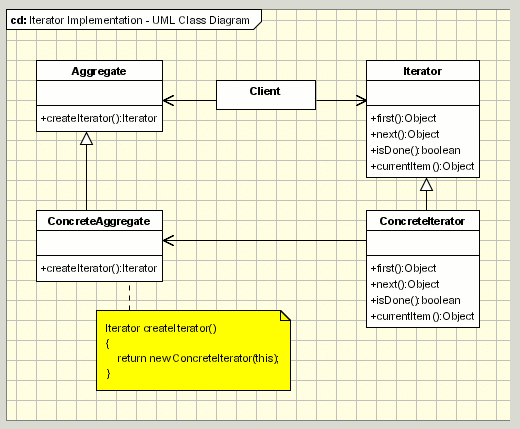
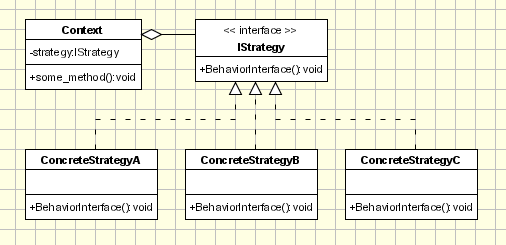
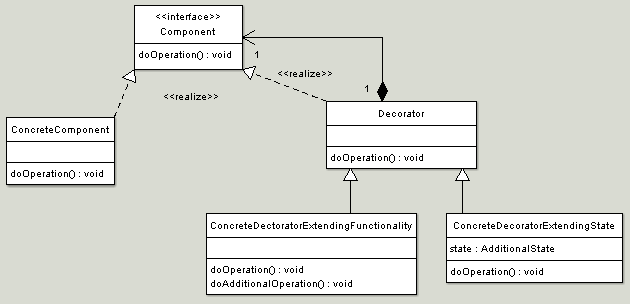
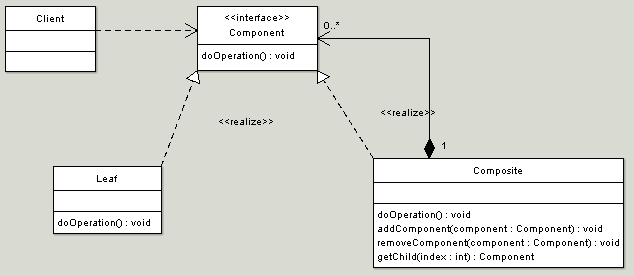
# Test Structure

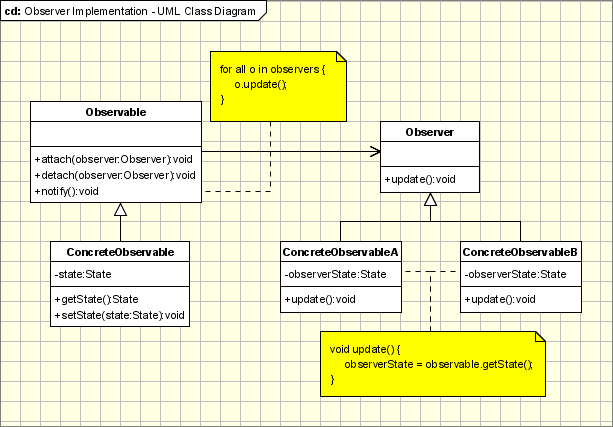
* Try to remember to import
* 4 questions of equal weight, based on the lectures and tutorial exercises
  + Object Oriented Design
    - Similar to assignment 1; good design in assignment 1 will help you in this question
    - All aspects of the design process - CRC cards, UML, code
  + Programming by Contract
    - Pre-conditions, post-conditions, invariants; their interaction with inheritance
    - The notion of a contract
  + Generic Types and Polymorphism
  + Design Patterns
* Concurrency will probably be in the exam as well.

## Design Patterns









# Labs

## Object-Oriented Programming

### Create an **Employee** class which has private fields for an employee's name and salary

**public** **class** **Employee** **implements** Cloneable {

**private** String name;

**private** **int** salary;

/\*\*

\* Creates a new Employee given a name and salary.

\* @param startEmployeeName

\* @param startSalary

\*/

**public** **Employee** (String startEmployeeName, **int** startSalary) {

**this**.name = startEmployeeName;

**this**.salary = startSalary;

}

}

### Create a **Manager** class that is a subtype of **Employee**, which also has a private field for a manager's hire date

**public** **class** **Manager** **extends** Employee {

**private** Calendar hireDate;

/\*\*

\* Creates a new Manager given a name, salary and hire date.

\* @param startEmployeeName

\* @param startSalary

\* @param hireDate

\*/

**public** **Manager** (String startEmployeeName, **int** startSalary, Calendar hireDate) {

**super** (startEmployeeName, startSalary);

**this**.hireDate = hireDate;

}

}

### Define constructors for the classes, and public getter and setter methods for the fields: document them using Javadoc

/\*\*

\* Sets the salary of the employee.

\* @param salary The salary of the employee.

\*/

**public** **void** **setSalary** (**int** salary) {

**this**.salary = salary;

}

/\*\*

\* Returns the salary of the Employee.

\* @return the salary of the Employee.

\*/

**public** **int** **getSalary** () {

**return** salary;

}

### Define the standard methods toString, equals and clone for **Employee** and **Manager**

* Employee should implement the Cloneable interface
* When you implement equals, you need to implement hashCode

**public** **class** **Employee** **implements** Cloneable {

...

**public** String **toString** () {

**return** **getClass**().getName() + **this**.name + "(" + **this**.salary + ")";

}

**@Override**

**public** Object **clone** () {

**try** {

**return** **super**.clone();

} **catch** (CloneNotSupportedException exc) {

exc.printStackTrace();

**return** **null**;

}

}

**@Override**

**public** **boolean** **equals** (Object obj) {

**if** (obj == **null**){

**return** **false**;

}

**if** (getClass() != obj.getClass()) {

**return** **false**;

}

Employee other = (Employee) obj;

**return** **this**.name == other.name && **this**.salary == other.salary;

}

**@Override**

**public** **int** **hashCode**() {

**int** hash = **0**;

hash += **7** \* name.hashCode();

hash += **13** \* salary;

**return** hash;

}

}

* The methods for **Manager** should call those for **Employee** using the **super** construct

**public** **class** **Manager** **extends** Employee {

...

