Project Design Document

CASE Tool

Version 1.7

Name	Student ID
Jose Luis Saldivar	5335906
Sahadat Hussein	6196187
Ryosuke Izu	5344107
Juan Jose Balcazar	6018351
Vuong Kien Phong	5347424

Table of Contents

1. Introduction

The purpose of this document is to collect, analyze, and define the high-level needs and features of CASE Tool, a visualized desktop application for for the creation of UCM's (use case maps). This document will describe the product functions, graphical user interfaces, Design details and constraints of the software. The details of how the CASE Tool fulfills required needs are detailed in the use case with collaboration and sequence diagram. The design will show a visual implementation plane of whole project.

1.1. Purpose

This Software Design Document (SDD) describes the plane to implement the software with a set of use cases, which is in partial fulfillment of the requirements of COMP 354. It is following the high-level requirements of the user interfaces, product functions, user descriptions, assumptions and dependencies, constraints, specific requirements and an analysis model. The analysis model will include use case diagrams, class diagrams, sequence diagrams, class model and description of classes with functionality in details. Furthermore, a detailed project design plan will be provided a complete documentation and plan to progress the implementation. This document is intended primarily for the members of Team 10 and the project coordinator, Dr. Juergen Rilling, as it will serve as a basis for the upcoming phases of the project.

1.2. Scope

This document only addresses the design of the CASE Tool that will be used as a basis for the implementation phase. Screen shots of the user interfaces will give one an idea on how the interface will look once it is completed. The use case diagrams will give an overview of the functions of the software and how the users will interact with the tools. The class diagrams will show the inter-relationships between the different objects in the software and the sequence diagrams will model the flow of logic within the software. The sequence diagrams and cooperation will specified all the functionality following requirements. They will help for implementation.

1.3. Definitions, Acronyms, and Abbreviations

Components

These are rectangular boxes representing data items such as components in a system performing particular functions or representing the systems themselves. Component boxes can be displayed within a general level component box containing these sub-boxes.

Responsibility Points

These indicate which components the use cases are operating on.

Scenario Paths

These are the lines representing the use cases. Each scenario path crosses over responsibilities and cannot exist without them.

1.4. References

1. A Software Design Specification Template, **Brad Appleton**, http://www.cmcrossroads.com/bradapp/docs/sdd.html#TOC_SEC3, 1997

2. Naval Battle Simulation System: A Case Study in Software Engineering, LinFang Wang, Presented in Partial Fulfillment of the Requirements the Degree of Master of Science at Concordia University, March, 2002

3. Course Group, Montrealopoly : Project Analysis and Development PlanVersion 1.3, montreal, 2003.

4. Shafique Ahmed, "COMP 354 TA" Concordia University, Interview, Hall building, Sept 24th 2008.

5. IBM, "Rational Rose Beginner's FAQ", IBM, http://www-01.ibm.com/software/rational (Current September 14 Oct, 2008)

6. Pressman, Roger S. Software Engineering: A Practitioner's Approach. 5th ed. Toronto: McGraw-Hill 2001.

7. *Writing Software Requirements Specification*.
http://www.techwr-l.com/techwhirl/magazine/writing/softwarerequirementspecs.html (Current September 11 Oct, 2008)

8. HST SE *Software Requirements Document* Template
https://cabig.nci.nih.gov/workspaces/TBPT/Templates/caBIG_Sw_Req_Doc_Temp.doc (Current September 11 Oct, 2008)

2. System Overview

Provide a general description of the software system including its functionality and matters related to the overall system and its design (perhaps including a discussion of the basic design approach or organization). Feel free to split this discussion up into subsections (and subsubsections, etc …).

