System Design Document

for C.R.A.P. the game

Version: 2.0
Date: 2016-05-29

Authors:
Andrea Buchholz
Rebecca Finne
Maija Happonen
Lisa Larsson

This version overrides all previous versions

Table of Content

1 Introduction

1.1 Design goals

The application is divided with clearly separated classes and packages so it is easy to know which classes to test. The application is also general, that makes it easy to add/remove models and frameworks quite easily. The application currently loads tmx-files as maps, made in the program Tiled (Tiled Map Editor, 2016), therefore the landscape of the game could easily be switched to a different setting than Chalmers campus Johanneberg. Creating maps in this tool also gives the possibility of adding multiple more types of specific objectlayers.

1.2 Definitions, acronyms and abbreviations

MVC - Model-View-Controller. A designpattern where the controller knows about the model and the view, the view only knows about the model, and the model does not know about either the controller or the view..
libGDX - Library for Gradle projects.
Gradle - Special kind of project helping with compiling etc.
Tiled map - A map build up by different tiles.
Mascot - The characters to be caught in the game.
Tmx-files - Type of file that tiled maps are saved as.
Collision - When the player object intersects an object in a specific objectlayer on the map depending on the player's and object's position, width and height.
Interaction - Talking to a non player character, NPC, in the game.

2 System design

2.1 Overview

The code will be built with a MVC design. Each model has its own controller and view if it is more complex and necessary, otherwise the GameController, CRAP and GameView takes care of the major functionality in the game mode.

2.2 Software decomposition

2.2.1 General

The program will, when possible, have one "main class" from each section (model, view and controller). That all the other classes will go through before something is changing or printed out.
The StateController controls the entire application according to which state that is active.
Depending on the state - the view and the controller will be updated accordingly.
The controller and view classes will be using methods from the frame work libGDX and the model will not use any external libraries.

Inputs from the user will be handled in the GameController-class and some other controller classes like HowToPlayController and MenuController. LibGDX input processor will be used to notify the program that something in typed on the keyboard.
The text for the mascots and carachters will be stored in a model class and will be accessed by the InteractionView and GameView. The mascots and other characters will have their own questioins and information.

Class Diagrams can be found in appendix A.

2.2.2 Decomposition into subsystems

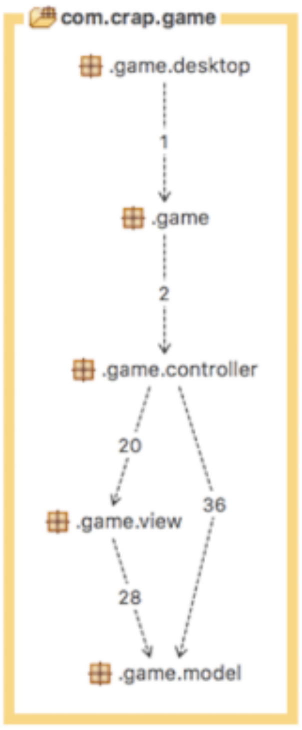
Not applicable.

2.2.3 Layering

When interacting a new frame is loaded on top of the other frame and the game will be paused during the interaction. However, the interaction-frame is always disposed when closed and created as a new frame if interaction occurs again. The StateController is handling the different frames which creates the layers.

2.2.4 Dependency analysis

Dependencies from the framework libGDX are used in the view and controller classes. The connection between the model, view and controller are as shown below.



2.3 Concurrency issues

Render-methods is called almost simultaniously in view and controller. Also, when checking collision in CollisionController both new world tiles, slow tiles and collision tiles are checked at the same time.

2.4 Persistent data management

The tmx-files and other files used in the game will be stored in a folder “assets” which is connected in the desktop and core through the Gradle dependencies.

2.5 Access control and security

Not applicable.

2.6 Boundary conditions

Not applicable.

