# SDD - System Design Document



**Version** 2.0

**Date** 31/05 - 2015


**Author**
Lars Niklasson
Anton Ingvarsson
Daniel Sunnerberg
Victor Christoffersson


# 1 Introduction

## 1.1 Design Goals

The architecture of the application should be as loosely coupled as possible to allow smooth exchange of frameworks/models. Models should be testable and tested as much as possible to ensure a high code quality and prevent future bugs.

The application will not be designed in a way that allows the game engine to be changed, since that would only result in overly generic code without any guaranteed profit. This since most game engines has different architectures.

### 1.2 Definitions, acronyms and abbreviations

Map - A driving track with surroundings

GUI - Graphical user interface

Java - Platform independent programming language.

2D - Two dimensional graphics

Game mode - Different ways of playing the game, with different goals

LibGDX - Game engine for java.

FPS (Frames Per Second) - How many times the screen is drawn upon per second

Time-Trial - A gamemode where the aim is to complete a lap around the track as fast as possible, trying to beat the fastest recorded time.

JAR - A runnable version of the program code, containing eventual resources

ZIP - A compressed archive containing a set of files

README - Text file containing solutions to common problems

JRE - Java Runtime Environment, software needed to run Java application.

Tiled - Map Editor

Box2D - Physics engine for LibGDX

AI - Artificial intelligence

MVC (Model View Controller) - A programming design pattern

# 2 System design

### 2.1 Overview

The game will run in the open-source engine LibGDX, using a slightly modified MVC model.

### 2.1.1 - Main architecture in LibGDX

The main class is called _2DRacingGame and extends LibGDX's "Game" class.
from the Game-class you can set different screens. The game has a number of different screens, the most important ones being GameScreen and MainMenuScreen. All the screens implements LibGDX's "Screen"-interface. To change what screen is being shown the Game#setScreen method is called from _2DRacingGame.

An explicit game loop does not exist due to the way LibGDX works. However one can regard each screen's Screen#render methods as different game loops.

### 2.1.2 - The "Game Loop"

Updating the state of the game objects and drawing them to the screen will be handled in mainly 2 classes. GameWorld, which holds all the objects, and GameRenderer which has a reference to a GameWorld object and draws everything to the screen.

The "loop" takes place in GameScreen's render method and is made up by 2 parts - (1) telling the GameWorld to run the physics simulation, take care of user input and update the game objects accordingly and (2) telling the GameRenderer to draw everything to the screen.

### 2.1.3 - Physics simulation

The application will be using the library Box2D which is used by LibGDX for physics simulation. Box2D simulates bodies linked to each other in the gameworld and how forces affect them. Specifically the vehicles and different ground materials (see 2.1.4).

### 2.1.4 - Vehicle/car/tire

The most important aspect of the actual gameplay is the vehicles, how they are built and how they are being controlled.

All the vehicles are derived from a common interface "Vehicle", which holds information such as position and direction of the vehicle. It also holds a reference to a "SteeringSystem" (see 2.1.5), which handles how to vehicle is controlled. The vehicles actually implemented in the game are all made up by one Box2D-body which is then linked to one or more Wheel-objects.

### 2.1.5 - Steering-systems

There are two types of steering-systems in the game, one that lets the user control the vehicle and one that lets the computer (AI) control the vehicle.

In the user case, each update, input from the user is checked by polling. Appropriate forces are then applied to the vehicles wheels, which makes the vehicle accelerate, turn, etc.

In the AI case, the system calculates the right direction and position to move to, based on a series of waypoints the vehicle follows, and applies appropriate forces to the body and wheels.

## 2.2 Software decomposition

### 2.2.1 General

Will be split up in following modules:

- AI - package for early AI-development, only in use for an easter-egg
- controllers - controllers for the classes - controller parts for MVC
- gameModes - game modes that defines the way the game is played
- vehicle - vehicles, wheels, and steering-systems are handled here
- helperClasses - utils package for help with math, box2D and scaling shapes
- models - contains models for player, vehicles, settings and scoreboard. Note: there all model-like classes in other packages as well, to avoid a huge model-package
- screens - viewparts of the MVC, contains the  in-game screen and all the menu screens
- game - contains the gameworld and gamerenderer. The core of the gameplay. (not menus)
- persistence - logic for saving class-instances to disc
- map - handles loading maps and creating mapobjects

### 2.2.2 Decomposition into subsystems

The only fully independent subsystem is the IO-classes, located in the persistence package.

