Object-Oriented Software Design

1 Overview

In this assignment, each group will complete an **object oriented design** of an application that manages a modern library called Tawe-Lib. This electronic system shall adhere to the detailed list of the functionality and requirement specifications provided in this document (Functional Specification).

2 Functional Specification

The main purpose of Tawe-Lib is to manage a modern library, by for example, keeping track of resources such as books, allowing users to borrow and return resources, keeping track of fines, etc. Such a system would normally (these days) be run as a web application, but to make this assignment manageable, Tawe-Lib will run as a stand alone PC application. Only one user can be logged in at a time. All data will be stored in files on the local computer – no networking or web technology will be involved.
It is up to you to design the data structures, algo-rithms and GUI involved in realising this system.

2.1 Users

There are two categories of users within the system: normal users and Librarians.

Each user has the following information associated with their account:

1. A username that uniquely identifies their account.
2. A first name and a last name.
3. A UK mobile telephone number.
4. A full UK address complete with postcode.
5. A profile image.

Additionally, Librarians also have:

- Employment date.
- Staff Number.

You should design and implement classes to manage this information. Also, you should create an graphical user interface that allows Librarians to create new accounts.

2.2 Resources

Resources available in the library fall into various categories: Books, DVDs, and Laptop Computers.

Different types of resources have different information associated with them. However all resources have the following information associated with them:

- Unique ID.
- Title.
- Year.
- A thumbnail image.

Each Book also has:

- Author.
- Publisher.
- Genre.
- ISBN.
- Language.

Each DVD also has:

- Director.
- Runtime.
- Language.
- List of Subtitle Languages.

Each Laptop Computer also has:

- Manufacturer.
- Model.
- Installed operating system.

You will also need to build a user interface for allowing Librarians to manage resources and copies (see Section 2.3). Librarians shall be able to create new resources and edit existing ones. When a Librarian creates a new resource (or edits an existing one) they must provide all the data (with the exception of optional data).

2.3 Multiple Copies

There are multiple (identical) copies of each resource available within the library. For example, there can be 20 copies of the resource (i.e., Book) *Java for Everyone by Cay Horstmann*.

Each copy of an item has a loan duration associated with it. This can be 1 day, 1 week, 2 weeks, or 4 weeks. When copies are borrowed no due date is set (see Section 2.6).

2.4 Browsing and Searching Resources

Users shall be able to browse all the resources within the library. That is, the user shall be able to look through the resources and have at least the unique ID, title and year displayed. The user shall be able to choose a resource and the system will provide more detailed information. For each resource, the user shall be able to see how many copies of the resource are in the system. For each copy, they shall be able to see if it is available or currently being borrowed (but not by whom).

Users shall also be able to search for resources based on entering partial information in a typical search mechanism. The user shall also be able to filter resources based on the type of resource to only show Books, DVDs, etc.

2.5 Borrowing Copies

Individual copies, not currently borrowed, can be borrowed by users by visiting the Issue Desk and interacting with a Librarian. Only Librarians have the authority to loan copies to users. The system will keep track of which user has borrowed the copy and the date it was borrowed. If the user borrowing the copy has outstanding fines on their account, or overdue copies, then the system will prevent any further items from being borrowed.

2.6 Due Dates and Requesting Items

When a particular copy of a resource is borrowed **no** due date is set. Instead the user may borrow the copy until someone requests it.

If all copies of a resource are currently being borrowed then a user may request that the first copy of the resource to be returned is reserved for them. They will be added to the resource's request queue. As copies become available they will be reserved for people in the queue (in queue order).

When someone is added to a resource's request queue the system will automatically set the due date of the oldest borrowed copy where no due date is currently set. The due date will be set to the latest of the following:

- tomorrow (the date after this request is being made); or

- the earliest date such that the borrower has had the copy for the loan duration. (This ensures that a borrower has access to the copy for at least the loan duration).

2.7 Returning Copies

Borrowed copies can be returned by users by visiting the Issue Desk and interacting with a Librarian. Only Librarians have the authority to process returns. Upon returning a copy, the system will:

1. Calculate if the returned copy is overdue. If it is overdue then a fine will be applied to the balance of the account as follows:
    - For Books and DVDs, the fine is £2.00 per day late, up to a maximum of £25.
    - For Laptop Computers, the fine is £10.00 per day late, up to a maximum of £100.
2. If the resource's request queue is empty then the copy will be marked as available.
3. If the resource's request queue is not empty then the copy will be reserved for the next person in the queue.

2.8 Paying Fines

A user may pay off current fines (fully or partial) by visiting the Issue Desk and interacting with a Librarian. Only Librarians have the authority to pay off fines. A user will be able make a payment between £0.01 and the full outstanding balance of the account (i.e., the outstanding total amount of fines). The balance of the account will be adjusted accordingly. If all the fines are paid off then the user can now resume borrowing items.

2.9 User Dashboard

Each user shall have their own dashboard where they can view the following:

- Borrowed Items. A list of items they are currently borrowing. This shall show the due date of each item (if set).
- Requested Items. A list of items that the user has requested, but are not currently available.
- Reserved Items. A list of items that they previously requested that are now available to pick up. They have been reserved for this user.
- The current balance of the account (i.e., the current amount of fines).
- Transaction History. A chronologically ordered list showing each transaction on their account. A transaction is either a fine (showing the date and time of the fine, the amount, the item that caused the fine, and the number of days the item was overdue) or a payment (showing the date and time of payment and the amount).

2.10 Viewing a Copies Borrowing History

Librarians shall be able to view a particular copy's borrowing/return history. This will show a chronological log showing each time the copy was borrowed or returned, by whom and the date and time of the event.

2.11 Viewing a Overdue Copies

Librarians shall be able to view a list of all overdue copies showing the borrower, and the number of days overdue.

2.12 Profile Images

Each user must have a profile image. Two kinds of profile images are supported:

- Avatars
- Custom Drawings

The system will have a set (say 5 or 6) built in Avatar images. The user can select one of these as their profile picture.

Alternatively, the user will be allowed to create a custom drawing. The system will provide the user with a built-in basic drawing environment that is similar to a paint program with at least the following functionality:

- Ability to draw straight lines

- Ability to draw a particle trace (i.e., clicking and dragging will draw "filled" circles at the location of the pointer).

