|  |
| --- |
| Software Engineering Assignment 2 |
| High Level Design |
| Software Engineering Group 11 |

|  |
| --- |
| Version: Final, Monday March 12th 2012 |

# High Level Design

Contents

[High Level Design 1](#_Toc319309252)

[Introduction 2](#_Toc319309253)

[Architectural Design 3](#_Toc319309254)

[Classes and Objects 4](#_Toc319309255)

[Startup 4](#_Toc319309256)

[Visualiser 4](#_Toc319309257)

[Game 4](#_Toc319309258)

[Statistics 4](#_Toc319309259)

[Tournament 4](#_Toc319309260)

[Player 4](#_Toc319309261)

[Ant Brain 4](#_Toc319309262)

[State 4](#_Toc319309263)

[Ant 4](#_Toc319309264)

[Ant Verifier 4](#_Toc319309265)

[World 4](#_Toc319309266)

[Cell 4](#_Toc319309267)

[World Verifier 4](#_Toc319309268)

[Common Tactical Policies 5](#_Toc319309269)

[Programming Style 6](#_Toc319309270)

[Guidelines 6](#_Toc319309271)

[Code Example 7](#_Toc319309272)

[Requirements Cross-Reference 8](#_Toc319309273)

[Detailed Design Plan 9](#_Toc319309274)

# Introduction

Design is an iterative process, and as such it needs to be start from an abstract level and work downwards, adding more detail until a blueprint suitable for a team of programmers is ready. The high level design is the middle stage of the design in the project. Its purpose is to take the information provided by the requirements analysis team and begin to turn into something more robust. The designs listed in this document are developed from certain criteria supplied in the customer requirements, as well as our own aims in the project.

The design is split up into eight different sections which are summarised below:

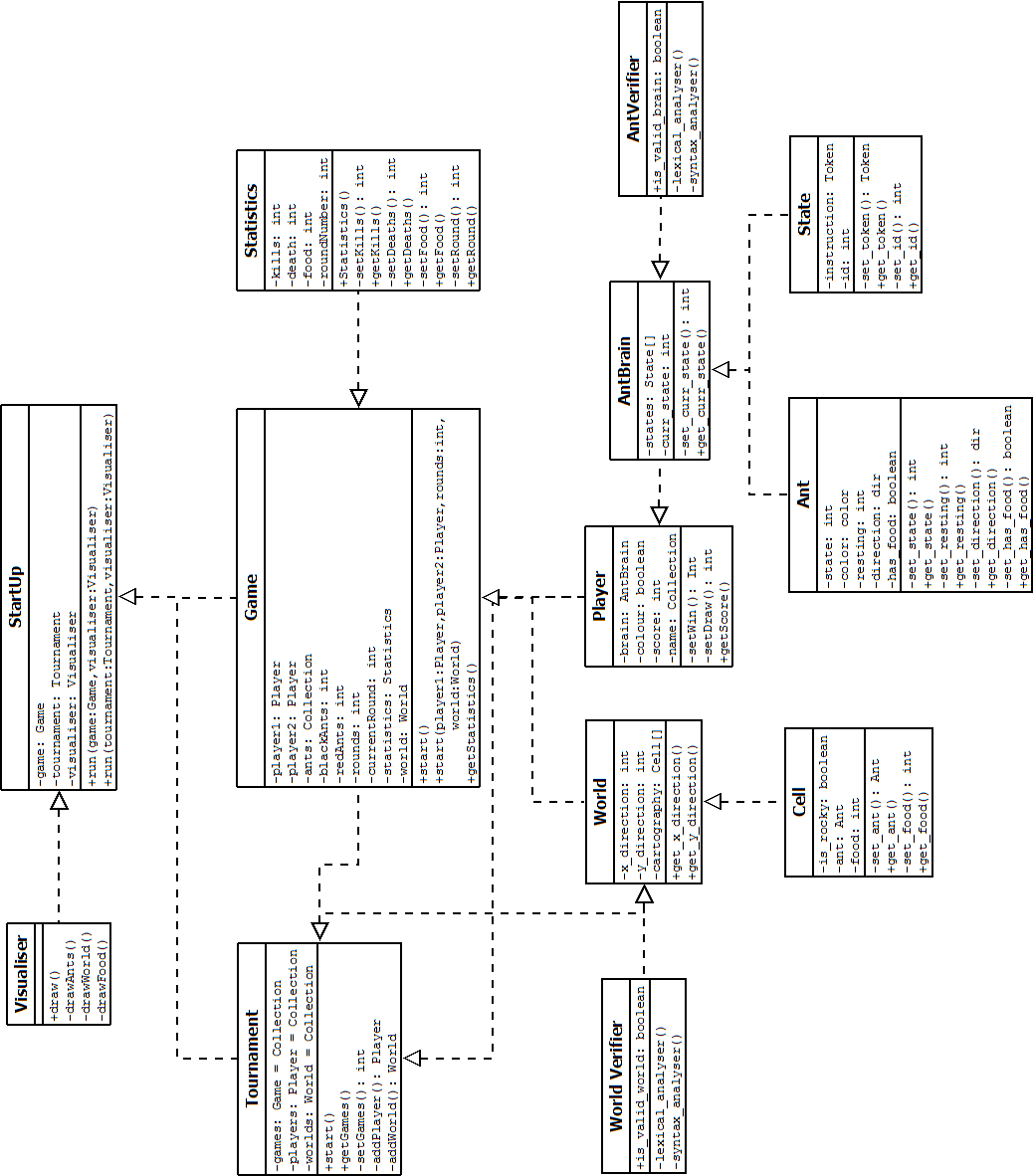
* A program that checks if an ant supplied by a player is syntactically well formed.
* A program that checks if an ant world is syntactically well formed and meets the requirements for ant worlds supplied by the player.
* A program that can visualise the ant world.
* A program that allows the generation of random but well-formed ant worlds.
* A program that allows 2 players to play.
* A program that creates and plays tournaments.
* An ant brain.

