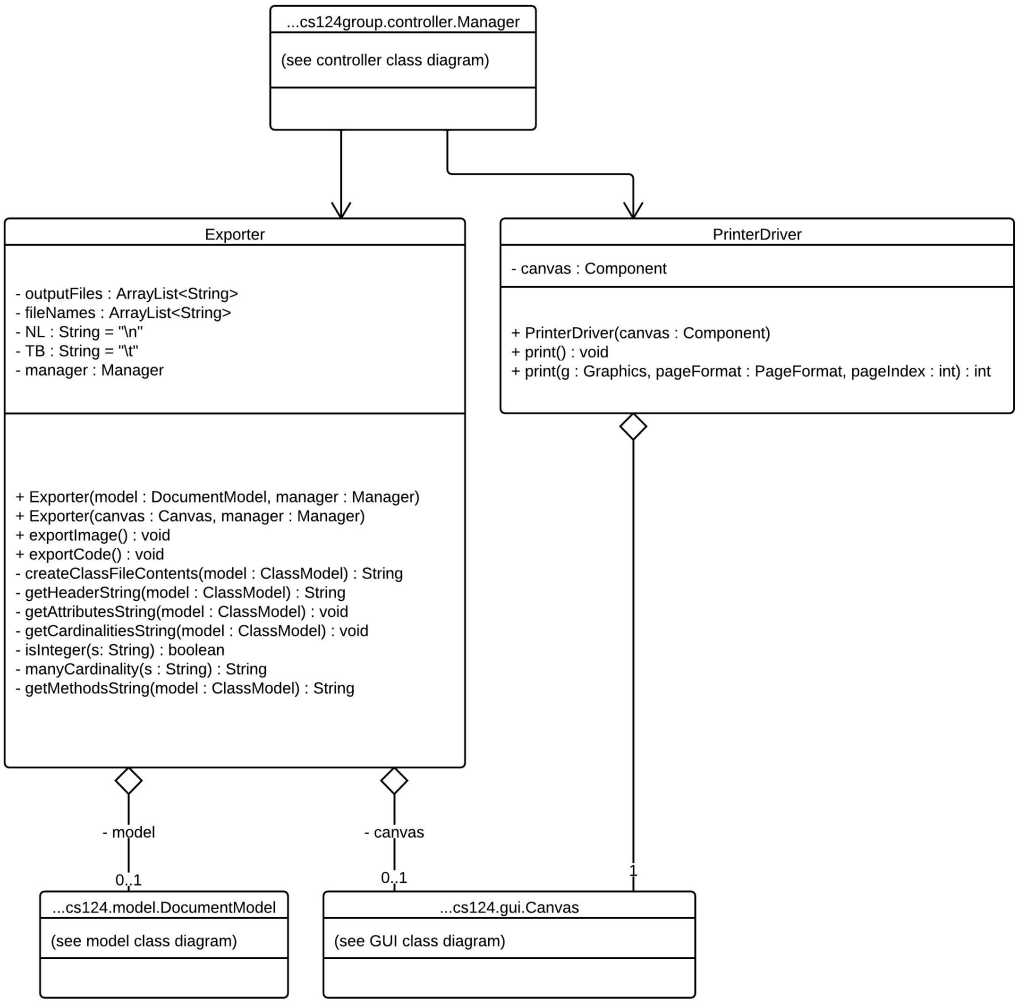
2.1.6 Undo

The classes in the undo package closely follow the Java API specification for in-built undo support. Each of these classes represents a type of undoable edit. The edits store information about the former state and the latter state of an object and toggle between the states as requested by the user. The undo manager, the Manager class in our case, stores these edits and calls their undo() or redo() methods as needed (this is implemented in javax.swing.UndoManager which our Manager class extends). This architecture is based on the Commander design pattern. The presentation name attribute of these edits is used to update the status bar at the bottom of the application window.