Once you have the basic functionality working to a high standard, you are welcome to expand upon it to earn higher marks.

2.13 Statistics

The dashboard shall show users statistics about how many items they borrow per week, per month and per year. These will be displayed both textually and graphically.

2.14 Other Properties of Tawe-Lib

The Tawe-Lib must have the following property:

- Data shall be persisted across runs of the system. For example if a book is borrowed or other changes made, then those changes are also present after the application has been closed and re-opened. You may save the data to local files when exiting the application. When you restart the application, the data will be loaded from the files. You may also use a database if you so choose.

Tawe-Lib should operate on a single machine. A user logs in to the program and performs actions which are saved locally to disk on that machine. The user logs out. A new user can then log in to view what has changed. When you start the implementation part of the project, please build the system up one step at a time: get one aspect working before starting another.

2.15 Extra Tawe-Lib Features

This section is not technically part of A1, but it will be part of A2. It will be helpful for you to plan for extensibility and hence know about this section.

To achieve top marks in A2, you will need to be creative. At a minimum, all the functionality of the Functional Specification should be completed to a high standard. All features should adhere strictly to the specification. You need to get all this working very well in order to get a low first class mark (in A2). In order to get higher marks, you would be required extend the implementation in a novel way. All extensions that do not violate the specification will be considered. Substantial extensions to the software, extra reading and learning, will be required to achieve a high first class mark (in A2).

For this assignment (A1), **do not include these extra features**. Simply design without the extra features, but be aware that they will be required for the implementation (in A2).

3 A1 Tasks

Your team is asked to provide a complete **design** for the entire system. This design must include at least one complete class hierarchy. Each team member is required to contribute to at least one class in the design. The designer of the class does not necessarily need to implement it in A2.

3.1 Design Document

Your Design Document proposes an object-oriented design for the *entire* application. In other words, your team incorporates the full functional specification into the design. The Design Document should be no more than 30 pages including text and diagrams.

The Design Document consists of the sections as detailed below.

3.1.1 Candidate Classes and Responsibilities

Provide a list of candidate classes and their responsibilities. This list is like the CRC cards in lectures but with a bit more detail. For each candidate class the team has identified, the following information (i.e., card) must be provided:

1. **Class Name** (in bold)
2. Rough idea behind the class.
3. Author.
4. SuperClass.
5. SubClasses.

6. Responsibilities: a list of services this class provides.

7. Collaborations: a list of classes with which the class communicates.

There should be one of these cards for every class that appears in your class diagrams (see Section 3.1.2).

3.1.2 Class Diagrams

After specifying the information in Section 3.1.1, draw the classes using UML Class Diagram notation. Your class diagrams must use appropriate arrows and UML syntax to represent relationship such as collaborations (i.e., associations, compositions, and aggregations) and generalisations/specialisations (i.e., inheritance).

You should carefully think about navigability and multiplicities when drawing the collaboration relationships. If you would like, you can use UML tools such as Papyrus[1] (or others).

When providing details about the attributes and operations, you must think carefully about type information and your design. The classes and methods should fit together and function to achieve the intended behaviour. Do not just add operations and collaborations without thinking about how the operations can actually carry out their jobs. You need to take time to think about what collaborations your classes need to have in order to provide the services they offer.

Each class in your design (See Section 3.1.1) must appear in at least one UML Class Diagram. Each class hierarchy identified during your design process must be depicted in a hierarchy in a UML Class diagram.

3.1.3 Hierarchy Descriptions

Each generalisation/specialisation (i.e., inheritance) relationship (see Section 3.1.2) must be accompanied by a short textual description that describes the "is-kind-of" relationship used. Make sure that your team provides justifications that backs up the choices. Make sure that abstract classes are distinguishable from concrete classes in your diagrams. Your team should be able to identify two (or more) class hierarchies.

3.1.4 Collaboration Descriptions

Each composition and aggregation relationship (see Section **??**) must be accompanied by a short textual description that describes the "is-made-up-of" relationship used. Make sure that your team provides justifications that backs up its choices.

3.1.5 Operation Descriptions

For each of your 5 most complex operations (within your classes) you must provide a short paragraph explaining what the operation should do overall, how the operation can be implemented, and how it has access to the data it needs (e.g., through collaborations).

All group members must review the final documents prior to submission. Each individual group member is responsible for understanding the entire contents of all documents. Submission indicates that all group members have read and approved the document unless a conversation that has been documented by your Academic Mentor indicates otherwise.

One member of each group, the **Secretary**, should lead the submission of the weekly Contribution Breakdowns and the final submission (i.e., the Design Document and Contributions Report) on behalf of the group. Points will be deducted for those submissions that do not follow the file naming conventions and required file formats.

[1] `http://www.eclipse.org/papyrus/`

Design Documentation

Group 7

Implementation Managers:

Test Managers:

Planning and Quality Manager:

Design Manager:

Customer Interface Manager:

January 20, 2012

Contents

Chapter 1

Candidate classes and responsibilities

1.1 AddressBook, Contact

AddressBook
Author: Yan Sun
SuperClass: -
SubClasses:-
Responsibilities: Initialise the address book. Change view between calendar and address book. View all current contacts. Manipulate contact details.
Collaborations: Event DigitalOrganiser

Protocols:
`public Boolean addContact(Contact c)`
`public Boolean editContact(Contact c)`
`public Boolean removeContact(Contact c)`
`public Boolean viewContacts()`

Unit Tests: Add a contact. Edit a contact. Remove a contact.

Contact
Author: Ceris Land
SuperClass:-
SubClasses:-
Responsibilities: Create new contacts. Get contact details. Validate contact details.
Collaborations: AddressBook

Protocols:
`public String setName(String n)`
`public String getName()`
`public Boolean validateName()`
`public String setAddress(String a)`
`public String getAddress()`
`public String setEmail(String e)`
`public String getEmail()`
`public Boolean validateEmail()`
`public String setHomeTel(String tel)`
`public String getHomeTel()`
`public String setFaxNo(String fax)`
`public String getFaxNo()`
`public String setURL(String url)`
`public String getURL()`
`public Boolean validateURL()`
`public String setOther(String o)`
`public String getURL()`
`public Boolean validateAll()`

Unit Tests: Test all valid/invalid inputs. Check the contact details are updated.

