Object Oriented Software Engineering: Practical software development using UML and Java

By Timothy C. Lethbridge and Robert Laganière
McGraw-Hill 2001
International ISBN 0-07-709761-0 US ISBN 0-07-283495-1

Public Answers to Selected Exercises – Version 1.9 (August, 2002)

The following contains answers to a **subset of the exercises** in the book, along with explanations of some aspects of the answers. In some cases the provided answers are more detailed those that would be expected from students. Also note that some questions are subjective, so readers may have responses different from those written here.

The answers provided here are publicly available on the website so as to enable students to practise. Generally speaking, the answers given here are parts a of multi-part questions; in exercises with many parts, answers may also be given to parts c and e. The full set of answers is only available to instructors as a separate password-protected document.

We update these answers from time to time, adding improved explanations, alternative answers, and extra exercises. You can contact us at tcl@site.uottawa.ca with suggestions. Items which we plan to complete or augment at a later date are marked in green.

Please look at our web site (www.lloseng.com) for other information about this book, including:
• A knowledge base containing many of the basic facts from the book
• Videos of Tim Lethbridge presenting lectures based on the book.
• The slides that have been prepared for professors who teach using the book

Chapter 1. Software and Software Engineering

E1.1 *p. 5 Classifying software.*
a)*Custom; real-time.
c)*Generic; real-time (but *soft* real-time).

E1.2 *p. 10 Stakeholder reactions to situations.*
a)*
- The **user** may be disappointed, since he or she might be looking forward to no longer having to do this particular type of work. On the other hand, he or she may be relieved about not being put out of work.
- The **customer** may be disappointed at not being able to save money; he or she may also be surprised, since many people believe that software systems are easy to develop; they underestimate the complexity of tasks that are to be automated. The customer might consult some other software engineer to obtain a second opinion.
- The **developers** will probably move on to other work.
- The development **managers** may be disappointed at not having the opportunity to do further work on the project.

E1.3 *p. 12 Prioritizing quality attributes*
a)***Reliability** will be paramount for the spacecraft software. It would be sad if the spacecraft did not make it into orbit after all that time, although no lives would be lost. **Efficiency** might be important since processors from 20 years earlier are far slower than today's devices. **Usability** will likely not be an issue since the software will run autonomously and report any feedback to experts; furthermore the software cannot be interactive since it takes considerable time to send signals to and from Pluto at the speed of light. **Maintainability** is also likely to be a minimal concern since this software is likely to only be used once. However, there remains the possibility that the software could be used on other systems, or will need to be changed as last-minute bugs are fixed.

c)***Maintainability** will likely be the most important concern since data processing software tends to evolve. **Reliability** will also be of considerable importance: Bill printing must be accurate, since it can be costly to rectify mistakes. **Usability** of the bills themselves will be important because ordinary people have to understand them. **Efficiency** should not be a concern.

Chapter 2. Review of object orientation

E2.1 *p. 32 Distinguishing among classes and instances*

The class names in the following answers could vary slightly. The points in parentheses are of lesser importance, and are for those who have read Chapter 5.

a)*Instance of **`AutomobileCompany`**. (or perhaps just **`Company`**)

c)*Class, subclass of **`Person`**. (In Chapters 5 and 6, we will see that it might be better to make this a subclass of **`PersonRole`**)

e)*Instance of **`Person`**.

E2.2 *p. 33 Detecting bad class names and improving them*

a)*Bad. Is this a particular vehicle (which might better be called something like **`Locomotive`**, or **`RollingStockConfiguration`**) or a scheduled run (**`RegularlyScheduledRun`**) that could use any vehicle, or the run on a particular day (**`SpecificRun`**)? We will discuss this kind of problem in more detail in Chapter 6 in the context of the Abstraction Occurrence pattern.

c)*Bad. The word 'Data' is inappropriate in a class name. Call it simply **`SleepingCar`**.

e)*Bad. It should not be plural: **`Route`** would be better.

E2.3 *p. 33 Naming different classes derived from words with multiple English meanings.*

a)***`Title`**: Describes published books independently of the number of copies **`CopyOfTitle`**: or perhaps **`LibraryItem`**: Represents physical books (as well as other things, such as videos) that the library places on its shelves.

c)***`RegularlyScheduledFlight`**: has a flight number, departure time, origin and destination and is flown every day; **`SpecificFlight`**: flies on a particular day.

E2.4 *pp. 34-35 Identifying attributes*

In this question and the next, there are many alternatives, a few possibilities are shown here. In particular sometimes certain attributes could be represented as associations instead – these are shown in parentheses.

a)***`name`**, **`description`**, (**`producer, distributer`**)

c)***`date`**, **`startTime`**, **`endTime`**, **`description`**, **`soundAlarmWhenStarting`**, (**`room`**)

e)***`callerNumber`**, **`calledPartyNumber`**, **`isConnected`**, **`startTime`**, **`currentCell`**, **`signalStrength`**, **`totalCost`**

E2.5 *p. 35 Identifying associations*

a)***`boughtFrom`**:**`ProductionCompany`**, **`producedBy`**:**`Producer`**, **`episodes`**:**`Episode`**, **`leadActors`**:**`Actor`**

c)* **`meetingRoom`**: **`MeetingRoom`**

e)***`caller`**:**`CallParty`**, **`called`**:**`CallParty`**

E2.6 *p. 35 Differentiating between variables and objects*

a)*Object

E2.7 *p. 36 Identifying operations*

In the following, operations that would probably be polymorphic are shown in italics, however, exactly which operations are polymorphic would depend on the design.

a)****`getArea`***, ***`getPerimeterLength`***, ***`getPoints`***, **`move`**, **`resize`**, **`rotate`**, **`flipHorizontally`**, **`flipVertically`**

E2.8 *p. 40 Identifying poor generalizations*

a)*Bad: You can't say 'A Canadian dollars is a money'; also **`CanadianDollars`** should be an instance of **`Currency`**

c)*Probably OK.

e)*Bad: **`Account12876`** would be instance of **`BankAccount`** (or some subclass of **`BankAccount`**)

E2.9 *No public answer*

E2.10 *pp. 41-42 Arranging potential classes into inheritance hierarchies (See also E5.21, p. 189)*
In the following, the hierarchies are shown using indented text to save space. Whether it would be appropriate to actually include all the classes in a given system would depend on the application's requirements. There are many possible variations on these answers. Note: See also E5.21 for additional exercises based on these problems.

a)*In this problem we create two separate hierarchies, **`Vehicle`** and **`PartOfVehicle`**. These could also have something like **`Machine`** as a common superclass, although the problem suggested creating separate hierarchies.

- **`Vehicle`**
  - **`LandVehicle`** (Added)
    - **`Car`**
      - **`SportsCar`**
    - **`Truck`**
    - **`Bicycle`**
  - **`AirVehicle`** (Added)
    - **`Aeroplane`**
  - **`AmphibiousVehicle`**
- **`PartOfVehicle`**
  - **`Engine`**
    - **`JetEngine`**
    - **`ElectricMotor`**
  - **`Wheel`**
  - **`Transmission`**

`Vehicle` could instead be divided into **`PoweredVehicle`** and **`UnpoweredVehicle`**; multiple inheritance could be then used for superclasses of the vehicle leaf classes.

c)***`Schedule`** (We will learn later that this may not be needed since the whole system stores the schedule)

- **`RegularlyScheduledTrip`** (Added as a superclass representing something that runs at a given time)
  - **`RegularlyScheduledExpressBus`** (renamed for clarity)
- **`BusRoute`**
- **`ActualTrip`** (Runs at a given time on a given day; could also be called 'Run')
  - **`CharteredTrip`** (Renamed for clarity)
  - **`ScheduledTrip`** (Added to properly complement **`unscheduledTrip`**)
  - **`UnscheduledTrip`** (Runs on a route, but not at the normal time)
- **`BusVehicle`** (Renamed from 'bus' to distinguish different types of bus)
  - **`LuxuryBus`**
  - **`TourBus`**

e)***`Currency`** (Canadian Dollars and US Dollars are Instances)

- **`ExchangeRate`** (Attributes or associations could be **`fromCurrency`**, **`toCurrency`**, rate)
- **`FinancialInstitution`** (Added)
  - **`Bank`**
  - **`CreditUnion`**
  - **`CreditCardCompany`** (Visa and MasterCard are instances)
- **`FinancialInstrument`**
  - **`ReusableFinancialInstrument`** (Added)
    - **`CreditCard`**
    - **`DebitCard`**
  - **`SingleTransactionInstrument`** (Added)
    - **`Cheque`**
- **`BankAccount`**
  - **`Loan`**
- **`ElectronicDevice`** (Added)
  - **`AutomaticTellerMachine`** (better than **`BankMachine`**)
- **`BankBranch`**

E2.11 *p. 47 Describing how polymorphic implementations of certain shape operations would work.*

a)***translate**: In **SimplePolygon** (as inherited by **Rectangle** and **RegularPolygon**), and **EllipticalShape** (as inherited by **Circle**), the method **translate** would move the **center**.

c)***getArea**: In **Rectangle**, the method **getArea** would return **height * width**. In **RegularPolygon**, it would compute the area by dividing the polygon into **numPoints** individual triangles (see the answer to E2.30 for the detailed algorithm). Once this area is calculated, the total area of the **RegularPolygon** would be calculated by multiplying the area of each triangle by **numPoints**. In **Circle**, the method **getArea** would return pi * $\textbf{radius}^2$.