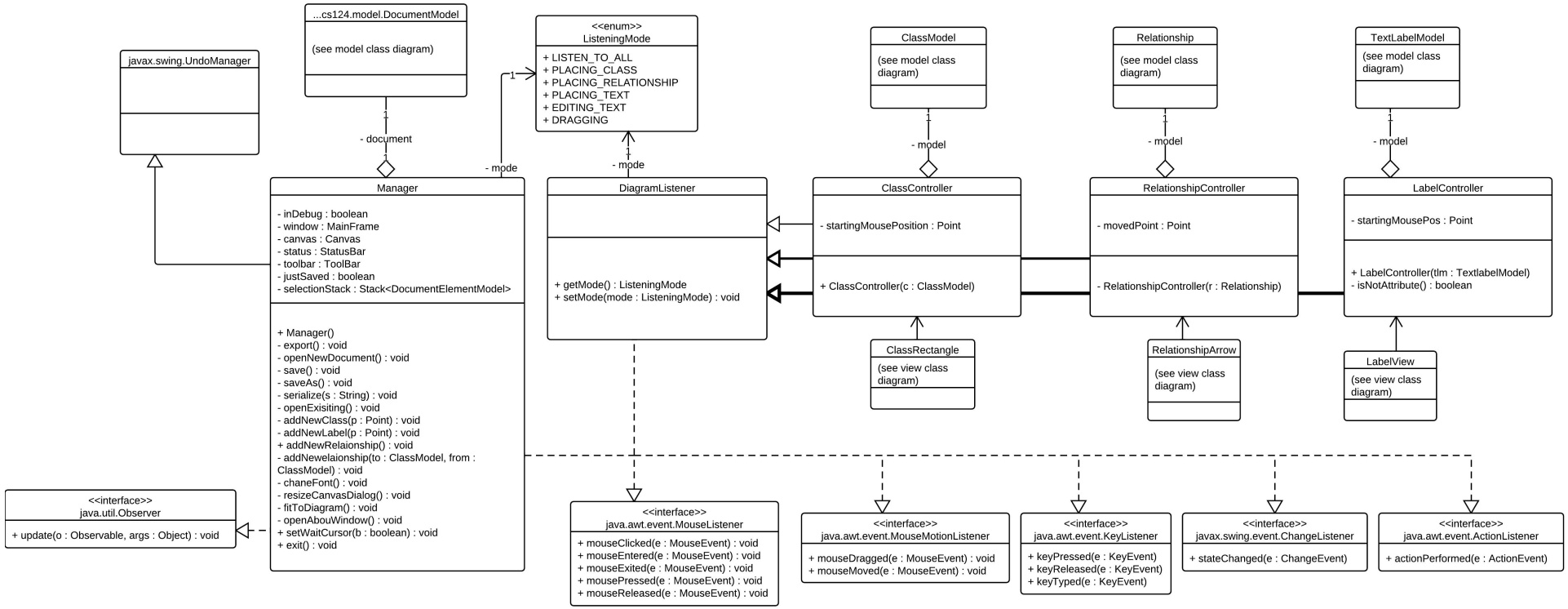
uk.ac.aber.dcs.cs124group.gui



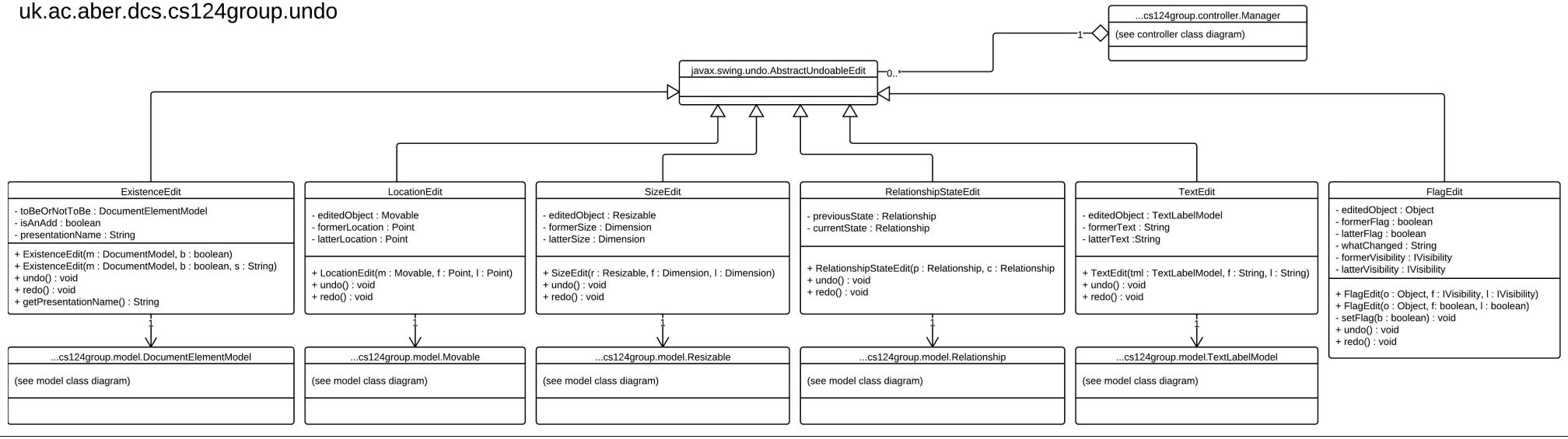
uk.ac.aber.dcs.cs124group.export



uk.ac.aber.dcs.cs124group.controller



uk.ac.aber.dcs.cs124group.undo



2.2 Algorithm Descriptions

2.2.1 Launching The Application

The application entry point is in the Run class. This class creates a new Runnable that gets inserted at the end of the event dispatch thread (this is a simple precaution — we don't want to throw exceptions right after we launch). The Runnable creates a new Manager object that creates and bootstraps the whole GUI and then creates a new document.

2.2.2 File Management Use Cases

Creating a new document
The command to open a new blank document is issued from the menu bar or when the application is launched. When the user clicks the appropriate menu item, the event gets sent to the Manager which calls its own openNewDocument() method. If any unsaved, user-initiated actions have been performed on the current document, a "do you want to save?" dialog appears. If the operation isn.t cancelled by the user (possibly after having saved), the canvas is emptied, the current DocumentModel loaded in the Manager is disposed of in favour of a new one, all undoable edits are discarded and new DocumentPreferences are set into the DocumentModel. Finally, the status bar is updated.

Saving a document
Again, the command to save a document is initiated in the menu bar. Depending on the exact menu item clicked, save() or saveAs() is called in the Manager. If a filename is not associated with the document in the DocumentPreferences object, saveAs() is called anyway. This method brings up a JFileChooser letting the user select where the document should be saved to. Once a filename is obtained, serialize() is called and the DocumentModel currently in the Manager is simply written to that file using default Java serialization.

Loading a document from a file
