Workshop: How to Design Class Hierarchies 2004

Viera K. Proulx

July 19, 2004

How to design class hierarchies: an introduction to object-oriented programming
Matthias Felleisen, Matthew Flatt, Robert Findler, Kathy Gray, Shriram Krishnamurthi, Viera K. Proulx
p. cm.
Includes index.
ISBN 0-262-06218-6 (hc.: alk. paper)
1. Computer Programming. 2. Electronic data processing.

QA76.6 .H697 2001
005.1'2—dc21 00-048169

2 Lab Monday pm:

Goals

Learn to use ProfessorJ

- Define classes using the *Insert Java Class* widget
- Make examples of data in the *Java Interactions Box*
- Explore the use of the *Test Case* tool
- Explore the use of the *Interactions* window

Design and use simple classes, classes with containment and unions:

- Translate information into data definitions
- Interpret data as information
- Translate class diagram into class definitions
- Represent Java classes as class diagrams

Simple classes

Exercise 2.1.1 The *Book* class.
Take a look at this problem statement:

> Develop a program that assists a bookstore manager. The program should keep a record for each book. The record must include its book, the author's name, its price, and its publication year.

This example is included in you notes.

- Use the Java class widget to define this class. Select *add class diagram* checkbox.
- In the Interactions box make instances of this class that represent the following information:

  1. Daniel Defoe, *Robinson Crusoe*, $15, 1719;
  2. Joseph Conrad, *Heart of Darkness*, $12, 1902;

3. Pat Conroy, *Beach Music*, $9, 1996.

- Use the Test Case tool to check that some of the fields of the instances you defined have the correct values.
- Use the Interactions pane to make an instance of your favorite book. Try some tests similar to those done iin the Test Case tool.

Exercise 2.1.2 Understanding data definitions.

Study this class definition:

```
class Image {
  int height /* pixels */;
  int width /* pixels */;
  String source /* file name */;
  String quality /* informal */;

  Image(int height, int width, String source, String quality) {
    this.height = height;
    this.width  = width ;
    this.source = source;
    this.quality = quality;
  }
}
```

Draw the class diagram.

The class definition was developed in response to this problem statement:

> Develop a program that creates a gallery from image descriptions. Those specify the height, the width, the source of the images, and, for aesthetic reasons, some informal information about its quality.

Interpret the following three instances in this context:

```
Image im1 = new Image(5, 10, "small.gif", "low");
Image im2 = new Image(120, 200, "med.gif", "low");
Image im3 = new Image(1200, 1000, "large.gif", "high");
```

What is the value of *im2.quality*, of *im3.height*?

Exercise 2.1.3 Translate the class diagram in figure 1 into a class definition. Also create instances of the class.

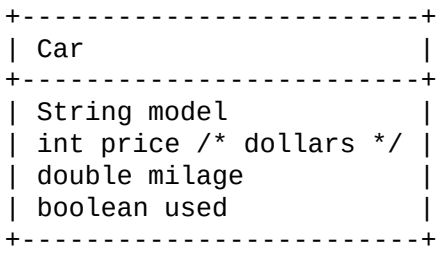


Figure 1: A class diagram for cars

Classes with containment

Note: The solution to the exercise 3.1.3 is in your handout. You may use it as a guide. Remember to include a class diagram and examples of instances for each class you design.

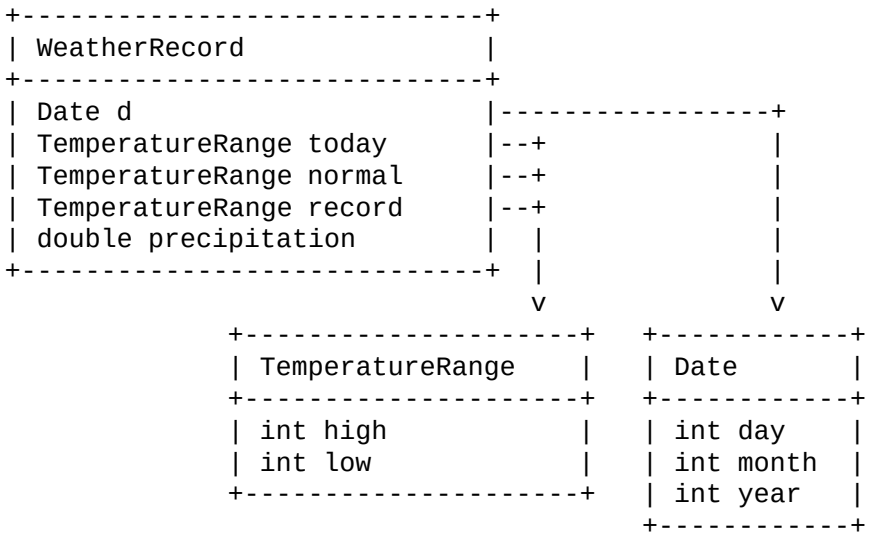


Figure 2: A class diagram for weather records

Exercise 3.1.1 Develop a data definition and Java classes for this problem:

> Develop a "real estate assistant" program. The "assistant" helps the real estate agent locate houses of interest for clients. The information about a house includes its kind, the number of rooms, its address, and the asking price. An address consists of a street number, a street name, and a city.