All of these are detailed with an architectural design and descriptions of each class.

This document also contains information about the programming style that will be used in the project, as well as a list of common tactical policies and a cross-reference with the requirements analysis. Finally, we have a short plan for the detailed design.

The motivation in the high level design was ultimately to create an architectural design that can be detailed and improved on in the future. As a result, every major class within the system has been identified along with appropriate methods, attributes and data types, but additional information such as data structures and most likely some refactoring will be included within the detailed design. Part of the architectural design was inspired by the “façade” data pattern which provides a simplified interface to a large body of code and classes.

# Architectural Design



In this class diagram, the classes are organised in a particular way. First of all, each class has a title (e.g. World). This is followed immediately by the classes attributes and then after that the methods are listed. The arrows designate that a class is interacting with another by using it to create objects (e.g. Tournament is creating Game objects). Each method and attribute is listed with a plus and a minus symbol. These symbols state that an attribute or method is either private (as shown with a minus) or public (as shown with a plus).

## Classes and Objects

Below is an explanation of the architecture featured on the previous page. The architecture itself was primarily inspired by the design patterns, “Model-View-Controller” (in terms of the overall structure of the design) and “Façade” (although this has been relegated mostly to the “StartUp” class).

### Startup

The purpose of this class is primarily to serve as an interface for users. It allows users to start a simple game or a tournament game. It also imports a visualiser which will display the game.

### Visualiser

This class will visualise the world and the elements within it.

### Game

This class allows a user to play a single ant game. It will also display statistics that have been gathered from the “Statistics” class.

### Statistics

The purpose of this class is to gather information on the performance of each team during a game.

### Tournament

This class creates a tournament by creating a multitude of games, players and worlds on which they can play.

### Player

This class plays the role of a single player by importing an ant brain and registering its performance in a tournament.

### Ant Brain

A representation of the ant brain during the game.

### State

A representation of the ant’s identity during the game.

### Ant

This class allows ants to perform a variety of functions within the game.

### Ant Verifier

Checks whether or not the ant brains syntax is valid

### World

This class sets the size of the world, and its overall style of cartography.

### Cell

This class creates objects which accurately represent the state of a cell within the world.

### World Verifier

This class determines whether or not a world is syntactically correct.

# Common Tactical Policies

The tactical policies can be split up into various different areas:

* The GUI; The GUI only plays a relatively small part in the overall function of the system and therefore any changes to it do not severely impact the infrastructure of the project. As such , the GUI is an element which can be altered frequently all the way up until the program itself has been finalised.
* The testing; The system of testing we will be using will not in fact impact the high level structure of the project. This is in spite of the fact that testing will be taking place frequently throughout the project life cycle. As the group are essentially satisfied with the high level design structure, there won’t be sweeping changes to its overall functionality despite how the testing occurs.
* Coding conventions; Although c ode conventions have been set out in this document, as long as programmers stick the idea of keeping code readable then there won’t be an issue with it affecting the project. On top of this, the style of code is isolated from the design and therefore has little change of causing disruption in the system.
* Algorithms; At this stage, the implementation of algorithms is the responsibility of the programming team. In the detailed level design, the nature of algorithms will be expressed but the actual implementation (for instance in pseudo-code) will not appear. As such, the implementation of algorithms does not radically affect the overall structure of the system.
* Source code and documents; The organisation of the projects documents and source code is within the hands of GitHub. As this is something everyone on the project uses to read and update documentation, and given how everyone will be using them from the same location, the organisation of these documents is essentially finalised from the moment they are uploaded. As such, this means that the decision to use GitHub does affect how the project is organised, but it isolates the system from widespread changes.