E2.12 *No public answer*

E2.13 *p. 47 Incorporating new classes into an existing class hierarchy that contains considerable polymorphism.*

a)***IsoscelesTriangle**: One might think of making this a subclass of **ArbitraryPolygon**, however that would be inappropriate since you don't want it to inherit methods such as **addPoint** and **removePoint**. A better solution is to make it a subclass of **SimplePolygon**. As attributes you would have to store the **baseLength** and **height**, or else you could store one of the two values for angles and the length of one of the sides, letting the other angle value and side be computed when needed. As methods, you would need **changeScale**, **setBaseLength**, **setHeight**, **getArea**, **getPerimeterLength**, **getVertices**, **getBoundingRect**, **getBaseAngle**, **getTopAngle**, and perhaps **setBaseAngle** and **setTopAngle**.

E2.14 *No public answer*

E2.15 *No public answer*

E2.16 *No public answer*

E2.17 *p. 49 Determining when dynamic binding is needed in a set of polymorphic methods.*

This exercise has turned out to be particularly useful to ensure students really understand the implications of polymorphism. Before assigning this exercise, it has proved to be necessary to explain the material on pages 48 and 49 particularly carefully, with several examples. Note that the exercise has the assumption, "that the compiler knows that no new classes or methods can be added to the hierarchy"; it is worth reminding students that this is not generally true in Java (you can add a subclass unless the class is declared **final**, and you cannot declare a non-leaf class to be **final**).

a)*Invoking **getPerimeterLength** on a **Rectangle** variable: No dynamic binding is needed since **Rectangle** is a leaf class and so the variable could only ever contain an instance of that class. The local method in **Rectangle** would always be called.

c)*Invoking **getBoundingRect** on a **Polygon** variable: Dynamic binding would be needed, since either the method in **Polygon** or the one in **Rectangle** might have to be executed depending on which class of object is placed in the variable at run time.

E2.18 *p. 51 Researching products that claim to be object-oriented to determine if they really are.*

The answer to this question will vary over time, depending on what products are available.

E2.19 *p. 53 Using documentation to look up library classes and thus better understand a program.*

This is a purely practical exercise. Its purpose is to encourage students to get in the habit of using documentation.

E2.20 *No public answer*

E2.21 *No public answer*

E2.22 *No public answer*

E2.23 *No public answer*

E2.24 *No public answer*

E2.25 *No public answer*

E2.26 *No public answer*

E2.27 *No public answer*

E2.28 *No public answer*

E2.29 *No public answer*.

E2.30 *No public answer*

E2.31 *No public answer*

Chapter 3. Basing software development on reusable technology

E3.1 *p. 63 Researching resources on the Internet that discuss aspects of reuse.*
The answer to this question will vary since web pages are continually being added, deleted and changed.

E3.2 *p. 63 Analysing information about reuse.*
For discussion only.

E3.3 *pp. 69-70 Determining the services that should be present in a framework.*
a)***Reservation framework**.
- Add an instance of whatever that can be reserved (e.g. a book, a place on a flight, a seat in a theatre)
- Add an instance of whatever the reservation is made on behalf of (e.g. a library patron, a passenger, a theatre-goer).
- Make a reservation
- Delete a reservation

E3.4 *p. 70 Determining differentiating features of framework-based applications, as well as its hooks and slots.*
a)***Reservation framework**.
- Differentiating features:
  - The classes of objects that can be reserved and their attributes, associations and other operations
  - The attributes of the reservation itself, and perhaps subclasses representing different types of reservation
  - The classes of objects that a reservation can be made on behalf of
  - The user interface
  - Rules regarding the reservation, such as who can make one, whether the item can be reserved, etc.
- Hooks:
  - A function that would be called when a reservation is complete (e.g. to send an email)
  - An function to call when reservations are 'full' that could be used, for example, to add a waiting list, or to give some form of notification to the user.
  - A function called to load a reservation from a database.
  - A function called to save a reservation.
- Slots:
  - There may be no slots in this system.

E3.5 *p. 70 Determining the range of applications that might benefit from a framework.*
a)***Reservation framework**.
- A library system, where you can reserve items that are already checked out
- Reservation of seats on any kind of transportation system
- Reservation of entertainment tickets

E3.6 *p. 70 Evaluating alternative approaches to designing a framework.*
a)*If you started with a vertical framework for a frequent flier program, you would have a lot of facilities already developed, some of which would be quite specific to the frequent flier domain. The following gives some ideas of the services, slots and hooks that might be provided; many variations on this answer are possible.

i. **Services**:
- Maintenance of frequent flier accounts to which points (miles) can be added and redeemed (i.e. adding new accounts; deleting accounts)
- Keeping of basic personal information (name, address, points) about each frequent flier, with methods to update this information (which would call the first hook below to allow managing of additional information)
- Keeping of a log of flights to be used when producing reports (would also be capable of recording other transactions, e.g. points awarded for renting a car).
- Methods to add, delete and query the number of points in the account.

- Generic mechanism, given two cities, that would calculate how many points would be required to fly between them for free for a given user. This would be called when customers are doing queries, and also when a person actually books a free flight to determine how many points to deduct. This would call the first slot below to do detailed calculations.
- Generic mechanism, given two cities, that would calculate how many points to credit to a particular frequent flier who pays for a ticket. It would be called when the user is making queries, and also when a person actually takes a flight, in order to credit points. This would call the second slot below for detailed calculations.
- Generic mechanism that runs every month, calling the slot (below) for producing reports, and the hooks (below) for deleting inactive accounts, and perhaps promoting users to different classes.

ii. **Slots** (code that must be provided)

- A method to evaluate rules regarding how many points are required to fly certain routes for free.
- A method to evaluate rules regarding how many points are obtained from flying certain routes (might be specific to certain classes of frequent flier, certain times, etc.)
- A method to produce a formatted printout of a frequent flier's account (each airline may want to make these appear stylistically different from other airlines). Note that instead of creating a slot, this functionality could be left out of the framework entirely.
- Note that the user interface would have to be provided, but would probably not actually appear as slots in the basic system; the user interface would be a separate subsystem that would simply call the services.

iii. **Hooks** (places where optional add-ons can easily be plugged in)

- Ability to manage additional types of information about frequent fliers (e.g. work address, email,), beyond the basic minimum information. Different airlines might want to keep different kinds of information for use in marketing etc.
- Ability to have different classes of users (e.g. prestige users who have accumulated large numbers of points)
- Mechanism to delete accounts after a certain time period with no activity (this might be activated by the monthly run that prints reports)
- Ability to add different kinds of points to be used for different purposes (e.g. some airlines, in addition to miles, also keep track of a separate count of miles flown in first class; such points might have different rules, such as expiring after the end of a year).

E3.7 *p. 70 Evaluating whether or not one should first develop a framework when designing an application.*

a)* Arguments **in favour** of developing a full frequent flier framework:

- Developing the framework improves your overall design. Since you would have to build flexibility into the framework to allow it to be used by others, you would benefit from that flexibility when you have to make changes yourself.
- As a consequence of the above, you would expect maintenance costs to be reduced; you would also expect to be able to respond faster to market-driven requirements changes.
- You might be able to sell your framework to others, or to provide frequent-flier services to other smaller airlines.
- Should two airlines using the same framework decide to work together in an alliance (or to merge) their systems would be more compatible with each other, reducing costs.

