Architecture Design

The Cave of Caerbannog

By
Team MIGI2
Taico Aerts, taerts, 4345797
Chiel Bruin, cbruin, 4368436
Bram Crielaard, bcrielaard, 4371755
Wytze Elhorst, welhorst, 4306228
Robin van der Wal, robinvanderwal, 4142918

Context project 2015-2016

FACULTY OF ELECTRICAL ENGINEERING, MATHEMATICS AND COMPUTER SCIENCE.

Instructors:	prof. dr. A. Hanjalic and dr. A. Bacchelli
Context coordinator:	dr. ir. R. Bidarra.
Teaching assistants:	S. van den Oever and J. van Schagen

May 2016

Table of contents

1 Introduction

This document will go into detail explaining how “The Cave of Caerbannog” is put together, what one would need to be able to run it, and what the main design goals behind the game are.

1.1 Design goals

In this chapter the main design goals for the final release will be discussed.

Manageability
As the game can change during development, or even after release, one of the design goals is manageability. To make the code more manageable it will be split in independent parts, which will work together to make the game run. Interfaces shall be used to make this happen. All components of the game should be interchangeable with different components which implemented the same interface.

Quality
The code must be guaranteed to be of high quality. This can be guaranteed because of the following agreements between members of the development team:

- All code, excluding the GUI, has to be tested to reach a line coverage of at least 80%. If this task is impossible an explanation will have to be given as to why this is the case.
- Git and Github will be used for all code related to the project. This ensures we can always revert to an older version if the code breaks, and that there is a single repository where all the code can be found. All pull requests will have to be reviewed and accepted by at least two members of the development team before they can be merged.
- All methods and classes have to include javadoc explaining their functionality.
- The code may not contain warnings generated by checkstyle/FindBugs/PMD. If a developer has a good reason to keep one of the warnings an explanation has to be given.
- The program will be built by a continuous integration system after every push to the online branch. Pull requests that do not build will not be accepted. Builds will fail when the code does not compile.

Reliability
The game must remain playable and not crash under normal circumstances. This goal is reached by extensive play testing and the aforementioned guarantee that the code must be of high quality.

2 Software architecture views

In this chapter the architecture of the game will be discussed. This will be done by decomposing the project into its independent subsystems, explaining which hardware is used and how it is used, and how concurrency problems have been dealt with.

2.1 Subsystem decomposition

The system is divided up into different subsystems according to the MVC pattern. This adds to our design goal of “manageability”. One of the reasons the MVC pattern is used is because multiple views will be presented on the same game state. The main reason the MVC pattern was chosen was because the development team is comfortable with this pattern, as they have used it numerous times before.

Model
The model of the game consists of all entities and the level structure of the game. These two parts are also joined together by a Game object that forms the core of the model.

View
This subsystem consist of the jMonkey engine, as required by the stakeholders.
If during development the development team realises the jMonkey engine is lacking in required features they will be added to the view subsystem.

Controller
This subsystem will be responsible for handling the user input and interacting with the different controllers. Some of these controllers also form the game thread of the currently running game.

2.1.1 Main framework

This section will explain the interaction between the controllers and the model, as this is the main framework of the game. For a more detailed overview of the structure of the program, see section 2.1.2 and see section 2.2.1 for the implementation of the web server.

Game state
Because the gameplay is divided in several separate parts we introduced GameStates.
These states are:

- Waiting - The game waits for all players to connect and choose a team.
- Running - The main game is running
- Paused - The main game is suspended.
- Ended - The game is finished but not jet back in the waiting state for a new round.

Each controller used in the program has a state that it manages. For example the WaitingController manages the waiting state. A controller also has the task to interface the renderer by providing methods to add drawables and Gui elements. An important part of adding and removing drawables is to add them to or remove them from the physics environment.

A special controller is the GameController, as this controller manages the running state and therefore the main game.