2.1. Use case diagram

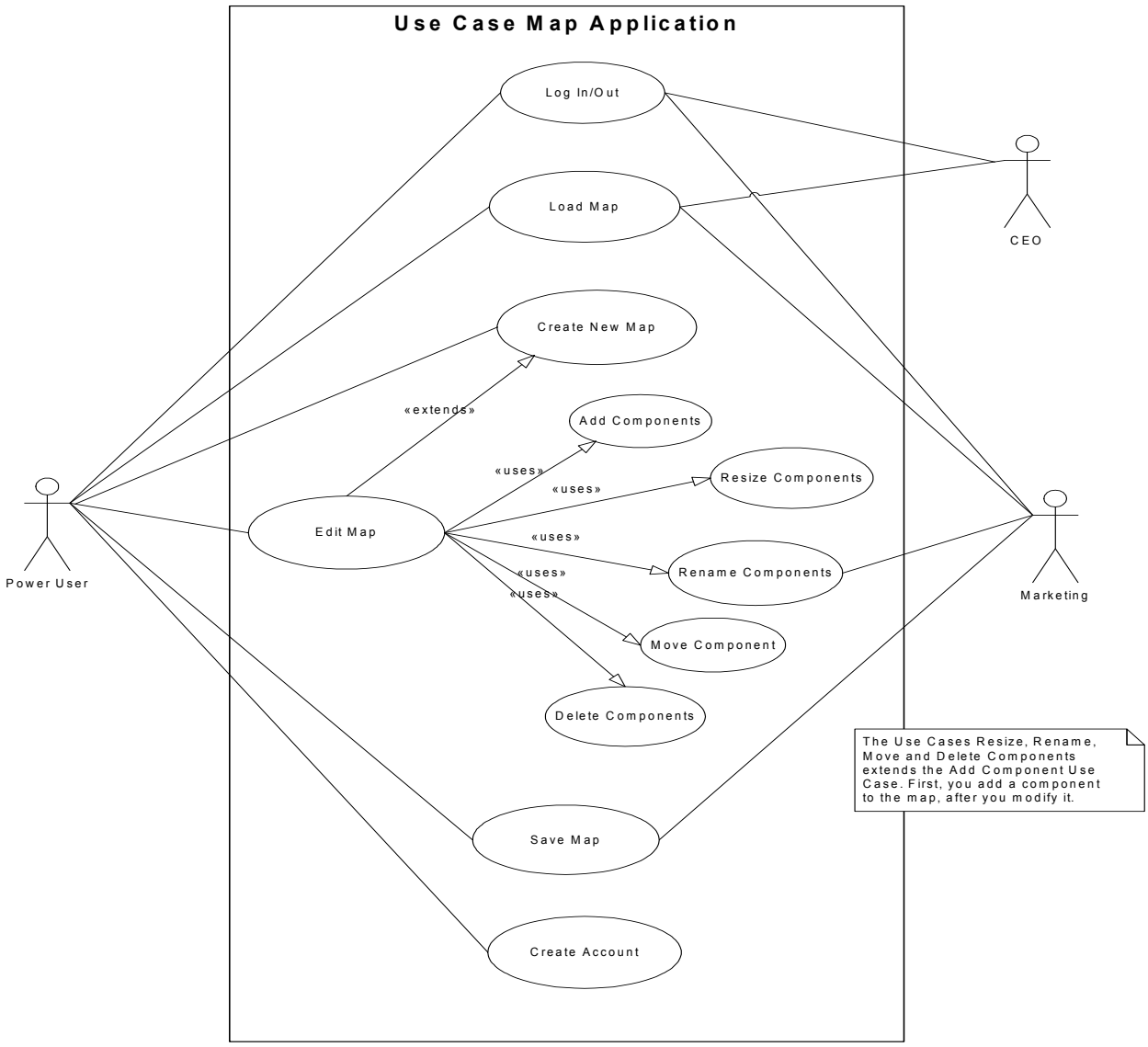

Figure 1: Use Case Diagram

2.1.1. Use case 1: Add new Component to the Map.

2.1.1.1. Sequence diagrams

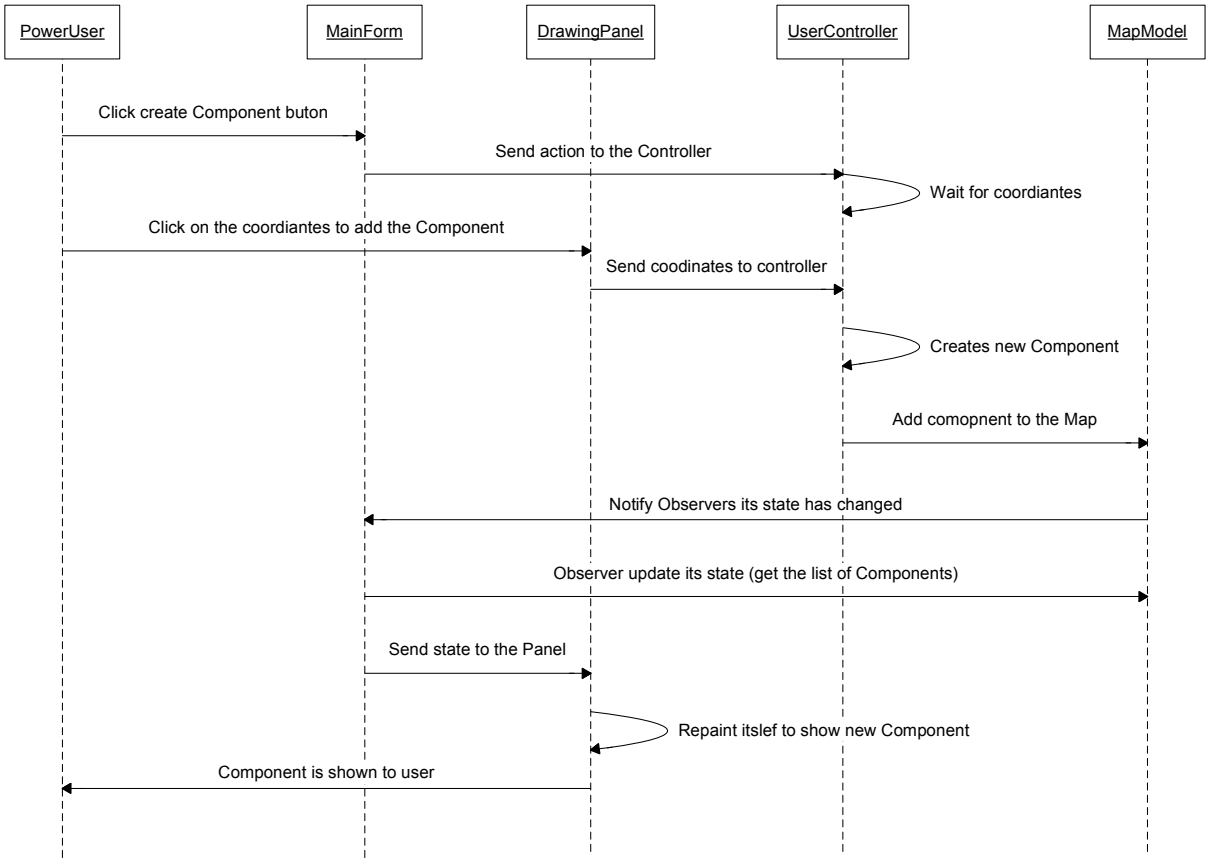

Figure 2 : Sequence Diagram: Add Component to the map

2.1.1.2. *Collaboration diagram*

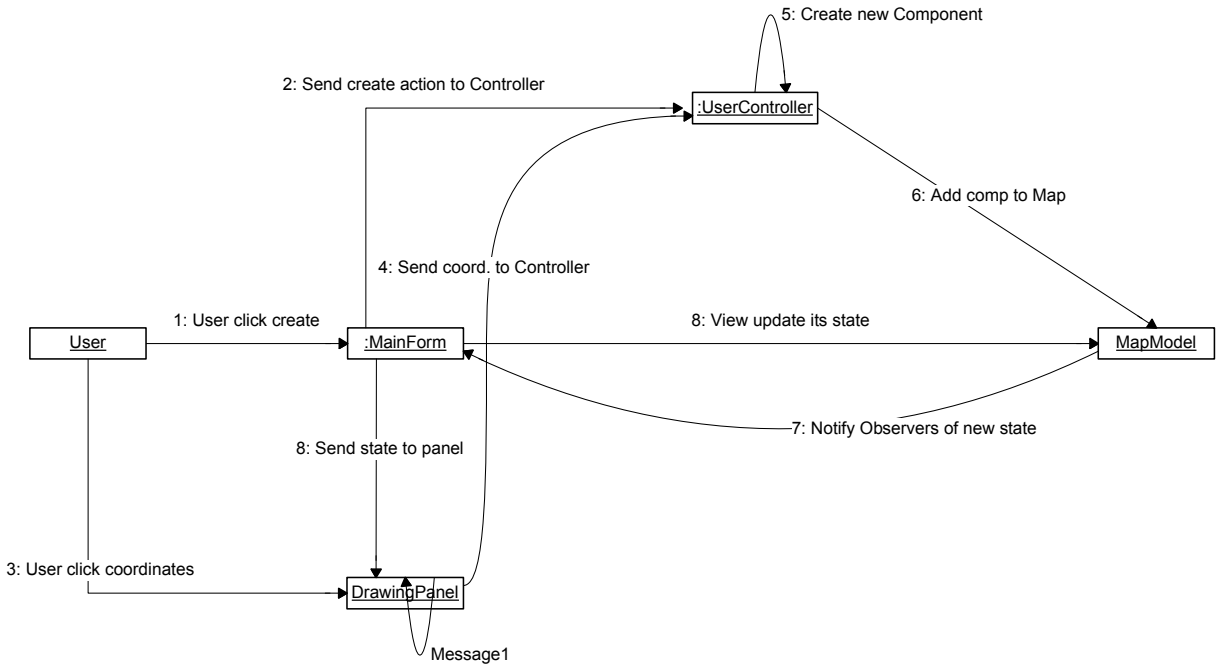

Figure 3 : Collaboration Diagram: Add Component to the map

2.1.2. Use case2: Resized

2.1.2.1. Sequence diagrams: Resized.

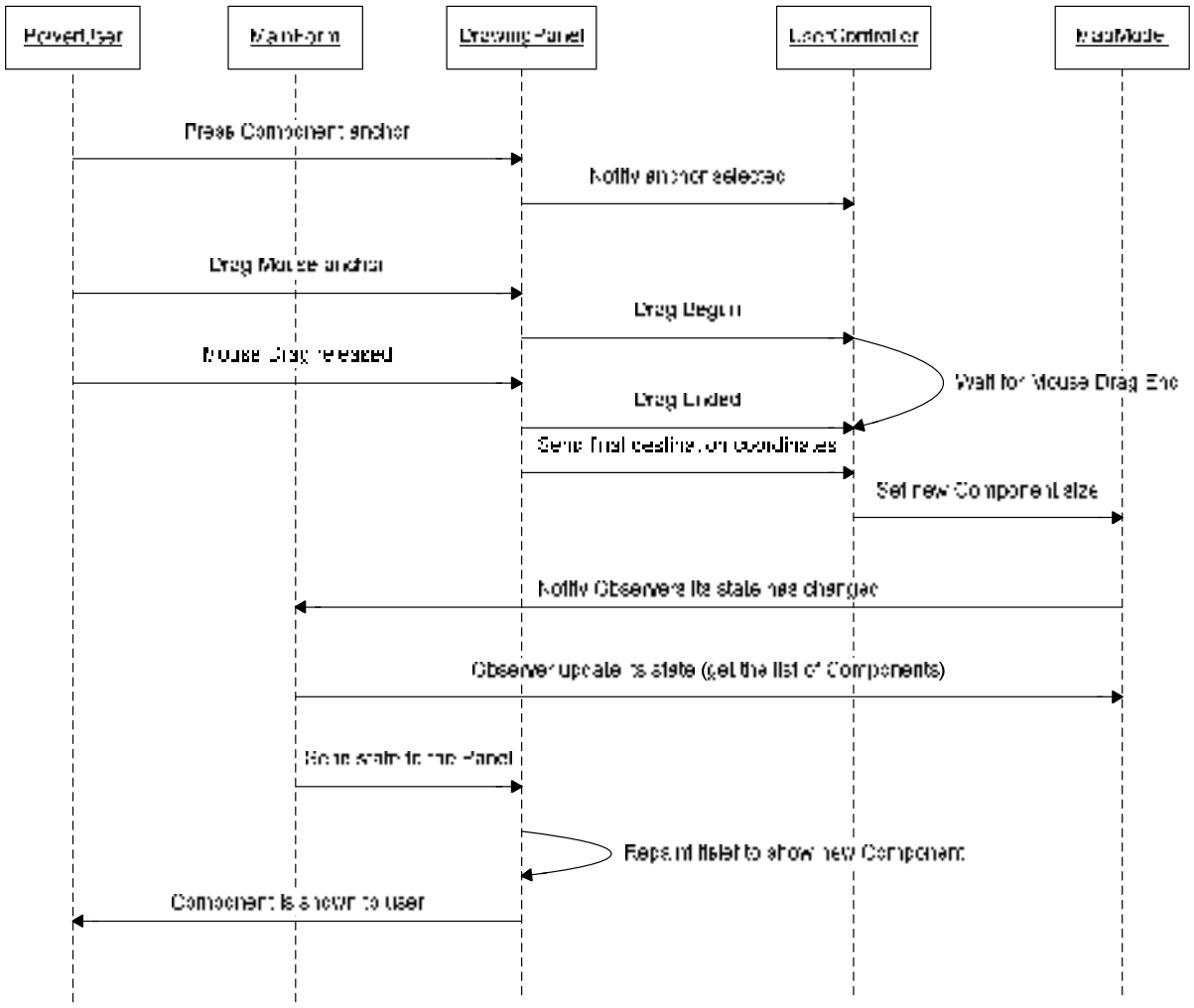

Figure 4: *Sequence diagrams:* **Resized**

2.1.2.2. *Collaboration diagram: Resized*

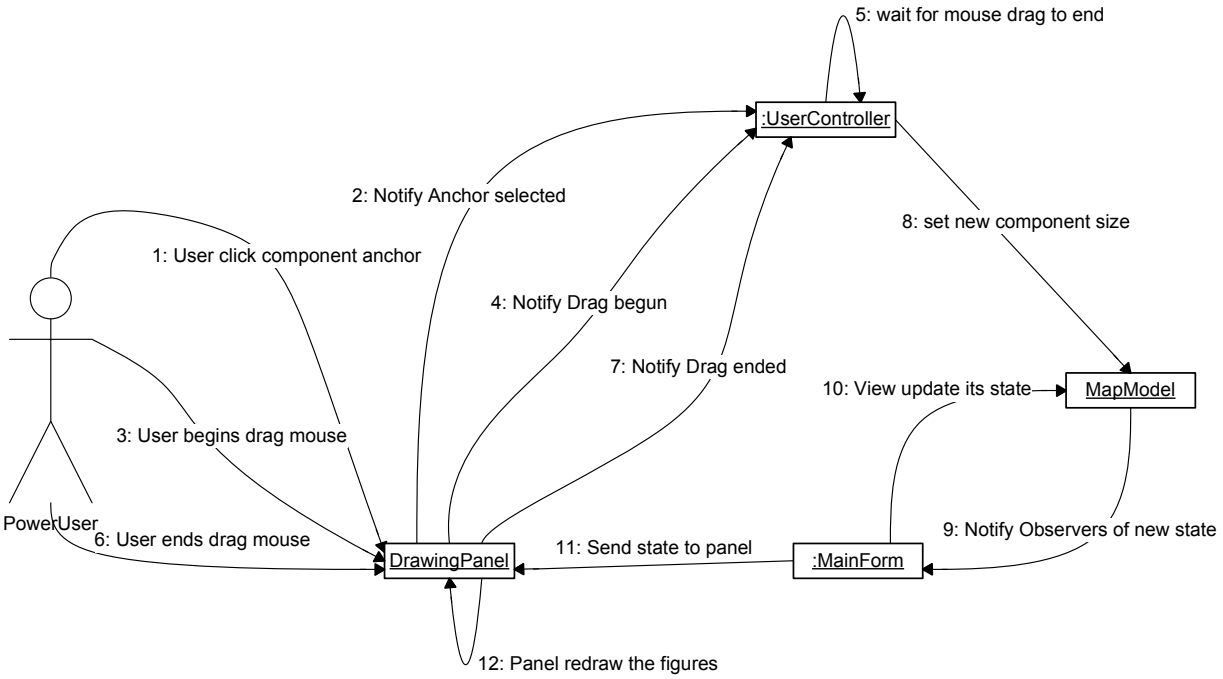

Figure 5 : Collaboration diagrams: Resized

2.1.3. Use case 3:User Login

2.1.3.1. Sequence diagrams