E3.8 *p.74 Evaluating whether an application should be designed as a client-server system or not.*

a)***Word processors** are not normally designed using a client-server architecture. The following are some of the reasons why not:

- People typically use word processors on laptop computers or other machines that may not be connected to any network.
- Although people do share word-processed data, most people work on their own local copy and then share it when they are finished.

It might be beneficial to allow a word processor to optionally connect to a server to do such things as load and save data, or to enable two people to edit the same document concurrently. However it would still be necessary to ensure that the user could operate the program while not connected.

E3.9 *p.74 Determining the services to be provided by a server.*

a)*An **airline reservation server** would be expected to do some of the following activities:

- Maintain the list of flights along with basic details of each flight (and allow changes)
- Maintain the prices (and allow changes, special offers etc.)
- Maintain the bookings (allow bookings to be made, changed and deleted)
- Allow people to query all of the above

E3.10 *p. 79 Evaluating the balance of work on the client and server side of a system.*

a)***For an airline reservation system**:

i. Server work: See exercise E3.9.

ii. Client work:
- Enter queries about available flights and display the results.
- Enter bookings
- Display and print itineraries

iii. Information transmitted from server to client:
- Results of queries (relatively simple text)

iv: Information transmitted from client to server:
- Simple commands and their arguments.

v: How the client's work could be minimized or maximized:
- The client might be interacting with several different reservation systems, producing overall itineraries that would combine the results from several servers. A server could be developed to provide this function – interacting with other servers and managing itineraries.
- The client could, on the other hand, be given a lot of 'intelligence' to work out itineraries and 'good deals'. It could find the best prices, and even run a bidding mechanism that would try to get airline servers to give better deals.
- The client could simply be a web browser with no intelligence at all about the frequent flier domain.

vi: Network effects of changing the client's workload:
- Putting more intelligence into the server (such as having the server interact with other servers) would have only a moderate effect on the quantity of information transmitted in the network as a whole.
- Adding extra intelligence to the client might result in large numbers of queries being transmitted, and large numbers of replies returned. The client would need to perform these queries in order to marshal the information it needs to make decisions.
- Having the client become nothing more than a web browser, would probably increase network traffic from the server to the client, since now the server would have to transmit images and all the other information that will make the UI of the client look nice.

E3.11 *p. 80 Writing protocols for a client-server system.*

There are clearly many possible protocols for these systems; the level of complexity could become substantial. the following shows some basic sets of messages that could be exchanged.

a)***Airline reservation system**. Note that a real system can be extremely complex; the following represents a simplified view of the kinds of messages that such a system would need to exchange.

Messages to server	**Possible replies to client**
• `addRegularFlight` *number planeType effectiveDate*	
	`confirmFlightAdded` number
	`flightNumberAlreadyExists`
• `addFlightLeg` *flightNumber origin leaveTime destination arriveTime*	
	`confirmLegAdded`
	`flightTimeConflictsWithAnotherLeg`
• `deleteRegularFlight` *number effectiveDate*	
	`confirmDeletion`
	`flightNumberDoesNotExist`

- **addFareClass** *fareCode*
  - **confirmFareClass**
  - **fareClassAlreadyPresent**
- **setFare** *origin destination classCode startDate endDate*
  - **confirmFare**
- **addBooking** *name date flight originCity destinationCity fareClass*
  - **confirmBooking** *confirmationNumber seat*
  - **bookingCannotBeMade** *reason*
- **deleteBooking** *confirmationNumber*
  - **confirmBookingDeletion**
  - **bookingNotCancelled** *reason*
  - **confirmationNumberInvalid**
- **queryFlightAvailability** *date origin destination fareClass earliestDepTime latestArrTime*
  - **availableFlight** date flightNumber fareClass
  - **noMatchingAvailableFlight**
- **queryBookings** *name date*
  - **confirmBooking** *confirmationNumber seat*
- **queryBookingsByNumber** *confirmationNumber*
  - **bookingInfo** *flightNumber date name origin destination fareClass seat*
- **queryPassengers** *flightNumber date*
  - **bookingInfo** *flightNumber date name origin destination fareClass seat*

Chapter 4. Developing requirements

E4.1 *p. 106 Listing information to be consulted when performing a domain analysis.*

a)***Police information system**
- Police officers and staff (interviews, brainstorming and observing work)
- Police training and procedures manuals
- Documentation of existing software and equipment used by the police (including competing products)
- Books about police methods
- Similar information about related activities such as court procedures, etc.

E4.2 *p. 106 Writing a short domain analysis for a system.*

We will not be providing any answers to this exercise since the answers will depend too much on the information consulted, the country in which the reader resides, and the date the information is gathered.

E4.3 *p.110 Defining and narrowing the scope of a system.*

In this following answers, a reasonably complete set of possible system functions is listed. The functions in bold are the ones that represent the minimal requirements of a first release. Of course, each person's answer will be somewhat different; however, the sample answers suggest the scope of good answers to this question. (See E9.1 for additional exercises based on these systems)

a)*Police information system
- Managing personnel information about officers and staff (note that a *generic* personnel management system could also be used to do this, although it would have to be hooked to the police system to enable some of the other functions mentioned below).
- **Managing basic information about areas and events to be patrolled**
- **Facilitating the manual scheduling of the duties of officers and staff**
- Automatically proposing schedules for some of the duties of officers and staff
- **Managing basic information about each case being investigated**
- **Managing the documents that have to be written regarding each case, and following each shift**
- Generating statistics and trend information about crime rates etc.
- Managing detailed information about suspects, witnesses and other people relevant to cases
- Providing links to drivers licenses and other databases to help police find information
- Providing links to court and criminal records

E4.4 *p. 110 Giving precise problem statements, considering high-level goals.*

a)*The new **police information system** will make operations of the police department more efficient by reducing the time and effort required to plan and execute work, as well the time required to enter and find information required in routine police work.

E4.5 *p. 115 Describing functional requirements of a system.*

These build on the answers to E4.3 and E4.4 above.

a)*Police information system

To be provided in a later version of this document

E4.6 *p. 118 Classifying requirements into functional or non-functional.*

a)*F. Describes the computation to be performed (some security requirements are non-functional, but this one seems to be in the functional category)

c)*NF. Constrains response time.

E4.7 *p. 118 Rewriting requirements so they are verifiable.*

In each part of this questions, we would expect an answer similar to one of the bulleted points. Some of the bulleted points show that there should be two or more separate requirements.

a)*Constraint on technology to be used:
- The C++ programming language must be used
- A strongly typed, block-structured programming language must be used that does not allow unrestricted access to memory.

c)*Constraint on availability and throughput

- - The system will accept connections 24 hours a day, every day, with a maximum downtime of 5 minutes a month, which can occur only between 12:30 a.m. and 1:30 a.m. North American Eastern Time.
  - When available, the system will at all times be able to manage 200 simultaneous connections, and process 2000 transactions per minute.

E4.8 *No public answer*.

E4.9 *p. 119 Writing non-functional requirements.*
These build on the answers to E4.3 - E4.5 above.
a)*Police information system

- When the user is connected to the over a local area network, the system shall respond with complete responses to all inputs within 0.5s at peak system load.
- When the user is connected over a wireless connection at 9600bps, the system shall respond with complete responses to all inputs within 3s at peak system load.
- The system shall be capable of handling 50 simultaneously connected users, and 300 user interactions (events originating from users) per minute.
- The system shall not consume more than 200MB of RAM while running at peak capacity.
- The system shall not consume more than 56Kbps of bandwidth per connected user (averaged over any 1s time period)
- The system shall achieve a reliability level of no more than 1 failure per 10 user hours, with none of those failures rendering the system totally inoperable.
- The system shall be available 99.9% of the time with no period of down time exceeding 2 minutes.
- The system shall be designed so that large numbers of additional functions can be added with little reprogramming.
- The system shall be designed so that it could be adapted to the needs of other police departments with minimal reprogramming.
- The server shall be able to run on a Windows 2000 (or higher) equipped system, with a 1.4 GHz or faster Intel processor, 300MB or more of RAM, and 15GB or more of disk space.
- The client software shall be able to run on EMP-4500 police car terminals, as well as on EMP-4500 emulators running on desktop computers.

E4.10 *p. 122 Practising interview techniques.*
This is a purely practical exercise.

E4.11 *p. 124 Practising brainstorming techniques.*
This is a purely practical exercise.

