

System design document for the Challenge Accepted project (SDD)

| | |
|--|----------|
| 1. INTRODUCTION | 1 |
| 1.1. DESIGN GOALS | 1 |
| 1.2. DEFINITIONS, ACRONYMS AND ABBREVIATIONS | 1 |
| 2. SYSTEM DESIGN | 2 |
| 2.1. OVERVIEW | 2 |
| 2.1.1. <i>Model functionality</i> | 2 |
| 2.1.2. <i>Event handling</i> | 2 |
| 2.1.3. <i>Rules</i> | 2 |
| 2.2. SOFTWARE DECOMPOSITION | 2 |
| 2.2.1. <i>General</i> | 2 |
| 2.2.2. <i>Layering</i> | 4 |
| 2.2.3. <i>Dependency analysis</i> | 4 |
| 2.3. CONCURRENCY ISSUES | 4 |
| 2.4. PERSISTENT DATA MANAGEMENT | 4 |
| 2.5. ACCESS CONTROL AND SECURITY | 5 |
| 2.6. BOUNDARY CONDITIONS | 5 |
| 3. REFERENCES | 5 |

Version: 2.0

Date: 2013-05-26

Author: Cecilia Edwall, Isabelle Frölich, Johan Gustavsson, Madeleine Appert

This version overrides all previous versions.

1. Introduction

1.1. Design goals

Our construction of the Challenge Accepted is tightly constructed because nothing changes in our game. The majority is hard-coded to make sure that we don't allow letting the game do any improvisation.

For usability see RAD.

1.2. Definitions, acronyms and abbreviations

- GUI, graphical user interface , the visual part of the application
- Java, platform independent programming language.
- JRE, the Java Run time Environment. Additional software needed to run an Java application.
- Host, the computer that the game will run on.
- Team, a team consists of at least two people but can contain as many as you like. There has to be at least two teams to compete and a maximum of 8 teams is allowed.
- Mission, when the team tries to answer as many questions or complete as many tasks as they have betted they could do.
- Turn, from when another teams is done with their mission til the current team is done with their mission. A team can only act during their turn (bet, do missions, change card and so on).

- Normal Turn, the turn when a team stands on a tile with a category.
- Challenge, the turn when a team stands on a white tile and will compete with another team.
- MVC, model view controller pattern.
- Cha, an abbreviation for Challenge Accepted.

2. System design

2.1. Overview

We use Javas framework, it isn't the best to use when you build games, but we have not created a complicated game so it works for our project.

Every event goes through our eventbus so our model and view aren't connected at all except through the event package.

2.1.1 Model functionality

The functionality is pretty straight forward and is easy to get an oversight of. The different turns are modeled so that they have a common abstract class to easily get an overview of their common functionality.

2.1.2 Event handling

All events go through our eventbus to get continuous updates during the game when the teams interact with the playingfield.

2.1.3 Rules

The rules are fixed and listed in the application for the teams to see during any part of the game.

2.2. Software decomposition

2.2.1. General

The application is decomposed to the following modules (see Figure 1):

- cha, is our main package.
- cha.gui is where all the views is designed.
- cha.domain has all model code.
- cha.event containing our events, for example, our eventbus is in that package.
- cha.domain in the test package wich includes all of the different test classes for the domain package.

We have used MVC to build our program to make it easy to do changes in the GUI without make changes in the model, you can see the packages and the dependencies in figure 1.

In figure 2 can you see the class diagram to see how the different classes in the domain interacts.