**@Override**

**public** String **toString** () {

**return** **super**.toString() + **this**.getHireDateString();

}

**@Override**

**public** Manager **clone** () {

Manager clone = (Manager) **super**.clone();

clone.setHireDate(**this**.getHireDate());

**return** clone;

}

**@Override**

**public** **boolean** **equals** (Object obj) {

**if** (obj == **null**) {

**return** **false**;

}

**if** (getClass() != obj.getClass()) {

**return** **false**;

}

Manager other = (Manager) obj;

**return** **this**.hireDate == other.hireDate && **super**.equals(other);

}

**@Override**

**public** **int** **hashCode**() {

**int** hash = **super**.hashCode();

hash += **11** \* hireDate.hashCode();

**return** hash;

}

}

* What output do you expect when getClass().getName() is called in the toString method of **Employee** with a **Manager** object?
  + This should return **Manager**.

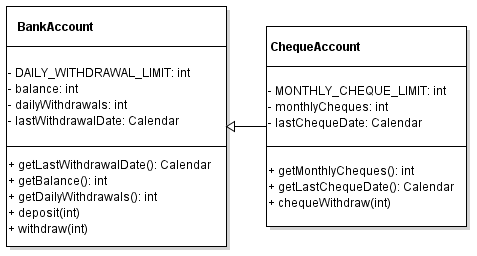
### Create a class for testing the **Employee** and **Manager** classes and define some tests for your methods

* What do you expect when you test whether an **Employee** is equal to a clone of the **Employee**?
  + They should be equal since their classes and fields are the same.
* What do you expect when you test whether a **Manager** is equal to an **Employee** with the same name and salary (and vice versa)?
  + They should not be equal since their classes differ.
* What do you expect when you test whether the name of an **Employee** is equal to the name of a clone of the **Employee**?
  + They should be equal.
* If you change the hire date of a clone of a **Manager**, is the hire date of the original **Manager** also changed?
  + No, because we use the Cloneable interface to make a deep copy in the clone function.

## Programming by Contract

### Draw a UML class diagram that incorporates the following requirements

* A **BankAccount** class for maintaining a customer's bank balance
  + Each bank account should have a current balance and methods implementing deposits and withdrawals
  + Money can only be withdrawn from an account if there are sufficient funds
  + Each account has a withdrawal limit of $800 per day
* A subclass of **BankAccount** called **ChequeAccount**
  + In addition to the constraints on **BankAccount**, there is a limit of 5 cheque withdrawals per month



### Give Java implementations of both classes

**import** **java.util.Calendar**;

**public** **class** **BankAccount** {

**private** **static** **final** **int** DAILY\_WITHDRAWAL\_LIMIT = **800**;

**private** **int** balance;

**private** **int** dailyWithdrawals;

**private** Calendar lastWithdrawalDate;

/\*\*

\* Creates a new BankAccount given a balance.

\* @param balance

\* @return BankAccount

\*/

**public** **BankAccount** (**int** balance) {

**super**();

**this**.balance = balance;

**this**.dailyWithdrawals = **0**;

Calendar now = Calendar.getInstance();

**this**.lastWithdrawalDate = now;

}

/\*\*

\* Gets the date of the last withdrawal from the BankAccount.

\* @return last withdrawal date of the BankAccount as a Calendar

\*/

**public** Calendar **getLastWithdrawalDate** () {

**return** lastWithdrawalDate;

}

/\*\*

\* Gets the current balance of the BankAccount.

\* @return current balance as an integer

\*/

**public** **int** **getBalance** () {

**return** balance;

}

/\*\*

\* Gets the total withdrawals from the BankAccount today.

\* @return total withdrawals as an integer

\*/

**public** **int** **getDailyWithdrawals** () {

**return** dailyWithdrawals;

}

/\*\*

\* Deposits money into the BankAccount.

\* @precondition depositValue is positive.

\* @postcondition balance increases.

\* This post-condition must be satisfied because the only operation in this method is addition.

\* @param depositValue The value to deposit in the BankAccount

\*/

**public** **void** **deposit** (**int** depositValue) **throws** Exception {

**if** (depositValue < **0**) {

**throw** **new** **Exception**("Deposit cancelled: Value to deposit must be positive.");

}

**this**.balance = **this**.balance + depositValue;

}

/\*\*

\* Withdraws money from the BankAccount.

\* @precondition withdrawalValue is positive, balance is positive, DAILY\_WITHDRAWAL\_LIMIT not reached.

\* @postcondition balance decreases.

\* @param withdrawalValue Amount to withdraw from the BankAccount

\* @throws Exception due to either insufficient funds or exceeding daily withdrawal limit

\*/

**public** **void** **withdraw** (**int** withdrawalValue) **throws** Exception {

/\*

\* How are limits enforced?

\* Last withdrawal date is used to determine when dailyWithdrawals is reset.

\* There are checks for whether there are sufficient funds to withdraw, whether the value to withdraw is positive and

\* whether the daily limit has been exceeded. These will throw exceptions.

\*/

// Check when last withdrawal occurred. If on a previous day, reset dailyWithdrawals to 0.

Calendar today = Calendar.getInstance();

**if** ((getLastWithdrawalDate().get(Calendar.DATE)) < today.get(Calendar.DATE) ||

(getLastWithdrawalDate().get(Calendar.MONTH)) < today.get(Calendar.MONTH) ||

(getLastWithdrawalDate().get(Calendar.YEAR)) < today.get(Calendar.YEAR)) {

**this**.dailyWithdrawals = **0**;

}

// Check withdrawalValue is positive.

