Victoria University of Wellington
School of Engineering and Computer Science

SWEN222: Software Design

The Excellent Adventure

(Group Project)

Worth 20% of Overall Mark

Software + Documents Due: Wednesday 14th October @ Midnight
Individual Reports Due: Friday 16th October @ Midnight
Demonstrations: 15th + 16th October

1 Introduction

You are to design and implement a program for a multi-player "graphical adventure" game. The objective of the game is to explore an imaginary world, collecting objects, solving puzzles, and performing actions to complete the game. This project will bring together all of the techniques you have been taught thus far.

You should undertake this project in teams of 4-5 people. We want you to work in teams for two different reasons: firstly, to experience what it is like to work as part of a team on a software development project; secondly, to be involved in a larger project without you having to do all the work yourself. Your team must be registered using the online team signup system:

`http://www.ecs.vuw.ac.nz/cgi-bin/teamsignup`

The team signup system will close on Friday 11th September @ Midnight, and you must ensure your team is registered by then.

NOTE: you may register incomplete teams (e.g. 1, 2 or 3 people) who wish to work together. *All incomplete teams will be combined together to form complete teams of 4-5 students.*

The Group Project will be conducted under the group work policy. We understand there are sometimes difficulties when working in teams, because of personality conflicts, and the pressures of time and workload. You will be able to resolve some such difficulties yourself, but we are willing to help. If your team is facing problems that you cannot resolve, then please seek our assistance.

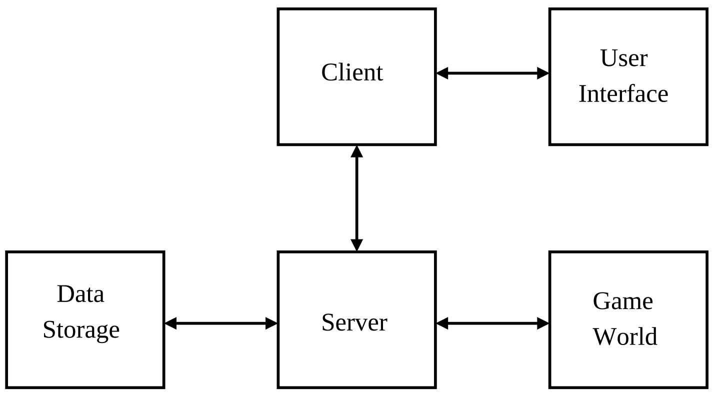


Figure 1: Illustrating the high-level architecture of the Adventure Game

2 Requirements

What follows are the main requirements for the adventure game. They are neither extremely detailed, complete nor consistent: your team must make reasonable assumptions where necessary or ask for clarification (e.g. on the forum).

The adventure game should follow a client-server architecture, with the high-level structure consisting of five main pieces, as illustrated in Figure 1. The main pieces of the design are:

- **The Game World package**. This is responsible for maintaining the current state of the game, such as where players and objects in the game currently are, and also for determining what actions are allowed within the game.
- **The User Interface package**. This is responsible for displaying the game world in a window, and for allowing the user to control the game (e.g. loading/saving games, moving the player, etc).
- **The Client-Server package**. This is responsible for communicating information between the client and server. Events triggered by a player (e.g. moving, picking up objects, etc) are propagated from the client to the server; likewise, the server broadcasts some, or all of the game state to ensure clients maintain a consistent view of the game world.
- **The Data Storage package**. This is responsible for reading and writing files that represent the stage of the game. The format used for this should be XML.

Each of these pieces will now be considered in more detail.

2.1 Game World

The adventure game world is made up of locations, which are either enclosed areas (i.e. rooms) or outside areas. Each location has a unique internal name and a short description (usually one or two lines). Each location has four walls — north, south, east and west — and may have any number of exits. Some of these exits may be blocked (e.g. a locked door) or hidden. They might also be one-way (e.g. a steep drop). The player needs to be able to explore the world, getting location descriptions and moving around using the exits.

The *game world* package is responsible for maintaining the *game state* and implementing the *game logic*. The game state is primarily made up of the game world itself, the current location of all objects in the world, the location of each player, the items being carried by each player and each player's score. The game logic controls what events may, or may not happen in the game world (e.g. "Can *that* player go through *this* door?", "Can *that* player pick up *this* object?", "Does *this* key open *that* door?", "Can *this* container hold *more* items?", etc.)