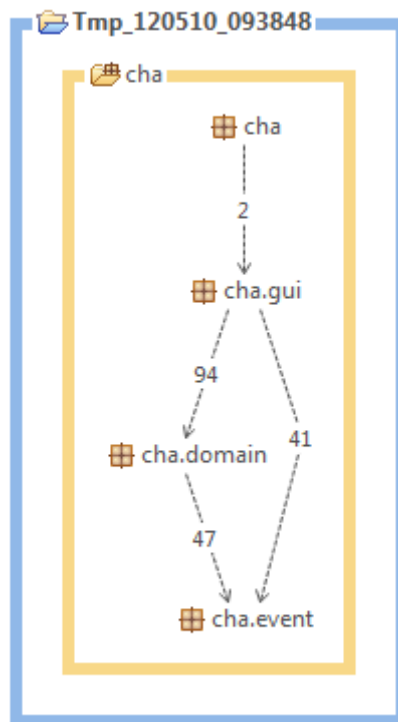


Figure 1. Layering and dependency analysis with STAN

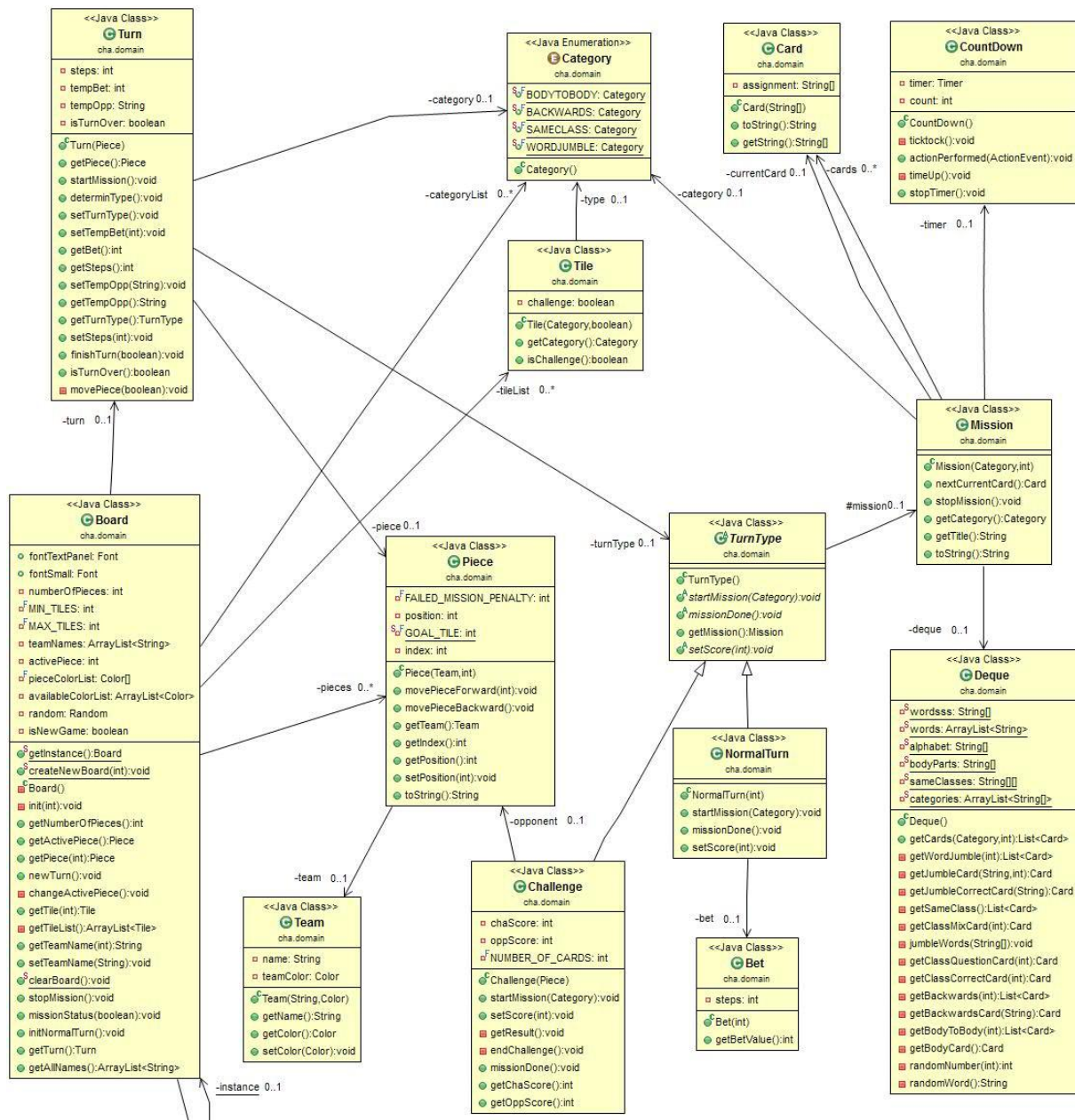


Figure 2. Class diagram for the more interesting classes

2.2.2. Layering

The layering is indicated in figure 1.

2.2.3. Dependency analysis

See figure 1. As you can see there are no circular dependencies.

2.3. Concurrency issues

N/A. It's a single threaded application. The Swing event thread will handle everything. For possible increased response there could be background threads. This will not raise any concurrency issues.

2.4. Persistent data management

N/A.

2.5. Access control and security

We have used the singleton pattern to make sure that you only have one instance of a game at a time. That way the instance can only be accessed by a certain method.

2.6. Boundary conditions

N/A.

3. References

Board game, "upp till bevis": <http://www.braspel.com/?id=317>

MVC: <http://www.braspel.com/?id=317>