**if** (withdrawalValue < **0**) {

**throw** **new** **Exception**("Withdrawal cancelled: Value to withdraw must be positive");

// Check balance has sufficient funds for withdrawal.

} **if** ((balance - withdrawalValue) < **0**) {

**throw** **new** **Exception**("Withdrawal cancelled: Insufficient funds");

// Check that daily withdrawal limit is not exceeded.

} **else** **if** ((withdrawalValue + **this**.dailyWithdrawals) > DAILY\_WITHDRAWAL\_LIMIT) {

**throw** **new** **Exception**("Withdrawal cancelled: exceeded daily withdrawal limit ($" + DAILY\_WITHDRAWAL\_LIMIT +")");

} **else** {

**this**.balance = **this**.balance - withdrawalValue;

**this**.lastWithdrawalDate = today;

dailyWithdrawals += withdrawalValue;

}

}

}

**import** **java.util.Calendar**;

**public** **class** **ChequeAccount** **extends** BankAccount {

**private** **static** **final** **int** MONTHLY\_CHEQUE\_LIMIT = **5**;

**private** **int** monthlyCheques;

**private** Calendar lastChequeDate;

/\*\*

\* Creates a new ChequeAccount given a balance and monthly cheques used.

\* @param balance balance of the ChequeAccount

\* @param monthlyCheques number of cheques used so far this month

\*/

**public** **ChequeAccount** (**int** balance, **int** monthlyCheques) {

**super**(balance);

**this**.monthlyCheques = monthlyCheques;

Calendar now = Calendar.getInstance();

**this**.lastChequeDate = now;

}

/\*\*

\* Returns the date when the last cheque was issued.

\* @return date when the last cheque was issued as a Calendar

\*/

**public** Calendar **getLastChequeDate** () {

**return** lastChequeDate;

}

/\*\*

\* Returns the number of cheques used this month.

\* @return the number of cheques used this month

\*/

**public** **int** **getMonthlyCheques** () {

**return** monthlyCheques;

}

/\*\*

\* Uses a cheque to withdraw from ChequeAccount.

\* @precondition withdrawalValue is positive, MONTHLY\_CHEQUE\_LIMIT not reached, balance has sufficient funds.

\* @postcondition balance decreases.

\* @param withdrawalValue value to withdraw using a cheque

\* @throws Exception if insufficient funds or exceeded daily withdrawal limit or if exceeded monthly cheque limit

\*/

**public** **void** **chequeWithdraw** (**int** withdrawalValue) **throws** Exception {

// Reset monthlyCheques if previous cheque was issued last month.

Calendar today = Calendar.getInstance();

**if** ((getLastChequeDate().get(Calendar.MONTH)) < today.get(Calendar.MONTH) ||

(getLastChequeDate().get(Calendar.YEAR)) < today.get(Calendar.YEAR)) {

**this**.monthlyCheques = **0**;

}

// Check if withdrawalValue is positive.

**if** (withdrawalValue < **0**) {

**throw** **new** **Exception**("Withdrawal cancelled: Value to withdraw must be positive");

// Check if MONTHLY\_CHEQUE\_LIMIT has been reached.

} **else** **if** ((**1** + **this**.monthlyCheques) > MONTHLY\_CHEQUE\_LIMIT) {

**throw** **new** **Exception**("Withdrawal cancelled: exceeded monthly cheque limit (" + MONTHLY\_CHEQUE\_LIMIT + ")");

} **else** {

withdraw(withdrawalValue);

**this**.lastChequeDate = today;

monthlyCheques += **1**;

}

}

}

* Explain how the limits on withdrawals are enforced within your system
  + Enforced using exceptions which are handled by the input system.

### Define a JUnit test case to test your implementations

**import** **static** org.junit.Assert.\*;

**import** **org.junit.\***;

**import** **banking.BankAccount**;

**public** **class** **BankAccount\_withdrawal** {

**@Test**

**public** **void** **withdraw\_normal\_balancedecrease**() {

BankAccount testBankAccount = **new** BankAccount(**100**);

**try** {

testBankAccount.withdraw(**100**);

} **catch** (Exception e) {

fail();

}

assertEquals(**0**, testBankAccount.getBalance());

}

**@Test**

**public** **void** **withdraw\_nofunds\_exception**() {

BankAccount testBankAccount = **new** BankAccount(**0**);

**try** {

testBankAccount.withdraw(**100**);

} **catch** (Exception e) {

// Success!

}

assertEquals(**0**, testBankAccount.getBalance());

}

**@Test**

**public** **void** **withdraw\_insufficientfunds\_exception**() {

BankAccount testBankAccount = **new** BankAccount(**50**);

**try** {

testBankAccount.withdraw(**100**);

} **catch** (Exception e) {

// Success!

}

assertEquals(**50**, testBankAccount.getBalance());

}

**@Test**

**public** **void** **withdraw\_negativeval\_exception**() {

BankAccount testBankAccount = **new** BankAccount(**600**);

**try** {

testBankAccount.withdraw(-**100**);

} **catch** (Exception e) {

// Success!

}

assertEquals(**600**, testBankAccount.getBalance());

}

**@Test**

**public** **void** **withdraw\_exceeddailylimit\_exception**() {

BankAccount testBankAccount = **new** BankAccount(**1000**);

**try** {

testBankAccount.withdraw(**900**);

} **catch** (Exception e) {

// Success!

