**1. (20 points) Final May 2015**

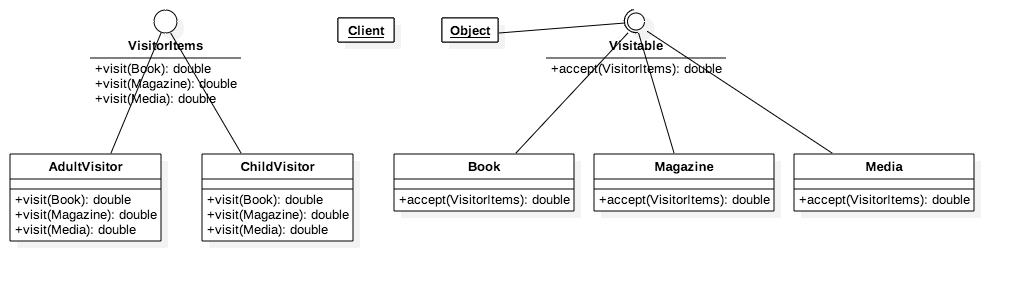
You are developing a library system for a Public Library. All items that its members (adults and children) can check out are books, magazines and media items. Loan periods for everything are 4 weeks. A fine is charged for each overdue item. Details are listed below:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Overdue Fines** | **Book/Day** | **Magazine/Day** | **Media/Day** | **Maximum Fine** |
| for Adults | $0.10 | $0.15 | $0.20 | $10.00 |
| for Children | $0.05 | $0.05 | $0.05 | $3.00 |

You now need to develop the FineCalculator class with the **Visitor Pattern**. You are also responsible for all other necessary classes/interfaces to make your FineCalculator work.

1) Draw the UML diagram for your solution with all necessary participants, responsibilities and relationships clearly specified.

2) Write Java code that shows all necessary attributes/methods with implementation logic for your design. (No need to provide test code).



Visitor.java

public interface Visitor {

public double visit(Book book);

public double visit(Magazine magazine);

public double visit(Media media);

}

AdultVisitor.java

public class AdultVisitor implements Visitor {

private double fine;

private double maxFine = 10;

@Override

public double visit(Book book) {

System.out.println("Book fine for Adult");

fine = book.getOverdue() \* 0.1;

if (fine >= maxFine) {

return maxFine;

}

return fine;

}

@Override

public double visit(Magazine magazine) {

System.out.println("Magazine fine for Adult");

fine = magazine.getOverdue() \* 0.15;

if (fine >= maxFine) {

return maxFine;

}

return fine;

}

@Override

public double visit(Media media) {

System.out.println("Media fine for Adult");

fine = media.getOverdue() \* 0.2;

if (fine >= maxFine) {

return maxFine;

}

return fine;

}

}

ChildVisitor.java

public class ChildVisitor implements Visitor {

private double fine;

private double maxFine = 3;

@Override

public double visit(Book book) {

System.out.println("Book fine for Child");

fine = book.getOverdue() \* 0.05;

if (fine >= maxFine) {

return maxFine;

}

return fine;

}

@Override

public double visit(Magazine magazine) {

System.out.println("Magazine fine for Child");

fine = magazine.getOverdue() \* 0.1;

if (fine >= maxFine) {

return maxFine;

}

return fine;

}

@Override

public double visit(Media media) {

System.out.println("Media fine for Child");

fine = media.getOverdue() \* 0.05;

if (fine >= maxFine) {

return maxFine;

}

return fine;

}

}

Visitable.java

public interface Visitable {

public double accept(Visitor visitor);

}

Book.java

public class Book implements Visitable {

private int loanPeriod;

private int loaned;

Book(int day) {

loaned = day;

}

public int getOverdue() {

if (loaned > loanPeriod) {

return loaned - loanPeriod;

}

return 0;

}

@Override

public double accept(Visitor visitor) {

return visitor.visit(this);

}

}

Magazine.java

public class Magazine implements Visitable {

private int loanPeriod;

private int loaned;