1.2 Event

Event
Author: Codrin Morhan
SuperClass: -
SubClasses: Deadline Lecture Meal Anniversary HappyHour Accident SocialEvent Meeting Other
Responsibilities: Provide event template. Set event data. Get event data. Validate event data. Have a generic icon. Be an abstract class.
Collaborations: View DigitalAddressBook

Protocols:
`set<attribute>(<attributeDataType> newValue) return void`
`get<attribute>() return <attributeDataType> attributeValue`
`validateDate() return Boolean`
`validateStartingTime() return Boolean`
`validateEndingTime() return Boolean`
`validateAll() return Boolean`

Unit Tests: Date is inside the required interval. Event must have a duration. Check validation works.

1.3 Deadlines

Deadline
Author: Adewale Odunlami
SuperClass: Event
SubClasses: WorkDeadline AssignmentDeadline MarkingDeadline BillPlayment
Responsibilities: Same as *Event* Have a specific icon.
Collaborations: Same as *Event*
Protocols: Same as *Event* except:
Unit Tests: Same as *Event* except: Starting and ending time are the same.

WorkDeadline
Author: Adewale Odunlami
SuperClass: Deadline
SubClasses: -
Responsibilities: Same as *Deadline* Have a specific icon.
Collaborations: Same as *Deadline*
Protocols: Same as *Deadline*
Unit Tests: Same as *Deadline*

MarkingDeadline
Author: Adewale Odunlami
SuperClass: Deadline
SubClasses: -
Responsibilities: Same as *Deadline* Have a specific icon.
Collaborations: Same as *Deadline*
Protocols: Same as *Deadline*
Unit Tests: Same as *Deadline*

AssignmentDeadline
Author: Adewale Odunlami
SuperClass: Deadline
SubClasses: -
Responsibilities: Same as *Deadline* Have a specific icon.
Collaborations: Same as *Deadline*
Protocols: Same as *Deadline*
Unit Tests: Same as *Deadline*

BillPayment
Author: Adewale Odunlami
SuperClass: Deadline
SubClasses: -
Responsibilities: Same as *Deadline* Have a specific icon.
Collaborations: Same as *Deadline*
Protocols: Same as *Deadline*
Unit Tests: Same as *Deadline*

1.4 Lecture, HappyHour, Anniversary

Lecture
Author: Adewale Odunlami
SuperClass: Event
SubClasses: -
Responsibilities: Same as *Event* Have a specific icon.
Collaborations: Same as *Event*
Protocols: Same as *Event* except:
Unit Tests: Same as *Event* except: Duration is longer than fifty minutes. Duration is shorter than three hours.

HappyHour
Author: Adewale Odunlami
SuperClass: Event
SubClasses: -
Responsibilities: Same as *Event* Have a specific icon.
Collaborations: Same as *Event*
Protocols: Same as *Event*
Unit Tests: Same as *Event* except: Duration must be exactly one hour.

Anniversary
Author: Codrin Morhan
SuperClass: Event
SubClasses: Birthday BankHoliday
Responsibilities: Same as *Event* Have a specific icon.
Collaborations: Same as *Event*
Protocols: Same as *Event*
Unit Tests: Same as *Event* except: Duration must be exactly one day.

1.5 Birthday, BankHoliday, Accident, SocialEvent, Concert

Birthday
Author: Codrin Morhan
SuperClass: Anniversary
SubClasses: -
Responsibilities: Same as *Anniversary* Have a specific icon.
Collaborations: View
Protocols: Same as *Anniversary*
Unit Tests: Same as *Anniversary* except: Duration is less than a day.

BankHoliday
Author: Codrin Morhan
SuperClass: Anniversary
SubClasses: -
Responsibilities: Same as *Anniversary* Have a specific icon.
Collaborations: Same as *Anniversary*
Protocols: Same as *Anniversary*
Unit Tests: Same as *Anniversary*

Accident
Author: Adewale Oduntami
SuperClass: Event
SubClasses: -
Responsibilities: Same as *Event* Have a specific icon.
Collaborations: View
Protocols: Same as *Event*.
Unit Tests: Same as *Event* except: Starting and ending time are the same.

SocialEvent
Author: Codrin Morhan
SuperClass: Event
SubClasses: Concert
Responsibilities: Same as *Event* Have a specific icon.
Collaborations: Same as *Event*
Protocols: Same as *Event*
Unit Tests: Same as *Event* except: Duration is less than ten days.

Concert
Author: Codrin Morhan
SuperClass: SocialEvent
SubClasses: -
Responsibilities: Same as *SocialEvent* Have a specific icon.
Collaborations: Same as *SocialEvent*
Protocols: Same as *SocialEvent*
Unit Tests: Same as *SocialEvent* except: Duration is less than a day.

1.6 Meeting, Appointment, Meal, Other, EventList

Meeting
Author: Codrin Morhan
SuperClass: Event
SubClasses: Appointment
Responsibilities: Same as *Event* Have a specific icon.
Collaborations: Same as *Event*
Protocols: Same as *Event*
Unit Tests: Same as *Event* except: Duration is less than a day.

Appointment
Author: Codrin Morhan
SuperClass: Meeting
SubClasses: -
Responsibilities: Same as *Meeting* Have a specific icon.
Collaborations: Same as *Meeting*
Protocols: Same as *Meeting*
Unit Tests: Same as *Meeting*

Other
Author: Codrin Morhan
SuperClass: Event
SubClasses: -
Responsibilities: Same as *Event* Have a generic icon.
Collaborations: View
Protocols: Same as *Event*
Unit Tests: Same as *Event*

EventList
Author: Codrin Morhan
SuperClass:-
SubClasses:-
Responsibilities: Be a list of events. Provide functionality for manipulating a list events.
Collaborations: Event View

Protocols:
`public void addEvent(Event e)`
`public void editEvent(Event e)`
`public void removeEvent(Event e)`
`public vodi removeAllEvents()`
`public void getAllEvents()`
`public void enumerateEvents()`

Unit Tests: Add an event. Edit an event. Remove an event. Remove all events. Enumerate the events.

Meal
Author: Adewale Odunlami
SuperClass: Event
SubClasses: -
Responsibilities: Same as *Event* Have a specific icon.
Collaborations: Same as *Event*
Protocols: Same as *Event*
Unit Tests: Same as *Event* except: Duration is at most one hour.