Represent the following examples using your classes:

1. Ranch, 7 rooms, $375,000, 23 Maple Street, Brookline;
2. Colonial, 9 rooms, $450,000, 5 Joye Road, Newton; and
3. Cape, 6 rooms, $235,000, 83 Winslow Road, Waltham.

Exercise 3.1.2 Take a look at the data definition in figure 2. Translate it into a collection of classes. Also create examples of weather record information and translate them into instances of the matching class.

Exercise 3.1.3 Revise the data representation for the book store assistant in exercise 2.1.1 so that the program keeps track of an author's year of birth in addition to its name. Modify class diagram, the class definition, and the examples.

Union

Exercise 4.1.1 Defining classes from the given information.
Consider a revision of the problem statement in exercise 2.1.2:

> Develop a program that creates a gallery from three different kinds of records: images (gif), texts (txt), and sounds (mp3). All have names for source files and sizes (number of bytes). Images also include information about the height, the width, and the quality of the image. Texts specify the number of lines needed for visual representation. Sounds include information about the playing time of the recording, given in seconds.

Develop a data definition and Java classes for representing these three records. Then represent the following examples with Java objects:

1. an image, stored in the file `flower.gif`; size: 57,234 bytes; width: 100 pixels; height: 50 pixels; quality: medium;

2. a text, stored in `welcome.txt`; size: 5,312 bytes; 830 lines;

3. a music piece, stored in `theme.mp3`; size: 40960 bytes, playing time 3 minutes and 20 seconds.

Exercise 4.1.2 Defining classes from the class diagrams.

Take a look at the data definition in figure 3. Translate it into a collection of classes. Also create instances of each class.

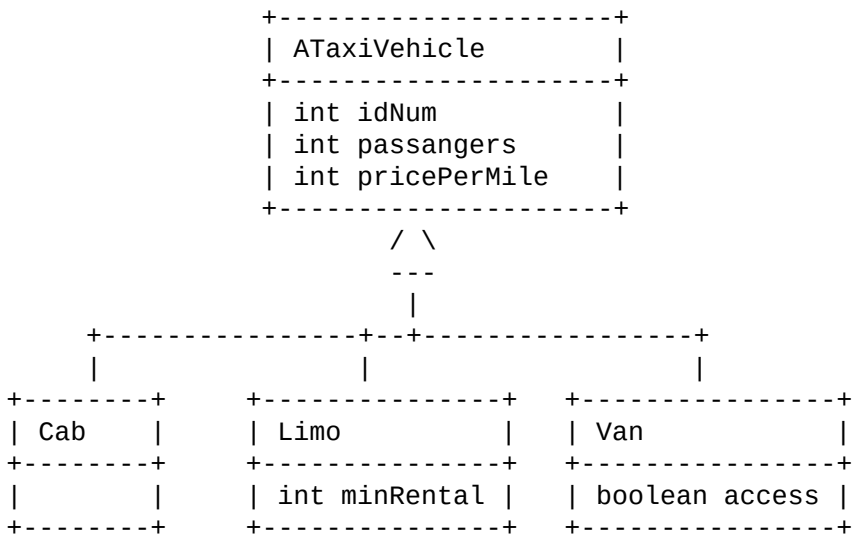


Figure 3: A class diagram for taxis

Exercise 4.1.3 Representing class hierarchies as class diagrams.

Draw a UML diagram for the classes in figure 4.

Exercise 4.1.4 You may want to think of other information that can be represented as simple classes, classes with containment, or union of classes. Write the problem description and ask you partner to design the classes. Or make the class diagram and ask your partner to make instances of these classes.

```
abstract class AMuseTicket {
Date d;
int price;
}
```

```
class MuseAdm
  extends AMuseTicket {
  MuseAdm(Date d,
          int price) {
    this.d = d;
    this.price = price;
  }
}
```

```
class OmniMax
  extends AMuseTicket {
  ClockTime t;
  String title;

  OmniMax(Date d,
          int price,
          ClockTime t,
          String title) {
    this.d = d;
    this.price = price;
    this.t = t;
    this.title = title;
  }
}
```

```
class LaserShow
  extends AMuseTicket {
  ClockTime t;
  String row;
  int seat;

  LaserShow(Date d,
            int price,
            ClockTime t,
            String row,
            int seat) {
    this.d = d;
    this.price = price;
    this.t = t;
    this.row = row;
    this.seat = seat;
  }
}
```

Figure 4: Some classes

3 Lab Tuesday am:

Designing classes that represent lists and trees

Containment in Union — Lists

Note: The solution to the exercise 5.1.2 is in your handout. You may use it as a guide. Remember to include a class diagram and examples of instances for each class you design.