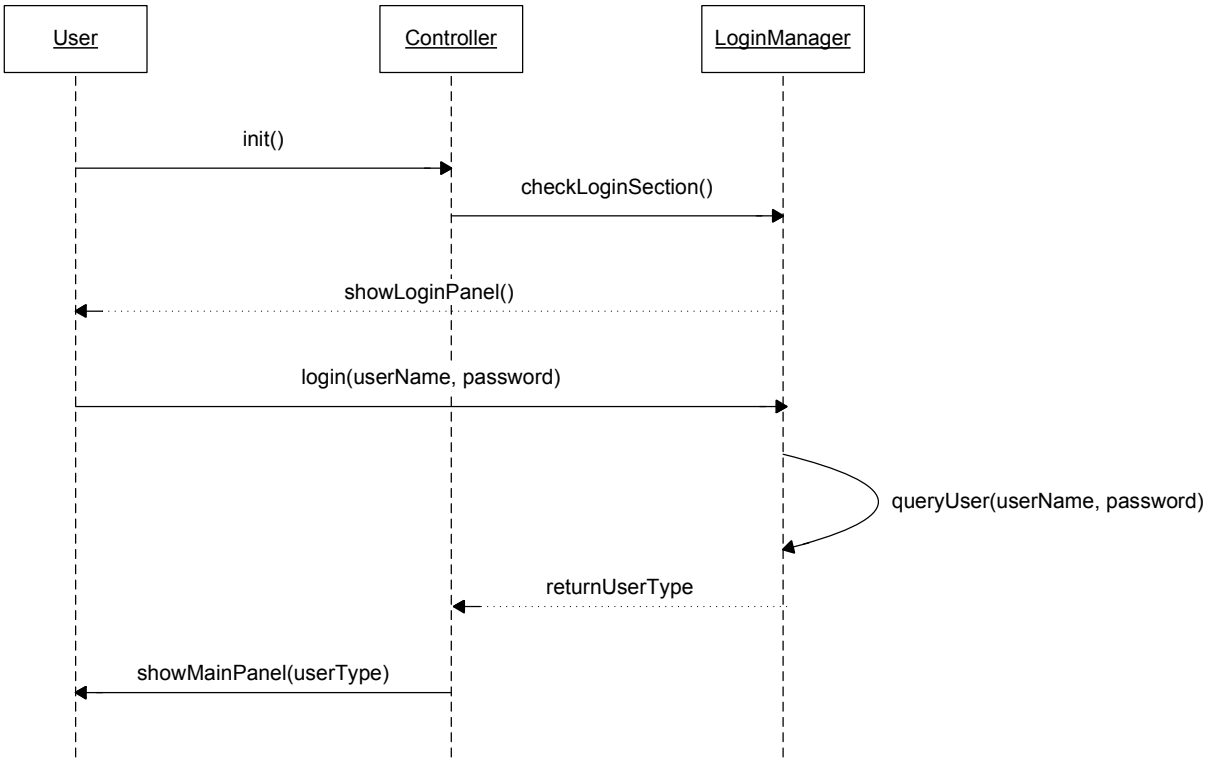

Figure 6: *Sequence diagrams:* **User Login**

2.1.3.2. Collaboration diagrn: User Login

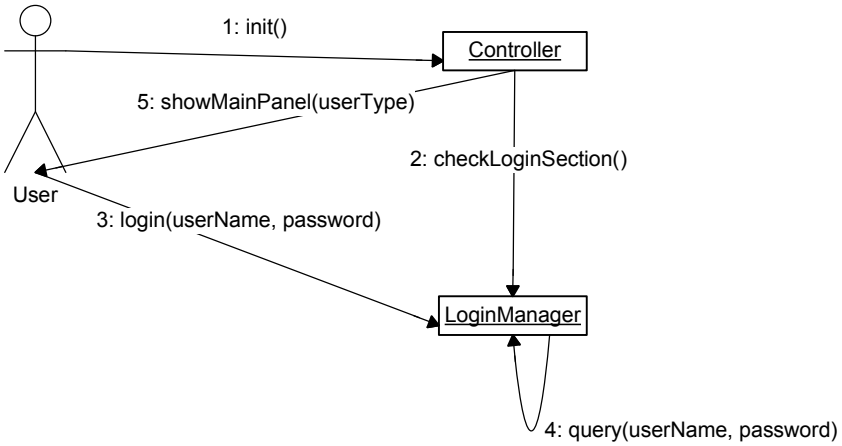

Figure 7 : Collaboration diagrams: User Login

2.1.4. Use case 4: Open New Map

2.1.4.1. Sequence diagrams: Open New Map

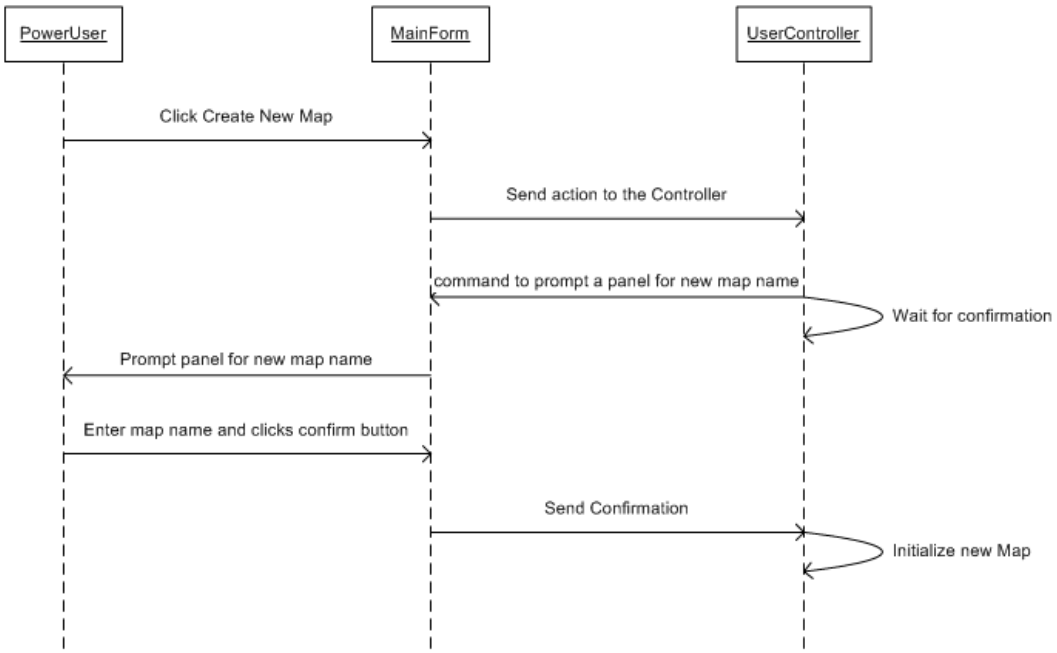

Figure 8: Sequence diagrams: Open New Map

2.1.4.2. Collaboration diagrams: Open New Map

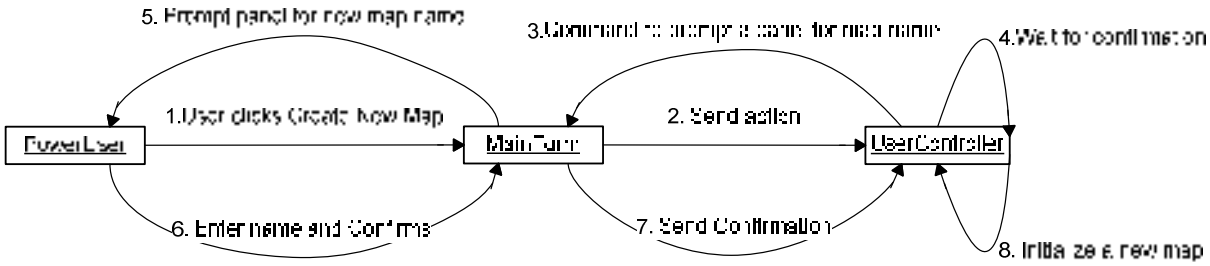

Figure 9: *Collaboration diagrams: Open New map*

2.1.5. Use case 5: save Map.

2.1.5.1. Sequence diagrams: save Map

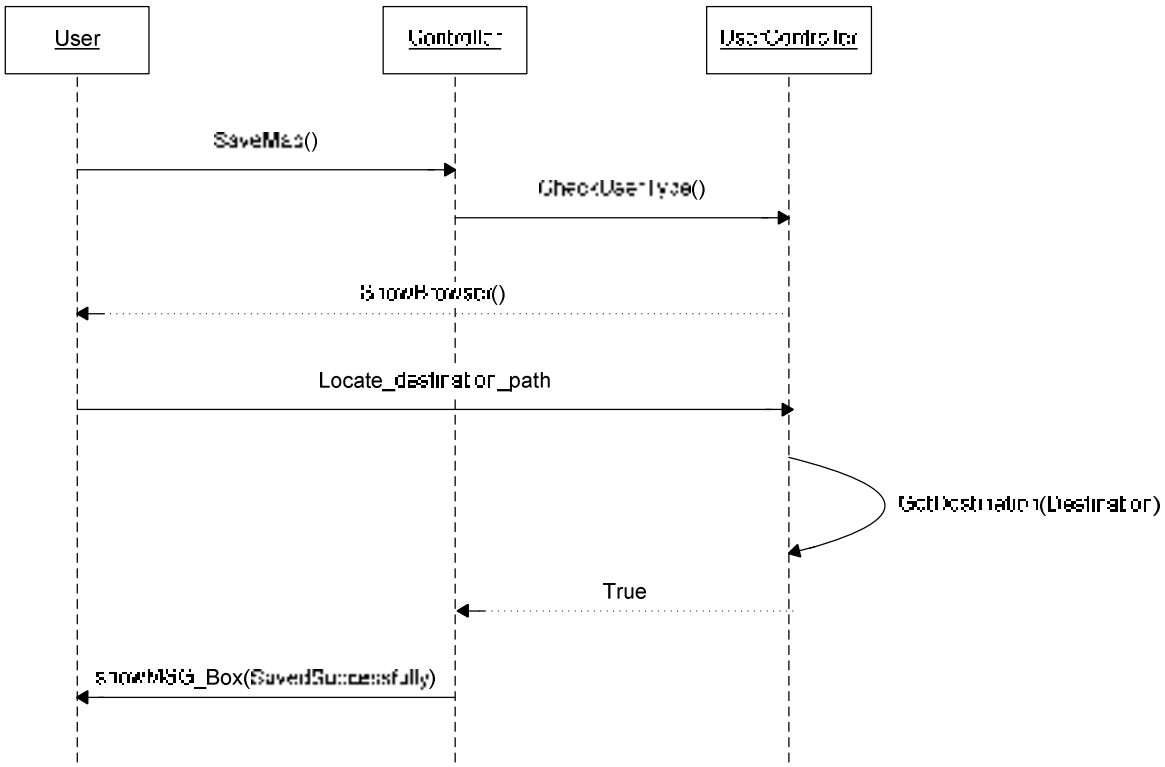

Figure 10: *Sequence diagrams: Save map*

2.1.5.2. *Collaboration diagrams: save Map.*

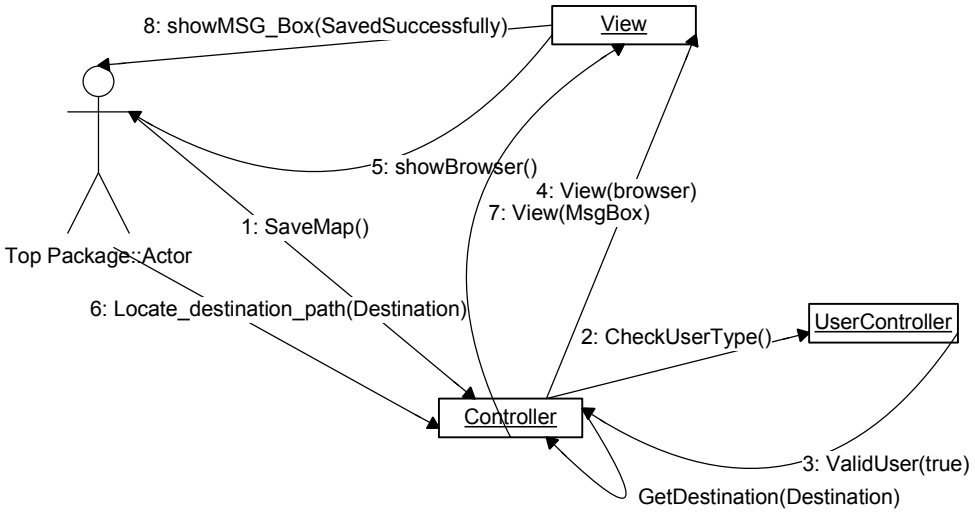