Loading a document is slightly trickier. We need to recreate views and controllers for all model objects and couple Observable-Observer pairs properly. When the command is issued by the user, the openExisting() method is called in the Manager. First, we need to dispose of the current document. The second stage is simply reading the Document-Model object from the file the user selected (again by means of a JFileChooser). It is loaded into the Manager which then: calls cleanUp() on the DocumentModel which will cause it to discard all non-existent elements, calls getView() on each DocumentElementModel, adds the returned DocumentElementViews to the Canvas, ties up Observers with Observables, reads DocumentPreferences on the loaded object and sets the document properties accordingly. The individual DocumentElementView constructors do very much the same for sub-elements (e.g. views are created and added for Attributes and so on). They also create and assign controllers.

Exporting a document to Java code
Exporting to code can be triggered from two locations, either from the MenuBar under Export or from the SideBar under "Export to Java...". This calls the Exporter constructor using DocumentModel that contains all the objects and information needed to create the code. Within the Exporter class there are five private methods which will create the code sections, createClassFileContents(), which pulls the following four methods together into one string to be used within the exportCode() method. getHeaderString() creates the class header along with any classes that will be extended or implemented, getAttributeString() will take all the variables written into the class to be created, get the visibility, name, type and any modifiers that will be added to it. getCardinalitiesString(), depending on the type of cardinalities and relationships, will create fixed or open arrays or ArrayLists, respectively with the ClassName as the type. getMethodsString() will take methods that the user enters, again taking visibility, name, any parameters and their types, and the return type of the method. The exportCode() method will then run through an ArrayList of all the filenames and Strings created and output each file as a .java filetype. The selection of output directory is facilitated by a JFileChooser.