Exercise 5.1.1 Consider a revision of the problem in exercise 3.1.1:

> Develop a program that assists real estate agents. The program deals with listings of available houses. ...

Make examples. Develop a data definitions for listings of houses. Implement the definition with Java classes. Translate the examples into Java instances.

Exercise 5.1.2 Consider a revision of the problem in exercise 2.1.1:

> Develop a program that assists a bookstore manager with reading lists. ...

The diagram in figure 5 represents the data definitions for classes that represent reading lists. Implement the definitions with classes. Create two book lists that contain at least one of the books in exercise 2.1.1 plus one or more of your favorite books.

Exercise 5.1.3 Take a look at figure 6, which contains the data definition for weather reports. A weather report is a sequence of weather records (see exercise 3.1.2). Translate the diagram into a collection of classes. Also represent two (made-up) two weather reports, one for your home town and one for your college town, in Java.

Containment in Union — Trees

Exercise 6.1.1 Consider the following problem:

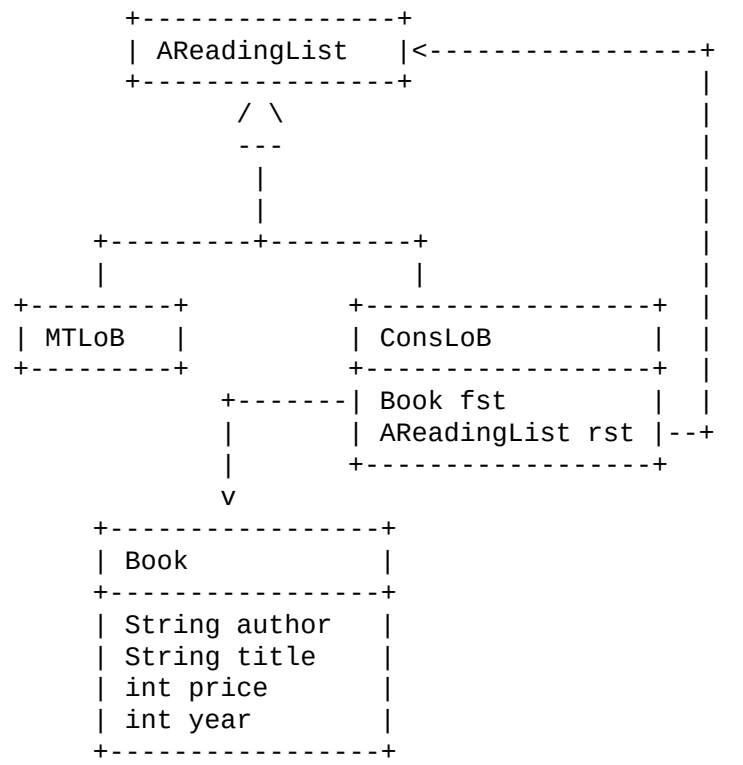


Figure 5: A class diagram for reading lists

> Develop program that helps with recording a person's ancestry tree. Specifically, for each person we wish to remember the person's name and year of birth, in addition to the ancestry on the father's and the mother's side, if it is available.

See Figure 7 for an example of the relevant information.

Develop the class diagram and the Java class hierarchy that represents the information in an ancestry tree. Then translate the sample tree into Java code. Also draw your family's ancestor tree as far as known and represent it as a Java object.