Figure 11 : *Collaboration diagrams: Save map*

2.2. Domain model

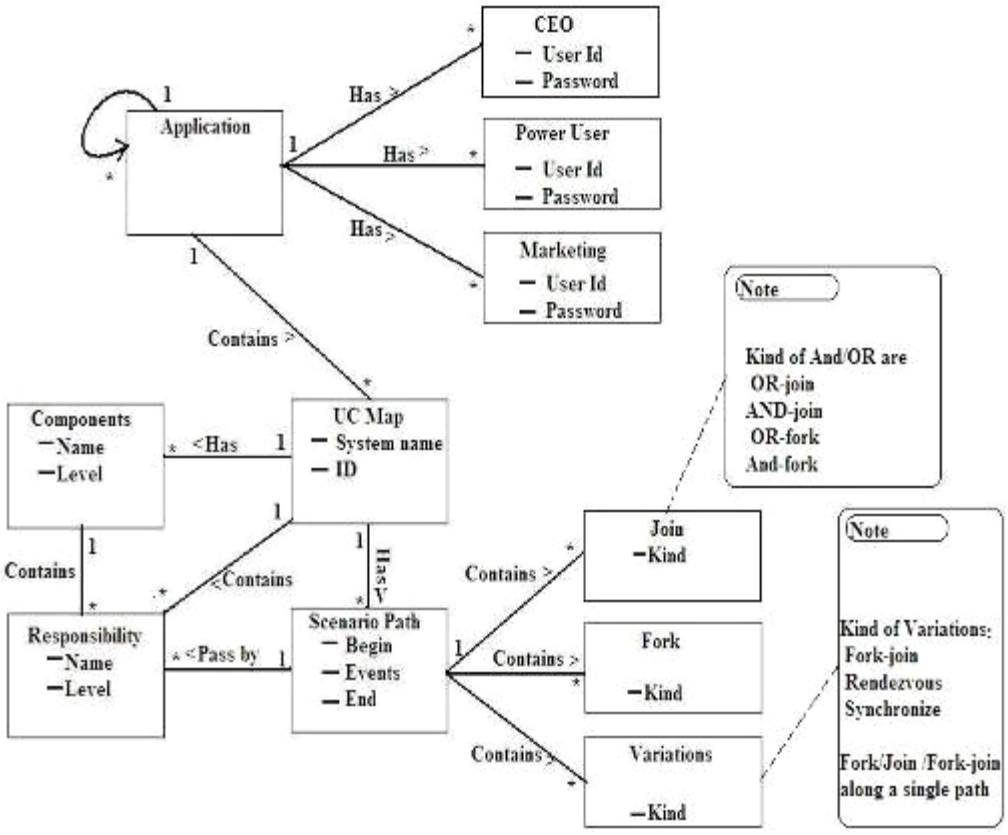

Figure 12 : Domain Model

2.3. User Interface design

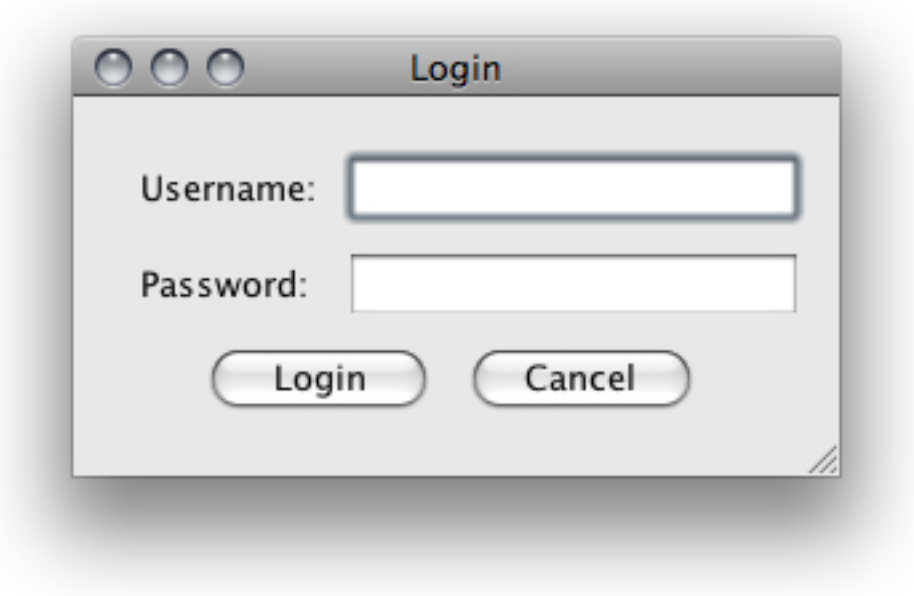

Figure 13: The login panel

Description: This is the first displayed panel when the user opens the application. User needs to provide his login ID and password, clicks on the **Login** button to issue his login session. If the **Cancel** button is clicked, the application will terminates.

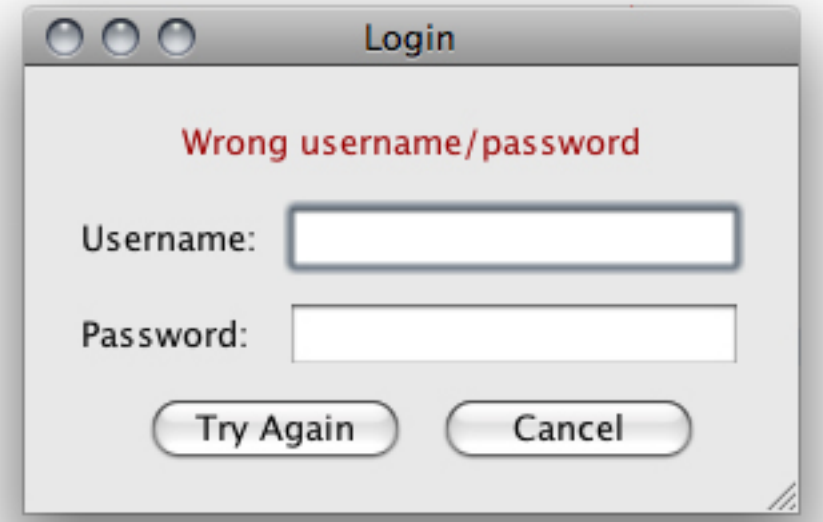

Figure 14: The login panel indicated with error message

Description: This panel will be prompted when user entered a wrong combination of login ID and password. The login process will be restarted again.