Each object has a unique internal name and a short description. Each object is either at a location, being carried by a player, or inside another object. Each player needs to be able to pickup/drop objects and examine them (to see their descriptions). Players must also be able to list their inventory (i.e. the objects they are carrying).

There can be many kinds of objects, including:

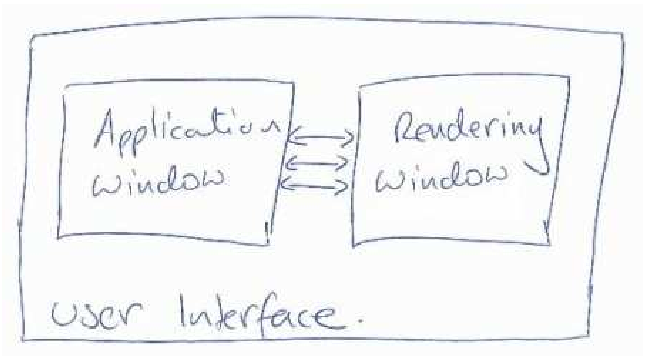
- **Furniture** or other decorative objects (e.g. trees) that can't be picked up or moved, but that provide extra background description or clues to puzzles.
- **Movable** objects (e.g. books) that can be picked up and moved to different rooms and positions within those rooms.
- **Containers** like bags, backpacks, boxes, chests, or suitcases that can have other objects inside them. The player should be able to open and close containers, put objects inside them or get them out. You must be able to put one container (like a wallet) inside another container (like a suitcase).
- **Keys** are special objects that let you unlock containers or locked doors.
- **Lights** let you see and move around safely in dark locations.
- **Maps** are special items that allow you to see the game world more completely. They can be especially helpful, for example, when trying to figure out where hidden locations are.
- **Magic** items (or perhaps technological gizmos) let the player do special things, such as fight monsters, teleport around the world, or solve other puzzles.
- **Treasures** win points for the player when they are put in the right place. For example, putting a disk into a computer to reveal a secret message would be awarded points.

These choices may not be mutually exclusive: perhaps one object is both a magic item and a treasure.

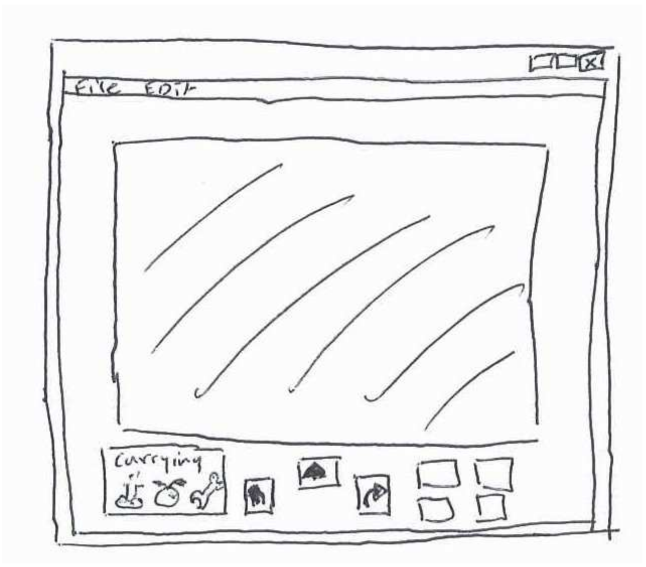
HINT: some of these objects are harder to implement than others. You should focus on getting the simplest to work first, before moving on to those which are more complicated.

2.2 User Interface

Your program should provide a Graphical User Interface through which each player can see the game world and control his/her actions. This is called the *user interface package* and should consist of two sub-packages, roughly as follows:



The *application window* is responsible for presenting appropriate windows to the user and allowing them to interact with the program. A typical application window looks something like this:



