System design document for the Challenge Accepted project (SDD)

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 Rules
The rules are fixed and listed in the application for the teams to see during any part of the
game.
2.1.2 Event handing
All events go through our eventbus to get continuous updates during the game when the teams
interact with the playingfield.

2.2. Software decomposition
2.2.1. General
The application is decomposed to the following modules (see Figure 1):
o cha, is our main package.
o cha.gui is where all the views is designed.
o cha.domain has all model code.
o cha.event containing our events, for example, our eventbus is in that
package.
O 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 changes in the
GUI without make changes in the model.
In figure 2 can you see our design model, it's not so different from the domain
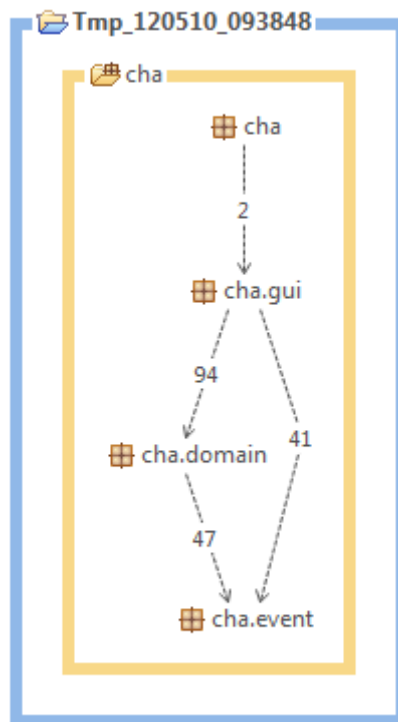model.

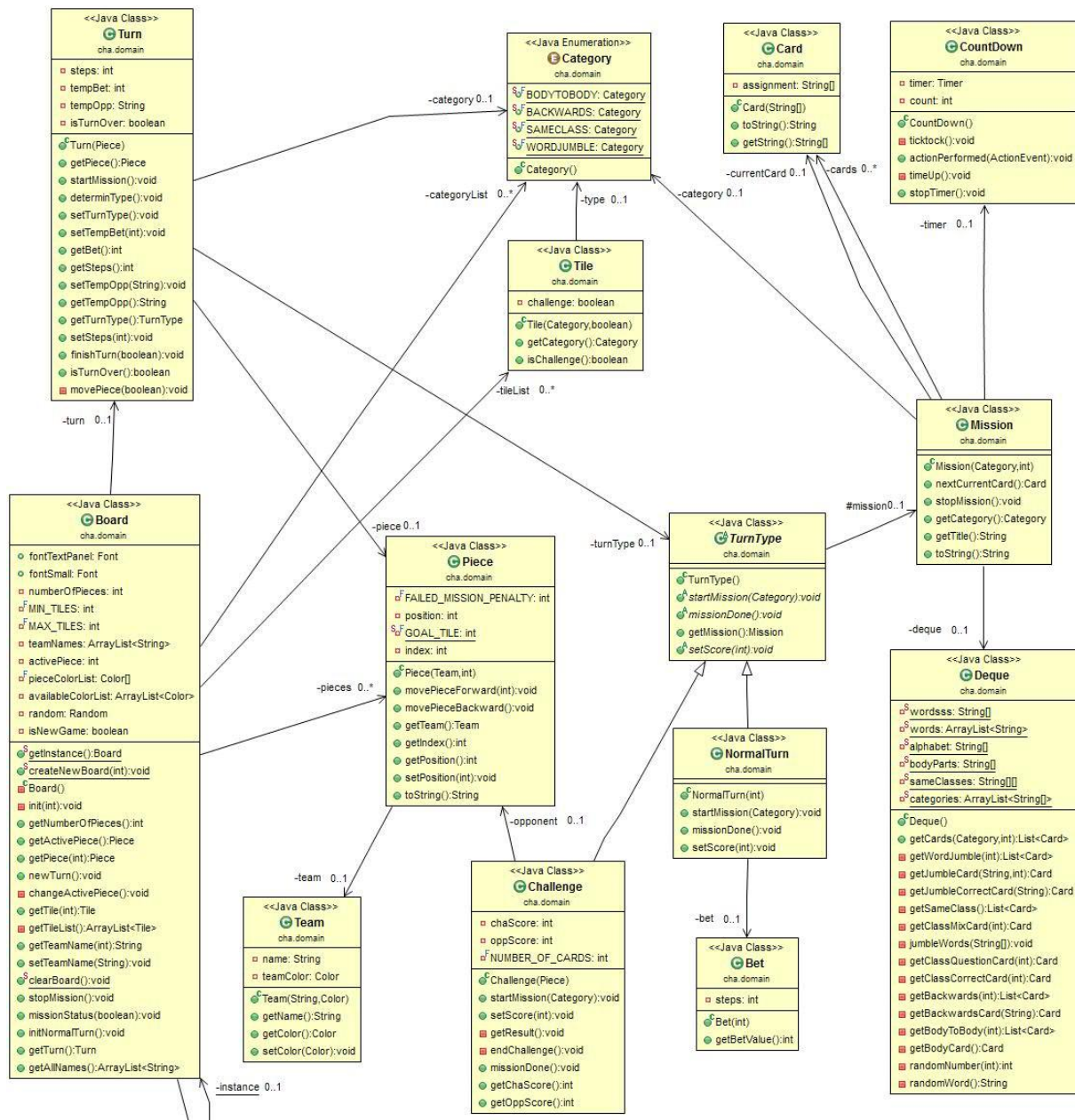Figure 1. Layering and dependency analysis with STAN