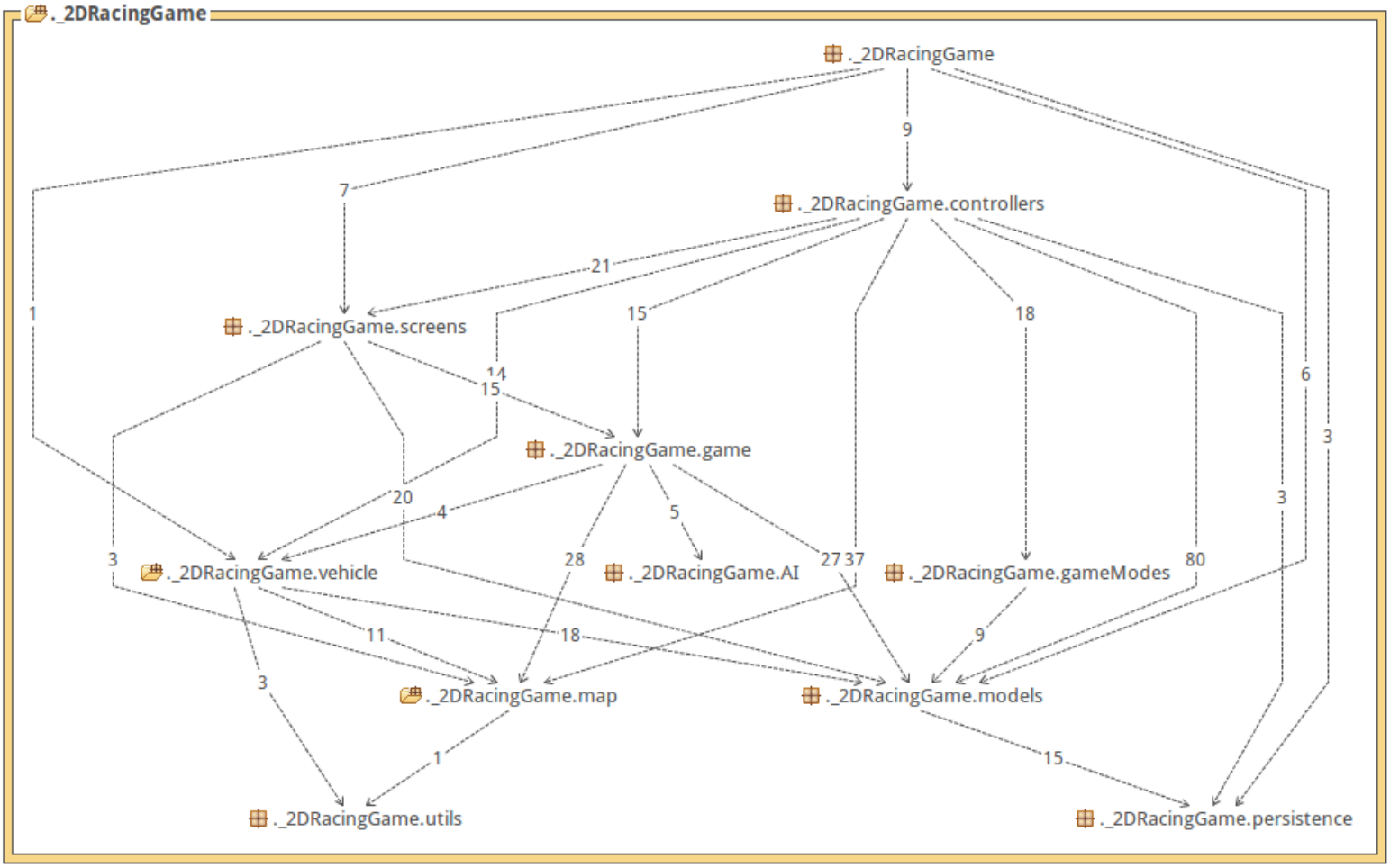
### 2.2.3 Layering

N/A

### 2.2.4 Dependency analysis

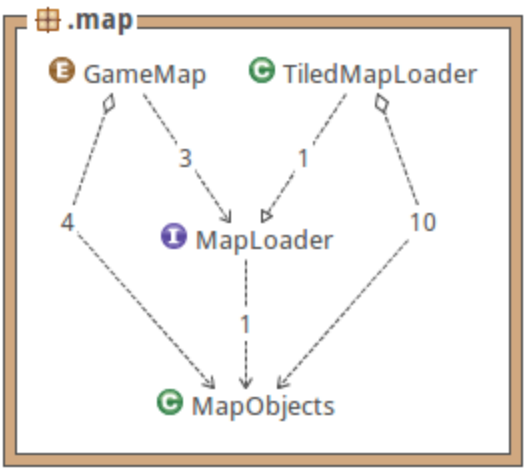
Dependencies are shown in the figures. There is no circular dependencies