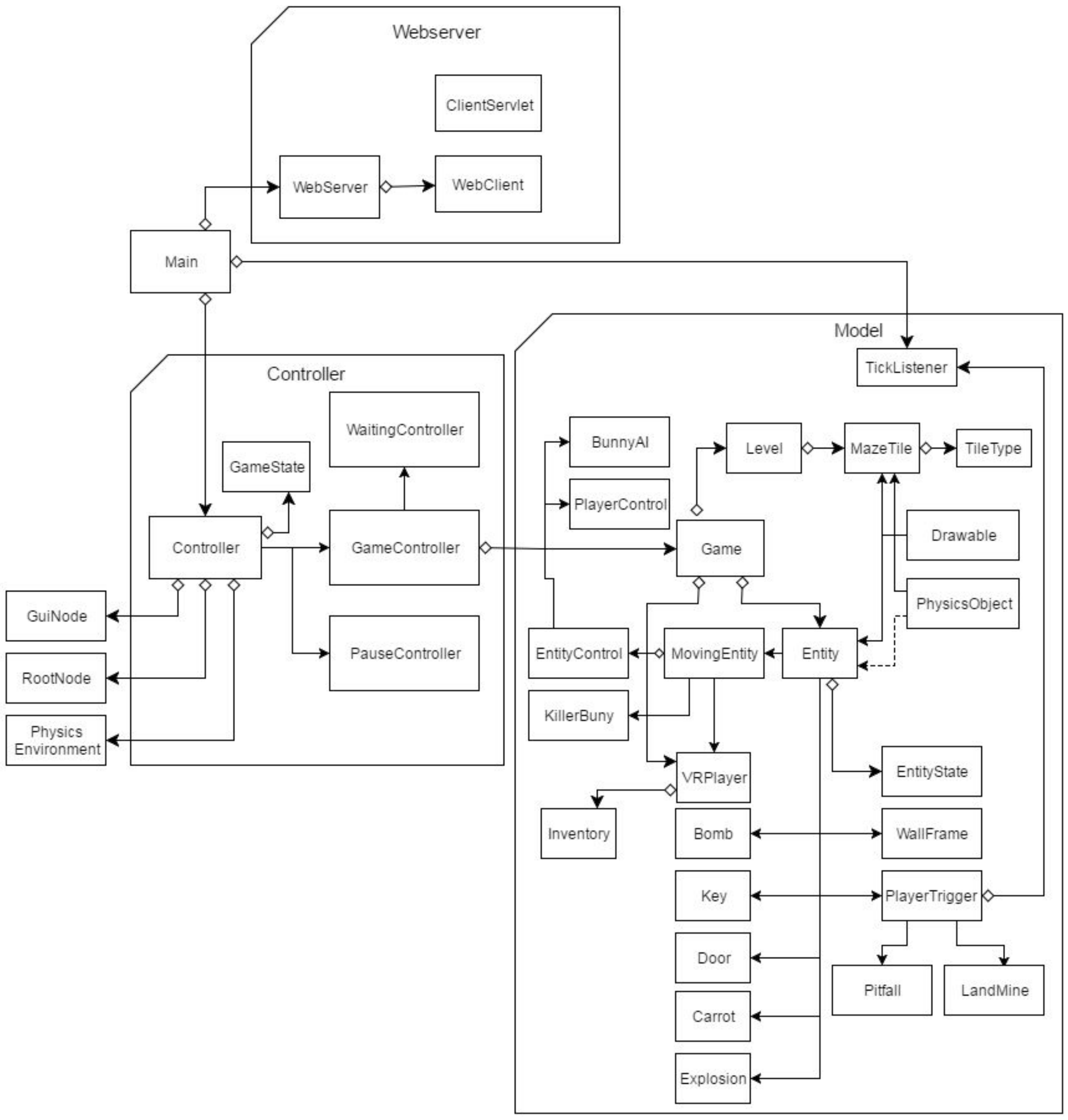
GameController
The GameController has several tasks to do for a game to run. Firstly it needs to create the game. This is done by loading a level, either from file or from a level factory and add this to a game object. Another task of the GameController is to update all entities contained in the game object. This means that all entities with state New must be added, those with state Alive must be updated and the entities with state Dead must be removed.
Updating entities is done by calling the update method on them.