1.7 Task

Task
Author: Simon Maling
SuperClass: -
SubClasses: -
Responsibilities: Creates itself. Changes itself. Validates itself. Sets itself to complete.
Collaborations: Time Date

Protocols:
`public Boolean setTask()`
`public Task getTask()`
`public Boolean validateTask()`
`public Boolean deleteTask()`
`public Boolean completeTask()`
`public Boolean unCompleteTask()`
`public ArrayList<Task> getIncompleteTasks()`
`public ArrayList<Task> getImportantTasks()`
`public ArrayList<Task> getAllTasks()`
`public ArrayList<Task> orderTasksDeadlineAscending()`
`public ArrayList<Task> orderTasksDeadlineDescending()`
`public ArrayList<Task> orderTasksName()`
`public ArrayList<Task> orderTasksCategory(String cat)`

Unit Tests: Create a task. Complete a task. Uncomplete a task.

1.8 TaskList

TaskList
Author: Simon Maling
SuperClass:-
SubClasses:-
Responsibilities: Create a list of tasks. Show itself. Update itself. Manipulate a list of tasks.
Collaborations: Time Date

Protocols:
`public Boolean setTaskList(TaskList tl)`
`public Boolean getTaskList()`
`public voiid showTaskList()`
`public ArrayList<Task> getIncompleteTasks()`
`public ArrayList<Task> getImportantTasks()`
`public ArrayList<Task> getAllTasks()`
`public ArrayList<Task> orderTasksDeadlineAscending()`
`public ArrayList<Task> orderTasksDeadlineDescending()`
`public ArrayList<Task> orderTasksName()`
`public ArrayList<Task> orderTasksCategory(String cat)`

Unit Tests: Add tasks to the list. Remove tasks from the list. Order the list.

1.9 View, Calendar

View
Author: Samuel Jenkins
SuperClass:-
SubClasses: MonthView WeekView DayView Multi-WeekView YearlyView MovingView
Responsibilities: Providing the correct view. Providing valid data. Handle user input. Set changes made to the data by the user. Be an abstract class.
Collaborations: Display

Protocols:
`public View createDisplay()`
`public View createDisplay(Object[] oObject)`
`public Boolean show()`
`public Boolean show(Boolean bShow)`

Unit Tests: Check to see that the View can display itself. Check to see that the View can hide itself.

Calendar
Author: Codrin Morhan
SuperClass:-
SubClasses:-
Responsibilities: Display calendar
Collaborations: Event View
Protocols: `public void display()`
Unit Tests: See if it displays itself correctly.

1.10 MonthView, WeekView

MonthView
Author: Lloyd Woodroffe
SuperClass: MovingViews
SubClasses:-
Responsibilities: Creating a correctly formatted monthly view. Showing events during a month in the monthly view. Editing the event data in the monthly view.
Collaborations: Event Period

Protocols:
`public View MonthlyView()`
`public View MonthlyView (Event[] oEvent)`
`public Boolean show()`
`public Boolean show(Boolean bShow)`
`public Boolean addEvent(Event Period oPeriod)`
`public Boolean addEvent(EventPeriod oPeriod, Event oEvent)`
`public Boolean editEvent(EventPeriod oPeriod, Event oEvent)`
`public Boolean removeEvent(EventPeriod oPeriod, Event oEvent)`

Unit Tests: Check that an event is added to an Event Period in the monthly view. Check that a specific event is added to an Event Period in the monthly view. Check that a specific event is edited in a particular Event Period in the monthly view. Check that a specific event is removed from a particular Event Period in the monthly view.

WeekView
Author: Lloyd Woodroffe
SuperClass: MovingViews
SubClasses:-
Responsibilities: Creating a correctly formatted weekly view. Showing event data into a weekly view. Editing the event data in weekly view.
Collaborations:-

Protocols:
`public View WeeklyView()`
`public View WeeklyView (Event[] oEvent)`
`public Boolean show()`
`public Boolean show(Boolean bShow)`
`public Boolean addEvent(Event Period oPeriod)`
`public Boolean addEvent(EventPeriod oPeriod, Event oEvent)`
`public Boolean editEvent(EventPeriod oPeriod, Event oEvent)`
`public Boolean removeEvent(EventPeriod oPeriod, Event oEvent)`

Unit Tests: Check that an event is added to a Event Period in the week view. Check that a specific event is added to a Event Period in the week view. Check that a specific event is edited in a particular Event Period in the week view. Check that a specific event is removed from a particular Event Period in the week view.

1.11 DayView, Multi-WeekView

DayView
Author: Lloyd Woodroffe
SuperClass: MovingViews
SubClasses:-
Responsibilities: Creating a correctly formatted daily view. Showing event data into a daily view. Editing the event data in daily view.
Collaborations:-

Protocols:
`public View DailyView()`
`public View DailyView (Event[] oEvent)`
`public Boolean show()`
`public Boolean show(Boolean bShow)`
`public Boolean addEvent(Event Period oPeriod)`
`public Boolean addEvent(EventPeriod oPeriod, Event oEvent)`
`public Boolean editEvent(EventPeriod oPeriod, Event oEvent)`
`public Boolean removeEvent(EventPeriod oPeriod, Event oEvent)`

Unit Tests: Check that an event is added to a Event Period in the daily view. Check that a specific event is added to a Event Period in the daily view. Check that a specific event is edited in a particular Event Period in the daily view. Check that a specific event is removed from a particular Event Period in the daily view.

Multi-WeekView
Author: Lloyd Woodroffe
SuperClass: MovingViews
SubClasses:-
Responsibilities: Creating a correctly formatted multi-week view. Showing event data into a multi-week view. Editing the event data in multi-week view.
Collaborations:-

Protocols:
`public View MultiWeekView()`
`public View MultiWeekView (Event[] oEvent)`
`public Boolean show()`
`public Boolean show(Boolean bShow)`
`public Boolean addEvent(Event Period oPeriod)`
`public Boolean addEvent(EventPeriod oPeriod, Event oEvent)`
`public Boolean editEvent(EventPeriod oPeriod, Event oEvent)`
`public Boolean removeEvent(EventPeriod oPeriod, Event oEvent)`

Unit Tests: Check that an event is added to a Event Period in the multi-week view. Check that a specific event is added to a Event Period in the multi-week view. Check that a specific event is edited in a particular Event Period in the multi-week view. Check that a specific event is removed from a particular Event Period in the multi-week view.