## Whole project

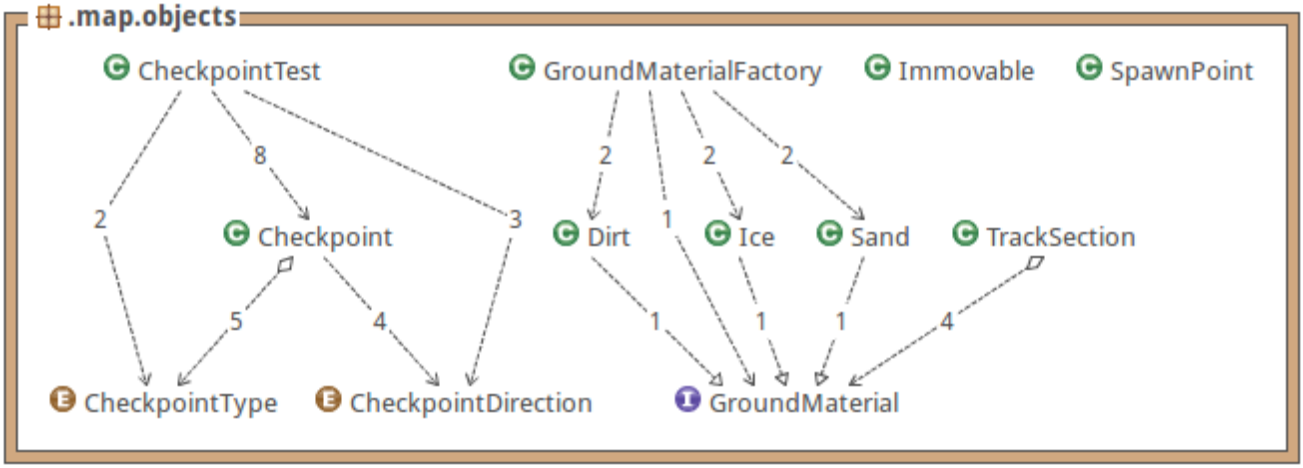



## Vehicle

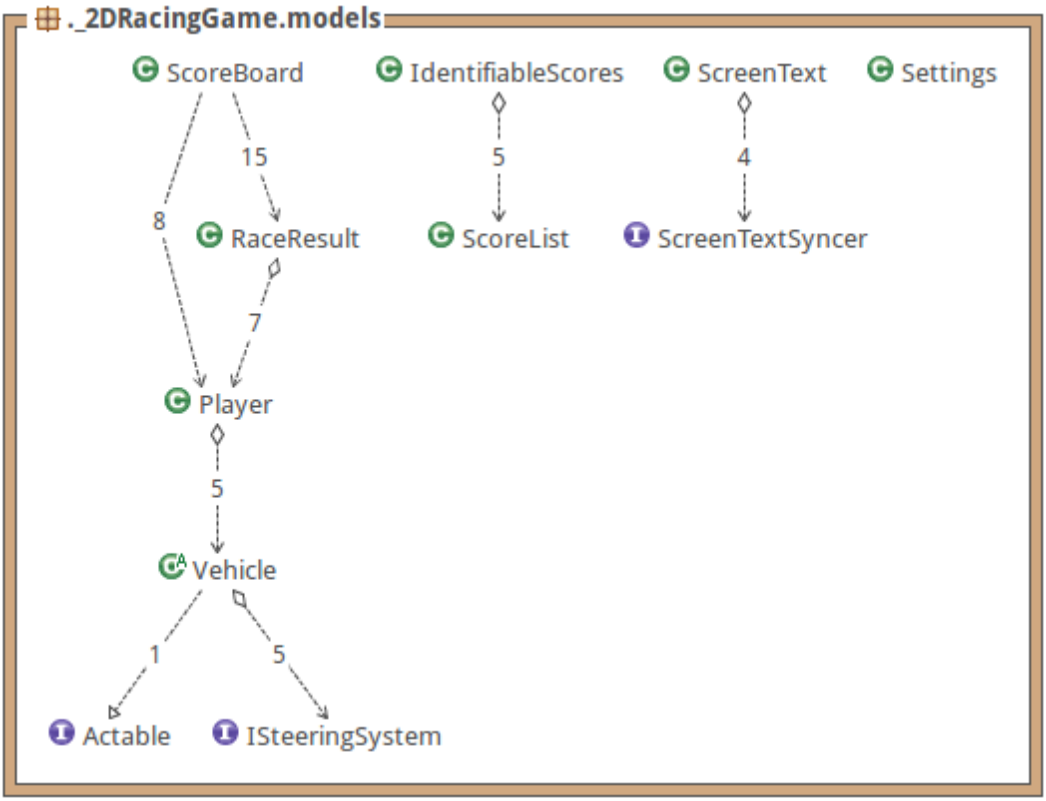
## Map




## Mapobjects




## Models




## 2.3 Concurrency issues

Playing a multiplayer game, multiple actions are supposed to happen (somewhat) concurrently. Our vehicle might be moved at the same time as the opponents. This is, thankfully, natively supported through LibGDX without changing anything. Actors and bodies (and similar) can be altered concurrently. The networking is fully handled by AppWarp, which is fully synchronized as well.

## 2.4 Persistent data management

Persistent data will be stored as serialized instances of a class in JSON-format. The storage location will be dependant of the device running the application, and is therefore handled by LibGDX and its IO-framework. Google's GSON will be responsible for serializing.

Most data will not be persistent, except for:

- Scores unique for the map and game mode
- Game settings

## 2.5 Access control and security

None. The multiplayer-server will completely trust the application to provide honest game updates.

### 2.6 Boundary conditions

NA. Game launched and exited as normal desktop application.

# 3 References

Badlogic Games. (2015). *LibGDX*

http://LibGDX.badlogicgames.com/

Box2D (2015). *A 2D physics engine for games*

http://box2d.org/