Entities
Entity is the abstract class that all entities in the game inherit. This class adds the concept of entity states to the drawable interface. This state is either new, alive or dead, representing an entity that is not yet added to the renderer, is already added to it or should be removed from the game respectively. By implementing the drawable interface each entity has a spatial that can be added to the renderer. Each entity can also choose to be a collidable object by implementing the PhysicsObject interface. A list of all the entities in the game at its current state is:

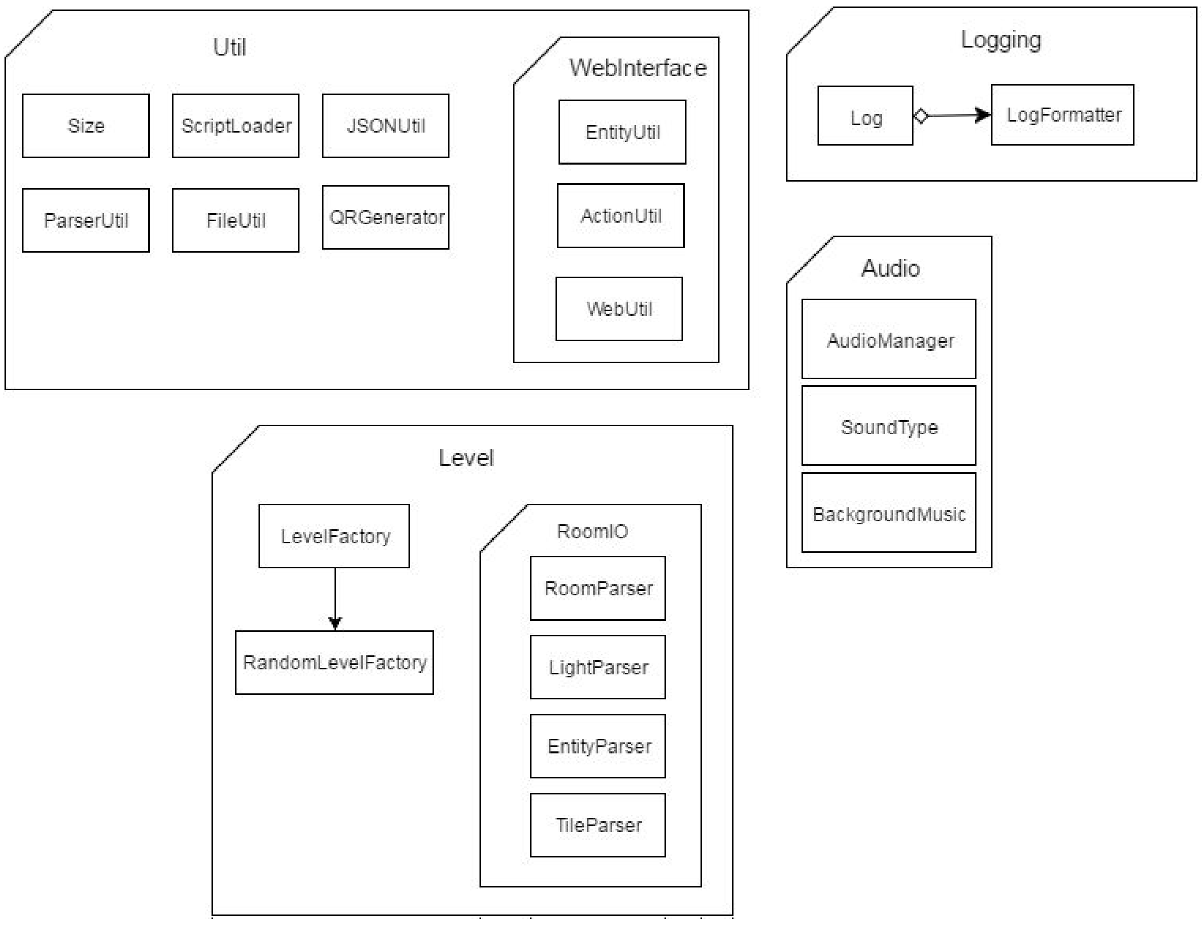
- VRPlayer
  The main player in the game. This entity has an inventory where bombs and keys could be stored. The player also has health that is reduced by taking damage.
- WallFrame
  A frame that can be placed on a wall that displays an image. This image can be used to show some information to the player.
- PlayerTrigger
  An entity that will trigger an action when the player is closer than the specified distance. This event is a TickListener and could be loaded from an external file.
- Key
  An entity that can be stored in the player’s inventory and can be used to open doors. The door can only be opened when the door has the same color as one of the keys in the player’s inventory.
- Bomb
  An entity that can be picked up and dropped by the player. A bomb will explode shortly after the player picks it up from the level.
- Door
  An entity blocking the occupying MazeTile for the player. A door can be removed/opened with a key that has the same color as the door.
- Carrot
  An entity that can be placed by elves that distracts the killer bunnies.
- LandMine
  An entity that explodes when the VR player steps on them. Can be seen by elves. Can be placed by dwarves.