Magazine(int day) {

loanPeriod = day;

}

public int getOverdue() {

if (loaned > loanPeriod) {

return loaned - loanPeriod;

}

return 0;

}

@Override

public double accept(Visitor visitor) {

return visitor.visit(this);

}

}

Media.java

public class Media implements Visitable {

private int loanPeriod;

private int loaned;

Media(int day) {

loanPeriod = day;

}

public double getOverdue() {

if (loaned > loanPeriod) {

return loaned - loanPeriod;

}

return 0;

}

@Override

public double accept(Visitor visitor) {

return visitor.visit(this);

}

}

VisitorTest.java

public class VisitorTest {

public static void main(String[] args) {

AdultVisitor fineCalc1 = new AdultVisitor();

ChildVisitor fineCalc2 = new ChildVisitor();

Book book = new Book(30);

Magazine magazine = new Magazine(36);

System.out.println(book.accept(fineCalc1));

System.out.println(book.accept(fineCalc2));

System.out.println(magazine.accept(fineCalc1));

System.out.println(magazine.accept(fineCalc2));

}

}

**2. (20 points) Final May 2015**

You are developing an application that deals with the following workflow –

1) LOCAL - Create a user record in the local database. (via DBFaçade).

2) REMOTE - Create a payment account on a remote server. (via BillPayFaçade).

3) REMOTE – Submit credit card info for payment method details (via BillPayFaçade).

Both the DBFaçade and BillPayFaçade are classes locally available to you. Below are 2 interfaces they implement respectively

**public interface** DBFacadeInterface {

**public void** save(User user);

**public void** remove(User user);

}

**public interface** BillPayFacadeInterface {

**public** Account createPaymentAccount(User user);

**public void** removePaymentAccount(Account account);

**public boolean** submitCreditCard(CreditCard card);

}

During the workflow, in case step 3) fails for any user, you must rollback step 1) and step 2) to maintain data integrity. Please design/implement the above requirements with the **Command Pattern**. You need –

**1)** Draw the UML diagram for your solution with all necessary **participants, responsibilities and relationships clearly specified**.

**2) Write Java code** to implement your design illustrated above. (Need to write code to show how to roll back steps 1&2 if step 3 in the workflow fails).

**~~3. (20 points) Midterm July 2016~~**

You are working on a framework for GUI application development. The main components that you develop include GUI widgets (buttons, textboxes, checkboxes, dropdown boxes, scrollbars, etc.) and GUI containers (a special container called Window and other regular containers). Your idea is to design the model that stores the containers and widgets in a tree structure using a Composite pattern.

The requirement is for each GUI, there is one and only one special top-level container (called a Window). All widgets and regular containers can be placed on the Window. Regular containers can hold widgets as well as other regular containers. Your design should allow adding/removing widgets onto/from containers and painting them on the screen by simply calling the paint() method on each of the widgets and/or containers.

The way it works is illustrated below –

Painting always starts from the top (the Window) by calling its paint() method first. Then below the top-level Window, if widgets are found, the widgets’ paint() methods are called one by one. If regular containers are also found, you should call the first container’s paint() method and all its children’s paint() methods before moving to the second container on the window. This process goes as deep as the tree structure until all widgets and containers are painted.

1). Design your tree structure using the Composite pattern with a structure diagram showing

participants and their interfaces or responsibilities. 2). Write Java code for your implementation. (For all different widgets, you can use one Widget superclass to represent all of them.)