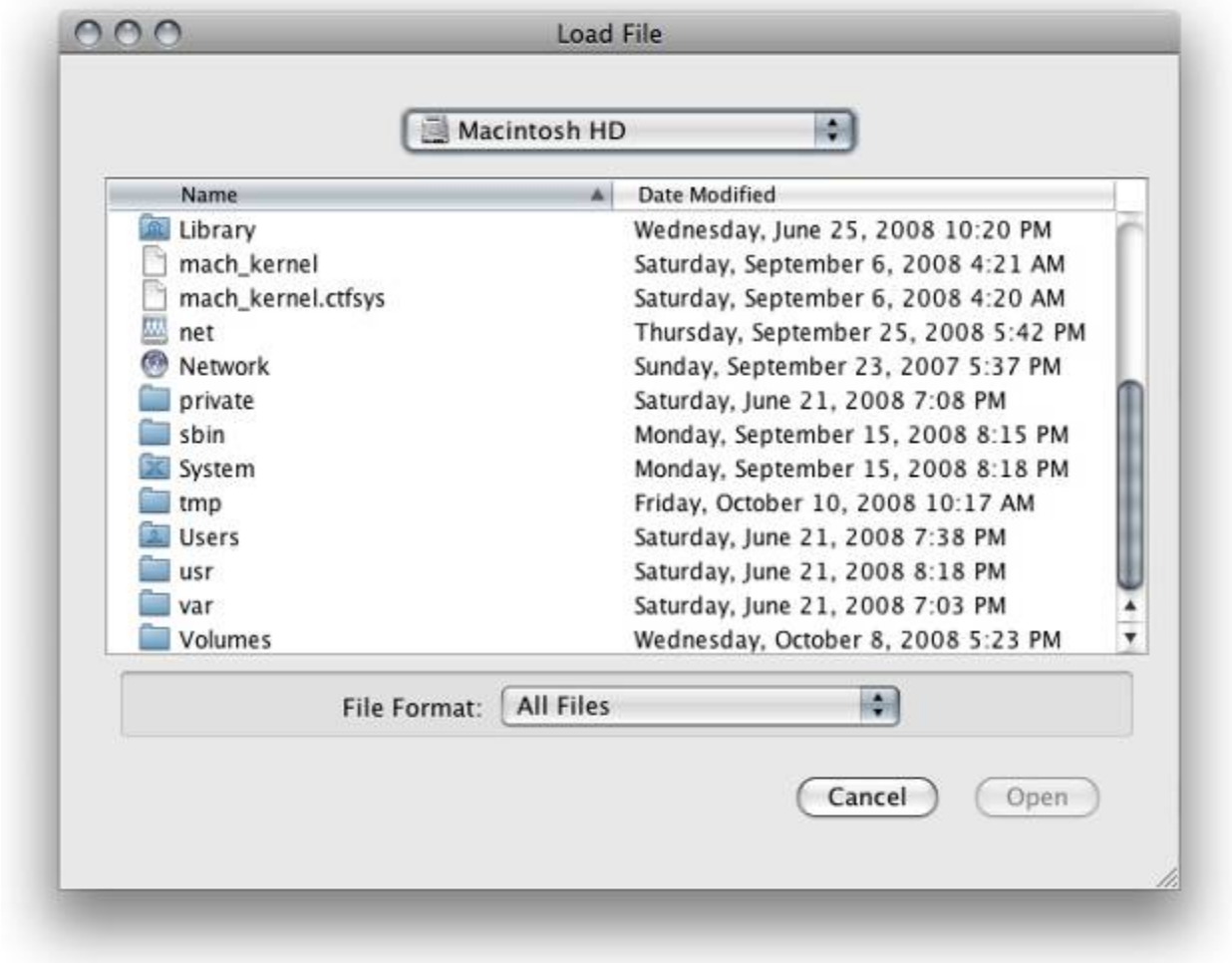

Figure 15: The load map GUI

Description: This panel will be prompted when user selected the **Load** menu. User loads a map by clicking on the map's name and clicks the **Open** button.

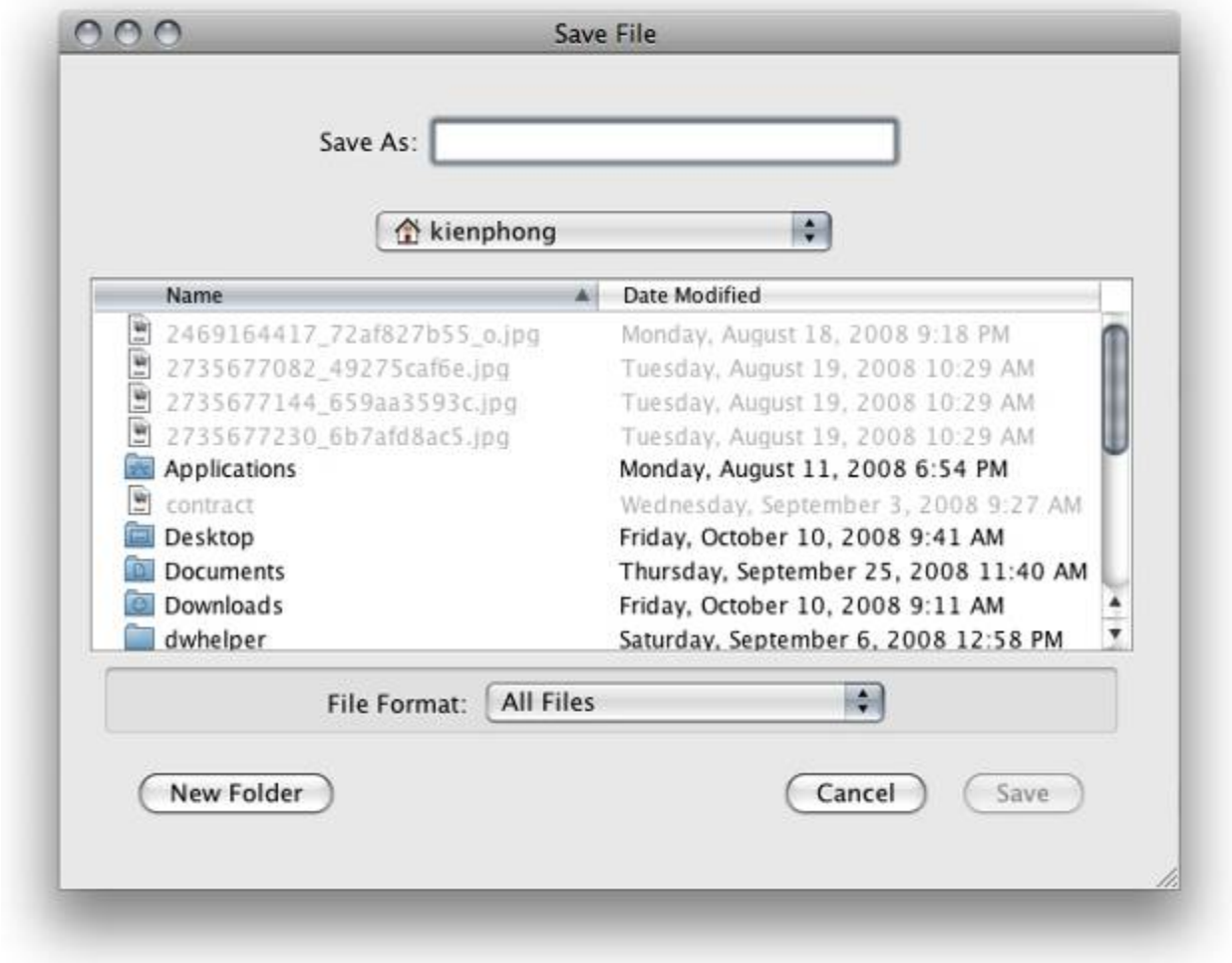

Figure 16: The save map GUI

Description: This panel will be prompted when user select the **Save** menu. User saves a map be giving a file name, the **Save** button will be enabled and user continues the saving process by clicking on the **Save** button.

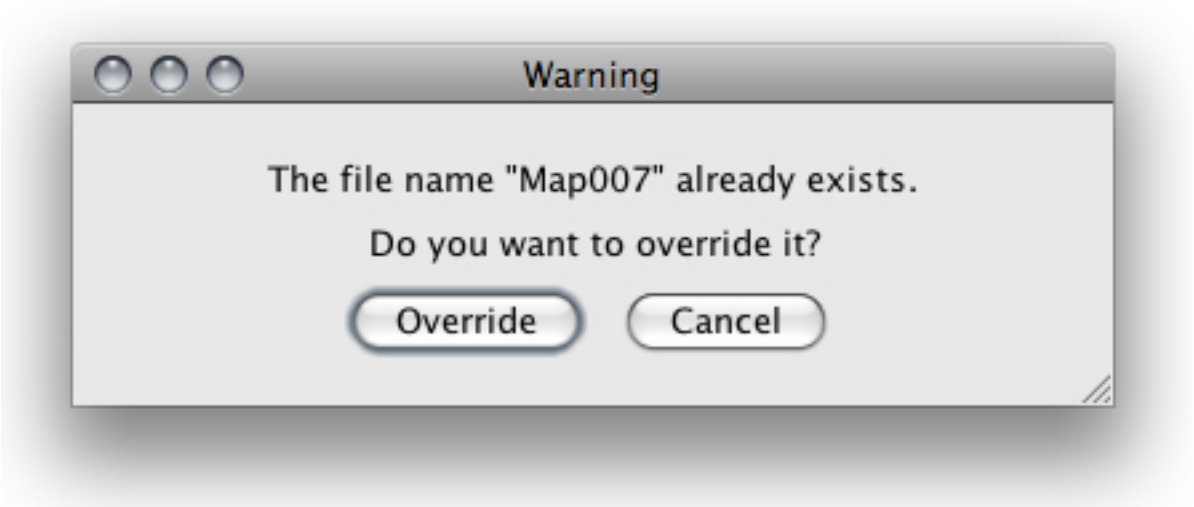

Figure 17: The Alert Panel indicated with error while saving map

Description: When a file with the same name already exists, user will be prompted with this Alert. User can choose to replace the old file by clicking on the **Override** button or cancel the saving process by clicking on the **Cancel** button.