# Programming Style

As every member of the group is well acquainted with it, Java has been selected as the programming language which will be used to implement the design. We will not be deviating from the traditional Java programming style. The primary focus is to maintain the readability of the code. Below is a list of the guidelines which will be followed during the projects development, and also a small example of some code to express how these guidelines will be implemented.

## Guidelines

* Use the correct grammar for declarations;
  + Classes begin with capital letters.
  + Attributes use small letters, with new words separated by underscores.
  + Methods use small letters, with a capital letter to signify the start of a new word.
  + Acronyms to appear in small letters.
  + “Final” attributes to appear solely in capital letters.
  + Enums to use capital letters.
* Each class must be fully annotated;
  + The beginning of a class should contain an author name, the date in which it was created, and some information about the purpose of the class.
  + Attributes can be labelled with comments next to them.
  + Methods must be explained with comments preceding them, and if need be, during them.
    - Must also contain information on the input and the output that is expected in the method.
  + Packages should be labelled with lower case letters.
* Accessors and mutators should be created for every global variable.
* Classes should be organised in the following order (although deviations may occur if needed);
  + Package declarations.
  + Import statements.
  + Class creation.
  + Attribute declarations.
  + Constructors.
  + Methods.
* Methods that use control structures must use proper indentation.

## Code Example

|  |
| --- |
| package example.package;  import java.util.example;  /\*\*  \* @author Author Name  \* @data Date  \* Class description goes here.  \*/  public class Example {  static final float EXAMPLE\_VARIABLE;  private int exampleGlobalVariable;  private int UmlDiagram;  private boolean isBoolean;    public Example(){  First Statement;  if() {  second statement;  }else if{  third statement;  }  }  public void setUml(int example){  UmlDiagram = example;  }  } |

# Requirements Cross-Reference

The requirements cross-reference takes the specification set out in the requirements document and compares it to the finalise high level design.

For the most part, the overall design is not significantly different from the requirements, although there are some slight differences.

|  |  |
| --- | --- |
| Class | Differences |
| GUI | Although the high level design accounts for a GUI, it does so in a class with a different name; Visualiser. |
| High Score | The High Score class has been changed to a class called Statistics which essentially performs the same function, (i.e. declaring a winner) but has more functionality. |
| Game | Requirements specification does not mention a class to simulate a single game. The high level design does. |
| StartUp | The high level design has rearranged some aspects of the requirements and added a class which will serve mostly as an interface for users; StartUp. |
| Player | The high level design has created a single object which will be used to represent players. The requirements has not listed this, although it has listed the elements that Player will import (i.e. State, AntBrain and Ant. |
| Tournament | The high level design renames the Tournament Game class to simply Tournament, as its simpler and conveys the same meaning. |
| World Verifier | The requirements analysis does not mention the World Verification class. The high level design does. |
| Ant Verifier | The requirements analysis does not mention the World Verification class. The high level design does. |

Outside of the differences listed in the table above, there is a correlation between the class design for the requirements and the high level design, particularly in terms of overall class functionality.

# Detailed Design Plan

The detailed design plan is built in order to give the programming side of a project a full blueprint which they can begin to start implementing. In order to do this at this stage in the project, we need to take the high level design document and derive the detailed level design from it. The aim of this plan is to provide some information on how we plan on doing this. It is broken down into several stages:

1. Team Feedback: By opening up the design side of the project to scrutiny from the rest of the team, we open ourselves up to new perspectives and can include elements that were forgotten or could be improved upon.
2. Testing: We can also start to test our design by implementing information about object behaviour. This allows us to refine our design into something more concise and we can find and eliminate issues as they appear. Documenting the information will also be extremely useful for the design team when it comes to understanding how the design works.
3. Refactoring: This is where we begin to improve on the overall structure of the architecture. By doing some additional research into design patterns and our own high level design, we can break down and re-arrange the architecture, if need be. As the programming team is to begin some early attempts at implementation after the high level design is complete, it is unlikely that any major changes will occur at this point in the project, unless it is worthy of significant change.
4. More Detail: Additional detail will be added to the design. As previously mentioned, part of this will entail the introduction of information about the object behaviour. However, we also intend to include information on particular data structures that will be used, as well as elaborating on each individual method that has been listed. Algorithms will be detailed in plain English rather than pseudo-code, as this allows the programming team a greater degree of freedom when it comes to implementing the design.
5. GUI: The detailed design plan needs to include information about the GUI that will be used to overlay the system. This is likely to be done via a collection of diagrams, or screenshots of a functionless GUI for demonstration purposes. Complete explanations of each element of the GUI will also be detailed.

This is by no means an exhaustive list, and these stages may not occur entirely in order. However, they are four guidelines which the next stage of the design will revolve around.