E4.12 *No public answer*

E4.13 *No public answer*

E4.14 *No public answer*

E4.15 *pp. 129-130 Describing the type of requirements document needed.*
a)***Software controlling a manned spacecraft sent to Mars**

- There would need to be a considerable amount of detailed technical requirements documentation. Technical terminology could be used since all readers would be expected to have a technical background. There would probably initially be a very high-level description of the mission's overall objectives used to obtain funding for the mission. Then there would be a requirements document for the high-level system requirements of the combined hardware-software system. Next, the system would be allocated into hardware and software subsystems, which in turn would each be divided into several subsystems. At each level there would probably be general and detailed requirements documents. The rationale for this level of detail is that a spacecraft is novel, is technically complex, will involve many different people with different types of expertise to develop, is difficult or impossible to change once launched, and is expensive (with high failure cost and hence a high level of required reliability).

E4.16 *pp. 135-136 Finding problems in requirements statements.*

a)***Simple interest calculation program**

- The requirements are not separated into individual points
- 'This is a handy utility' is too informal
- It is not clear whether the fields are updated upon every keystroke, or whether the user has to hit 'enter' or click on some button to cause an update of the fields.
- It is not clear whether the monthly payments are at the start or end of each month (this effects the computation).
- (The computations themselves are well-known by accountants, so it is legitimate to exclude them)
- There is no information about the platform on which the program runs,
- Maximum and minimum dollar amounts and interest rates that can be handled should be specified.
- Precision of calculations should be specified.
- What happens if the user enters invalid keystrokes?
- It would be desirable to describe the user interface in a little more detail.

E4.17 *p. 136 Rewriting requirements to remove defects.*

a)***Simple interest calculation program**

- Functional Requirements
  - When the system starts, a window appears, whose content and function is describe below.
  - The window has the title 'Simple Interest Calculation', as well as standard close and minimize icons in the title bar.
  - The window has labels 'Principal:', 'Annual interest rate:', 'Interest paid at the end of each month:'. The right edges of these are vertically aligned, and each is followed on the right by an input field
  - The principal and interest fields are capable of accepting 10 characters of input, the interest rate field is capable of containing 8 characters.
  - The user may enter numeric values in each field; as each key is typed (including the delete key), the other fields are updated instantly. An empty field is taken as having the value zero.
  - Editing the interest rate causes a change to the monthly payment.
  - Editing the monthly payment causes a change to the interest rate
  - Editing the principal causes a change to the monthly payment
  - The user may use the mouse to move the insertion point or select characters in any field, as in standard Windows applications.
  - The system can handle dollar amounts from -999999.99 to 9999999.99, and interest rates from -999.99% to 999.99%. The percent symbol is automatically added if the user does not specify it.
  - All calculations are rounded to the nearest cent. Two decimal places are always shown in all three places.
  - Any edit to a field that would result in an invalid input, or a computed result that could not be displayed in the other fields, will not be accepted – the system will beep and display an error message at the bottom of the window describing the error. Error messages include:

    The largest value accepted for <field> is xxx
    The smallest value accepted for <field> is xxx
    Input must be numeric
    Input is only accepted to two decimal points
    A <field> input value of xxx would result in an invalid value for <field>
- Non-functional Requirements
  - The program will run on all Microsoft Windows operating systems starting with Windows 95 and Windows NT 4.0 (on all CPUs of at least 33Mhz).
  - Response time will appear instantaneous to users.

E4.18 *No public answer*

E4.19 *No public answer*

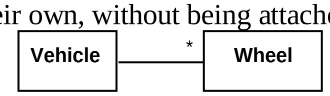
E4.20 *No public answer*

Chapter 5. Modelling with classes

E5.1 *No public answer*

E5.2 *p. 160 Designing associations among classes*
Note that the exercise says 'take care to specify … labels for the associations'; however, we have only shown labels where the default 'has' would not suffice. In many of the following, differences in class names are possible.

a)*We will see later on that aggregation could be used here; however, at this point in the book the following would be a reasonable answer. Making the multiplicity on the **`Vehicle`** end 0..1 might be even better, allowing for wheels to exist on their own, without being attached to vehicles.



E5.3 *p. 160 Explaining the consequences of associations in terms of creation and destruction of instances*
This exercise has several parts that are strongly analogous to each other, and which turn out to be candidates for association classes, which are described on page 161.

a)*A **`Vehicle`** could exist, and then a **`Wheel`** could be created and associated with it; conversely, the wheels could exist first. Similarly, the order in which these associated objects are destroyed does not matter.

c)*Under normal circumstances in the real world, a country exists over a longer period of time than any of its heads of state. However, if we imagine an information system storing biographical information, then a person might be created in the system who later on is recorded as the head of state of a country. So there are no necessary constraints on the order of creation or destruction of these instances.

e)*This is similar to parts b and d. The **`Registration`** would be created after the **`Member`** and the **`Activity`**, and would have to be destroyed no later than these associated instances. (In some circumstances, the **`Member`** and the **`Registration`** could be created together).

E5.4 *p. 160 Reading associations in both directions.*
The exact phrasing of answers to this question will, of course, vary considerably. This exercise has proven particularly valuable at getting students to check their work and detect errors in their class diagrams. Creating these answers even helped us make improvements to our answers to E5.2. Different answers to E5.2 would result in different answers to this question.

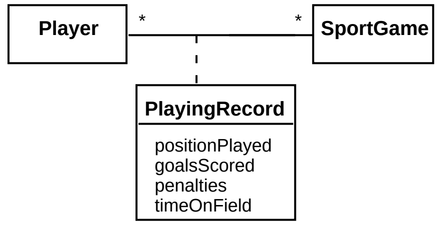
a)*
- A **`Vehicle`** can have no **`Wheel`**s, or an unspecified number of **`Wheel`**s.
- A **`Wheel`** always belongs to exactly one **`Vehicle`**.

c)*
- A **`Country`** has either zero or one known **`HeadOfState`**.
- A **`HeadOfState`** can head any number of **`Country`**s

e)*
- A **`Member`** can register in any number of activities (can have any number of **`Registration`**s)
- Each **`Registration`** involves registering exactly one **`Member`** in exactly one **`Activity`**
- An **`Activity`** can have an arbitrary number of **`Registration`**s

E5.5 *p. 162 Adding association classes.*
The question doesn't require a list of attributes, but we show attributes below to help explain the purpose of the association class.

a)*A reasonable association class might be called **`PlayingRecord`** or something similar. The attributes needed would depend on the type of sport, but might include **`positionPlayed`**, **`goalsScored`**, **`penalties`** and **`timeOnField`**



E5.6 *No public answer*

E5.7 *No public answer*

E5.8 *No public answer*

E5.9 *No public answer*

E5.10 *No public answer*

E5.11 *No public answer*

E5.12 *p. 175-176 Writing OCL constraints.*

a)*Referring to Figure 5.11, p. 164

```
context  RecordingCategory inv:
subcategory->forAll(r |
  self <> r)
```

E5.13 *p. 178 Designing how operations would be performed*

a)*In this example it is ambiguous whether **getSiblings** should include half siblings or just full siblings; we assume the former.

getSiblings (a method of **Person**) would be performed as follows:

1 Obtain the **parents** (a **Union**), by following the association
2. For the **femalePartner** of the **Union**:
   2.1 For each of her **Union**s
      2.1.1 Put the **child**ren of the **Union** in the result set (excluding duplicates and the self object)
3. Repeat step 2 for the **malePartner**
4. Return the result set

c)*Step siblings are children of a person's step-parent which are not half-siblings. To solve this problem we will first define a method of **Person** called **getStepParents**, as follows:

1. Obtain the parents (a **Union** we will call **mainUnion**).
2. For the **femalePartner** of the **mainUnion**:
   2.1 For each of her **Union**s other than the **mainUnion**
      2.1.1 Put the **malePartner** in the result set
3. Repeat step 2 inverting **femalePartner** and **malePartner**
4. Return the result set

Now we can define the **getStepSiblings** method of **Person**:

1. For each **stepParent** obtained by calling **getStepParents**
   1.1 Add each of the **child**ren of the **stepParent** to the result set
2. Call **getSiblings** to obtain the set of **siblings**
3. Subtract the **siblings** and self from the result set
4. Return the result set

E5.14 *p. 178 Writing OCL constraints for the genealogical example.*

a)*

```
context  Person inv:
dateOfDeath = ''
or dateOfBirth = ''
or dateOfBirth <= dateOfDeath
```

E5.15 *No public answer*

E5.16 *No public answer*

E5.17 *No public answer*

E5.18 *p. 184 Making a preliminary list of classes by listing noun phrases in requirements.*

In the following, we simply list a reasonable initial set of classes based on a first scan of the nouns in the problem descriptions. Other lists are possible. Not all the nouns shown will end up in the final answers, shown in the answer to E5.20. Classes where there is some initial doubt are marked '??'.

a)*Bank account management system (p. 448)

- **`Service`** ??
- **`BankAccount`**
- **`Branch`**
- **`Client`**
- **`MortgageAccount`**, **`ChequingAccount`**, **`CreditCardAccount`**
- **`Property`** ??
- **`CreditCard`**
- **`JointAccount`** ??
- **`Privilege`** ??
- **`Division`**, **`SubDivision`** ??
- **`Manager`**, **`Employee`** ??