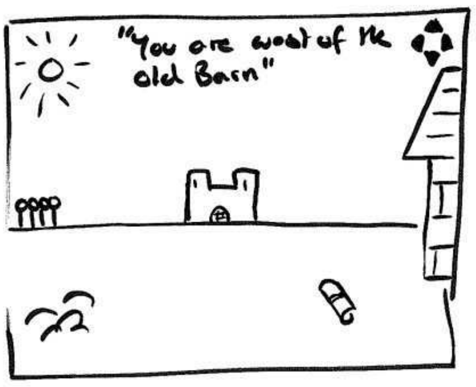
The application window should provide appropriate menus, buttons and right-click windows allowing the user to perform actions in the game. You should provide commands that allow the player to move between locations (e.g. "go north"); "take" and "drop" objects; "look" to see a location's description; "examine" objects; "list" the players inventory etc. This is just the start, however, as your game may need more specialised commands — perhaps to move in other directions or other ways (e.g. "jump", "run", "push", "kick"), manipulate special objects (e.g. "push button", "swipe card", "polish lamp"), or to complete puzzles ("pay shopkeeper", "throw ring into fire"). Sometimes, normal commands may do special things in certain positions.

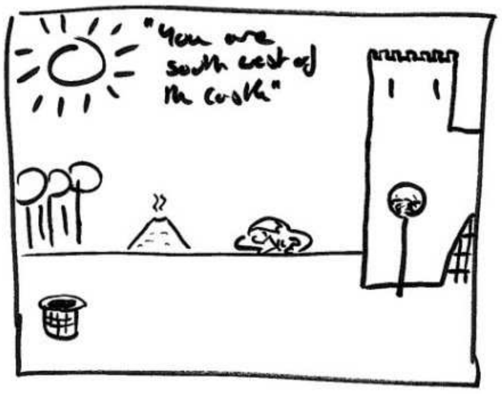
HINT: A simple approach to user input would be to list the available actions in separate buttons next to the game view. You could supplement this with a menu system. A more advanced approach might allow the user to click on individual objects (with the mouse) to see their descriptions. Right-clicking could bring up the list of available actions on that object, etc.

The *rendering window* is responsible for providing a simple 3-dimensional view of the game world, with locations presented from a side-on perspective. Typically, the application window provides a canvas on to which the game world view is rendered. Each location should have four orientations of view (i.e. north, south, east and west), which are shown according to how the location was entered. For example, a location entered from the north is shown with the view facing south.

Objects in the current location should be shown in the foreground, whilst those at locations in the distance should be shown in the background. To get a sense of perspective, objects which are further away should be drawn smaller. Objects should be represented using simple bitmaps or polygons.

The following images illustrate what your game view might look like. However, it does not have to be identical to these and, in practice, may be quite different.





HINT: Do not spend time on a proper 3-dimensional view with accurate perspective etc. Instead, use a simple "painters algorithm" which draws objects in the background first, and then those in the foreground on top.

HINT: The player should be able to put objects down in a room and view them from other rooms. From some orientations, an object may be partly obstructed by other objects, or doors which are in the way.

HINT: Outside spaces may be modelled using multiple "rooms" which, effectively, have "see-through" walls. Thus, it should be possible to see objects in the distance — and, of course, they will appear smaller.

HINT: Remember that each location only ever needs to be drawn from four viewpoints. You do not need to support viewing a location from any angle (as is done in popular first-person games).

HINT: The player does not need to be able to move within a location.

HINT: A simple approach to use a static background (e.g. a photograph) which appears 3-dimensional. However, this is not a good solution, as it does not utilise the 3rd dimension properly.

2.3 Client-Server Architecture

The *client-server* package is responsible for managing all communication between the server and any clients currently enabled. For any given game, there should be one server running, as well as one client per player. The idea is that the server can be located on one machine, whilst the clients may be scattered across many machines — and all communication happens via the network.

The server and/or clients should be *fault-tolerant* and be able to handle unexpected disconnections gracefully. A client is started by running the appropriate Java class from the command-line, passing in the location of the server to connect to. Similarly, the server is started with a given game file to use as the starting point and, optionally, the minimum and/or maximum number of clients to accept in the game.

HINT: The Pacman game discussed in lectures provides a good example of a client-server architecture. All of the components needed for this package are present in the Pacman game.

HINT: One of the key challenges here is to determine a sensible network protocol to use. This needs to transmit all necessary information regarding the current game state, whilst remaining efficient.

HINT: Given that the game has a client-server architecture, and is running in real time, it should be possible for one player to see other players as they move around the game. Thus, when another player enters a room, he/she immediately becomes visible to others players currently in that room.

2.4 Data Storage

The *data storage* package is responsible for loading and saving game files. Game files must be stored in an XML format, and must completely describe the state of the game world at any particular point. In particular, it should be possible to completely close down the game, and then later restart it from exactly the same point as before.