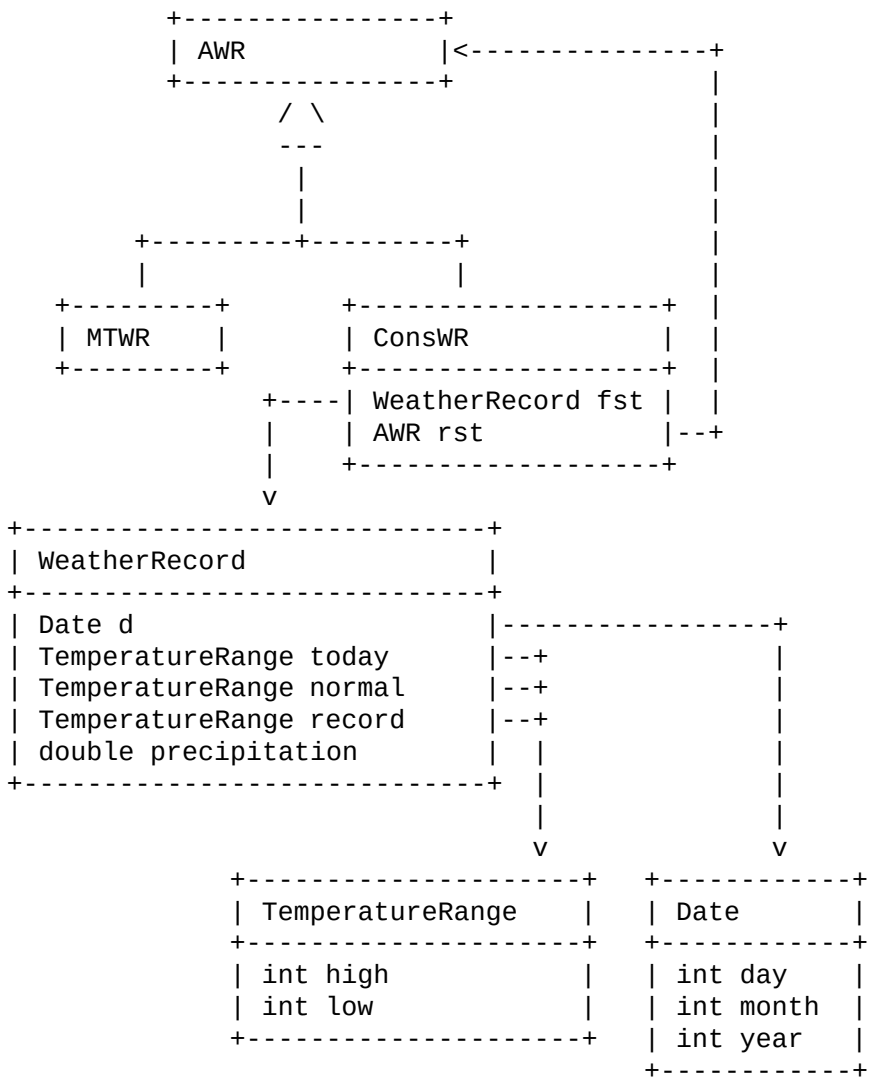


Figure 6: A class diagram for weather reports

Exercise 6.1.2 Take a look at the class diagram for a program that manages a phone tree (like those for a soccer team)(Figure 8). To inform the team about rainouts and schedule changes, the league calls the coach, who in

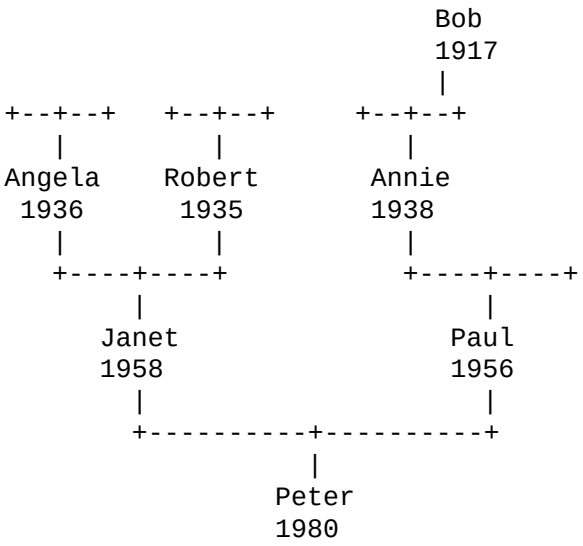


Figure 7: A family tree

turn calls the team captain. Each player then calls at most two other players.

Translate the following examples into pictures of phone trees:

```
Player coach = new Player("Bob", 5432345);
Player p1 = new Player("Jan", 5432356);
Player p2 = new Player("Kerry", 5435421);
Player p3 = new Player("Ryan", 5436571);
Player p4 = new Player("Erin", 5437762);
Player p5 = new Player("Pat", 5437789);
  APT empty = new MTTeam();
  APT pt =
      new PhoneTree(
          p2,
          new PhoneTree(p3, empty, empty),
          new PhoneTree(p4,
                        new PhoneTree(p5, empty, empty),
                        new PhoneTree(p1, empty, empty)));
Coach ch = new Coach(coach, pt);
```