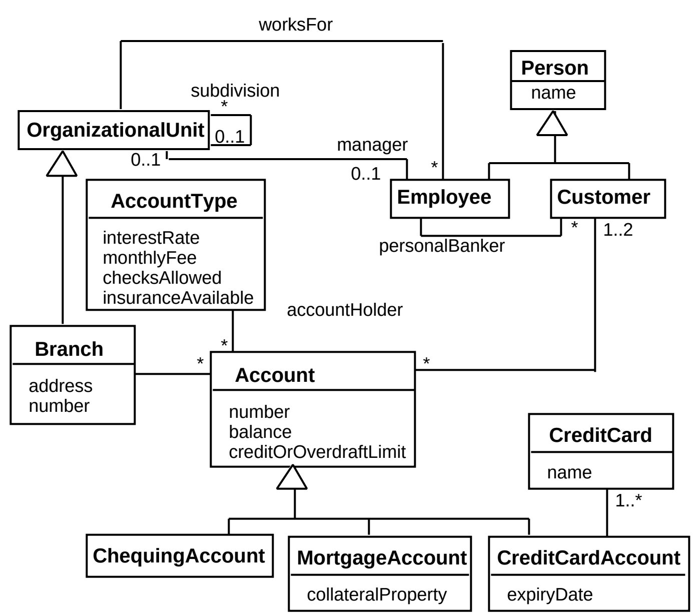
Nouns not on the list because they will probably not be classes in the system:

- System (because it is not part of the domain)
- OOBank (an instance)
- address, branch number (attributes)
- account number, balance, credit limit, overdraft limit (attributes)
- interest rate, monthly fee (attributes)
- husband, wife (information in an example only)
- cheque, insurance (possibly beyond the scope, may become classes later)
- Planning, Investments, Consumer, Consumer Division (instances)
- personal banker (association)

E5.19 *No public answer*

E5.20 *p. 189 Adding generalizations or interfaces to a class diagram.*

a)***Bank account management system:** See 5.21e for a related problem.



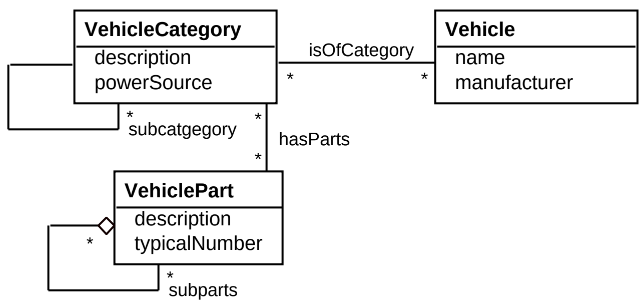
The distinction between **`AccountType`** and **`Account`** is important (so that each **`Account`** does not need to store its own interest rate). There is a good chance, however, that students will not make this distinction in their initial answers.

The following are some variations:

- A customer is not always a person, so the generalization here may need to be changed so the top class is **`LegalEntity`**, which has **`Customer`** as a subclass (or the generalization can be removed entirely)
- There can be separate **`Department`** and **`Division`** classes, although the solution shown, using **`OrganizationUnit`** is more flexible). It is important, however, not to add a separate class for each division, as that would be very inflexible.
- Each individual credit card may have its own number.
- **`CollateralProperty`** and **`Address`** may be classes instead of simple attributes
- You might add a separate **`Privilege`** class associated to **`AccountType`**, instead of the attributes **`checksAllowed`** and **`InsuranceAvalable`**.

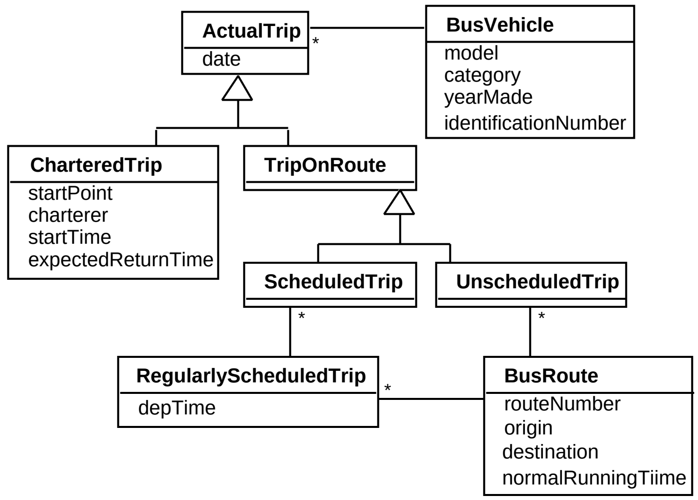
E5.21 *p. 189 Additional exercises in drawing class diagrams, based on E2.10 (p. 41).*
Since each of these lacks detail regarding the requirements, we have had to suggest the attributes and associations that should be present. Note that each of these represents only part of a final system – many details (e.g. additional classes) have been omitted.

a)***Vehicles**: We assume data is being maintained for hobbyists interested in data about a wide variety of vehicles and their parts. In this answer, we decided to follow the pattern found in Figure 5.12 – creating a **`VehicleCategory`** class instead of modelling each type of vehicle as its own class. Students who created the class hierarchy in E2.10 might simply transcribe it here; however, it is a good learning exercise to show that although the hierarchy of E2.10 might have been good for preliminary domain modelling, the answer below is much more flexible (it allows new vehicle types to be added without extra programming). Note also that question E6.2 is related to this.



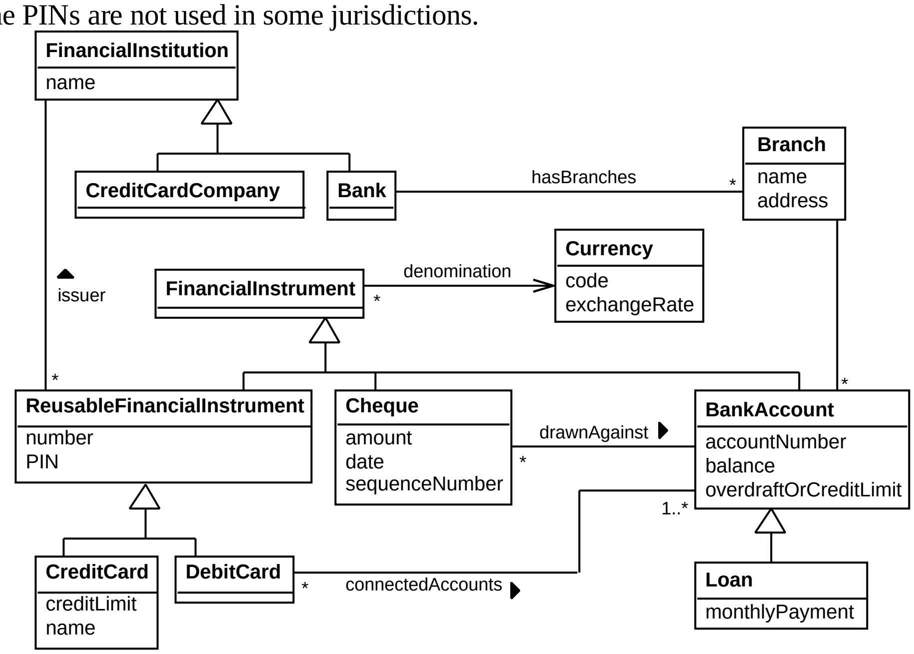
c)***Bus scheduling**: We assume a system for a bus company that runs regular routes as well as charters. This has much in common with the Airline system (see the answer to E5.26). Note the following points:

- The subclasses of **`BusVehicle`** have been merged, and the attribute category is used to determine whether the bus is a luxury one or not.
- Although it would seem that **`TripOnRoute`** would have no distinguishing features, and thus not need to exist as a separate class, in fact it could contain the abstract operation **`getRoute`**, which would be implemented differently in its two subclasses.
- Information about bus stops not included in this model, but would be rather important.



e)***Financial institution**: See 5.20a for a similar problem (that also includes the notion of customers which are omitted here). Note the following points about this answer:

- We have decided to make **`BankAccount`** a kind of **`FinancialInstrument`**; this is a bit different from what we did in the answer to E2.10e, but has the advantage that it allows the denomination association to inherit nicely.
- It is well known that debit cards have PINs. Some have argued that credit cards don't. In fact they do – but the PINs are not used in some jurisdictions.