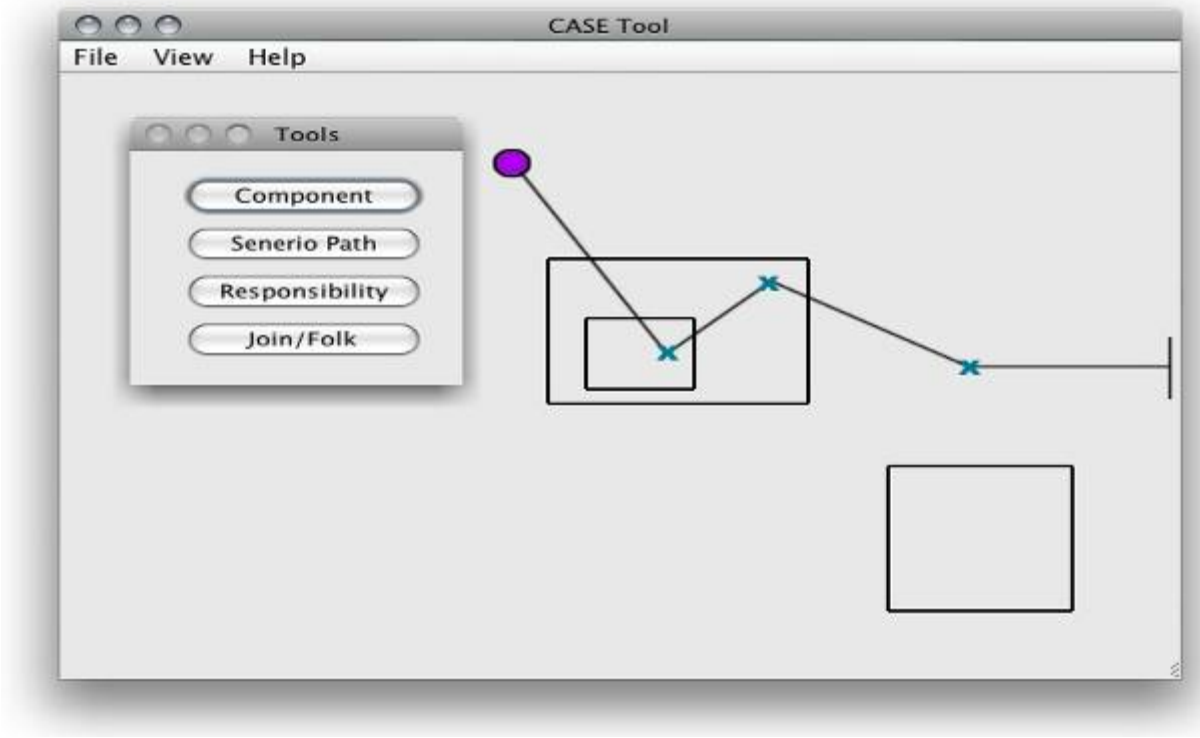

Figure 18: The main GUI while processing a map

Description: This is the main GUI of the application. The main GUI provides the user with four buttons for create different items for UCM. Clicking on the button will result in adding a item of that type in the current map. The user can later decide to resize/move/rename/delete the items by clicking on every individual one. The menu on the top area of the GUI provides the user accessibilities to the Save/Create/Load/Help sections.

3. Design consideration: Class Diagram

Our class diagram shows the advantages of our design:

Composite pattern:

"Composite pattern" is used for the creation of Components. All Components implements the same interface, so you can call the same operation over both composites and individual objects.

It makes the design flexible because you can add new Components to the hierarchy by sub classing the UCMComponent class.

Observer pattern:

"Observer pattern" is used between the MapModel and the MainForm. When the MapModel changes the MainForm get notify and updates its state. Make our design flexible because you can make add other Observers at runtime; and it makes the classes *loosely coupled* because they know very little about each other and they are able to talk.

Strategy pattern:

"Strategy pattern" is used between the MapModel and the Controller. This makes the design flexible because if you want to add a new userController you just extend the IController interface. It also allows the application to *interchange the Controller dynamically*.

Reason for our design

We will use the MVC (Model, View and Controller) paradigm for the following reason:We have a GUI (*the View*) that draws the Use Case Maps (*the Model*) through a set of operations that depends on the privileges of the user (*the controllers*).

- It has been proved by "good practices" that the MVC paradigm is flexible, easy to extend, easy to maintain and easy to test. Also it is easy to perform the refactoring of one of its components.
- Each component (the view, the model and the controller) has high cohesion, because each of them has a defined task to accomplish.

- The MVC pattern is a set of patterns working together; this ensures we are following good practices.
- We will use the “Factory pattern” and “The Observer Pattern” as part of the MVC paradigm. Also on the detailed diagram you will see we use the “Composite Pattern” to compose the UCMaps.

The MVC Paradigm

Following the MVC Paradigm, out design will divided in three main packages: the Model, the View and the Controller.