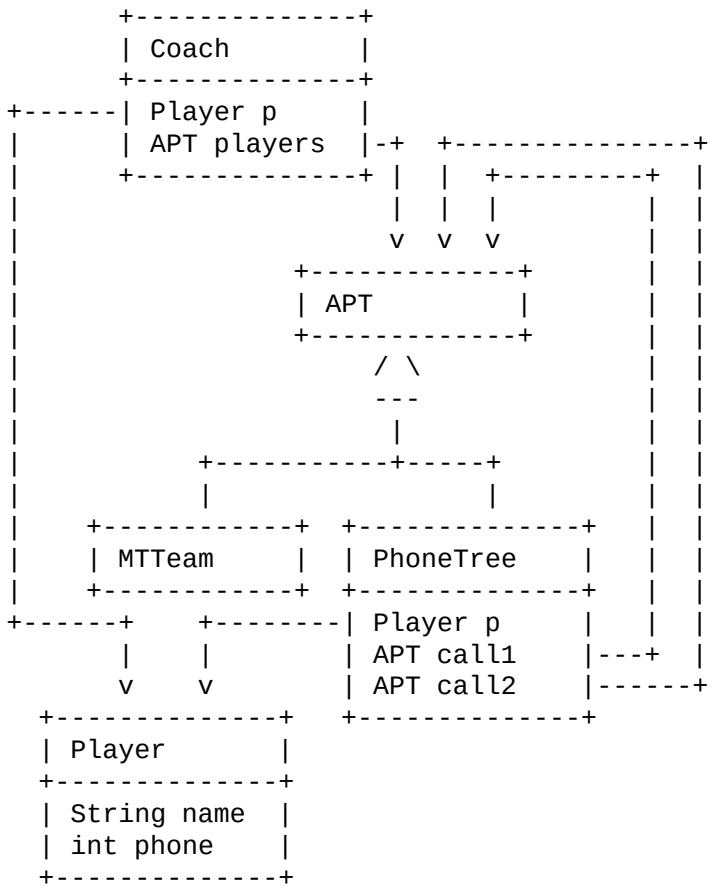


Figure 8: A class diagram for a phone tree

Now develop Java code that corresponds to the given data definition.

Exercise 6.1.3 Think of how you would design the classes that represent a grade book. Follow the design recipe to design Java classes that represent this information.

4 Lab Tuesday pm:

Design methods, build a test suite, introduction to templates

Methods for simple classes and classes with containment

Exercise 7.1.1 Remember the class *Image* from exercise 2.1.2 for creating Web pages. Develop the following methods for this class:

1. *isPortrait*, which determines whether the image is taller than wider;
2. *size*, which computes how many pixels the image contains;
3. *isLarger*, which determines whether one image contains more picture than some other image.

Exercise 7.1.2 Develop the following methods for the class *House* from exercise 3.1.1:

1. *isBigger*, which determines whether one house has more rooms than some other house;
2. *thisCity*, which checks whether the advertised house is in some given city (assume we give the method a city name);
3. *sameCity*, which determines whether one house is in the same city as some other house.

Don't forget to test these methods.

Exercise 7.1.3 Here is a revision of the problem of managing a runner's log (see figure 9):

> Develop a program that manages a runner's training log. Every day the runner enters one entry concerning the day's run. . . . For each entry, the program should compute how fast the runner ran. . . .

Develop a method that computes the pace for a daily entry.

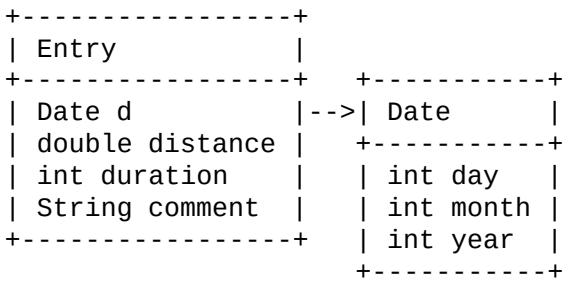


Figure 9: An entry for a runner's log

Methods for composition

Note: The classes for the exercise 8.1.1 are in your handouts. Add the methods to these classes.

Exercise 8.1.1 Recall the problem of writing a program that assists a book store manager (see exercise 3.1.3). Develop the following methods for this class:

- *currentBook* that checks whether the book was published in 2003 or 2002;
- *currentAuthor* that determines whether a book was written by a current author (born after 1940);
- *thisAuthor* that determines whether a book was written by the specified author;
- *sameAuthor* that determines whether one book was written by the same author as some other book;
- *sameGeneration* that determines whether two books were written by two authors born less than 10 year apart.

Exercise 8.1.2 Exercise 3.1.2 provides the data definition for a program that keep track of weather records. Develop the following methods:

1. *withinRange* that determines whether today's *high* and *low* temperatures were within the normal range;
2. *rainyDay* that determines whether the precipitation is higher than some given value;
3. *recordDay* that determines whether the temperature broke either the high or the low record;
4. *warmerThan* that determines whether one day was wormer than another day;
5. *lowerRecord* that determines whether the record low for one day was lower than for some other day.

5 Lab Wednesday am:

Methods for Unions

Exercise 9.1.1 Recall problem 4.1.1: that described a revision of the problem in exercise 2.1.2:

> **Problem** Develop class definitions for a collection of classes that represent several different kinds of files. All files have name and size given in bytes. Image files (gif) also include information about the height and width of the image, and the quality of the image. Text files (txt) include information about the number of characters, words, and lines. Sound files (mp3) include information about the playing time of the recording, given in seconds.

1. Develop the method *timeToDownload* that computes how long it takes to download a file at some network connection speed, typically given in bytes per second. The size of the file is given in bytes.
2. Develop the method *smallerThan* that determines whether the file is smaller than some maximum size that can be mailed as an attachment.
3. Develop the method *sameName* that determines whether the name of a file is the same as some specified name.