E5.22 *p. 192 Determining responsibilities for each class.*

It is important that these be simple phrases. Suitable classes for the responsibilities are given in parentheses. In some cases the word 'singleton' is used to indicate that the responsibility could be added to a singleton class not shown in the answer to E5.20. There are sometimes several choices for suitable classes; all suitable choices are listed, with the suggested one in italics. The plus symbol indicates that there should be several polymorphic implementations in subclasses

a)*Bank account management system.

- Adding a new customer (*singleton*, static **`Customer`**)
- Changing details of a customer (***`Customer`***)
- Opening an account (static ***`Account`*** +)
- Crediting an amount to an account (***`Account`***)
- Debiting an amount from an account (***`Account`***)
- Changing the credit or overdraft limit on an account (***`Account`***)

- Issuing a new credit card (***CreditCardAccount***)
- Adding a new employee (***OrganizationalUnit***)
- Changing details of an employee (***Employee***)
- Assigning a personal banker to a customer (***Customer***, **Employee**)
- Adding a new branch (*singleton*, static **Branch**)
- Changing the address of a branch (***Branch***)
- Changing an employee's reporting relationship (***Employee***)
- Changing the divisional structure in the organization (***OrganizationalUnit***)
- Adding a new account type (*singleton*, static **AccountType**)
- Changing details of an account type (***AccountType***)

E5.23 *No public answer*

E5.24 *p. 194 Adding operations to a class diagram.*
These could be listed inside class boxes; however, we have just shown them as simple lists following each class.
a)*Bank account management system

- **Account**
  Account(overdraftLimit, accountType)
  credit(amount)
  debit(amount)
  changeOverdraftLimit(amount)
- **ChequingAccount**
  ChequingAccount(overdraftLimit, accountType)
- **MortgageAccount**
  MortgageAccount(initialBal, accountType, propertyDescription)
- **CreditCardAccount**
  CreditCardAccount(accountType)
  addCard(name)
- **AccountType**
  changeInterestRate(newRate)
  changeMonthlyFee(newFee)
  changeChequingPrivilege(checksAllowed)
  changeInuranceDetails(insuranceDescription)
- **Person**
  (no operations at this level of analysis)
- **Employee**
  changeName(name)
  changeOrganizationalUnit(newOrganizationalUnit)
- **Customer**
  assignPersonalBanker(employee)
- **Branch**
  changeAddress(newAddress, newPhoneNumber)
- **OrganizationalUnit**
  addEmployee(name)
  changepParentUnit(newParentUnit)

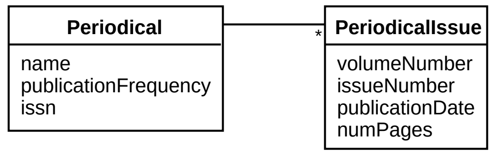
E5.25 *No public answer*

E5.26 *No public answer*

E5.27 *No public answer*

Chapter 6. Using design patterns

E6.1 *p. 206 Applying the Abstraction-Occurrence pattern.*
a)*



E6.2 *No public answer*

E6.3 *No public answer*

E6.4 *No public answer*

E6.5 *p. 221-222 Determining the pattern(s) that would apply in specific design circumstances.*
a)***Adapter** (in your inheritance hierarchy create an adapter class that whose methods delegate to those in the class you want to reuse)

Note, parts a also uses delegation, but this is not the dominant pattern here.

E6.6 *No public answer*

E6.7 *No public answer*

E6.8 *No public answer*

E6.9 *No public answer*

Chapter 7. Focusing on users and their tasks

E7.1 *p. 232 Reflecting on the usability and usefulness of everyday software.*
This exercise is highly subjective; answers will depend on the software chosen, the version of the software and the reader's personal experience with the software.

E7.2 p. 234 Determining the types of users who will use systems.
a)*An air traffic control system would be used by highly trained air-traffic controllers and their managers. They would be using the system intensively throughout their working day. Some aspects of the system might also be used by pilots, government aviation authorities and airport administrators.

E7.3 p. 237 Writing use cases
a)*The sequence of operations could differ from bank to bank.

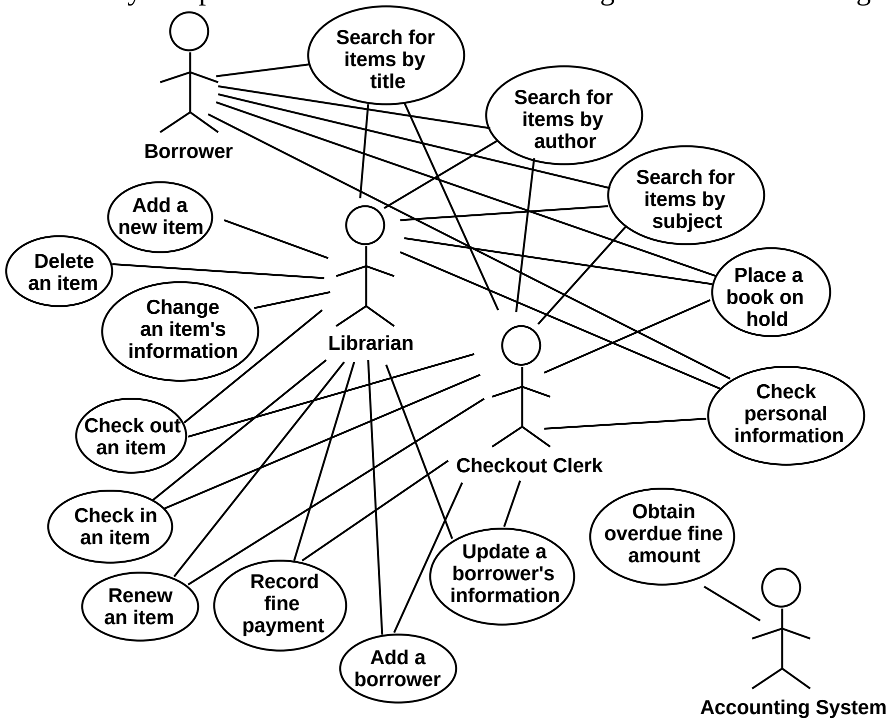
Use case: Pay bill at ATM.
Actors: Any member of the general public with a bank account and bills to pay (minimum likely age is 16)
Goals: To efficiently and accurately pay his or her bill
Preconditions: The actor must have an account at the bank. The bank must accept the bill in question for payment. The actor must have their bank card and know their PIN. The actor must have sufficient funds in their account.
Steps:

Actor actions	**System responses**
1. Insert card	2. Prompt for PIN
3. Enter PIN	4a. Display list of transactions
	4b. Prompt for desired transaction
5. Specify 'pay bill'	6. Prompt for details of bill
7. Enter details of bill	8. Prompt to deposit bill using an envelope
9. Deposit bill	10a. Accept deposit
	10b. Debit account
	10c. Print receipt
	10d. Prompt to see if there is another transaction
11. Indicate no other transaction	12. Eject card
13. Take card	14. Display welcome message