}

assertEquals(**1000**, testBankAccount.getBalance());

}

}

## Object-Oriented Design

* Consider a university enrolments system with the following requirements
  + Students enrol in courses that are offered in particular semesters
  + Students receive grades (pass, fail, etc.) for courses in particular semesters
  + Courses may have prerequisites (other courses) and must have credit point values
  + For a student to enrol in a course, s/he must have passed all prerequisite courses
  + Course offerings are broken down into multiple sessions (lectures, tutorials and labs)
  + Sessions in a course offering for a particular semester have an allocated room and timeslot
  + If a student enrols in a course, s/he must also enrol in some sessions of that course

### Design an object-oriented system to implement the above requirements

* Define one or more use cases, e.g. for a student enrolling in a course that has a prerequisite that s/he has passed: Student enrolling in course with prerequisites passed
  + Student chooses classes
  + Class checks that prerequisites are passed
  + Student chooses sessions
  + Session checks that no clashes occur
  + Student enrols
* Provide CRC cards for your main classes

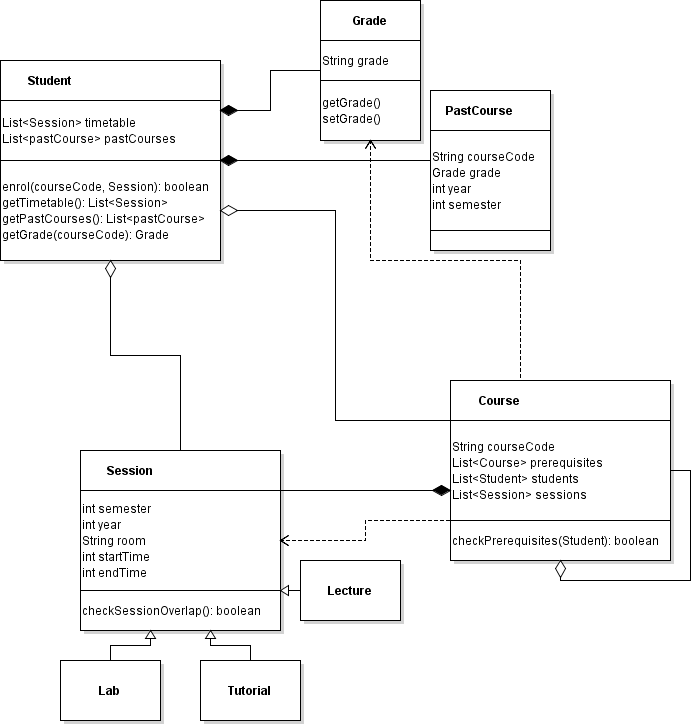
|  |  |
| --- | --- |
| *Class* | **Student** |
| *Responsibilities* | Stores timetable for current semester  Stores past courses and metadata  Enrols |
| *Collaborators* | Grade, Course, Session |

|  |  |
| --- | --- |
| *Class* | **Grade** |
| *Responsibilities* | Represent the grade a student receives for a course |
| *Collaborators* | Student, Course |

|  |  |
| --- | --- |
| *Class* | **Course** |
| *Responsibilities* | Stores course code, prerequisites, students and sessions  Checks if student satisfies prerequisites |
| *Collaborators* | Grade, Session, Student |

|  |  |
| --- | --- |
| *Class* | **Session** |
| *Responsibilities* | Stores session details for a course, for example, semester, room and class time  Ensures that student doesn't have timetable clashes |
| *Collaborators* | Course, Student, session types (Lab, Lecture, Tutorial) |

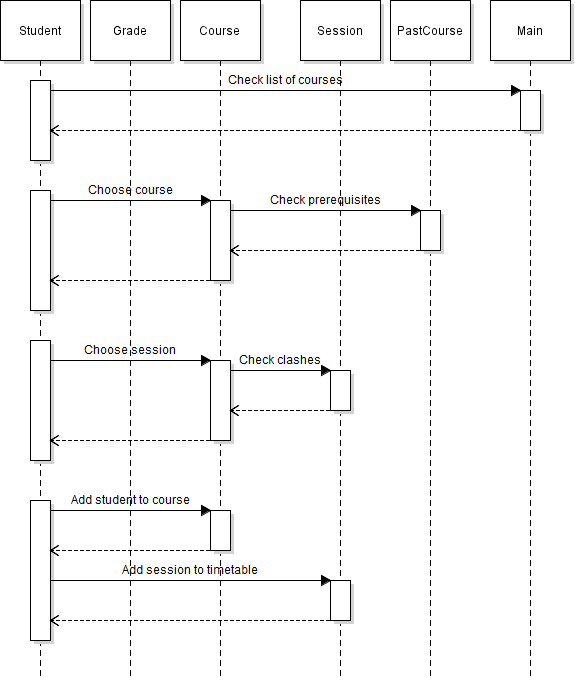
* Draw a UML class diagram for your initial design



* Clearly explain how your design distinguishes between a course and an instance of a course in a particular semester
  + Course represented by Course class, whereas instance of course is represented by the Session class.
* Identify any assumptions and design trade-offs you have made
  + Grades stored as strings only - need custom comparators.

### Consider the above use case for a student enrolling in a course that has a prerequisite that s/he has passed

* Describe using a walkthrough how this use case would be handled in your system: Student enrolling in course with prerequisites passed
  + Student chooses classes: check list of courses stored in main
  + Course checks prerequisites: use Course.checkPrerequisites(Student studentName)
  + Student chooses sessions: get sessions from the Course class
  + Session checks that no clashes occur: use Session.checkSessionOverlap(Student)
  + Student enrols: use enrol(courseCode, Session) to add student to Course and to add Session to timetable
* Define a UML sequence diagram corresponding to this walkthrough



### implement your design using Java, making sure that your code conforms to your design

**import** **java.util.\***;

**public** **class** **Student** {

**private** List<Session> timetable;

**private** HashMap<String, PastCourse> pastCourses; // string is the course code

**public** **Student**() {

**this**.timetable = **new** ArrayList<Session>();

**this**.pastCourses = **new** HashMap<String, PastCourse>();

}

...

**public** **boolean** **enrol**(Course c, Session s) {

**if** (!c.doesSessionExist(s)) **return** **false**;

**if** (!c.checkPrerequisites(**this**)) **return** **false**;

**if** (s.doesSessionOverlap(**this**)) **return** **false**;

c.addStudent(**this**);

**this**.addTimetable(s);

**return** **true**;

}

}

**import** **java.util.\***;

**public** **class** **Course** {

**private** String courseCode;

**private** HashMap<Course, Grade> prerequisites;

**private** List<Student> students;

**private** HashSet<Session> sessions;

**public** **Course**(String courseCode) {

**this**.courseCode = courseCode;

**this**.prerequisites = **new** HashMap<Course, Grade>();

**this**.students = **new** ArrayList<Student>();

**this**.sessions = **new** HashSet<Session>();

}

...