1.12 YearlyView

YearlyView
Author: Samuel Jenkins
SuperClass: MovingViews
SubClasses:-
Responsibilities: Creating a correctly formatted yearly view. Showing event data into a yearly view. Editing the event data in yearly view.
Collaborations:-

Protocols:
`public View YearlyView()`
`public View YearlyView (Event[] oEvent)`
`public Boolean show()`
`public Boolean show(Boolean bShow)`
`public Boolean addEvent(Event Period oPeriod)`
`public Boolean addEvent(EventPeriod oPeriod, Event oEvent)`
`public Boolean editEvent(EventPeriod oPeriod, Event oEvent)`
`public Boolean removeEvent(EventPeriod oPeriod, Event oEvent)`

Unit Tests: Check that an event is added to a Event Period in the yearly view. Check that a specific event is added to a Event Period in the yearly view. Check that a specific event is edited in a particular Event Period in the yearly view. Check that a specific event is removed from a particular Event Period in the yearly view.

1.13 MovingView, EventData

MovingView
Author: Samuel Jenkins
SuperClass:-
SubClasses: DayView WeeklyView Multi-WeekView MonthlyView YearlyView
Responsibilities: Update view regularly.
Collaborations:-
Protocols: `public Boolean updateDisplay()`
Unit Tests: Check that the view is updated without error at timely intervals.

EventData
Author: Samuel Jenkins
SuperClass: Views
SubClasses:-
Responsibilities: Get the data to be held from the Events.
Collaborations: DayView WeeklyView Multi-WeekView MonthlyView YearlyView Period
Protocols: `public EventPeriod[] getAllEventData(Period period)`
Unit Tests: Check that all events for a view are added to that period.

1.14 Date

Date
Author: Ceris Land
SuperClass: -
SubClasses:-
Responsibilities: Get date of system clock. Set the current date. Display current date. Compare two dates. Compare date to today.
Collaborations: DigitalOrganizer Calendar Event EventList AddressBook Task

Protocols:
`public DateFormat getDate()`
`public Boolean setDate()`
`public Boolean displayDate()`
`public Boolean isDateAfter(DateFormat d)`
`public Boolean isDateAfter(DateFormat d1, DateFormat d2)`
`public Boolean isDateBetween(DateFormat d1, DateFormat d2)`

Unit Tests: Test if date is got correctly. Test if date is after or between given dates.

1.15 DigitalOrganiser

DigitalOrganiser
Author: Yan Sun
SuperClass:-
SubClasses:-
Responsibilities: Control the main flow of execution of the application. Controls user input. Initialise the application. Initialise all other necessary resources. Exit cleanly; save data upon exit. Get and store who the current user is.
Collaborations: UserInterface User Date Time Store

Protocols:
`public Boolean Initialise()`
`public DateFormat getTime()`
`public DateFormat getDate()`
`public Boolean setCurrentUser(User u)`
`public User getCurrentUser()`
`public Boolean initialiseCalendar(User u)`
`public Boolean initialiseEvent(User u)`
`public Boolean initialiseAddressBook(User u)`
`public Boolean initialiseAll(User u)`
`public Boolean initialiseTask(User u)`
`public Boolean exit(Boolean b)`
`public Boolean exit()`

Unit Tests: Possible only when all the subsystems are available.

1.16 Time

Time
Author: Ceris Land
SuperClass:-
SubClasses:-
Responsibilities: Get time. Set time. Display time. Compare time. Compare time to current time. Determine if time is between two times.
Collaborations: DigitalOrganiser UserInterface Event EventList Task

Protocols:
`public DateFormat getTime()`
`public Boolean setTime()`
`public Boolean displayTime()`
`public Boolean isTimeAfter(DateFormat d)`
`public Boolean isTimeAfter(DateFormat d1, DateFormat d2)`
`public Boolean isTimeBetween(DateFormat d1, DateFormat d2)`

Unit Tests: See if time is got correctly. Test if a time is after/between.

1.17 Store, File

Store
Author: Simon Maling
SuperClass: -
SubClasses: File
Responsibilities: Save data. Backup data. Restore data. Get a saved file. Be an abstract class.
Collaborations: DigitalOrganiser User

Protocols:
`public int getStoreType()`
`public Boolean setStoreType(String s)`
`public String getStoreLocation()`
`public Boolean setStoreLocation(String s)`
`public Boolean save()`
`public Boolean backup()`
`public Boolean restore()`
`public Boolean load()`

Unit Tests: Save a file. Exit program and check file exists. Try backupping.

File
Author: Simon Maling
SuperClass: Store
SubClasses:-
Responsibilities: Decide which file to load. Create a new file. Write to a file. Save file. Delete file. Restore backup file.
Collaborations: DigitalOrganiser User

Protocols:
`public Boolean newFile(String s)`
`public Boolean inputFile()`
`public Boolean writeFile()`
`public Boolean saveFile()`
`public Boolean appendFile()`
`public Boolean deleteFile()`
`public Boolean restoreFile()`

Unit Tests: Test all the methods.

1.18 UserInterface, User

UserInterface
Author: Simon Maling
SuperClass:-
SubClasses:-
Responsibilities: Initiates GUI. Updates GUI. Control display of calendar,address book and tasks.
Collaborations: DigitalOrganiser Event AddressBook TaskList

Protocols:
`public void showCalendar()`
`public void showCalendar()`
`public void showTasks()`

Unit Tests: Testing depends on other classes/subsystems.