HINT: The data storage package should be robust and report errors for incorrectly formed game world files.

HINT: There are several good Java packages available for reading, writing and verifying XML. Examples include: JDOM, XOM, DOM4J and Woodstox.

2.5 Hints for Design and Development

We expect you to design your adventure game prototype working as a team, and applying what has been discussed in lectures, including UML class and sequence diagrams and CRC-card role play. Remember that these techniques should allow you to experiment with design alternatives before you make large investments in implementation. You should use the design techniques to develop your design to such a point where you feel confident in starting implementation.

HINT: Your initial design does not need to cover the complete functionality you are aiming for. A good approach is to start with a simple design which gives the basic functionality you need (e.g. a collection of locations with simple objects in them). Then, once you have this working, you could consider adding a richer variety of objects and tackling harder aspects of the game (such as special objects, plot tokens etc).

Plan your time on this project carefully, especially time spent on implementation. Make certain you understand the basic technology before starting to create code. Implement code in stages. Get each stage working successfully before working on the next stage. Get simple versions working before going on to full versions. Be prepared to abandon the less critical features if you run out of time. Devote attention to well designed and cleanly implemented code rather than focusing on adding fancy features.

Even after the initial team design effort, you should still regularly communicate as a team to track your progress in implementation. Especially as the due-date approaches, you should make efforts as a team to fulfill the requirements.

One of the main problems in a project like this is inter-operability. You should be careful to ensure that it is clear what classes/methods each person will provide. You should attempt to integrate your program as soon as possible and as often as possible. Do not leave this until the last moment as there will certainly be problems to overcome. You should also consider having code written by one person tested by another, as this always raises unexpected issues.

2.6 Extra for Experts

You could also consider implementing one or more extra features (your client would like these, but doesn't know how important they are):

- Simulating the passing of time as the player explores the world. Perhaps some things are only possible at some times (e.g. shops that open and close; sunrise and sunset etc).
- Having other characters that "wander around". These can range from random encounters (the goblin than appears from nowhere and gives you a gold coin) to more realistic people that move around the map according to some predefined rules.
- A combat system for simulating fighting between player/monsters. For example, each might have a certain amount of health points and strength, with each strike deducting so many health points of the opponent.
- The ability to talk or interact with other characters (such as exchanging objects, trading etc.)
- Money
- Weather
- Weight or size limits, so that containers/players can only carry a certain amount of things (you shouldn't be able to put a large rock inside a wallet for example).

3 Workload

There are five main components in the design of the adventure game:

1. **Application Window**. Covers construction of the application windows, menus and user input.
2. **Rendering Window**. Covers rendering of the game view display.
3. **Game State + Logic**. Covers the internal structures needed to represent current state of the game, and the rules used in determining what actions are and are not allowed at a particular state of the game.
4. **Client-Server Architecture**. Covers all setup and communication between the server and all clients.
5. **Data storage**. Covers loading and saving of game world files in an XML format.

Each member of your team should choose one of these components as being their responsibility to implement. If your team has only four people, then you may choose not to implement the data storage package.

HINT: Some components are harder than others, so choose carefully! Rendering is the hardest, with Data Storage being probably the easiest. The others are somewhere in between.

4 Assessment

There are several elements involved in the assessment of your project, as follows:

- **Integration day**.
- **Design documents** and **code** for the program, including JUnit tests, Javadoc and an appropriate README file.
- **Presentation**.
- **Individual Report**.

4.1 Integration Day

On or before Integration Day (Friday 2nd October @ Midnight) your team must demonstrate a working program to one of the lab tutors. This program must demonstrate functionality from each of the five main components of the system, although it does not need to be complete. Thus, there should be an application window containing some buttons and the rendering window; the rendering window should be able to draw the game world view to some degree; the data storage package should be capable of reading files in the basic file format given; and, finally, there should be some evidence of the game state and game logic.