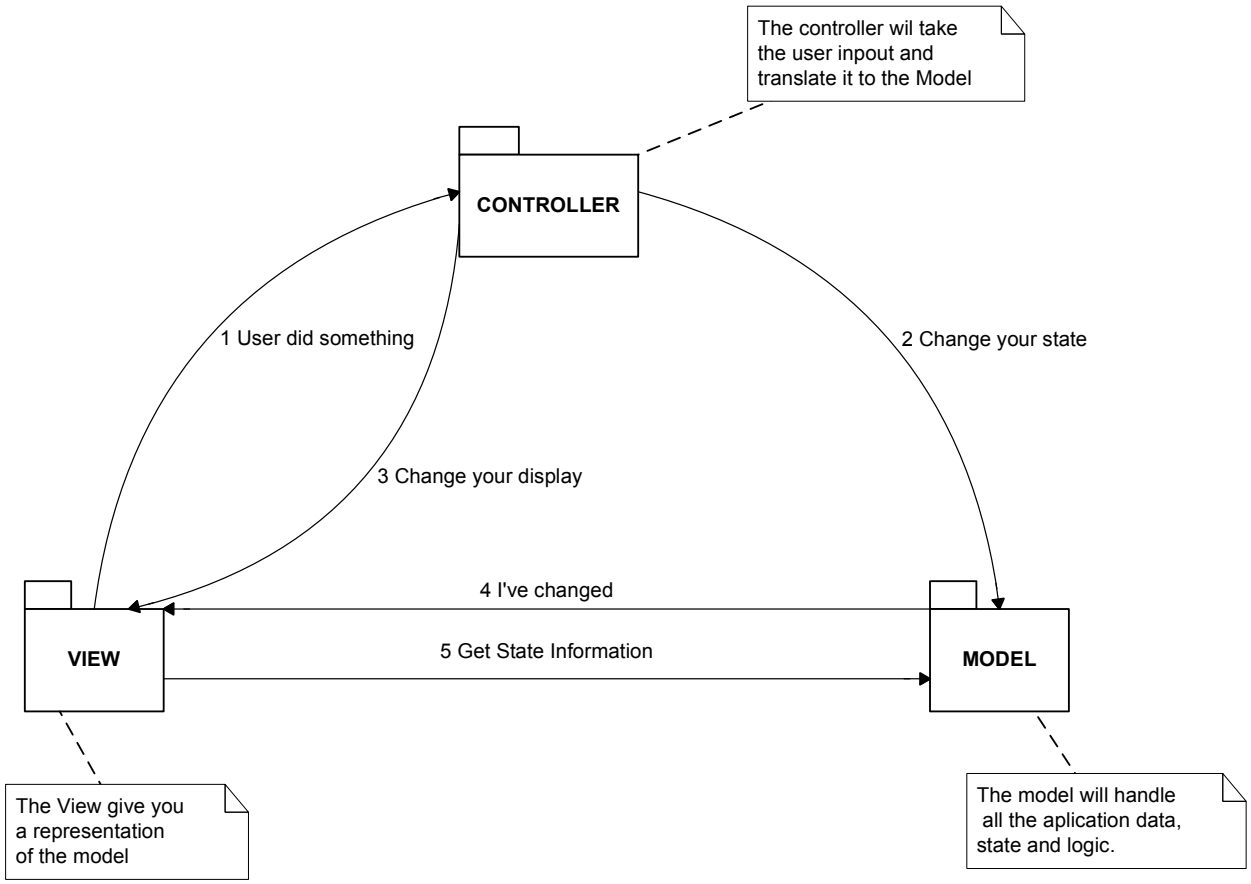

Figure 19: ***The MVC Paradigm***

MVC for the Application

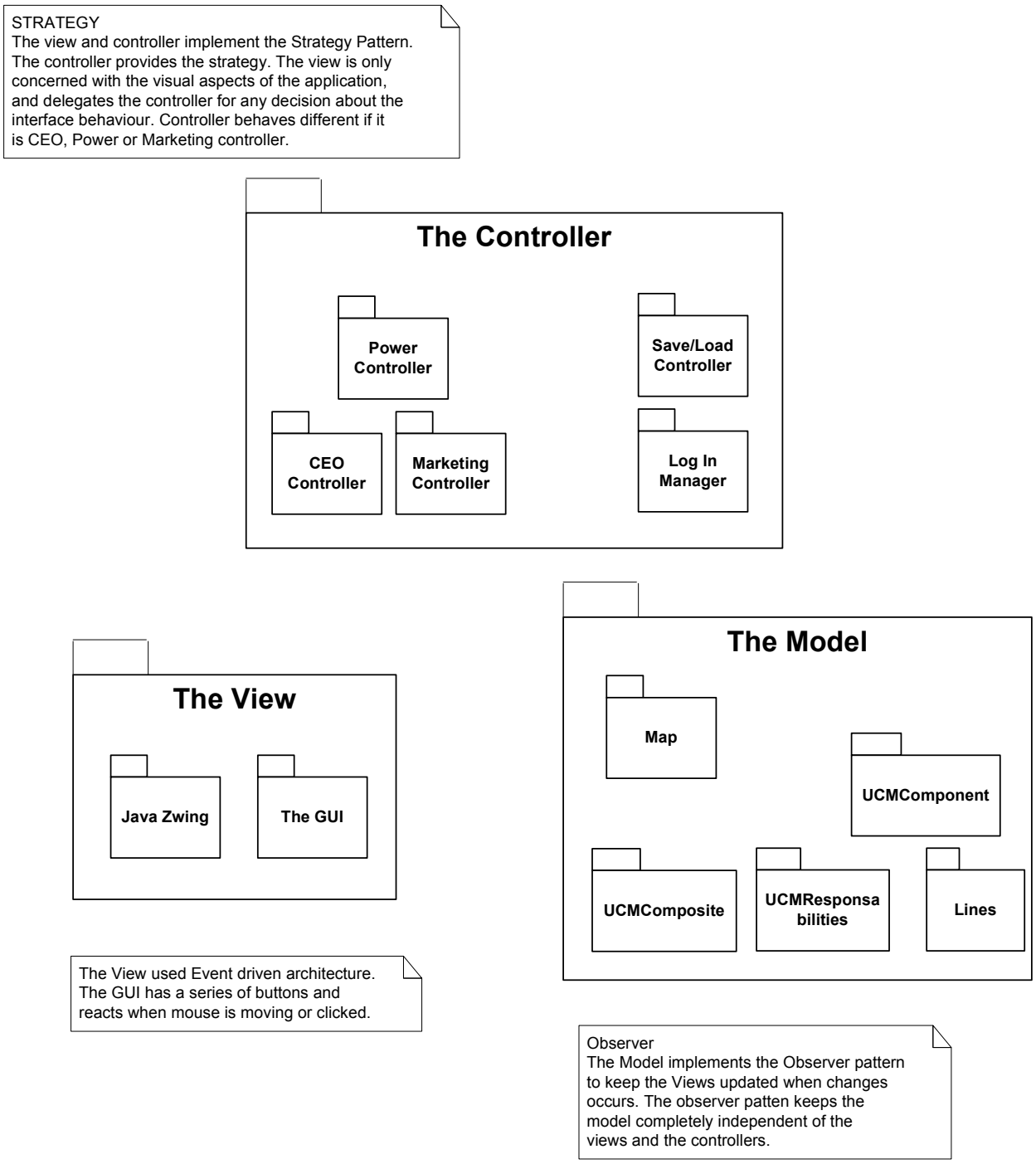

Figure 20: MVC diagram with Class view

Advantages:

- This design is easy to extend. In case you want to add a new kind of User Controller just create a new class and put it on the Controller packages. Detailed explanation on the next section.
- Again, it is easy to extend. If a new View wants to be updated when the Model state has changed the Observer pattern will solved it easily. Detailed explanation on the next section.
- Reuse is good. You can easily reuse any part of the design because is loose coupled.
- High Cohesion, each package is responsible for a concrete task.

3.1. Class Diagram 1

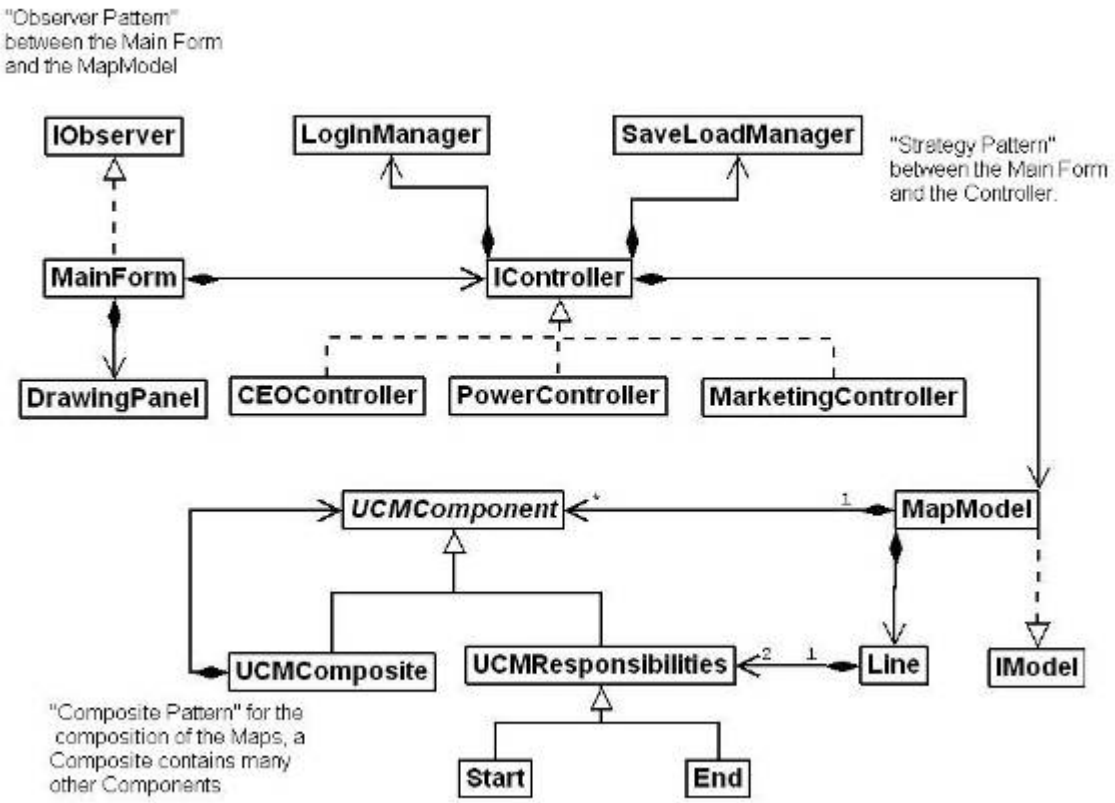

Figure 21: Simple Class diagram with function

3.2. Class diagram 2

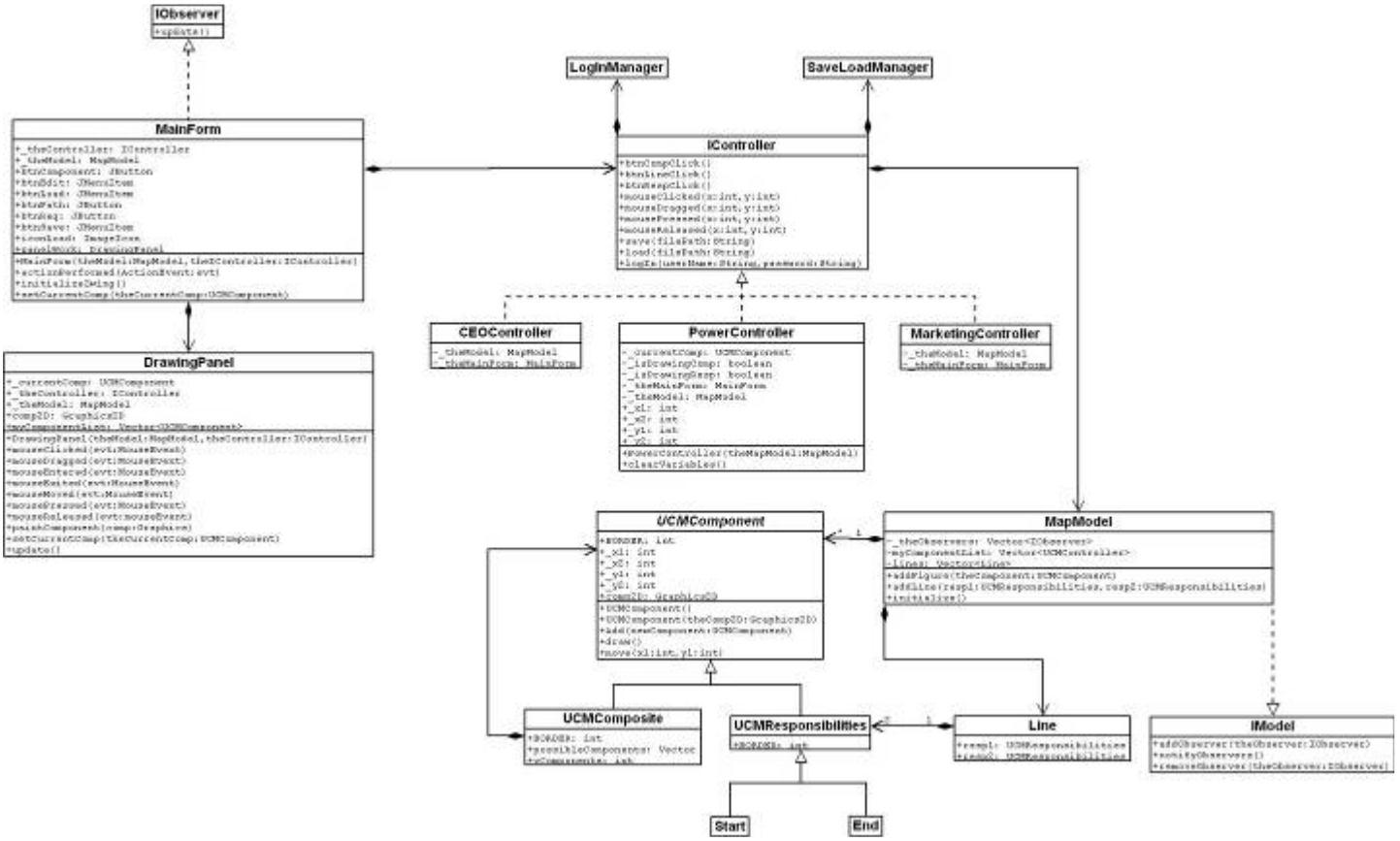

Figure 22: details Class diagram with function

4. Brief Class descriptions:

4.1 MainForm:

Contains the GUI used to interact with user and other functionality to and request data to update its status.

4.2 DrawingPanel:

Contains more detailed functionality of the GUI. It receives data from MainForm and returns transformed data to MainForm.

4.3 CEOController / PowerControls / MarketingControler:

Set of functionalities that each type of user has. Different functionality is enabled depending on which kind of user is currently logged in.

4.4 LoginManager:

Manages all logging functionalities and visual logging data.

4.5 SaveLoadManager:

Performs save and load functionalities.

4.6 MapModel:

Entity controlling what is in the drawing area of the GUI and where.

4.7 UCMComponent:

Parent class type component holding common functionality and variables used by all kinds of components.

4.8 UCMComposite:

Child class of UCMComponent. Defines and creates a box component to be displayed on the GUI.

4.9 UCMResponsibilities:

Child class of UCM Component. Defines and creates responsibility components to be displayed on the GUI.

4.10 Line:

Creates and defines line components to be displayed on the GUI.

5. *Classes details (pseudo code)*

5.1 IModel Class
(Interface)

Package

Model

Description

Using "Observer pattern". This interface must be extended by any Model that follows the Observer Pattern.