Exercise 9.1.2 Recall the exercise 4.1.2: Take a look at the data definition in figure 10. Translate it into a collection of classes. Also create instances of each class.

1. Develop the method *fare* that computes the fare in a given vehicle, based on the number of miles travelled, and using the following formulas for different vehicles.
   (a) passengers in a cab just pay flat fee per mile
   (b) passengers in a limo must pay at least the minimum rental fee, otherwise they pay by the mile
   (c) passengers in a van pay $1.00 extra for each passenger
2. Develop the method *lowerPrice* that determines whether the fare for a given number of miles is lower than some amount.

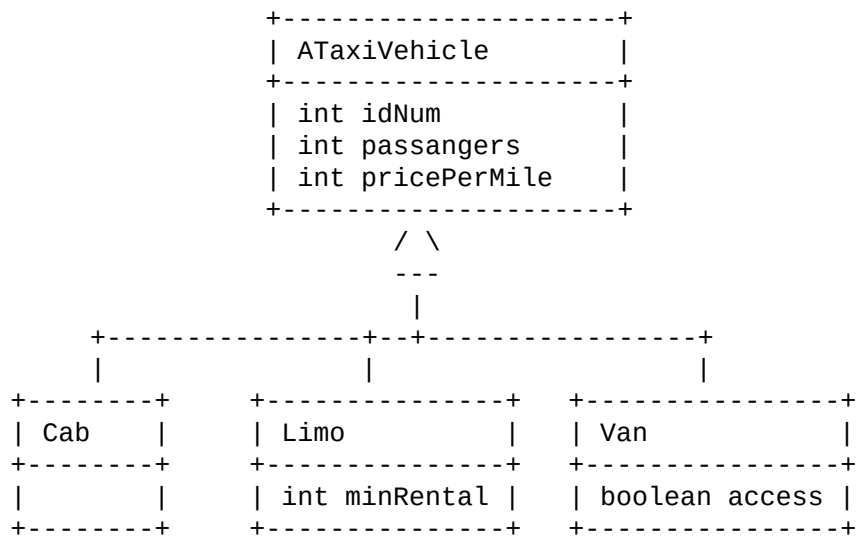


Figure 10: A class diagram for taxis

3. Develop the method *cheaperThan* that determines whether the fare in one vehicle is lower than the fare in another vehicle for the same number of miles.

Exercise 9.1.3 Consider this revision of the problem in exercise 2.1.1:

> **Problem** Develop a program that assists a bookstore manager in a discount bookstore. The program should keep a record for each book. The record must include its title, the author's name, its price, and its publication year. In addition, the books There are three kinds of books with different pricing policy. The hardcover books are sold at 20% off. The sale books are sold at 50% off. The paperbacks are sold at the list price.

1. Develop the class hierarchy to represent books in the discount bookstore.

2. Develop the method *salePrice* that computes the sale price of each book.

3. Develop the method *cheaperThan* that determines whether on book is cheaper than another book.

4. Develop the method *sameAuthor* that determines whether some book was written by the specified author.

Methods for lists

Exercise 10.1.1 Develop a program that assists a bookstore manager in a discount bookstore (see exercise 9.1.3).

1. Develop the class hierarchy to represent a list of books in the discount bookstore.

2. Describe in English examples of three book lists and represent them as objects in this class hierarchy.

3. develop the method *price* that computes the total for the sale, based on the sale price of each book.

4. Develop the method *thisAuthor* that produces a list of all books by this author in the bookstore list.

Exercise 10.1.2 Develop the following additional methods for the river system.

1. Develop the method *maxlength* that determines the length of the longest river segment.

2. Develop the method *confluences* that counts the number of confluences in the river system.

3. Develop the method *locations* that produces a list of all locations on this river - the sources, the mouths, and the confluences.

Methods for trees and similar structures

- Refer to the problem 6.1.1: develop methods to count the number of ancestors, to determine whether there is a person with some name in the tree, etc.

- Refer to the problem 6.1.2: Develop methods to find whether a given player is in the list, count the players, etc.