Exporting to an image
Another Exporter constructor will take the canvas and render it into a BufferedImage, and create a graphic from this. A JFileChooser will allow the user to change the image type of .png, .gif or .jpg. and save the file accordingly.

Printing a Diagram
This will be fairly similar to exporting an image, using the canvas a constructor parameter. A print() method will again create a graphic to be used, but will be scaled depending on the default paper size from PageFormat before printing to the selected printer.

2.2.3 Software Design Use Cases

Creating a class (includes setting class name)
When the "New Class" button is clicked on the side bar, the Manager receives the event and sets ListeningMode.PLACING_CLASS on itself. The coordinates of the next mouse event that will be received from the canvas will be sent to the addNewClass() method in the Manager. This method resets the listening mode, creates a new ClassModel, assigns a view (ClassRectangle) to it, bootstraps Observer-Observable pairs and adds the view to the canvas. It also creates a new ExistenceEdit holding the new ClassModel (more on undoable edits in the section about undo functionality). As soon as the class is created, the TextLabelModel that acts as the name label is asked to enable editing on itself so the user can type in the name of the class immediately (this happens in the ClassRectangle constructor which also creates and assigns a ClassController).

Adding attributes to a class
This use case is not handled by the Manager as the request does not come from the GUI or the canvas. The request to add a new attribute comes from a class rectangle right-click menu and so the appropriate ClassController will receive the event. The ClassController then calls requestNewDataField() or requestNewMethod(), as appropriate, and the ClassModel's view (ClassRectangle) is notified. addAttributeToModel() is called in the ClassRectangle which creates the Attribute object, assigns a LabelView object to it, adds the LabelView to itself, adds the Attribute to the ClassModel, couples the Attribute with the LabelView and immediately enables editing on the new Attribute. An ExistenceEdit is also created and sent to the Manager.

Behind the scenes: Derive attributes from UML representation
Instead of using standard String manipulations, we decided to match them against a Regex pattern, as some people enter UML notation differently and a lot of the notation is not mandatory. Therefore we created 3 different Regular Expressions: one for matching Data fields; another for the "shell" of the method (the method without any parameters); and finally one for the method parameters. Then using Java's incredibly powerful Regex engine, which automatically splits the up the matched string into specified groups, we can individually pull out parts of the string to test them against UML patterns and set the variables accordingly. If the Attribute representation doesn't conform to UML, the whole Attribute is flagged as invalid.

Editing modifiers and flags
Modifiers and flags can be set or unset through right-click menus on class rectangles or attributes. The appropriate ClassController (or LabelController for Attributes) receives the event, sets the appropriate modifier or flag (or combination thereof) by calling a setter in the model. This creates a FlagEdit and sends it to the Manager. The font style is overridden accordingly on the name label of the class or on the attribute string, whichever is applicable.

Adding relationships
Adding a relationship between two classes is a three-stage process. It can be triggered either from the sidebar or from a class right-click menu. Either way, the Manager (which implements Observer only for this purpose) receives the request and sets ListeningMode.PLACING_RELATIONSHIP on itself. If the request comes from a class, that ClassModel is added to the selectionStack in the Manager. If it comes from the sidebar, addNewRelationship() with no arguments is called in the Manager. The Manager will now listen to changes in the paint state of ClassModels. Once any one ClassModel passes into SELECTED state (as set by a ClassController), it is added to the selectionStack and from this point, both ways of the process follow the same path. The listening mode does not change, and the Manager waits for a second class to be selected. Once that happens, addNewRelationship() is called with both ClassModels and a new Relationship along with a RelationshipArrow is created in much the same way as a class. Listening mode is set back to LISTEN_TO_ALL, the selectionStack is emptied and an ExistenceEdit is created.