4.2 Design Documents, Code, JUnit Tests, Javadoc and README

The design documents, program code, JUnit tests and Javadoc manual must be submitted by midnight on Wednesday 14th October @ Midnight. We require:

- The CRC-cards for your design
- UML class diagrams for the whole program.
- Program code, including JUnit tests and Javadoc

Your program code should be in an appropriate directory (i.e. package) structure. It should be well-documented internally with the name and student ID of the author(s) clearly marked at the top of each Java file. Remember to present your code well: code that is difficult to understand will be penalised. Your code must work on the ECS lab machines as you will have to demonstrate it on them. You will be heavily penalised if you cannot demonstrate a working program at the final presentation. We also expect that you will have a reasonable number of JUnit tests for the various classes in your program and that you have generated a suitable a Javadoc manual to complement your program source.

Your program code should include a README file which details exactly how to start up your game and provides any information necessary on how to play the game (this information is essential for the tutors who will test your game).

HINT: Your UML documents must be easy to read. For example, the main piece of information a class diagram should convey is the class hierarchy. To help achieve this, you should avoid listing every method in each class. Instead, include only those which constitute the most important features of your implementation. Typically, there will only be at most 3-4 important methods per class.

HINT: For a B+ standard project, we would expect at least one test per non-trivial (public) method in the game world and data storage packages. Examples of trivial methods include: constructors that simply assign parameters to fields; and getter/setter methods. Observe that tests don't need to be provided for the GUI, since it's not easy to test a user interface in JUnit! Likewise, properly testing the networking component is very difficult, and is beyond what we could expect in the time available.

You are expected to use subversion when developing your project. You must create your subversion repository in "/vol/projects/". The exact location to use will be sent out once the team signups have been finalised and the corresponding UNIX groups created.

4.3 Presentation

Each group will be required to give a short (approx: 10mins) presentation demonstrating their project, and (briefly) discussing its design. Marks will be awarded on an individual basis for clear communication, and for how effectively the solution solves the given problem. During the presentation, each member of the group is expected to speak. **The presentations will take place on 15th + 16th October**.

4.4 Individual Report

Your individual report must be submitted by mid-night on Friday 16th October @ Midnight. The report itself should be private — you should not consult with your group members over the report. The report should consist of:

- A statement explaining your individual contribution to the project, listing each important part which you developed (max 100 words).
- A detailed appraisal of the overall design for your component of the project (max 1000 words, may additionally include diagrams/images). In particular, the report should clarify: 1) the main responsibilities of your component; 2) how your component interacts with other components; 3) aspects of your design which demonstrate quality; 4) aspects of your design which could be further improved; 5) what alternative approaches were considered. In addition, you are free to discuss other points as you feel appropriate.
- A ranking of the contribution of each person in the team. To do this, list the name and ECS email address of each team member (including yourself) and give them a score (out of 5) for their contribution. For example:
    1. Sue Student sams@ecs.vuw.ac.nz 3
    2. Lazy Larry larry@ecs.vuw.ac.nz 1
    3. Busy Bernice berniceb@ecs.vuw.ac.nz 5
    4. Dave Dedicated daved@ecs.vuw.ac.nz 5

4.5 Grade

You will receive an individual grade (A+ ... E) to recognise the whole of your project work. As a guide, the grade will reflect:

- Project Passes Integration Day: 10%
- Project Design and Implementation (scope and quality): 40%
- Individual Contribution: 30%
- Individual Report: 20%

A reasonably well written, properly integrated project which meets the basic requirements in this document can expect a grade of B+. For a higher grade, you will need to have a good overall object-oriented design, clear program code in a recognised style, and/or implemented extra features.

4.6 Submission

The code, JUnit tests and Javadoc manual should be submitted electronically using the *online submission system* from the course webpage. It must be packaged in an executable Jar file and MUST include all the source code necessary to compile and run the program.

NOTE: any external libraries used in the project should be included as part of the submission. Thus, it should be possible to test your game without downloading and installing anything.

In addition to the source code, various pieces of design documentation are required. These should be submitted as either PDF or PNG files; other formats will not be accepted. The required documents are:

- The CRC-cards for your design
- UML class diagrams for the whole program.
- Program code, including JUnit tests, Javadoc and README

4.7 Late Penalties

The lectures and assignments will have given you the opportunity to develop experience in design and programming, and the deadline has been set to give you time to complete this project. Accordingly, you should have no problem in completing this project on time. Furthermore, as the final deadline is already as late as possible and arrangements for marking are very tight, extensions can only be given in exceptional circumstances. If you do not submit on time without prior consent you should expect to receive no marks for the project.

5 Marking Guide

What follows is a detailed marking guide for the main components of the project.