Public functions

notifyObservers	abstract
Description	Indicates each observer that the Model state has changed.
Precondition	The model class changes its state.
Post Condition	All observers update their state.
Pseudo code	public void notifyObservers();

addObserver	abstract
Description	Add an Observer to the present Model.
Precondition	Model has been initialize
Post Condition	Observer is added to the Observer list in the Model
Pseudo code	public void addObserver(IObserver theObserver);

removeObserver	abstract
Description	Remove the Observer from the present Model.
Precondition	Observer already was added to the Observer list
Post Condition	Observer is removed from the Observer list
Pseudo code	public void removeObserver(IObserver theObserver);

5.2 *MapModel Class*

(implements IModel)

Package Model

Description

This class contains all the data and logic of the model. It is going to keep the list of Components that are drawn on the Map.

Private data members

Name	Type	Description
myComponentList	Vector<UCMComponent>	Keep a list of the components to draw.
_theObservers	Vector<IObserver>	Keep a list of the observers

Public functions

Initialize	**(implemented)**
Description	Initialize private variables.
Precondition	none
Post Condition	none

Pseudo code	public void initialize()
	{
	myComponentList = new Vector<UCMComponent>();
	_theObservers = new Vector<IObserver>();
	}

addObserver	**(implemented)**
Description	Add an observer to the list of Observers.
Precondition	See IModel Interface above.
Post Condition	See IModel Interface above.

Pseudo code	public void addObserver(IObserver theObserver)
	{
	_theObservers.add(theObserver);
	}

removeObserver	(implemented)
Description	Remove an observer from the list of Observers
Precondition	See IModel Interface above.
Post Condition	See IModel Interface above.

Pseudo code	public void removeObserver(IObserver theObserver)
	{
	if(_theObservers.contains(theObserver))
	_theObservers.remove(theObserver);
	}

notifyObservers	(implemented)
Description	Notify all Observers that the Model state has changed
Precondition	See IModel Interface above.
Post Condition	See IModel Interface above.

Pseudo code	public void notifyObservers()
	{
	for(IObserver theObserver: _theObservers)
	{
	theObserver.update();
	}
	}

getState	(implemented)
Description	Return the Vector containing all figures in the Map.
Precondition	Map has been initialized.
Post Condition	Vector was returned to Observer.

Pseudo code	public Vector<UCMComponent> getState()
	{
	return myComponentList;
	}

addFigure	(implemented)
Description	Add a figure to the list of Components.
Precondition	Map has been initialized.
Post Condition	Figure is correctly added and all Observers are notified of the change.

Pseudo code	public void addFigure(UCMComponent theComponent)
	{
	myComponentList.add(theComponent);
	this.notifyObservers(); // <- Notify all Observers.

	}

getSelectedUCMComponent	**(implemented)**
Description	Get the selected figure. If no figure has been selected it returns null.
Precondition	None.
Post Condition	UCMComponent is returned.

Pseudo code	public UCMComponent getSelectedUCMComponent(int coordx, int coordy)
	{
	UCMComponent selectedComponent = null;
	int _xm = coordx;
	int _ym = coordy;
	
	// Scan all elements recursevly on the vector of figures to get the object
	//selected
	}

5.3 IObserver Class
(Interface)

Package View

Description

Using "Observer pattern". This interface must be extended by any Observer that follows the Observer Pattern.

Public functions

Update	**abstract**
Description	The update method is used by Observer to update the data they need from the Observer
Precondition	The Class that extends this Interface must be added to the list of Observers of the Model class
Post Condition	The class will update its List of Figures
Pseudo code	public void update();

5.4 UCMComponent Class
(Interface)

Description

An interface for all the elements to be drawn on the map.

Constants

None

Private data members

Name	Type	Description
comp2D	Graphics2D	Java drawing utility
BORDER	int	Width of the line
_x1	int	Upper left horizontal coordinator of the element
_x2	int	Lower right horizontal coordinator of the element
_y1	int	Upper left vertical coordinator of the element
_y2	int	Lower right vertical coordinator of the element

Public functions

UCMComponent	
Description	default constructor
Input	none
Output	UCMComponent object
Pseudo code	{ }

UCMComponent	
Description	constructor that initialize the Graphics2D component
Input	a Graphics2D object
Output	UCMComponent object

Pseudo code	{
	comp2D = newGraphics2D;
	}

getComponentSelected	
Description	get the selected component
Input	x and y coordinators of the mouse
Output	the selected UCMComponent
Pseudo code	{ if((xm >= _x1) && (xm <= _x2) && (ym >= _y1) && (ym <= _y2))

	return this;
	else
	return null;
	}

Move	
Description	move the component to the new coordinates.
Input	new x and y coordinators
Output	none

Pseudo code	{
	int tempX1 = this._x1;
	int tempY1 = this._y1;
	this._x1 = x1;
	this._y1 = y1;
	this._x2 = this._x1 + this._x2 - tempX1;
	this._y2 = this._y1 + this._y2 - tempY1;
	}

Draw	**(abstract)**
Description	draw the component
Input	None
Output	none
Pseudo code	// depends on implemented class

drawAsSelected	**(abstract)**
Description	draw the component as selected (highlight this element)
Input	none
Output	void: highlight this element
Pseudo code	// depends on implemented class

Add	**(abstract)**
Description	add a component inside this component
Input	an UCMComponent
Output	none
Pseudo code	// depends on implemented class

getType	**(abstract)**
Description	get the type of Component
Input	none
Output	a string that represents the type of this element
Pseudo code	// depends on implemented class

getWidth	
Description	get the width of Component
Input	none
Output	width of the component (int)

Pseudo code	{
	return Math.abs(_x2 - _x1);
	}

getHeight	
Description	get the height of Component
Input	none
Output	height of the component (int)

Pseudo code	{
	return Math.abs(_y2 - _y1);
	}

Continue on next page…

5.4 UCMResponsabilities Class
(extends UCMComponent)

Description

A class for the “Reasonability” element of UCM

Constants

None

Private data members

Name	Type	Description
comp2D	Graphics2D	Java drawing utility (inherited)
BORDER	int	Width of the element’s border
_x1	int	Upper left horizontal coordinator of the element (inherited)
_x2	int	Lower right horizontal coordinator of the element (inherited)
_y1	int	Upper left vertical coordinator of the element (inherited)
_y2	int	Lower right vertical coordinator of the element (inherited)

Public functions

Draw	**(implemented)**
Description	draw the responsibility
Input	none
Output	none

Pseudo code	{
	comp2D.setColor(Color.PINK);
	this.comp2D.draw(
	new Rectangle2D.Float(
	this._x1,
	this._y1 ,
	this._x2 - this._x1,
	this._y2 - this._y1
	)
	);
	}

	}

Add	**(implemented)**
Description	add a component to this component
Input	an UCMComponent component
Output	none

Pseudo code	{
	// responsibility cannot contain any element
	}

getType	**(implemented)**
Description	get type of this element
Input	none
Output	a string that represents the responsibility component

Pseudo code	{
	return "UCMResponsabilities";
	}

drawAsSelected	**(implemented)**
Description	draw the component as selected (highlight this element)
Input	none
Output	void: highlight this element

Pseudo code	{
	comp2D.setColor(Color.RED);
	this.comp2D.draw(
	new Rectangle2D.Float(
	this._x1,
	this._y1 ,
	this._x2 - this._x1,
	this._y2 - this._y1
	)
	);
	}

6. Time Log Tables

Group member: Juan

Date	Task	Duration
October 28, 2008	Meeting I for Design Documentation	2 hours
October 30, 2008	Meeting II for Design Documentation	2 hours
October30, 2008	Application of MVC paradigm	2 hours
November 1, 2008	Architectural Diagram and Class Model Diagram discussion I	1 hours
November 2, 2008	Application of Observer and Strategy patterns	2 hours
November 4, 2008	Class Model Diagram discussion II	2 hours
November 6, 2008	Classes Detailed Design discussion	2 hours
November 8, 2008	Design Documentation finalization	2 hours
	Total	**15 hours**

Group member: Sahadat

Date	Task	Duration
October 28, 2008	Meeting I for Design Documentation	2 hours
October 29, 2008	Prepare table of contents and chapter 1	1 hours
October 30, 2008	Meeting II for Design Documentation	2 hours
November 1, 2008	Architectural Diagram and Class Model Diagram discussion I	2 hours
November 3, 2008	Design Documentation prepare	2 hours
November 4, 2008	Class Model Diagram discussion II	2 hours
November 6, 2008	Classes Detailed Design discussion	2 hours
Novemver7, 2008	Case sequence and collaboration diagram	1.5 hours
November 8, 2008	Design Documentation finalization	2 hours
November 9, 2008	Design Documentation finalization	2 hours
	Total	**18.5 hours**

Group member: Jose

Date	Task	Duration
October 28, 2008	Meeting I for Design Documentation	0.5 hours
October 30, 2008	Meeting II for Design Documentation	1 hours
November 1, 2008	Architectural Diagram and Class Model Diagram discussion I	1 hours
November 4, 2008	Class Model Diagram discussion II	2 hours
November 6, 2008	Classes Detailed Design discussion	2 hours
November 8, 2008	Design Documentation finalization	3.5 hours
	Total	**10 hours**

Group member: Ryo

Date	Task	Duration
October 28, 2008	Meeting I for Design Documentation	0.5 hours
October 30, 2008	Meeting II for Design Documentation	1 hours
November 1, 2008	Architectural Diagram and Class Model Diagram discussion I	3 hours
November 4, 2008	Class Model Diagram discussion II	3 hours
November 6, 2008	Classes Detailed Design discussion	2 hours
November 8, 2008	Design Documentation finalization	4.5 hours
	Total	**14 hours**

Group member: Kien

Date	Task	Duration
October 28, 2008	Meeting I for Design Documentation	0.5 hours
October 30, 2008	Meeting II for Design Documentation	1 hour
November 1, 2008	Architectural Diagram and Class Model Diagram discussion I	3 hours
November 4, 2008	Class Model Diagram discussion II	3 hours
November 6, 2008	Classes Detailed Design discussion	2 hours
November 8, 2008	Design Documentation finalization	3 hours
	Total	**12.5 hours**

----------End----------