Changing the relationship type
The type of the relationship can be changed in the relationship right-click menu. The appropriate RelationshipController receives the event, creates a RelationshipStateEdit to store the current state of the Relationship and changes the type of the model. The view is then repainted.

Adding cardinalities / labels to relationships
See section on adding Attributes to ClassModels. The process of adding sub-elements to Relationships follows exactly the same process, especially since Attribute, Cardinality and RelationshipLabel all inherit from TextLabelModel.

Removing elements

Removing elements is simple. A class or a relationship may only be removed through the right-click menu but any textual element may also be removed by deleting all text and pressing ENTER. Both produce the same result — the userRemove() method is called on the appropriate DocumentElementModel which sets the "exists" variable to false and creates an ExistenceEdit, sending it to the Manager. Observers are notified and when a view learns that its model has been removed, it sets itself to invisible, removes itself from the parent container but keeps a reference to it in the "suspendedParent" variable. It is important to note that the whole MVC triad of the element remains in the program and the DocumentElementModel remains in the document until cleanUp() is called on DocumentModel. This is so that undoing a remove can easily bring back an element into existence.

Editing textual elements

Whenever the user double-clicks a LabelView, the LabelController will set editing to true in the model. Through update(), this will call enableEdit() in LabelView. This method removes the whole view from its parent, replaces it with a JTextArea at the same location, requests immediate focus for the JTextArea and selects all the text within it. The LabelController assigned to this LabelView will now listen to the JTextArea as well. Once the user presses ENTER, this process is rolled back by the exitEdit() method. The text of the label is set to the text of the JTextArea and the label is re-added to the parent component. Also, a TextEdit is created and sent to the Manager. If no text is in the JTextArea as it is validated, the model is userRemove()'d.

2.2.4 Diagram Editing Use Cases

Moving and resizing class rectangles

Moving class rectangles is simple enough. Whenever the mouse is dragged on a ClassRectangle component, the ClassController updates the location of the ClassModel which then notifies its observers. A LocationEdit is created and sent to the compoundEdit residing in the ClassModel. Once dragging is finished, the whole compound edit is sent to the Manager. The ClassRectangle changes its own location in accordance and the Canvas is forced to re-layout. However, we need to think about Relationships as well. This is where the RelationshipEndPoint class comes in. Each RelationshipEndPoint is defined as having a precise position on the border of the class rectangle. When this changes its (x,y) coordinates, the (x,y) coordinates of the RelationshipEndPoints can be updated easily as well. For this reason, Relationships implement Observer. Whenever a ClassModel changes location, the new location is passed to RelationshipEndPoints through the update() method in Relationship. The endpoints are then forced to update their coordinates to the new rectangle. Resizing class rectangles works in a similar way. Whenever the user mouses over the bottom-right corner of the class rectangle, the cursor is set to system default resize cursor and the element paint state is set to MOUSED_OVER_RESIZE. While this paint state is set, any dragging motion will result in the class rectangle being resized. The rest of the process follows that of moving class rectangles, except instead of LocationEdits, SizeEdits are being created.

Text labels outside any other element

Adding explanatory text labels to a class diagram is a useful feature. It is implemented by simply creating a new TextLabelModel from the Manager (in much the same way as any other element). Please refer to the section on modifying textual elements — free labels adhere to this model.

Controlling relationship arrows

Relationship arrows only receive mouse events when the mouse is actually on the arrow, even though the actual component size is many times larger. This is done by overloading the contains(Point) method (for further details on this method, please refer to the javadoc). Whenever the user clicks on a point that is already present in the ArrayList of Points in the Relationship object, the Point can be moved around. If this point is an endpoint and therefore attached to a class rectangle, the overloaded move(int, int) method in RelationshipEndPoint ensures it stays there. If a user clicks and drags somewhere else, a new point is added to the line. The addPoint() method is called in Relationship. It checks all the existing segments and determines to which one the new point should belong. It then inserts the Point object into the ArrayList at the appropriate index. Removing a point is done by double-clicking on it. The RelationshipController simply calls removePoint() with the mouse event coordinates and the method determines whether a point should be removed, and if so, which one. Updating the view is done in the usual way.