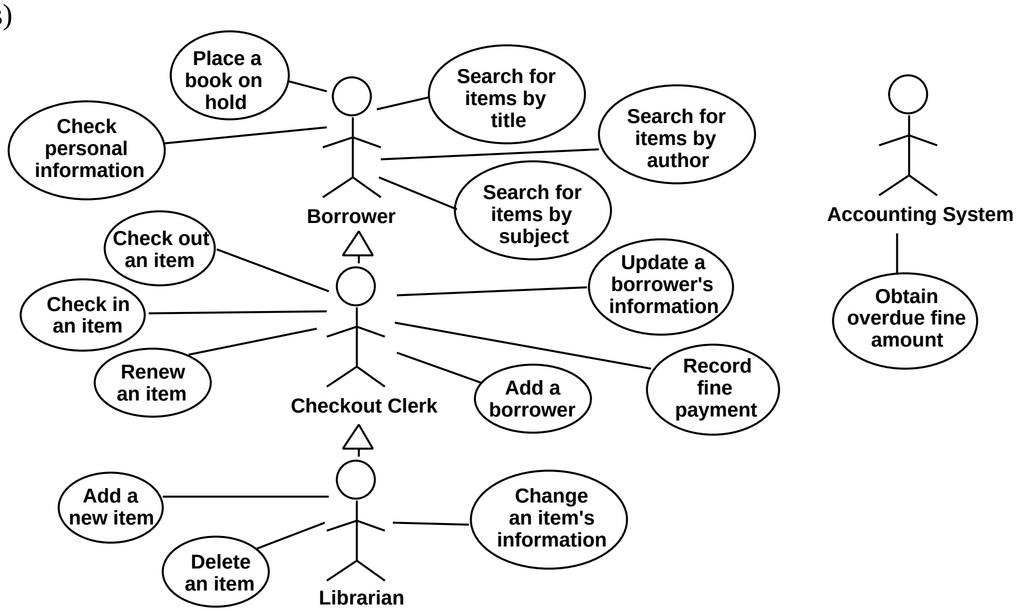
Postconditions: The system has stored the bill physically, and has a record of the transaction. The actor has a receipt for the transaction.

E7.4 p. 238 Drawing use case diagrams

a)***Library system** (based on example 4.9, page 126) We give two answers for this: The first does not use generalization, which had not yet been discussed in the text in the context of use case diagrams. This diagram is excessively complex so would not be worth drawing in a real software engineering project.



The second version of the answer uses generalization, showing how the diagram can be simplified (fewer lines)



E7.5 *p. 240-241 Writing use cases for the car park system.*

a)*

Use case: Exit car park by paying using a debit card.
Goals: To leave the parking lot after having paid the amount due using a debit card
Preconditions: In addition to 'exit car park' preconditions: The driver must have available his or her debit card and know the PIN. The driver must have sufficient funds in his or her bank account.
Related use cases:
Specialization of: Exit car park
Steps:

Actor actions	**System responses**
1. Drive to exit barrier, triggering a sensor.	2a. Detect presence of a car.
	2b. Prompt driver to insert his or her card.
3. Insert ticket.	4. Display amount due.
5.1 Insert debit card.	6.1 Prompt for account (chequing/savings).
5.2. Specify account.	6.2. Prompt for PIN.
5.3. Specify PIN.	6.3a. Debit account.
	6.3b. Eject debit card.
	6.3c. Prompt driver to take debit card.
5.4. Remove debit card.	6.4. Raise barrier.
7. Drive through barrier, triggering a sensor.	8. Lower barrier.

Postconditions: In addition to 'Exit car park' postconditions: The driver's bank account has been debited.

E7.6 *No public answer*

E7.7 *p. 242 Listing use cases and prioritizing them.*

a)*A telephone answering machine.

- Record outgoing message
- Record incoming message
- Listen to incoming messages
- Skip back to previous message
- Erase a message
- Skip to next message
- Record a conversation

E7.8 p. 257 *Creating error messages for a user interface.*

a)*There are unlikely to be any error messages on this screen!

E7.9 *p. 257 Determining whether response time is adequate at a web site.*

The answer to this exercise depends on the design of the web site chosen (and that web site's response time may change from time to time).

E7.10 *p. 257 Preparing a paper prototype.*

This a practical exercise with no specific right answer.

E7.11 *No public answer*

E7.12 *p. 258 Performing heuristic evaluations of user interfaces.*

The answers to this exercise depend on the programs chosen, the versions of those programs, and the parts of the programs evaluated.

E7.13 *p. 258 Performing a heuristic evaluations of a web site.*

The answers will depend on the web site chosen.

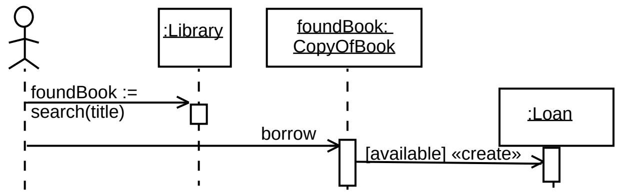
E7.14 *p. 258 Having multiple people conduct separate heuristic evaluations.*
This is another subjective exercise.

E7.15 *p. 259 Performing heuristic evaluation of a GANA prototype and updating the UI.*
This is another subjective exercise.

E7.16 *p. 260 Performing an evaluation of a UI by observing users.*
This is another subjective exercise.

Chapter 8. Modelling interactions and behaviour

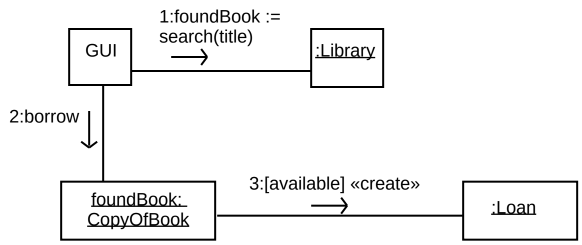
E8.1 *p. 273 Drawing sequence diagrams.*
a)***Library example**: This assumes that there is a **`Library`** singleton class acting as a Façade.



E8.2 *No public answer*

E8.3 *p. 275 Drawing more collaboration diagrams*
These correspond to the answers to E8.1

a)*Library example



E8.4 *No public answer*

E8.5 *p. 279 Determining the state the system would be in after a sequence of events.*
a)*It would be in state **`Cancelled`**.
Rationale:
Initial = **`Planned`**;
`openRegistration` -> **`OpenNotEnoughStudents`** (**`classSize`**=0);
`requestToRegister` -> **`OpenNotEnoughStudents`** (**`classSize`**=1);
`requestToRegister` -> **`OpenNotEnoughStudents`** (**`classSize`**=2);
`cancel`->**`Cancelled`**

E8.6 *No public answer*

E8.7 *No public answer*

E8.8 *No public answer*

E8.9 *No public answer*

E8.10 *No public answer*

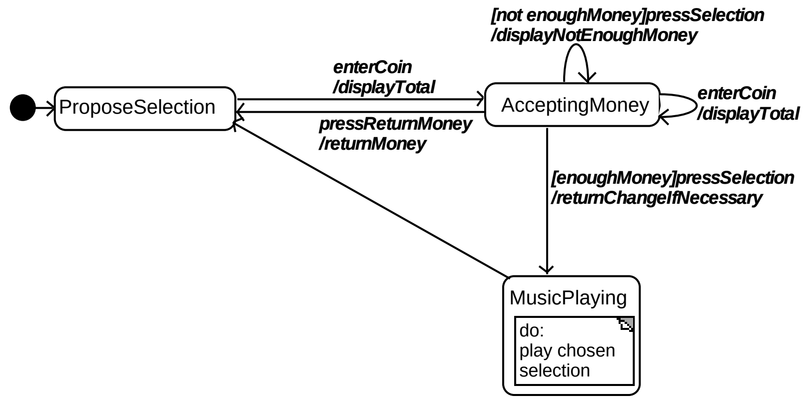
E8.11 *No public answer*

E8.12 *No public answer*

E8.13 *No public answer*

E8.14 *p. 283-284 Drawing and extending state diagrams.*

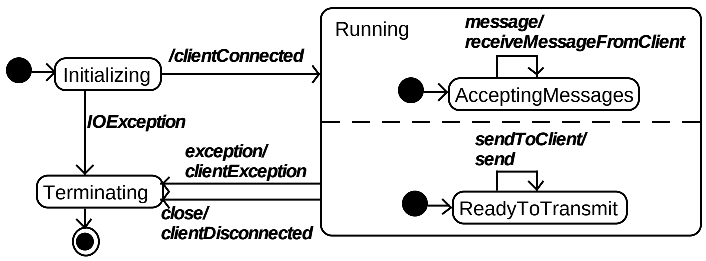
a)***Jukebox**.



E8.15 *No public answer*

P8.2 *p. 291 Drawing state diagrams of OCSF and SimpleChat*

a)***`ConnectionToClient`** in the OCSF Framework: Note that events such as **`setHost`** have not been modelled here.