- KillerBunny
  An entity that chaises the VR player around. Will prioritise Carrots over the VR player. Can be placed by dwarves
- Pitfall
  An Entity that the drops the player below the map when stepped on. Can be seen by elves. Can be placed by dwarves.
- Explosion
  Currently a growing entity that damages the player. Should be replaced with an animation instead.
- VoidPlatform
  An entity that the player can walk on. This entity can be placed in the “void” by elves to give the VRPlayer a floor to walk on.
- Treasure
  A static entity which completes the game. The VRPlayer wins the game if it comes within a certain range of this entity.
- Torch
  An entity which represents a light source.
- InvisibleWall
  An entity which represents a wall which is invisible to the VRPlayer. Players on their mobile phones can see this entity.
- DamagedWall
  An entity which represents a wall which can be blown up.

2.1.2 Overview of classes



This is a quick overview of all game related classes currently in the program. Note that this is a very shallow UML diagram, only containing inheritance and the most important components of a class. This UML diagram does not contain the utility classes, these can be found on the next page. All entities added within the last sprint have not been added to the entities package, as these aren’t 100% final yet.



These classes are utility classes used by the rest of the program. Drawing all the dependencies between the main classes of the game and the utility classes would result in an unreadable UML, thus they have been displayed separately. As the maze generation has changed a massive amount of these util classes these will be updated in the next sprint.

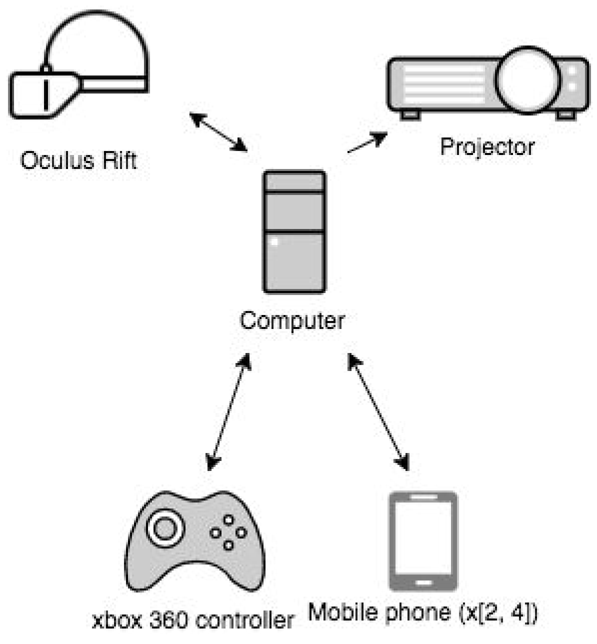
2.2 Hardware/software mapping

There are multiple hardware components which need to interact with each other. There is a single computer on which the game runs, which also acts as a server for the mobile phones. Connected to the computer are the following components:

- Oculus Rift (x1)
- Xbox 360 controller (x1)
- Mobile phone (x[2, 4])
- Projector (x1)

One player uses the Oculus Rift to view the game, and the xbox 360 controller to control the game.

Four other players use the projector to view an image of the map, and mobile phones to control the game. The mobile phones in turn display relevant information to what is currently going on and the relevant controls for the player.



2.2.1 Web Server

A large part of our game revolves around the multiplayer aspect. There will be one player that wears the oculus, and up to four players that participate in the game from their mobile phones. To communicate with these mobile phones, we use an embedded HTTP server.

The communication revolves around continuous status update requests from the clients to the server. The server will respond to these requests with relevant status information, with which the web page as seen by the clients is updated.