// YOUR CODE FOR PROB 25 GOES HERE

**4. (20 points) Midterm July 2016**

You are developing a model for displaying different contents on a client device. Suppose there are only 5 different content types (from most specific to most generic) - image, post, category, archive, and front-page. And correspondingly there are 5 different templates that can be used to display the content types. Below is a table that shows which template is good for which content types.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | Image | Post | Category | Archive | Front-page |
| ImageTemplate | Yes | No | No | No | No |
| SingleTemplate | Yes | Yes | No | No | No |
| CategoryTemplate | Yes | Yes | Yes | No | No |
| ArchiveTemplate | Yes | Yes | Yes | Yes | No |
| GenericTemplate | Yes | Yes | Yes | Yes | Yes |

But the rule is you must always use the most specific template to display any content type. For example, for the ‘Category’ content type, the ‘CategoryTemplate’, the ‘ArchiveTemplate’ and the ‘GenericTemplate’ are all possible templates, according to the above table. But since the ‘CategoryTemplate’ is the most specific one among the 3, you must use this one.

1). Use the **Chain of Responsibility pattern** to design the model with a structure diagram, showing all the necessary participants and their relationships and key behavior.

2). Build the chain of templates to make sure the rule is always followed.

3). Implement 1) and 2) with Java code.

You can use the skeleton code for Content for your implementation

public class Content {

private String contentType; //image, post, category, archive, front-page

//…

public Content(String type){

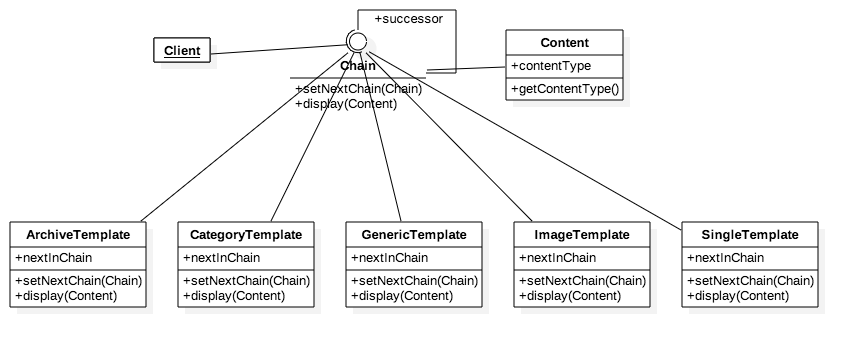
this.contentType = type;

}

//…

}

// YOUR CODE FOR PROB 26 GOES HERE



Content.java

public class Content {

    private String contentType;

    public Content(String type) {

        this.contentType = type;

    }

    public String getContentType() {

        return contentType;

    }

}

Chain.java

public interface Chain {

public void setNextChain(Chain nextChain);

public void display(Content content);

}

ArchiveTemplate.java

public class ArchiveTemplate implements Chain {

    private Chain nextInChain;

    @Override

    public void setNextChain(Chain nextChain) {

        nextInChain = nextChain;

    }

    @Override

    public void display(Content content) {

        if (content.getContentType() == "Image" || content.getContentType() == "Post" || content.getContentType() == "Category" || content.getContentType() == "Archive") {

            System.out.println("Content type: " + content.getContentType());

        } else {

            nextInChain.display(content);

        }

    }

}

CategoryTemplate.java

public class CategoryTemplate implements Chain {

    private Chain nextInChain;

    @Override

    public void setNextChain(Chain nextChain) {

        nextInChain = nextChain;

    }

    @Override

    public void display(Content content) {

        if (content.getContentType() == "Image" || content.getContentType() == "Post" || content.getContentType() == "Category") {

            System.out.println("Category template: " + content.getContentType());

        } else {

            nextInChain.display(content);

        }

    }

}

GenericTemplate.java

public class GenericTemplate implements Chain {

    private Chain nextInChain;

    @Override

    public void setNextChain(Chain nextChain) {

        nextInChain = nextChain;

    }

    @Override

    public void display(Content content) {

        if (content.getContentType() == "Image" || content.getContentType() == "Post" || content.getContentType() == "Category" || content.getContentType() == "Archive" || content.getContentType() == "Front-page")

            System.out.println("Generic template: " + content.getContentType());

        else {

            System.out.println("Only works for Image, Post, Category, Archive, Front-page");

        }

    }

}

ImageTemplate.java

public class ImageTemplate implements Chain {

    private Chain nextInChain;