Chapter 9. Architecting and designing software

E9.1 *p. 299 Dividing a system into subsystems (see also E4.3, p. 110).*
There will be several ways of dividing these into separate systems. The following systems suitable for the requirements for are expressed as a simple hierarchical list.
a)*Police information system
- User interface layer/client
- Main functional layer/server
  - Scheduling subsystem
  - Case management subsystem
  - Personnel management subsystem
  - Statistics subsystem (may not be implemented in early releases)
- Document repository
- Database subsystem
- Subsystem providing interfaces to other systems

E9.2 *No public answer*

E9.3 *No public answer*

E9.4 *No public answer*.

E9.5 *No public answer*

E9.6 *p. 312 Categorising aspects of designs according to the coupling they exhibit.*
a)*Common coupling (all methods that access the variables are coupled)

E9.7 *p. 313 Determining ways to reduce coupling.*
a)*It would be better to have methods **`getMinClassSize`** and **`getMaxClassSize`** available in **`CouseSection`** – this is probably a reasonable place to use static methods, even though we recommend in general that they be avoided. Access to the static variables should not be public.

E9.8 *No public answer*

E9.9 *pp. 321-322 Choosing an architecture, basing the decision on quality requirements.*
In parts a and b, the objectives are not met by any of the architectures; this raises some debate since we have to choose which requirements to relax; we really need further study and discussion with the stakeholders.

a)*Choose architecture A, although architecture C might also be a reasonable choice – further study is probably needed. Rationale:
- Runs on windows: Clearly eliminates D (A, B, C left)
- Works on a 30 Kbps connection or faster: Eliminates B and perhaps eliminates A, however we will leave A under consideration for now since it is close to this threshold (A, C left)
- Works on a 500 MHz machine or faster: Clearly eliminates C. Leaving only architecture A even though it does not quite meet the highest priority objective: bandwidth efficiency.

E9.10 *No public answer*

E9.11 *No public answer*

E9.12 *No public answer*

E9.13 *No public answer*

E9.14 *p. 339 Designing the architecture of systems.*
a)*A corporate payroll system.
To be provided.

Chapter 10. Testing and inspecting to ensure high quality

E10.1 *No public answer*

E10.2 *p. 353 Determining equivalence classes for testing purposes.*

a)***Telephone number.** Note that it is good practice in software engineering to not require telephone numbers to be in any particular format; this is because telephone number formats can change over time, and because formats differ from country to country. Also, people need to enter strings such as 'ext' to specify an extension; therefore in the following we are being very liberal about what could be entered. You may choose to be more restrictive.

Equivalence classes of valid data
1 Any string of printable ASCII characters (including letters, numbers and special symbols)

Equivalence classes of invalid data
2 The empty string (assuming the input is required)

c)***A time zone (numeric or alphabetical)**

Equivalence classes of valid data:
1 Any sequence of three upper-case letters (time zones can be specified in many different ways depending on the country; we are thus being liberal about allowed input)
2 A string of the format shh[:]mm, where
s can be '+' or '-',
hh can be 00 to 14 (time zones near the date line can sometimes be greater than 12 hours from UTC)
mm can be 00 or 30

Equivalence classes of invalid data:
3 An empty string (assuming input is required)
4 A string of length 1 or 2
5 A numeric time zone without the '+' or '-'
6 A numeric time zone where the hour is greater than 14
7 A numeric time zone where the minutes range from 01 to 29
8 A numeric time zone where the minutes are other than 00 or 30
9 A sequence of alphabetic characters of length greater than 3
10 A character string mixing digits and letters
12 A character string with characters other than digits, letters or ':', '+ and '-' in the appropriate spots.

E10.3 *p. 355 Determining equivalence classes when there are multiple inputs.*
The following also incorporates the answer to E10.4 (marked - Boundary cases)

a)***Personal information form**
Date of Birth: We will consider this as three separate inputs with a required format of yyyy/mm/dd; you may have picked a different format.

Year
1 Valid: integers from 1880 to 9999 (Assume this is a system for people alive at the time the system is used
- Boundary cases: 2000, 2001, 1880, 9999
2 Invalid: Integers from 0 to 99 (we treat this as a separate case since there may be some code that incorrectly handles 2-character dates, which we have decided not to allow)
- Boundary cases: 0, 1, 99
3 Invalid: Integers from 100 to 1879
- Boundary cases: 100, 1879
5 Invalid: Empty string
6 Invalid: String with non-numeric characters

Month
1 Valid: 01 to 12

- Boundary cases 01, 12
2 Invalid: 1 to 9 (assume we explicitly require two digits)
- Boundary cases 1, 9
3 Invalid: 00 and 0
4 Invalid: 13 to 99
- Boundary cases 13, 99
5 Invalid: Empty string
6 Invalid: String with non-numeric characters

Day
1 Valid: 01 to 28
- Boundary cases 01 and 28
2 Valid: 29 in February of a leap year, or in any other month any year.
3 Invalid: 29 in Frebruary of a year other than a leap year.
4 Invalid 30 (in February)
5 Valid: 30 (in some month other than February)
6 Invalid: 31 (in a 30-day month or in February)
7 Valid: 31 (in a 31-day month)
8 Invalid: 32-99
- Boundary cases 32, 99
9 Invalid: 00 and 0
10 Invalid: 1 to 9 (2 characters required)
- Boundary cases 1, 9
11 Invalid: Empty string
12 Invalid: String with non-numeric characters

Street Address:
Valid: Character strings of length 2-40 (assumed limits) containing upper and lower case characters, numbers, comma, period, hyphen, single spaces, accents and diacriticals
Boundary case: Strings of length 2 and 40
Invalid: Character strings containing non-printable characters
Invalid: Character strings with more than one embedded space
Invalid: Strings of length 0-1

City, Country and Postal code
Same as street address, except that country and postal code can have 0 characters

Home Telephone Number
Valid: Empty string
Valid: String of characters including digits, '+', '(' ')', '-', 'x' and space
Invalid: Other strings of characters

E10.4 *p. 355 Determining boundary values.*
See the lines marked 'boundary cases' in the answer to E10.3

E10.5 *No public answer*.

E10.6 *p. 358-359 Tables of conditions for testing purposes.*
a)*Message forwarding
To be provided.

E10.7 *No public answer*

E10.8 *pp. 365-366 Classifying and testing for numerical defects.*
a)*Assuming a floating point value will be exactly equal to 3.0, when it might be slightly different from this.

E10.9 *No public answer*

E10.10 *No public answer*

E10.11 *p. 372 Designing stress and unusual-situation tests.*

a)*A new web browser

- Loading a truly massive web page full of complexity.
- Loading a web page containing a large amount of JavaScript.
- Loading a web page containing several large Java applets.
- Loading a web page full of html errors
- Loading a web page with obsolete html constructs
- Loading a web page with the latest complex html and cascading style sheet constructs
- Loading a web page that contains an extremely large number of levels of nested html tags.
- Loading a web page containing tables with an extremely large number of columns and rows.
- Loading a web page demanding a lot of width of the resulting output
- Loading a very large web page that contains a large number of embedded objects.
- Testing the browser on an extremely slow connection.
- Loading a very large number of web pages at once in multiple windows.
- Running on a variety of different versions of the target operating system.
- Running on a machine with too little memory
- Running on a machine with too little disk space for caching purposes.
- Loading random pages over and over again for a long time (using a driver)
- Running on a very small screen
- Dealing with repeated failures of connections

E10.12 *p. 376 Writing test cases.*

a)*Message forwarding

To be provided.

E10.13 *No public answer*.

E10.14 *No public answer*.

E10.15 *p. 387 Conducting code inspections.*

This is a purely practical exercise

Chapter 11. Managing the software process.

E11.1 *p. 401 Reviewing group work in order to improve process.*
This is practical work only

E11.2 *No public answer*

E11.3 *No public answer.*

E11.4 *p. 419 Creating a team structure for different software projects.*
a)*Replacement of the income tax system of a country (unless the country is a tiny island nation) would be a very big undertaking, requiring hundreds of people. Some kind of **hierarchy** would certainly be required. There would probably be a group of system architects and overall project planners who would divide the system into its parts, and plan the interfaces among the parts. Then there would be a variety of sub-projects. Some of the sub-projects might be managed using the **chief-programmer** or **egoless style**.

E11.5 *No public answer*

E11.6 *No public answer*

E11.7 *No public answer*

E11.8 *No public answer*