5.1 Project Design and Implementation (50%)

The assessment of the project as a whole is based primarily on a demonstrated use of good software design, engineering, and programming practices. This includes:

- **Requirements (30 marks)**. Marks will be awarded for how well the project meets the given specification above. This includes, but is not limited to, the following items:
  - **Objects.** e.g. *furniture, containers and treasure*
  - **Pickup / Drop Objects** *e.g. ability to pickup/drop objects, and spatial positioning of objects*
  - **Navigation** e.g. *ability to move between and/or within rooms, hidden or locked doors, map, etc.*
  - **Base Rendering** e.g. *pseudo-3D view, different location views, etc*
  - **Painters Algorithm** e.g. *views dynamically generated, objects further away painted first, can see objects in distance (particularly moveable objects)*
  - **Application Window** e.g. *layout, menu bar, inventory display, etc*
  - **Interaction** e.g. *point and click selection, right-click menus, etc*
  - **Client-Server Architecture** e.g. *does it work, can we see player avatars, and updates made by other players, etc*
  - **Data-Storage** e.g. *load different game worlds, load/save current game, etc*
  - **Extras** e.g. *passing of time, non-player characters, weight/size limits, , etc*
- **Design Documents (20 marks)**. The design documents should effectively communicate the projects' design. Good diagrams avoid clutter by showing only the most important information. They are well presented, and easy to read/follow. Furthermore, diagrams should focus on correctly identifying the most important issues in the design and communicating them effectively.
- **Implementation (50 marks)**. Marks will be awarded for various aspects related to the program itself including, but not limited to, correctness and style. Submitted projects are expected to follow the appropriate style guide, and include comments suitable for documentation purposes (e.g. Javadoc), as well as general understanding. Submitted projects are also expected to provide a range of JUnit tests to give confidence that the program is correct. The breakdown of marks is as follows:
  - **Naming (5 marks).** The choice of names for key concepts in the design and implementation is critical. This includes (but is not limited to) *class names*, *method names* and *variable names.* The names should succinctly indicate what the item is for.
  - **Encapsulation (5 marks).** Classes should be encapsulated to properly separate interface from implementation. Use of `protected` fields is acceptable. However, use of `public` fields should be justified (except for e.g. `static final` fields which are being used as constants).
  - **Code Reuse (5 marks).** Good code should avoid duplication as much as possible. Inheritance and sub-division into methods should be used to avoid duplication of code and promote reuse.

- **Testing (5 marks).** A range of JUnit tests should be provided. As indicated above:

  > "*For a B+ standard project, we would expect at least one test per non-trivial (public) method in the game world and data storage packages.*"

- **Commenting (10 marks).** Commenting is especially important. In particular, most public methods and classes should include an appropriate JavaDoc comment. In some cases, this may be copied directly from the documentation of a base class or interface (or via `@Override`, assuming appropriate documentation on the superclass).

  Other than JavaDoc, internal commenting should be also be provided to aid in understanding the code. This should include comments directly on the code within methods.

- **Inheritance (5 marks).** Inheritance should be used to help structure the program. This includes (but is not limited to) *promoting reuse*, and *identifying classes which are specialised versions of others.*
- **Packaging (5 marks).** Proper use of packaging should also be employed to help structure the program. We might expect to see each of the main components put into their own packages.
- **Design Patterns (5 marks).** Good use of design patterns should help improve code structure. Patterns which are used should be clearly identified in the code base.
- **Existing Libraries (5 marks).** Good use of the Java standard library can help to make code more robust and compact. In particular, reimplementing functions which are already provided by the standard library unnecessarily increased code complexity.

5.2 Individual Contribution (30%)

The individual contribution refers to the *quality* and *amount* for a student's contribution. In assessing this, information will be gathered from a variety of sources. These include (but are not limited to):

- The student's individual report, and those of the other team members.
- The version control logs. In particular, it is expected that students use version control in a sensible fashion, such as providing appropriate commit messages and committing relatively frequently.
- The source itself. In particular, it is expected that students identify which files they contributed to. This can be done by adding the login to the authors list at the top of each file.
- The presentation.

5.3 Individual Report (20%)

Marks will be awarded for clear communication and a good discussion of the design, and for covering the five items listed in §4.4. Problems with grammar and other typographical mistakes will be penalised.