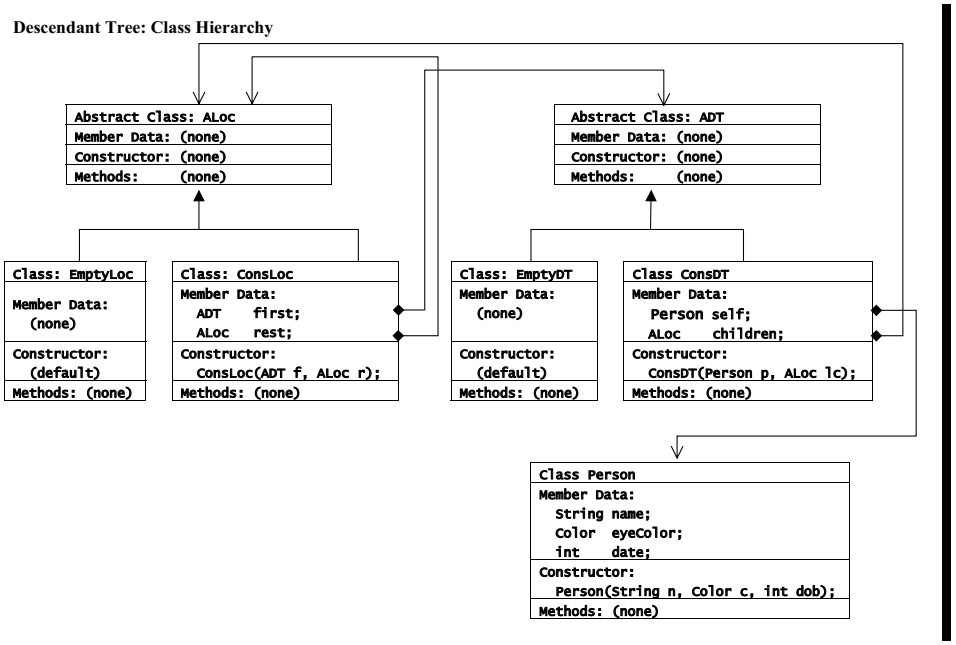
6 Lab Wednesday pm:

Simple drawing.

Designing methods for complex class hierarchies.

Exercise 11.1.1 Given a class diagram and the code for the data definitions, that represent the descendant tree, develop the following methods:

- *find*, which produces the *Person* object for a person with the given name, or *null*, if no person in the tree has this name.
- *children*, which produces a list of children for a *person* with the given name, or an empty list of children, if a *Person* with this name is not in the tree.
- *count*, which counts how many people are listed in the descendant tree



Exercise 11.1.2 A GUI *Component* is one of the following:

- Checkbox
- Textfield
- OptionsView
- ColorView
- Table

Each component contains a label and some additional data.
The data for a *Table* is one of

- empty
- list of *Row*s

A *Row* is one of

- empty
- list of *Component*s

Data for each of the remaining components is the default value to be displayed, specified as a String, and the preferred width and height.

- Draw the UML diagram for this collection of classes.
- Develop the templates for methods needed to count the number of primitive GUI elements in a given GUI *Component*.
- Develop the templates for the methods needed to determines the height of a given GUI *Component*. The height of the table is the sum of the heights of the *Row*s. The height of a *Row* is the maximum size of components in the list of *Component*s.

Exercise 11.1.3 A *WebPage* consists of a

- String *header*,
- and a *Body b*.

A *Body* is a list of *HTML* elements.
A *HTML* element is either

- a String *word*
- or a *Link*.

A *Link* consists of

- a String *word*
- and a *WebPage*.

- Draw the UML diagram for the collection of classes that represent web pages.
- Develop the classes.
- Develop the method *allWords*, which produces a list of all words in the web page
- Develop the method *pages*, which produces the list of *immediate* words on a page. That is, it consumes a *WebPage* and produces a list of String. An *immediate* word on a list of *HTML* elements is defined as follows:
  - an *HTML* element that is a *word* is the *immediate* word
  - for an *HTML* element that is a *Link*, the method extracts the *word* from the *Link*.
- Develop the method *occurs*, which determines whether the given *word* occurs in the web page or its embedded pages.

7 Lab Thursday am:

Design a simple game and implement it in the class that extends the `World` class.

- **Worm Game:** A worm (consisting of a head and a list of segments) moves in 'its' direction on each tick, unless the user selects s different direction through the key stroke. A food morsel appears at random in the play area. If the worm eats the food, it grows by a new segment. The game ends when a worm either runs into the wall, or it 'eats itself', i.e., the head attempts to move is such way that it would eat a part of itself.
- **UFOs:** An UFO is falling from the sky - moving slightly sideways as the wind blows. The user can move a gun platform left or right, and shoot a shot with the keystroke of letter 'x. Keep shooting, till you either hit the UFO, or the UFO lands on the earth. Add more shots, more UFOs, etc.
- **Ant Game:** An ant travels through the play area controlled by the arrow keys. As it moves, is looses weight from hunger. When it finds food (a number of food morsels appear at random in the play area), it grows bigger. The game ends when the ant is too small to live, or get too big to move. (You choose what is too small or too big). It can also end when the ant hits the wall.
- **Star Thalers:** Star Money, Star Thalers by the Grimm Brothers

  There was once upon a time a little girl whose father and mother were dead, and she was so poor that she no longer had a room to live in, or bed to sleep in, and at last she had nothing else but the clothes she was wearing and a little bit of bread in her hand which some charitable soul had given her. She was good and pious, however. And as she was thus forsaken by all the world, she went forth into the open country, trusting in the good God.