Moving labels and cardinalities on Relationships works the same way as moving any other element. The only difference is that the movement is restricted — a Cardinality may never be further than a certain distance from its associated endpoint. Similarly, a RelationshipLabel cannot leave its Relationship. This constraint must be enforced not only for user actions but also for when relationships move as a result of something else moving. For this reason, Cardinalities and RelationshipLabels both have a fixed relative position to a reference point on the relationship arrow. When this reference point moves, realign() methods are called in the sub-elements and these reposition themselves. This way, a relationship cannot "run away" from its cardinalities.

Undo and redo functionality

The undo functionality in the application is built atop the Java UndoManager framework. The Manager class extends UndoManager which, in turn, implements UndoableEditListener. DocumentElementModel has an addUndoableEditListener() method through which the Manager is assigned as a destination for UndoableEditEvents to each and every element. Whenever a user-initiated action is acted upon, an UndoableEdit is created and sent to the fireUndoableEvent() method in DocumentElementModel. This method wraps the edit in an event and sends it to the Manager. From there on, undoing is defined by the Java API. Our undo package defines six types of UndoableEdits, all extending the AbstractUndoableEdit class. All of these store information about the previous state and the current state of the model object in question and override the undo() and redo() methods, toggling the states as needed.

A note on existence. As removed objects stick around in the program, it is relatively easy to bring them back into existence using the resurrect() method defined in DocumentElementModel. They can also be removed again, this time by using remove(). However, once cleanUp() is called on DocumentModel, they disappear from it. Trying to undo removes after that might still work but will create unpredictable behaviour when saving is thrown into the mix. When a document is exported into java code, it is cleaned up as we don't want removed elements to appear in the actual code. This is why exporting disables all undos and redos on edits created before exporting was initiated. This is clearly a bug in the application, albeit forced, and we will be looking to fix it in the future.

A note on relationship states. Undoing actions on relationships works slightly differently than the other edits. In order to avoid having to create lots of edit classes for the many operations relationships allow to be performed on them, a whole relationship is simply cloned into a RelationshipStateEdit. When undoing, the current object simply copies all of its data from the cloned previous version.

Resizing the canvas

The Canvas is contained in a JPanel that has DiagramLayout set on it. This means that it will respect the canvas' preferred size no matter what it may be. It allows us to let the user resize the canvas to his liking. A canvas can either be snapped to the minimum size to contain all of its components through the fitToDiagram() method in the Manager or a resize dialog can be thrown up to let the user specify an exact size in pixels. This is implemented in the resizeCanvasDialog() function in the Manager and a separate ResizeDialog class. The workings of this functionality are trivial. More information on it may be found in the javadoc.

Changing the font

When the user changes the font name or the font size on the toolbar, the changeFont() method is called in the Manager. This method simply sets the new font into DocumentPreferences and it is up to the Observers of DocumentPreferences to set their own font accordingly. Each top-level element has a clause in the update() method that lets them respond to changes in font. Also, top-level elements have to force their own font into their sub-elements by directly calling their setFont() method.

Changing the zoom

This is a feature that was partially implemented at the very early stages of development. As the program grew, the support for the zoom feature throughout the program waned and it was decided that it would be completed later on. However, due to time constraints, this "later on" never materialized as implementing zoom got eclipsed by much more important features to implement and debug, such as undos. Completing the zoom may or may not be very difficult; however, there just wasn't enough time to try it out. The zoom slider in the GUI is intentionally disabled and all methods relating to zoom are currently deprecated throughout the code. We are planning on fixing this in the future.