User
Author: Yan Sun
SuperClass:-
SubClasses:-
Responsibilities: Create a new user. Store user name and password. Delete an user. Record who is the current user. Associate events,calendar and address book with user. Check password.
Collaborations: DigitalApplication

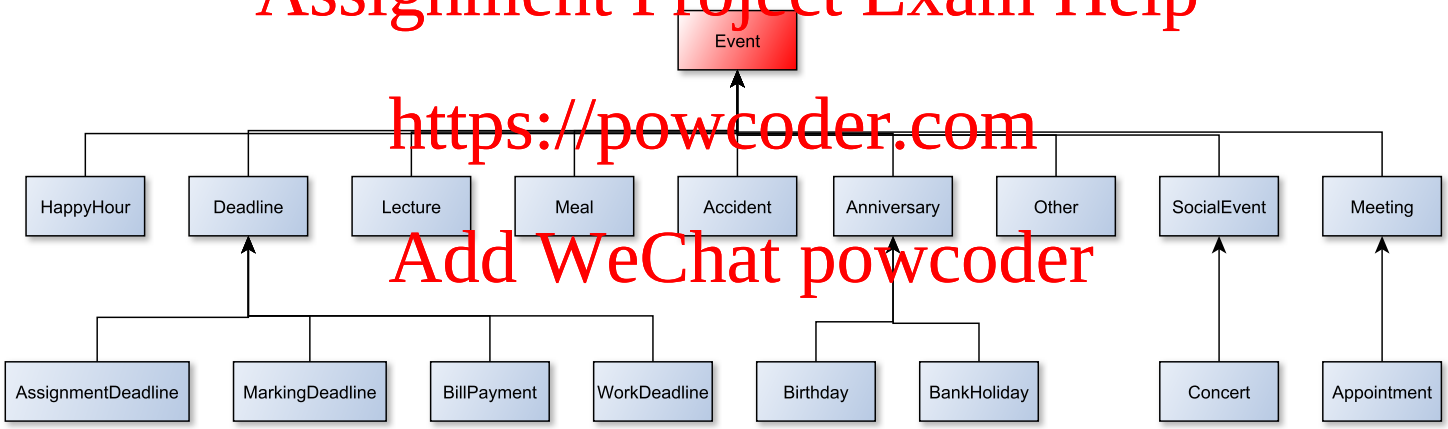
Protocols:
`public User getUser()`
`public Boolean newUser()`
`public Boolean deleteUser()`
`public Boolean setPassword()`
`public Boolean comparePassword(String pass)`

Unit Tests: Create an user. Delete an user. Check user validity.

Chapter 2

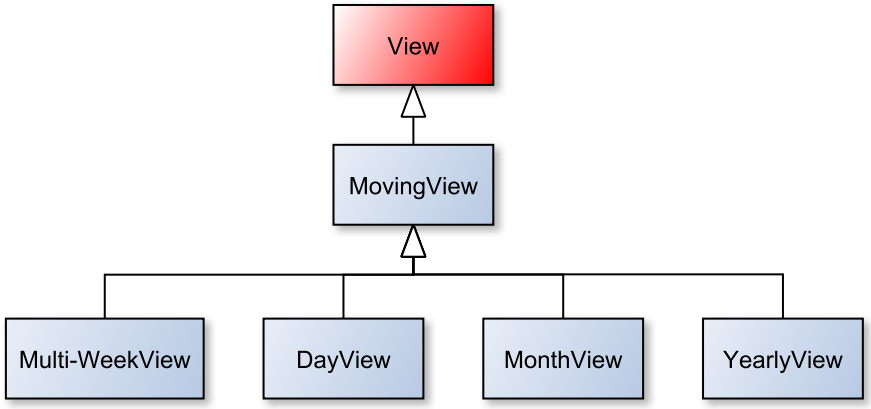
Class Hierarchy Diagrams and Descriptions

2.1 Types of Events



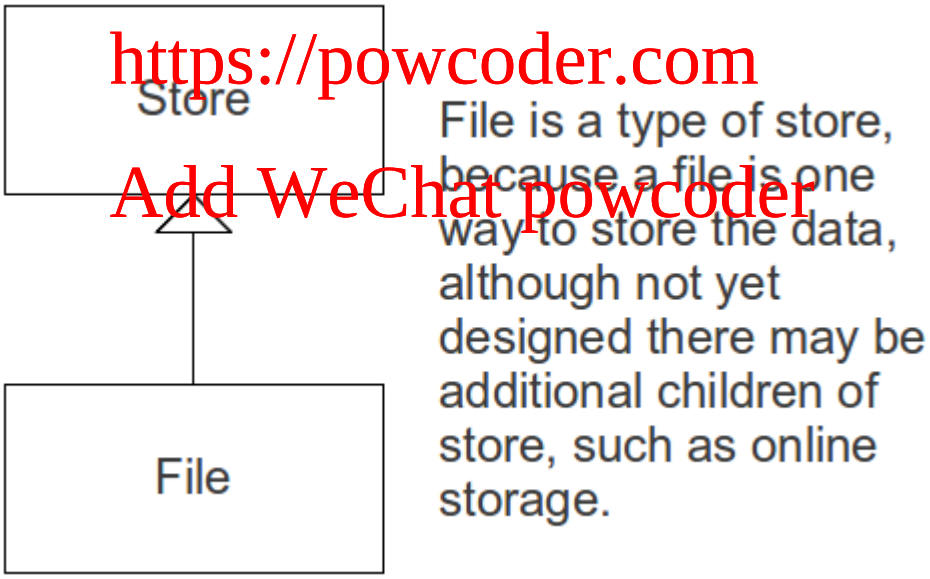
HappyHour/Deadline/Lecture/Meal/Accident/Anniversary/Other/SocialEvent/Meeting *is-kind-of* Event.
AssignmentDeadline/MarkingDeadline/BillPayment/WorkDeadline *is-kind-of* Deadline.
Birthday/BankHoliday *is-kind-of* Anniversary.
Concert *is-kind-of* SocialEvent.
Appointment *is-kind-of* Meeting.

2.2 Types of Views



Moving view *is-kind-of* View.
Multi-Week/Day/Month/Yearly View *is-kind-of* MovingView.