**public** **boolean** **doesSessionExist**(Session s) {

**return** sessions.contains(s);

}

**public** **boolean** **checkPrerequisites**(Student s){

**for** (Course **c:** prerequisites.keySet()) {

**if**(!s.getPastCourses().containsKey(c.getCourseCode())) **return** **false**;

**if**(s.getPastCourses().containsKey(c.getCourseCode()) &&

s.getGrade(c.getCourseCode()).compareTo(prerequisites.get(c)) == -**1**) **return** **false**;

}

**return** **true**;

}

}

**public** **class** **PastCourse** {

**private** String courseCode;

**private** Grade grade;

**private** **int** year;

**private** **int** semester;

**public** **PastCourse**(String courseCode, Grade grade, **int** year, **int** semester) {

...

}

...

}

**public** **class** **Session** {

**private** **int** semester;

**private** **int** year;

**private** String room;

**private** **int** day;

**private** **int** startTime;

**private** **int** endTime;

**public** **Session**(**int** semester, **int** year, **int** day, String room, **int** startTime,

**int** endTime) {

...

}

...

**public** **boolean** **doesSessionOverlap**(Student s) {

**boolean** sessionOverlap = **false**;

**for** (Session **n:** s.getTimetable()) {

**if** (**this**.overlapsWith(n)) {

sessionOverlap = **true**;

}

}

**return** sessionOverlap;

}

**public** **boolean** **overlapsWith**(Session s) {

**if** (**this**.year != s.getYear()) **return** **false**;

**if** (**this**.semester != s.getSemester()) **return** **false**;

**if** (**this**.day != s.getDay()) **return** **false**;

**if** (**this**.startTime <= s.getStartTime() && s.getStartTime() < **this**.endTime) **return** **true**;

**if** (s.getStartTime() <= **this**.startTime && **this**.startTime < s.getEndTime()) **return** **true**;

**return** **false**;

}

}

**public** **class** **Grade** **implements** Comparable<Grade>{

**private** String grade;

**public** **Grade**(String grade) {

**this**.grade = grade;

}

...

**@Override**

/\*\*

\* Order: HD > DN > CR > PS > FL

\*/

**public** **int** **compareTo**(Grade g) {

**int** thisGrade = **0**;

**int** otherGrade = **0**;

**if** (**this**.grade.equals("HD")) thisGrade = **5**;

**if** (**this**.grade.equals("DN")) thisGrade = **4**;

**if** (**this**.grade.equals("CR")) thisGrade = **3**;

**if** (**this**.grade.equals("PS")) thisGrade = **2**;

**if** (**this**.grade.equals("FL")) thisGrade = **1**;

**if** (g.getGrade().equals("HD")) otherGrade = **5**;

**if** (g.getGrade().equals("DN")) otherGrade = **4**;

**if** (g.getGrade().equals("CR")) otherGrade = **3**;

**if** (g.getGrade().equals("PS")) otherGrade = **2**;

**if** (g.getGrade().equals("FL")) otherGrade = **1**;

**if** (thisGrade == otherGrade) **return** **0**;

**if** (thisGrade > otherGrade) **return** **1**;

**if** (thisGrade < otherGrade) **return** -**1**;

**else** {

System.out.println("Invalid Grade " + **this**.grade + " compared to " + g.getGrade());

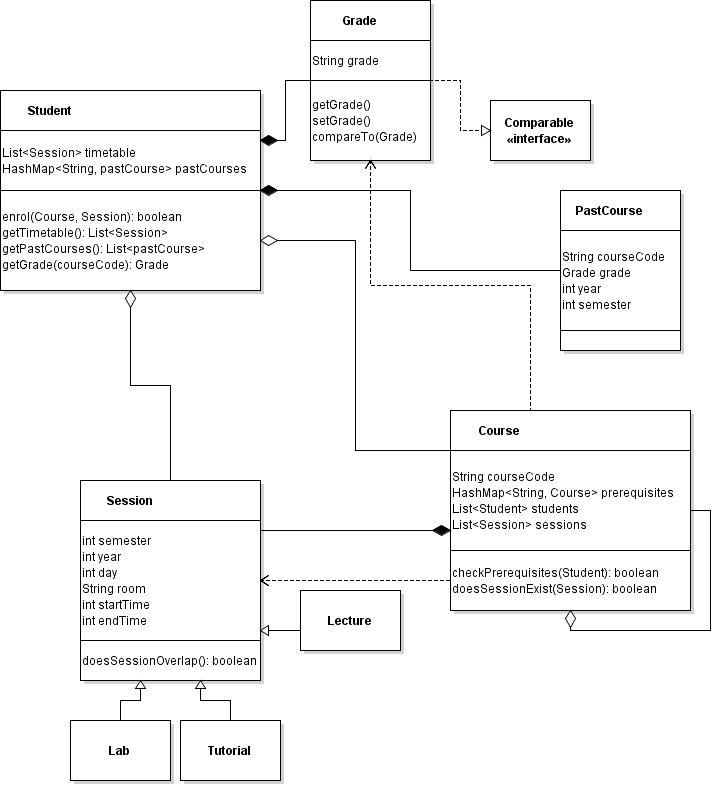
**return** **0**;

}

}

}

* Had to change representation of prerequisites and past courses. Could be avoided by stating data types passed in sequence diagram.