  Then a poor man met her, who said, "Ah, give me something to eat, I am so hungry."

  She handed him the whole of her piece of bread, and said, "May God bless you," and went onwards.

  Then came a child who moaned and said, "My head is so cold, give me something to cover it with."

So she took off her hood and gave it to him. And when she had walked a little farther, she met another child who had no jacket and was frozen with cold. Then she gave it her own, and a little farther on one begged for a frock, and she gave away that also.

At length she got into a forest and it had already become dark, and there came yet another child, and asked for a shirt, and the good little girl thought to herself, "It is a dark night and no one sees you, you can very well give your shirt away," and took it off, and gave away that also.

And as she so stood, and had not one single thing left, suddenly some stars from heaven fell down, and they were nothing else but hard smooth pieces of money, and although she had just given her shirt away, she had a new one which was of the very finest linen. Then she put the money into it, and was rich all the days of her life.

Exercise: stolen fair and square from TS!2 workshop
Develop a game program based on the story of "Star Money, Star Thalers."

The program should consume a natural number and drop that many thalers (from the top of the world) on the girl (at the bottom of the world), one at a time. The thaler should move randomly to the left or right and downwards, but should always stay within the boundaries of the world (canvas). The girl should react to 'left and 'right keystrokes, moving a moderate number of pixels in reaction but always staying completely within the boundaries of the world.

8 Lab Thursday pm:

Software provided:

- Classes that represent a list of books with author information, where author information includes the name and year of birth.
- Interfaces `IFilter` and `IFilter2` with examples of use.
- File **objlists.java**

Goals

- Learn to define and work with lists of `Objects`
- Learn to define interfaces and design classes that implement them
- Learn to use interfaces to design classes that encapsulate only behavior

Details

1. A list of `Objects`
   - Open the file objlists.java.
   - Draw by hand a class diagram that represents these classes.
   - Add two of your favorite books (and their authors) to the examples and to one of the lists in the examples and add tests that use your examples.
   - Make examples of lists of `Authors`.
   - Add the method `remove(Object obj)` that produces a list with only the first occurrence of the given object removed from the list.
2. Using an interface
   - `class Book` implements the `interface IFilter`. Study the code. Study the code that implements `orMap`. Add two more test cases to the test suite.
   - Modify the `class Author`, so it implements the `interface IFilter` to select authors born after 1945.
   - Add tests that verify the implementation of the methods `orMap` and `howMany` on your list of authors.

9 Lab Friday am:

Implementing interfaces - representing functions

1. Defining objects that encapsulate behavior.

   - Study the code for the `class CheapBook` that implements the IFilter2 interface, and its use in orMap2.
   - Design the method `andMap` that determines whether all items in the list satisfy the predicate encapsulated in an object that implements `IFilter2` interface.
   - Design the `class ContemporaryAuthor` that implements the `interface IFilter2` to select authors born after 1945.
   - Test your class in the context of `andMap` and `orMap`.

2. The power of abstraction.

   - Define interface `ITransform` that encapsulates a method with signature `Object transform(Object)`.
   - Design the methods `apply` for the list of `Object`s classes that produces a new list of `Objects`, applying the transform method to every object in the list.
   - Design the method `authortitle` that transforms a `Book` object into a `String` that contains the title and the author's name.
     Hint: given a `String s1` and `s2`, `s1.concat(s2)` produces a `String` that concatenates `String s1` and `String s2`.
   - Use the methods and classes you designed to produce a list of titles of all cheap books from a given list of books. (You will need one more class!)

10 Lab Friday pm:

Inner classes; Free time

Note: This is a preliminary version of this exercise.

> **Problem** University keeps records for all students and instructors at the university. For each person it records the name and id number. Each instructor also has a title, and is a member of some department (for now, we just know the name of the department). Each student has a major and grade point average (GPA).

Define the class hierarchy to represent students and instructors at the university, as well as a list of all people, all students, and all instructors.

Define the following methods for these classes:

- *sortStName* that sorts the students in a list of *Student*s by their name.
- *sortStGPA* that sorts the students in a list of *Student*s by their GPA.
- *sortInsName* that sorts the instructors in a list of *Instructor*s by their name.
- *sortInsDept* that sorts the instructors in a list of *Instructor*s by their department.

Work on refactoring this code as follows:

- Study the Java *Comparator* interface.
- In the class *Person* define a *Comparator* object that compares the persons by their name.
- In the class *Student* define a *Comparator* object that compares the students by their GPA.
- In the class *Instructor* define a *Comparator* object that compares the persons by their department.
- Refactor your code to use a list of *Object*s and rewrite the *sort* method so that it receives as a parameter an object in the class that implements *Comparator* interface.
- Add method to find whether an object is in the list.