2.3 Types of Storage

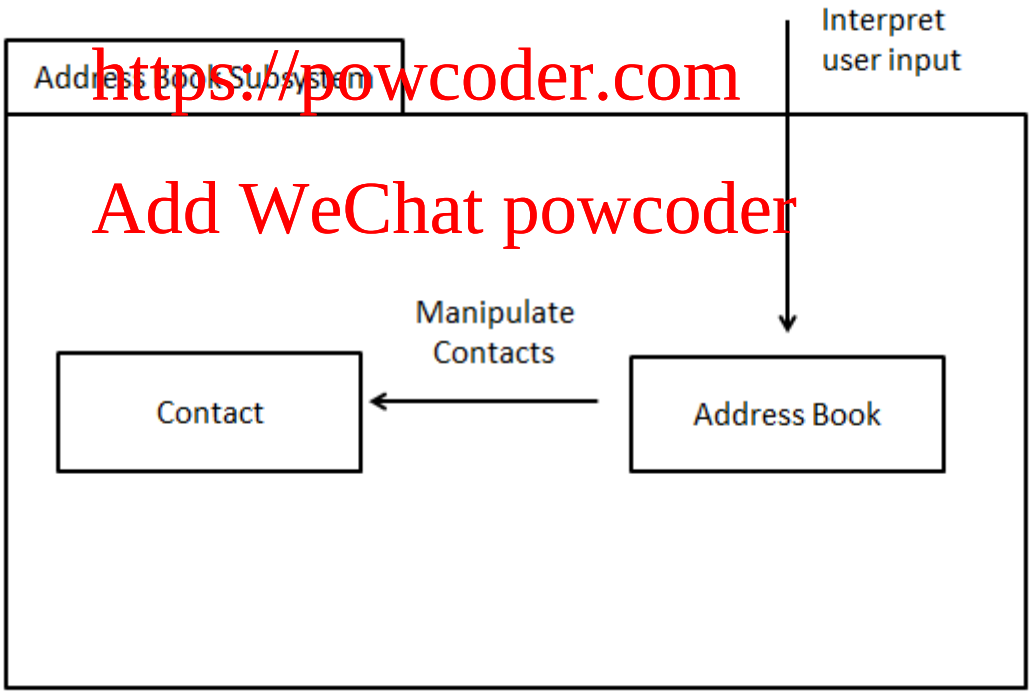


File *is-kind-of* Store.

Chapter 3

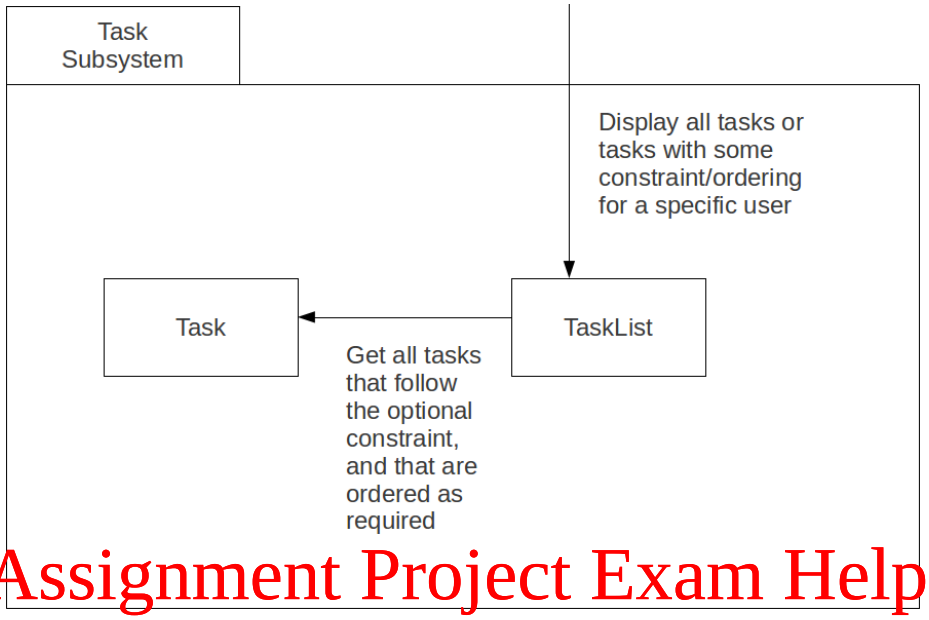
Sub-system Diagrams and Descriptions

3.1 Address Book Sub-system



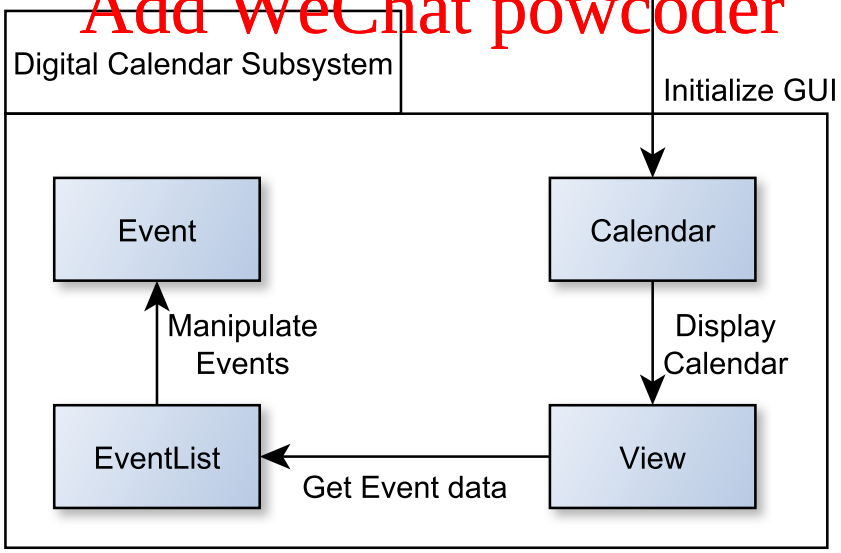
Contact *is-part-of* AddressBook.