## Sets

### Define a **Set<E>** interface type that can handle elements of a generic type **E**

**import** **java.util.\***;

**interface** **Set**<E> **extends** Iterable<E>{

// set membership operations

**public** **void** **add**(E s);

**public** **void** **remove**(E s);

**public** **boolean** **contains** (E s);

// accessors

**public** **int** **getSize**();

**public** **Class** **getType**();

**public** List<E> **getItems**();

// basic operations on sets

**public** **boolean** **subset**(Set<E> ms);

**public** Set<E> **intersect**(Set<E> ms);

**public** Set<E> **union**(Set<E> ms);

// other

**public** **boolean** **equals**(Object o);

**public** **int** **hashCode**();

**public** Iterator<E> **iterator**();

}

### Define a class realizing the **Set<E>** interface that uses an **ArrayList<E>** to store the elements

**import** **java.util.\***;

**public** **class** **MySet**<E> **implements** Set<E>{

**private** ArrayList<E> items;

**private** **final** Class<E> type;

**public** **MySet**(Class<E> type){

**this**.items = **new** ArrayList<E>();

**this**.type = type;

}

**@Override**

**public** **int** **getSize**() {

**return** items.size();

}

**@Override**

**public** Class<E> **getType**() {

**return** **this**.type;

}

**@Override**

**public** Iterator<E> **iterator**() {

**return** items.iterator();

}

**@Override**

**public** **void** **add**(E s) {

**if** (!items.contains(s)) {

items.add(s);

}

}

**@Override**

**public** List<E> **getItems**() {

**return** **this**.items;

}

**@Override**

**public** **boolean** **subset**(Set<E> ms) {

Iterator<E> iter = ms.iterator();

**while** (iter.hasNext()) {

E elem = iter.next();

**if** (!(**this**.contains(elem))) {

**return** **false**;

}

}

**return** **true**;

}

**@Override**

**public** Set<E> **intersect**(Set<E> ms) {

Iterator<E> iter = ms.iterator();

Set<E> intSet = **new** MySet<E>(**this**.type);

**while** (iter.hasNext()) {

E elem = iter.next();

**if** (**this**.contains(elem)) {

intSet.add(elem);

}

}

**return** intSet;

}

**@Override**

**public** Set<E> **union**(Set<E> ms) {

Iterator<E> iter1 = **this**.iterator();

Iterator<E> iter2 = ms.iterator();

Set<E> unSet = **new** MySet<E>(**this**.type);

**while** (iter1.hasNext()) {

E elem1 = iter1.next();

unSet.add(elem1);

}

**while** (iter2.hasNext()) {

E elem2 = iter2.next();

unSet.add(elem2);

}

**return** unSet;

}

**@Override**

**public** **boolean** **equals**(Object o) {

**if** (o == **this**) **return** **true**; // Check if identical

**if** (o == **null**) **return** **false**; // Check if null

**if** (!(o **instanceof** MySet<?>)) **return** **false**; // Check if same class

MySet<?> other = (MySet<?>) o; // Check if same generic type

**if** (!other.getType().equals(**this**.getType())) **return** **false**;

**@SuppressWarnings**("unchecked") // compiler can’t check cast

MySet<E> o2 = (MySet<E>) other;

**for** (E **e:** o2.getItems()); // causes ClassCastException

**if** (other.getSize() != **this**.getSize()) **return** **false**; // check size

Iterator<E> iter = **this**.iterator(); // check elements are the same

**while** (iter.hasNext()) {

E elem = iter.next();

**if** (!o2.contains(elem)) **return** **false**;

}

**return** **true**;

}

**@Override**

**public** **int** **hashCode**() {

**return** items.hashCode();

}

**@Override**

**public** **void** **remove**(E s) {

Iterator<E> iter = **this**.iterator();

E elemToRemove = **null**;

**while** (iter.hasNext()) {

E elem = iter.next();

**if** (elem.equals(s)) {

elemToRemove = elem;

}

}

**if** (elemToRemove != **null**) {

**this**.items.remove(elemToRemove);

}

}

**@Override**

**public** **boolean** **contains**(E elem2) {

Iterator<E> iter = **this**.iterator();

**while** (iter.hasNext()) {

E elem = iter.next();

**if** (elem.equals(elem2)) {

**return** **true**;

}

}

**return** **false**;

}

}

* Explain how your code enforces the class invariant that all elements of a set are distinct
  + In the add function, I check if the set already contains the element before adding it.

### By having **Set<E>** extend **Iterable<E>**, define a method iterator() thaT returns an iterator over sets

**import** **java.util.\***;

**interface** **Set**<E> **extends** Iterable<E>{

...

**public** Iterator<E> **iterator**();

}

**import** **java.util.\***;

**public** **class** **MySet**<E> **implements** Set<E>{

...

**@Override**

**public** Iterator<E> **iterator**() {

**return** items.iterator();

}

}

### Write a test class for **Set<E>** that uses a **Scanner** to read elements from an input file, then add them to various sets (e.g. of type **String)**

**public** **void** **testScanner**() {

ArrayList<Set<Character>> sets = **new** ArrayList<Set<Character>>();

**try** {

BufferedReader br = **new** BufferedReader(**new** FileReader("input.txt"));

**for** (String line = br.readLine(); line != **null**; line = br.readLine()) {

Set<Character> set = **new** MySet<Character>(Character.class);

**for** (**int** i = **0**; i < line.length(); i++) {

set.add(line.charAt(i));

}

sets.add(set);

}

br.close();

} **catch** (IOException e) {

e.printStackTrace();

}

}

## Graphs

* A *directed graph* is a set of nodes and a binary relation over this set (that node *a* is related to node *b* can be represented by an arrow from *a* to *b*)

### Define a generic **Graph<E>** interface type that handles nodes of a generic type **E**

**import** **java.util.Set**;

**public** **interface** **Graph**<E> { // Graph has objects of arbitrary type E

// Basic graph operations

// Accessors (getters)

**public** **int** **size**();

**public** Set<Node<E>> **getNodes**();

// Mutator (setters)

**public** **void** **addNode**(E a);

**public** **void** **removeNode**(E a);

**public** **void** **addConnection**(E from, E to);

**public** **void** **removeConnection**(E from, E to);

// Complex graph operations

**public** **boolean** **contains**(E a); // handy because standard Java term

**public** **boolean** **isConnected**(E a, E b); // a and b are graph Nodes

}

### Choose one of the graph representations and provide a class that implements the **Graph<E>** interface type in this way