    @Override

    public void setNextChain(Chain nextChain) {

        this.nextInChain = nextChain;

    }

    @Override

    public void display(Content content) {

        if (content.getContentType() == "Image") {

            System.out.println("Image template: " + content.getContentType());

        } else {

            nextInChain.display(content);

        }

    }

}

SingleTemplate.java

public class SingleTemplate implements Chain {

    private Chain nextInChain;

    @Override

    public void setNextChain(Chain nextChain) {

        nextInChain = nextChain;

    }

    @Override

    public void display(Content content) {

        if (content.getContentType() == "Image" || content.getContentType() == "Post") {

            System.out.println("Single template: " + content.getContentType());

        } else {

            nextInChain.display(content);

        }

    }

}

TestChain.java

public class TestChain {

    public static void main(String[] args) {

        Chain c1 = new ImageTemplate();

        Chain c2 = new SingleTemplate();

        Chain c3 = new CategoryTemplate();

        Chain c4 = new ArchiveTemplate();

        Chain c5 = new GenericTemplate();

        c1.setNextChain(c2);

        c2.setNextChain(c3);

        c3.setNextChain(c4);

        c4.setNextChain(c5);

        Content request = new Content("Post");

        c1.display(request);

    }

}

**5. (40 points) Midterm March 2015**

You are developing a framework for building GUI applications. The idea is to use an object model to store all GUI widgets (buttons, textboxes, checkboxes, dropdown boxes, scrollbars, etc.) and containers of them (containers and windows). To hold the widgets and manage their layout, you use a container that associates a widget manager with it (which manages positions and sizes of widgets that are placed on the container). But containers can also be placed on other containers so you can manage layout easily. To make a GUI executable/displayable, all widgets and containers have to be put on one and only one window widget that is a special container. So the model that stores all widgets, containers and a window is actually a data structure that starts from the window which can hold widgets and containers which in turn can hold other widgets and other containers so on and on. One important rule though is any container cannot hold itself and any 2 containers cannot hold each other.

To manage widgets layout/size on a container, you need to develop widget managers. The idea is to have 3 managers to start with – the central (position) manager, the quadrant manager and a sequential manager. At runtime, the GUI developer can specifically set a widget manager for a container or let it to the framework to handle it. That means if no widget manager is set for a container, the framework should first try applying the central manager (good for containers with 1 or 2 widgets only), then try quadrant manager (if a container has a total of exactly 4 widgets and/or containers) if central manager is not a good choice. If the quadrant manager still cannot manage the widgets nicely, the framework will pick the sequential manager as the choice of last resort (that is to simply lay the widgets sequentially one after another).

To display a GUI developed with this framework, you need to develop a displaying module that always starts from the window widget. Then it moves down to all the widgets and containers that are placed on it until all widgets and containers are displayed. When designing/implementing the displaying module, you have to keep in mind that multiple platforms are considered (Windows, Macintosh, and Linux with possibilities of other new platforms to add later on). But one class that common to all platforms is that you need to provide a class that goes through all widgets/containers stored in the data structure and call the corresponding displaying API to display them correctly.

You are responsible to design/implement the following components with design patterns we have learned so far in the course.

1) (25 points) The data model that stores all widgets/containers for a GUI application that allows application developers to easily manage widgets on the GUI.

2) (20 points) The widget-managing module that manages positions and sizes.

3) (20 points) The displaying API/implementations for rendering on different platforms.

4) (15 points) The ‘common’ class that helps the displaying API to render all widgets/containers stored in the data model.

// YOUR CODE FOR PROB 26 GOES HERE

**6.** You are designing an order processing system for a web-based business. All normal domestic orders will have to be shipped by FedEx. All large bulk orders are handled separately within the warehouse and might go to different shipping carriers. Most Favored Customers (your largest customers) may have special requirements for shipping and tracking, and they’re going to get what they want. All other international orders are handled in a generic manner. Draw both the class and object structures of your design and provide some skeleton code that shows how it works.