3.2 Task Sub-system



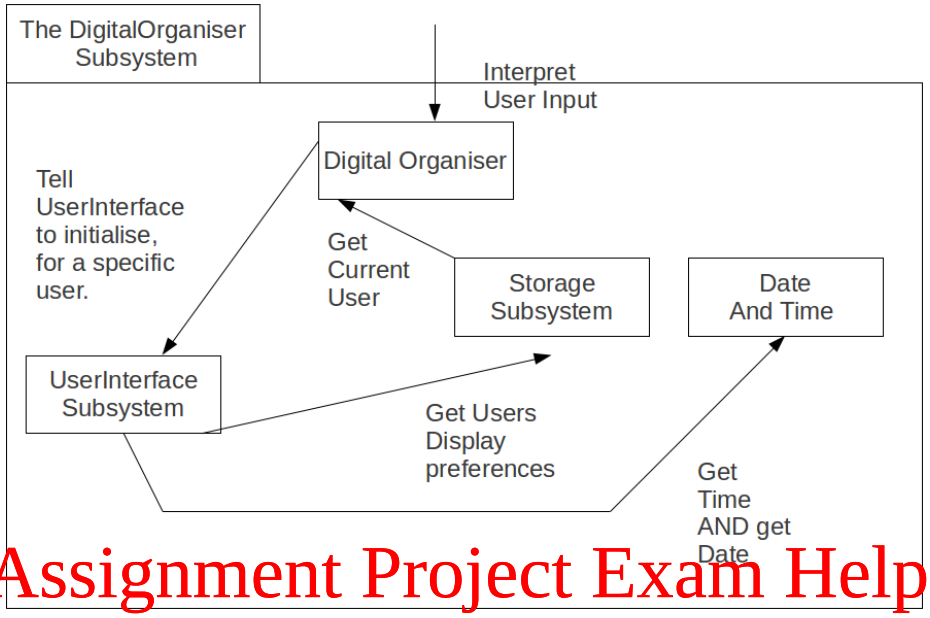
Task *is-part-of* TaskList.

3.3 Digital Calendar Sub-system



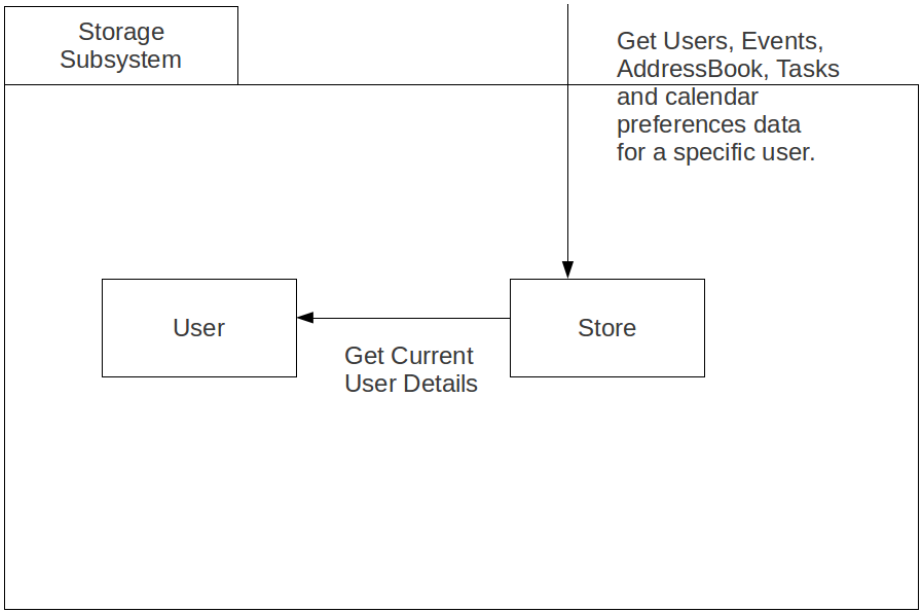
Event *is-part-of* EventList. EventList *is-part-of* Calendar Views. Calendar Views *is-part-of* Calendar.

3.4 Digital Organiser Sub-system



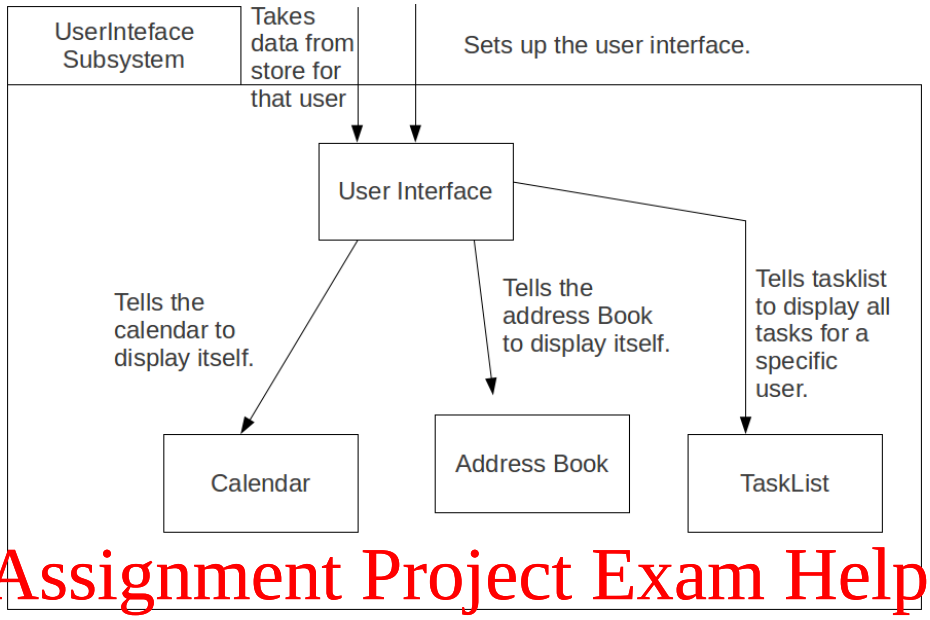
UserInterface *is-part-of* DigitalOrganizer.
The Storage Sub-system *is-part-of* DigitalOrganizer.
The Storage Sub-system *is-part-of* UserInterface.
Date and Time Sub-system *is-part-of* UserInterface.

3.5 Storage Sub-system



User *is-part-of* the Store.

3.6 User Interface Sub-system



TaskList *is-part-of* UserInterface.
AddressBook *is-part-of* UserInterface.
Calendar *is-part-of* UserInterface.

Chapter 4

Data File Format

We will use serialization for our files and we will use this to save objects directly to the files. We will use an ascii text file, the file will be named as either usernameCURRENT or usernameBACKUP[DATE]. The first name for when the file is the current file being used and the second one for when a backup is being made.

The file: Saving to the file will be done by the store class. The file will be saved to by saving the users store object to the file. This will include the date created as a date period object, and the last saved date also as a date period and time period object.(this will be replaced by the current date every time the file is saved). The store holds everything for the specific user.

The store object will include: The user object containing the username, and preferences of that user. Backups will be exactly the same except they will have backup and the backup date in the filename. This will mean that there are multiple files (one for each users).