**import** **java.util.\***;

**public** **class** **AdjacencyListGraph**<E> **implements** Graph<E>{

**private** **int** size;

**private** HashMap<E, Node<E>> nodes;

**public** **AdjacencyListGraph**() {

**super**();

**this**.size = **0**;

**this**.nodes = **new** HashMap<E, Node<E>>();

}

**@Override**

**public** **int** **size**() {

**return** size;

}

**@Override**

**public** **void** **addNode**(E a) {

Node<E> n = **new** Node<E>(a);

nodes.put(n.getNodeObj(), n);

size++;

}

**@Override**

**public** **void** **removeNode**(E a) {

List<Node<E>> connections = nodes.get(a).getConnections();

**for** (Node<E> **n:** connections) {

n.removeConnection(nodes.get(a));

}

nodes.remove(nodes.get(a));

size--;

}

**@Override**

**public** **void** **addConnection**(E from, E to) {

nodes.get(from).addConnection(nodes.get(to));

nodes.get(to).addConnection(nodes.get(from));

}

**@Override**

**public** **void** **removeConnection**(E from, E to) {

nodes.get(from).removeConnection(nodes.get(to));

nodes.get(to).removeConnection(nodes.get(from));

}

**@Override**

**public** **boolean** **contains**(E a) {

**return** nodes.get(a) != **null**;

}

**@Override**

**public** **boolean** **isConnected**(E a, E b) {

**return** nodes.get(a).isConnected(b);

}

**public** Set<Node<E>> **getNodes**() {

**return** **new** HashSet<Node<E>>(nodes.values());

}

}

**class** **Node**<E>{

**private** E nodeObj;

**private** HashMap<E, Node<E>> connections;

**public** **Node**(E nodeObj) {

**super**();

**this**.nodeObj = nodeObj;

**this**.connections = **new** HashMap<E, Node<E>>();

}

**public** List<Node<E>> **getConnections**() {

**return** **new** ArrayList<Node<E>>(connections.values());

}

**public** **boolean** **isConnected**(E a){

**return** connections.containsKey(a);

}

**public** **boolean** **isConnected**(Node<E> n) {

**return** connections.containsValue(n);

}

**public** **void** **addConnection**(Node<E> connectedNode) {

connections.put(connectedNode.getNodeObj(), connectedNode);

}

**public** **void** **removeConnection**(Node<E> connectedNode) {

connections.remove(connectedNode);

}

**public** E **getNodeObj**() {

**return** nodeObj;

}

}

* If a graph has *n* nodes, an adjacency matrix is an *n* × *n* matrix with entry 1 (or some other data value) in the *i*, *j* element if node *i* is related to node *j*
* The adjacency list is a list whose elements are lists, where for element *i* of the main list, the associated list is of those nodes *j* such thatnode *i* is related to node *j*

### Define some suitable class invariants for your implementation of the **Graph<E>** interface type and show that they are satisfied

* Size is always greater than or equal to 0
  + Initially, graph contains no nodes and its size is 0
  + Size increases when a node is successfully added
  + Size increases when a node is successfully removed
* Number of edges is always at most for n vertices (when undirected graph is complete)
  + Can only have 1 connection between two nodes

## Basic Search Algorithms

### Apply the Strategy pattern so that the successors of a node are sorted using an anonymous class that implements the **Comparator** interface type

List<Node<String>> connections = curr.getConnections();

**final** Collator col = Collator.getInstance();

Comparator<Node<String>> com = **new** Comparator<Node<String>>() {

**@Override**

**public** **int** **compare**(Node<String> n, Node<String> m) {

**if** (col.compare(n.getNodeObj(), m.getNodeObj()) == -**1**) **return** -**1**;

**if** (n.getNodeObj().equals(m.getNodeObj())) **return** **0**;

**if** (col.compare(n.getNodeObj(), m.getNodeObj()) == **1**) **return** **1**;

System.out.println("Node<String> objects cannot be compared");

**return** **0**;

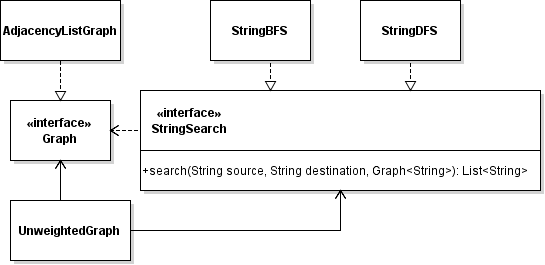
}

};

Collections.sort(connections, com);

* This **Comparator** is either in the method that computes the successors of a node or the method that adds those successors to the **Queue** – is either of these options better?
  + The method that compute the children of a node is part of the Node<E> class, so this would reduce encapsulation by asking Node<E> to perform search-related actions.
  + The method that adds children to the queue is the search function, so this option is better.

### Draw a UML class diagram of your program, making sure your design and code conforms to the Strategy pattern



## Problem-Solving Algorithms

### Apply the Strategy pattern so that the heuristic used by A\* search is passed as a parameter to an appropriate method

**public** **interface** **Heuristic**<E> {

**public** **int** **distanceLeft**(Node<E> source, Node<E> dest);

}

**public** **interface** **Graph**<E> { // Graph has objects of arbitrary type E

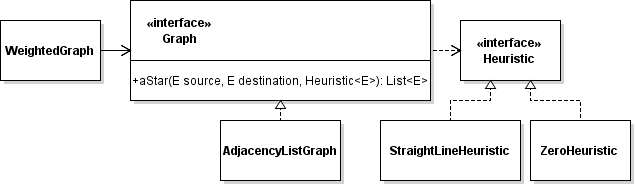
...

**public** List<E> **aStar**(E source, E destination, Heuristic<E> h);

}

* Are there any trade-offs associated with the choice of class or method?
  + Need to code more but more flexible when adding in new strategies.
  + Need to make **Heuristic** object in context.