Communication

The communication between the server and the clients happens in the following way:

1. The webserver is started with the game
2. A client connects to server.
    a. If the game is in progress, the client is denied from participating in the game with an error message.
    b. If the game is full, that is, when four other clients have already connected, then the client is denied from participating in the game with an error message.
    c. Otherwise, the client is added to the game, and is presented with a team selection page (2). The server will send a persistent cookie to the client for further tracking. This cookie will contain an unique id for this client.
3. The client selects a team
    a. If the client does not have a valid unique id, then they are handled as in phase 1.
    b. Otherwise, the server changes the team of the client to the selected team, and sends a confirmation.
    c. The client starts a timer to send update requests on a set interval of 1 second.
4. When the client receives in a status update that the game has started
    a. The client requests the map from the server.
    b. The server sends the map encoded as a json object.
    c. The client's web page is updated to show the map of the game, state (5).
5. When the client receives a status update while the game is in progress
    a. The client's web page is updated to reflect the new changes.
6. When the client receives in a status update that the game has ended
    a. The client's web page is updated to show the end game statistics
    b. The client stops the timer, and no longer sends update requests to the server.

Status updates

Status updates are sent as json objects from the server. Status updates include the following information:

- The current game state (WAITING, STARTED, PAUSED, ENDED)
- The client's team (ELVES, DWARFS, NONE)
- If the client is in the ELVES team:
    - A list of all tile locations that have been explored.
    - A list of all entities on the map that an elf should be able to see.
- If the player is in the DWARFS team:
    - A list of all entities on the map that a dwarf should be able to see.

Client tracking
Since HTTP is a stateless protocol, and because ip addresses can change, we use cookies to track which request comes from which client. When a client connects to the game, and is allowed to join, the server sends a cookie with a unique id to the client. The client will in turn send this cookie to the server with each of their requests (this is done automatically by the client's browser).

Map
Below is an image that shows what a client "sees", as it is at the end of sprint 3. Please keep in mind that this view will likely change when more of the functionality is implemented.