Turn — `<<Java Class>>` — cha.domain
- steps: int
- tempBet: int
- tempOpp: String
- isTurnOver: boolean
- Turn(Piece)
- getPiece():Piece
- startMission:void
- determinType():void
- setTurnType():void
- setTempBet(int):void
- getBet():int
- getSteps():int
- setTempOpp(String):void
- getTempOpp():String
- getTurnType():TurnType
- setSteps(int):void
- finishTurn(boolean):void
- isTurnOver():boolean
- movePiece(boolean):void

Category — `<<Java Enumeration>>` — cha.domain
- BODYTOBODY: Category
- BACKWARDS: Category
- SAMECLASS: Category
- WORDJUMBLE: Category
- Category()

Card — `<<Java Class>>` — cha.domain
- assignment: String[]
- Card(String[])
- toString():String
- getString():String[]

CountDown — `<<Java Class>>` — cha.domain
- timer: Timer
- count: int
- CountDown()
- ticktock():void
- actionPerformed(ActionEvent):void
- timeUp():void
- stopTimer():void

Tile — `<<Java Class>>` — cha.domain
- challenge: boolean
- Tile(Category,boolean)
- getCategory():Category
- isChallenge():boolean

Board — `<<Java Class>>` — cha.domain
- fontTextPanel: Font
- fontSmall: Font
- numberOfPieces: int
- MIN_TILES: int
- MAX_TILES: int
- teamNames: ArrayList<String>
- activePiece: int
- pieceColorList: Color[]
- availableColorList: ArrayList<Color>
- random: Random
- isNewGame: boolean
- getInstance():Board
- createNewBoard(int):void
- Board()
- init(int):void
- getNumberOfPieces():int
- getActivePiece():Piece
- getPiece(int):Piece
- newTurn():void
- changeActivePiece():void
- getTile(int):Tile
- getTileList():ArrayList<Tile>
- getTeamName(int):String
- setTeamName(String):void
- clearBoard():void
- stopMission():void
- missionStatus(boolean):void
- initNormalTurn():void
- getTurn():Turn
- getAllNames():ArrayList<String>

Piece — `<<Java Class>>` — cha.domain
- FAILED_MISSION_PENALTY: int
- position: int
- GOAL_TILE: int
- index: int
- Piece(Team,int)
- movePieceForward(int):void
- movePieceBackward():void
- getTeam():Team
- getIndex():int
- getPosition():int
- setPosition(int):void
- toString():String

Team — `<<Java Class>>` — cha.domain
- name: String
- teamColor: Color
- Team(String,Color)
- getName():String
- getColor():Color
- setColor(Color):void

TurnType — `<<Java Class>>` — cha.domain
- TurnType()
- startMission(Category):void
- missionDone():void
- getMission():Mission
- setScore(int):void

NormalTurn — `<<Java Class>>` — cha.domain
- NormalTurn(int)
- startMission(Category):void
- missionDone():void
- setScore(int):void

Challenge — `<<Java Class>>` — cha.domain
- chaScore: int
- oppScore: int
- NUMBER_OF_CARDS: int
- Challenge(Piece)
- startMission(Category):void
- setScore(int):void
- getResult():void
- endChallenge():void
- missionDone():void
- getChaScore():int
- getOppScore():int

Bet — `<<Java Class>>` — cha.domain
- steps: int
- Bet(int)
- getBetValue():int

Mission — `<<Java Class>>` — cha.domain
- Mission(Category,int)
- nextCurrentCard():Card
- stopMission():void
- getCategory():Category
- getTitle():String
- toString():String

Deque — `<<Java Class>>` — cha.domain
- wordsss: String[]
- words: ArrayList<String>
- alphabet: String[]
- bodyParts: String[]
- sameClasses: String[][]
- categories: ArrayList<String[]>
- Deque()
- getCards(Category,int):List<Card>
- getWordJumble(int):List<Card>
- getJumbleCard(String,int):Card
- getJumbleCorrectCard(String):Card
- getSameClass():List<Card>
- getClassMixCard(int):Card
- jumbleWords(String[]):void
- getClassQuestionCard(int):Card
- getClassCorrectCard(int):Card
- getBackwards(int):List<Card>
- getBackwardsCard(String):Card
- getBodyToBody(int):List<Card>
- getBodyCard():Card
- randomNumber(int):int
- randomWord():String

Association labels: -category 0..1, -categoryList 0..*, -type 0..1, -category 0..1, -currentCard 0..1, -cards 0..*, -timer 0..1, -turn 0..1, -tileList 0..*, -piece 0..1, -turnType 0..1, #mission 0..1, -deque 0..1, -pieces 0..*, -team 0..1, -opponent 0..1, -bet 0..1, -instance 0..1

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 are there 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
N/A. Our game is way to simple, so we don't need to have access control or security.
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