### Draw a UML class diagram of your program, making sure your design and code conforms to the Strategy pattern



## Design Patterns

* Each **Item** has a given price, and the price of an **Assembly** is just the total price of all the parts in the assembly

### Use the Composite pattern to write Java classes for an **Assembly** and an **Item** with methods for calculating the total price of any part

**public** **interface** **Part** {

**public** **int** **getPrice**();

**public** **void** **addPart**(Part p);

**public** **void** **removePart**(Part p);

**public** Part **getChild**(**int** index);

}

**public** **class** **Item** **implements** Part {

**private** **int** price;

**public** **Item** (**int** price) {

**this**.price = price;

}

**public** **int** **getPrice**() {

**return** **this**.price;

}

**@Override**

**public** **void** **addPart**(Part p) {

}

**@Override**

**public** **void** **removePart**(Part p) {

}

**@Override**

**public** Part **getChild**(**int** index) {

**return** **null**;

}

}

**import** **java.util.\***;

**public** **class** **Assembly** **implements** Part {

**private** **int** price;

**private** ArrayList<Part> children;

**public** **Assembly** (**int** price) {

**this**.price = price;

**this**.children = **new** ArrayList<Part>();

}

**public** **int** **getPrice**() {

**int** sumPrices = **this**.price;

**for** (Part **p:** children) {

sumPrices += p.getPrice();

}

**return** sumPrices;

}

**@Override**

**public** **void** **addPart**(Part p) {

children.add(p);

}

**@Override**

**public** **void** **removePart**(Part p) {

children.remove(p);

}

**@Override**

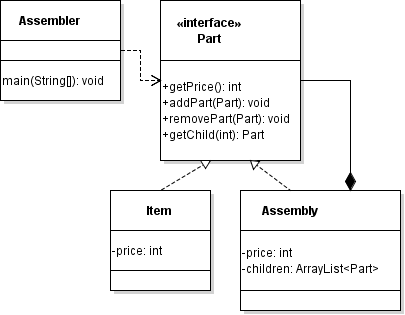
**public** Part **getChild**(**int** index) {

**return** children.get(index);

}

}

#### Draw a UML class diagram for your program, making sure your code conforms to the Composite pattern



### Use the Decorator pattern to allow discounted prices: discounts can apply to both basic and assembled parts, even already discounted parts

**public** **class** **Discount** **extends** Decorator {

**private** **int** discount; // discount as a percentage

**public** **Discount**(Part p, **int** discount) {

**super**(p);

**this**.discount = discount;

}

**@Override**

**public** **int** **getPrice**() {

**return** p.getPrice() \* discount / **100**;

}

**@Override**

**public** **void** **addPart**(Part p) {

**this**.p.addPart(p);

}

**@Override**

**public** **void** **removePart**(Part p) {

**this**.p.removePart(p);

}

**@Override**

**public** Part **getChild**(**int** index) {

**return** **this**.p.getChild(index);

}

}

**public** **class** **Discount** **extends** Decorator {

**private** **int** discount; // discount as a percentage

**public** **Discount**(Part p, **int** discount) {

**super**(p);

**this**.discount = discount;

}

**@Override**

**public** **int** **getPrice**() {

**return** p.getPrice() \* discount / **100**;

}

**@Override**

**public** **void** **addPart**(Part p) {

**this**.p.addPart(p);

}

**@Override**

**public** **void** **removePart**(Part p) {

**this**.p.removePart(p);

}

**@Override**

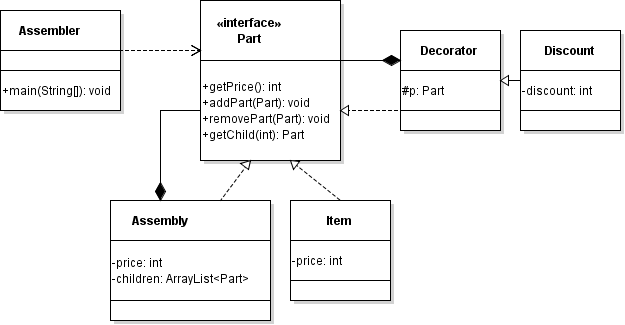
**public** Part **getChild**(**int** index) {

**return** **this**.p.getChild(index);

}

}

#### Extend the UML class diagram for your program, making sure your code conforms to both the Composite and Decorator patterns



## Concurrency

// Producer implements Runnable

Runnable run1 = **new** Producer();

Thread thread1 = **new** Thread(run1);

thread1.start();

* Explain why the implementation of the **BoundedQueue** class is not threadsafe
  + Different threads can add and remove from the queue at the same time. As a result, you cannot accurately keep track of the head and the tail of the queue.

### Use a re-entrant lock with two conditions to make the implementation threadsafe

**import** **java.util.concurrent.locks.\***;

**public** **class** **BoundedQueue**<E> {

...

**private** **final** ReentrantLock lock;

**private** **final** Condition spaceAvail;

**private** **final** Condition valueAvail;

**public** **BoundedQueue**(**int** capacity) {

...

lock = **new** ReentrantLock();

spaceAvail = lock.newCondition();

valueAvail = lock.newCondition();

}

**public** **void** **add**(E newValue) **throws** InterruptedException {

lock.lock(); // 1 acquire lock

**try** {

**if** (size == elements.length) { // 2 checks if queue is full

spaceAvail.await(); // 3 since queue is full, wait for space available

}

... // add element

valueAvail.signal(); // 4 signal value available before returning

} **finally** {

lock.unlock(); // 5 release lock when finished

}

}

**public** E **remove**() **throws** InterruptedException {

lock.lock(); // 1 acquire lock

**try** {

**if** (size == **0**) { // 2 checks if queue is empty

valueAvail.await(); // 3 since queue is empty, wait for value available

head = **0**;

}

... // remove element

spaceAvail.signal(); // 4 signal space available before returning

**return** r;

} **finally** {

lock.unlock(); // 5 release lock when finished

}

}

}