3 Testing

Test Done	Expected Result	Actual Result	Comments
		Drawing	
Adding a new class rectangle	New class rectangle created at where the mouse is clicked with default values	As expected	
Adding a second class to the canvas	Creating a new class rectangle at a different location to the first	As expected	
Adding a class on top of an existing class	Will not do anything	As expected	It will not dispose of the event, so if you click on some white space with no class rectangles on, it shall create a new class rectangle
Adding an independent label to the diagram	Creating a note to the diagram which isn't a part of the UML	As expected	There is no reference to them in the exported code
Drawing a relationship between two classes	Create a relationship between two classes with default values (A uses type and no cardinalities)	As expected	Later functionality might be adding it to the closest corner, as it defaults to the top left, to make it look nicer and easier to edit.
		Class Rectangles	
Adding a new class name	Changing the name of the class from the standard "NewClass" to the "HelloWorld"	As expected	
Adding a Data Field	Rightclick on the class diagram and click on Data Field, An editable version of the label comes up. Change to "- test : String" and press enter	As expected	
Adding a Method	Similar to Data field, but in a section below with the method UML syntax	As expected	
Adding a second Data Field	Should place a new data field below the existing data field	As expected	
Changing the class modifier to "Abstract"	Right clicking the Rectangle and choosing "Abstract" from the modifiers menu	As expected	However choosing the "Final" option will not show any difference in the font, as UML defines no notation.
		Attributes	
Double click to edit	Makes the label editable and hitting ESC or Return will stop editing	As expected	Doesn't lose focus when mouse is clicked somewhere else, and is still editable if we place a new class rectangle, for instance.
Changing the Modifier to both static and final	Right clicking on the attribute brings up the modifier menu. Do this twice and choose static and final. (Doesn.t matter which order)	When choosing static, the text changes and the option shows, but when final is chosen, it reverts to the option none but the text keeps the same.	This is a simple bug to fix . as it.s just cosmetic option in the popup.
Removing the Attribute	Right click and select remove on an attribute will delete it	As expected	Can also be done by deleting all text and pressing the return key

Relationships			
Moving a point around a rectangle	Move a point from the top left to botom right	As Expected	The point doesn't move off the side of the rectangle, and cannot be pulled off.
Creating a cardinality from one class to another	Make the value of the cardinality 0..*	As expected	The cardinality starts off on top of the class rectangle, but is a cosmetic bug which can be fixed at a later time
Change the Relationship type from Uses to Implements	Draws a filled in arrow at the start class	As expected	
Add a label	Create a new label on the relationship which simulates a data field on the class diagram	As expected	
Adding a new point onto the relationship	Click and drag a new point off the relationship to move the line	As expected	All labels (cardinalities and attribute label) still keep relational positioning and not able to delete the point.
Exporting			
Exporting a UML Class Diagram	Two classes have been created Monster (with a private variable scariness and a public method scare), and abstract class Evil (with a final static variable "markOfTheBeast" and abstract function Burn) Monster inherits Evil	Code shows everything there	
Exporting a UML Class Diagram with cardinalities	Two classes have been created (Duck and Pond) with no variables or methods. A composition relationship has been drawn with a cardinality on duck being 0..* (as there can be many ducks on a pond). A label has been created on the relationship called -ducks.	Code shows everything which we designed.	
Undos and Redos			
Undoing a Class Rectangle deletion when a relationship was deleted beforehand	2 Classes with a relationship between them. Delete the relationship and then delete the class rectangle. Undoing should only bring back the class relationship.	The relationship is resurrected with the class model.	Fixed when found, and implemented into version v1.0.0
Add/Delete class diagram	Upon undo of add, the Class is removed, upon deletion the class is replaced	As expected	
Resize Class	if class is resized, undo should revert the class back to its previos size	As expected	
Moving the position of a class	when moved, undo should replace the Class to its previous location	As Expected	

Undos and Redos			
Add/Delete relationships between 2 classes	Upon undo of add, relationship is removed, upon deletion relationship is replaced	As expected	
Change relationship Type	Relationship to revert back to it previous setting	As Expected	
Add/Remove cardinality	Upon add the cardinality is removed, upon deletion cardinality is replaced	As Expected	
Change cardinality	Upon Changing the text of a cardinality, the undo will revert the cardinality back to its previous state	As expected	
Add/Remove Data fields	Upon undo, opposite effect should take place	As Expected	
Add/Remove Methods	upon undo, opposite effect should take place	As Expected	
Edit Text labels	Revert back to previous state	As Expected	