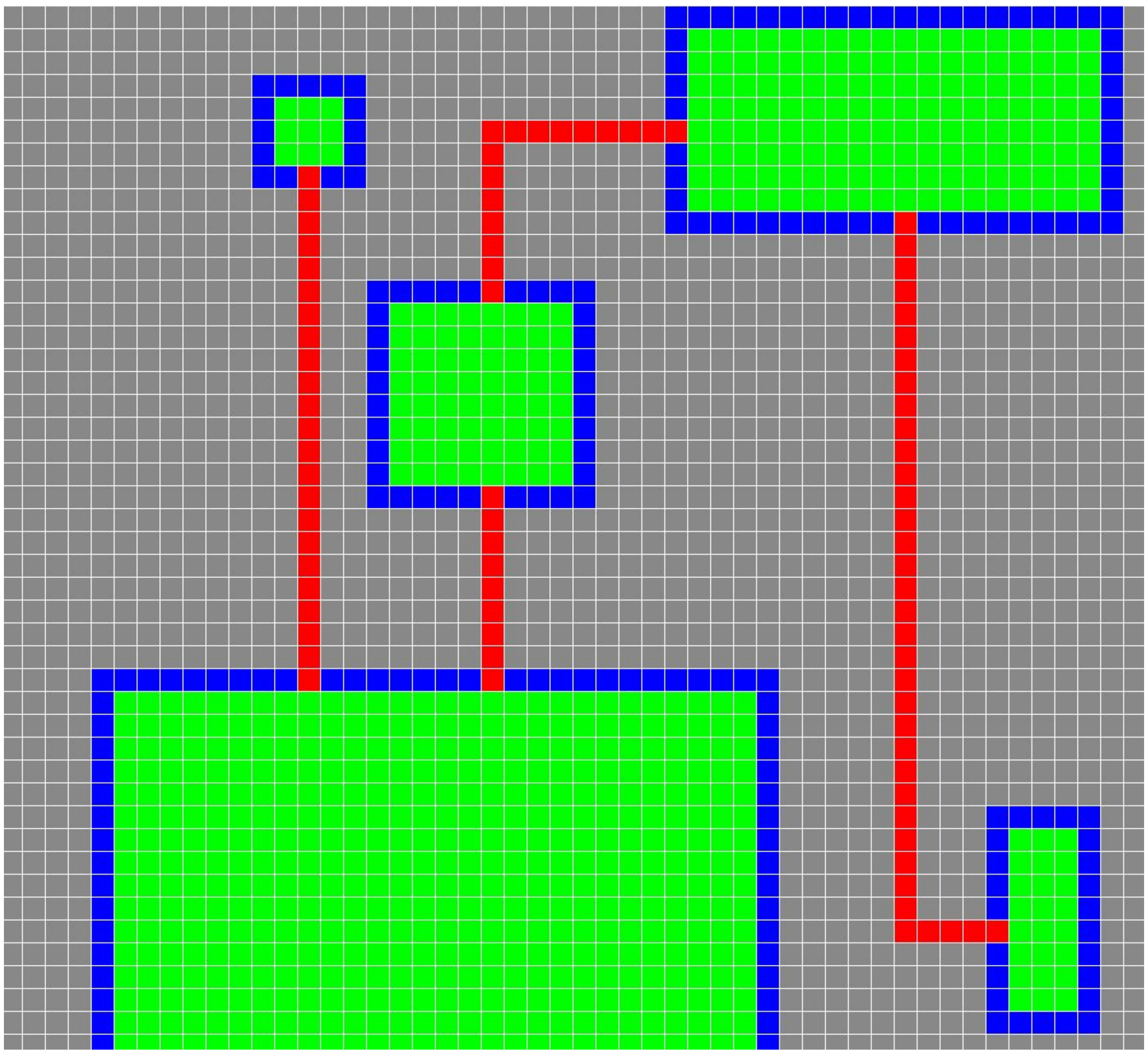


Figure 2: The map view of a web client.

2.3 Persistent data management

We will use persistent data management for two aspects of our game:

- The building blocks for the different levels, the rooms.
  The rooms will be stored in xml/json files. These files will not change during gameplay, but changes can be made during development and playtesting.
- The game settings.
  Settings are stored in a plain text format (yml, xml, json or properties), and can be updated by the player in the menu.

We do not use any kind of Database Management System, since we do not have large amounts of data we have to store, and updates in the data are scarce.

2.4 Concurrency

As there are multiple people playing the game at the same time concurrency is a something we have to consider.

The webserver is currently the only multithreaded class. Every request to the webserver is handled in its own thread, to guarantee a quick response. (Do note that threads are reused where possible)

Currently our concurrency problems are mostly fixed by using concurrent data types when possible. We also make extensive use of iterators, so we can remove items from lists/sets without the game breaking.

2.5 Maze generation

The current algorithm for the maze generation is not the final algorithm which will be used, so do keep that in mind.

The algorithm is extremely simple and works as follows:

1. Place a room randomly in an unoccupied space.
2. Create a corridor to the previously placed room, if there is one.
3. Repeat until the requested amount of rooms have been placed.

2.6 Dependencies

Below is a list of dependencies, and the reason we need them/what they are used for.

Dependency	**Reason**
Lombok	Adds convenient short ways to create try catch blocks (SneakyThrows), getters, setters, constructors, etcetera. Simply a useful developer's tool.
Jetty	Embedded web server, which is used for the web server of our game.
JSON	JSON parser/writer, to easily work with the JSON format, for both room loading and the web server.
Kenglxn QRGen	For generating QR codes.

3 Glossary

Branch
A split from the main code to which people can push changes and additions to the code.

Checkstyle
A static analysis tool which determines if the code complies with the style rules agreed upon by a development team.

FindBugs
A static analysis tool which detects possible bugs in java code.

Git
A version control system used for software development.

Github
A web-based git repository hosting.

GUI
Graphical User Interface, an interface which allows users to interact with electronic devices.

Merge
Blending of the code between two different branches.

MVC
Stands for “Model-view-controller”, A software design pattern for implementing user interfaces on computers. It splits the code into three different parts which all work together but could be interchanged for different parts.

PMD
A static analysis tool which uses a rule-set to determine if the code is erroneous.

Pull request
A request to add or change code and deliverables.

Push
Uploading code to a git repository.