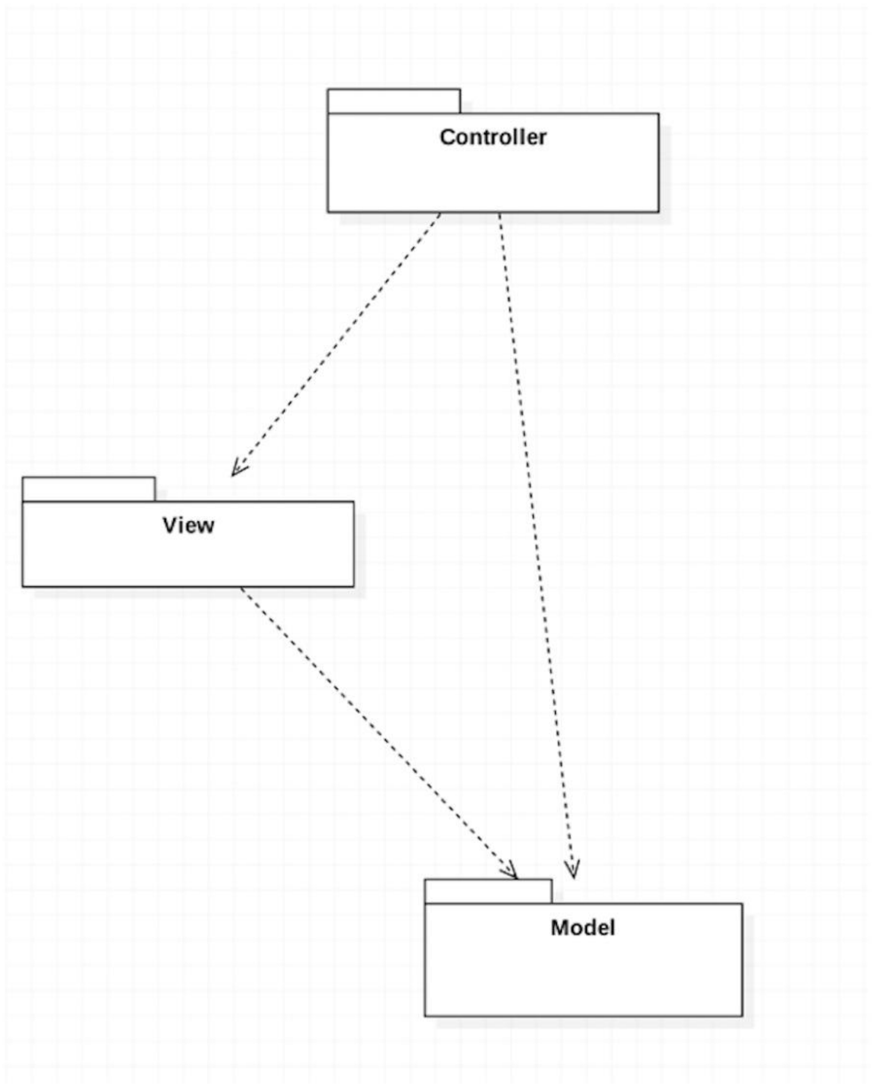
References

Tiled Map Editor. (2016). *Tiled Map Editor*. [online] Available at: http://www.mapeditor.org/ [Accessed 19 May 2016].

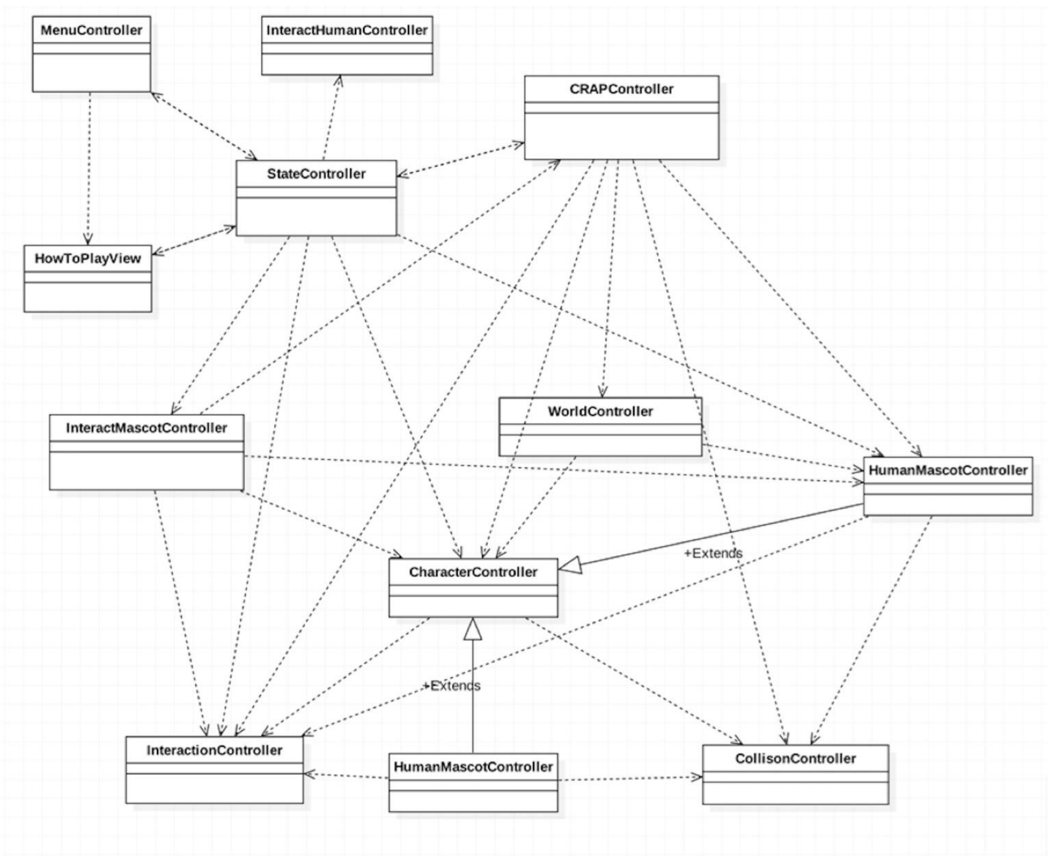
Appendix

Appendix A

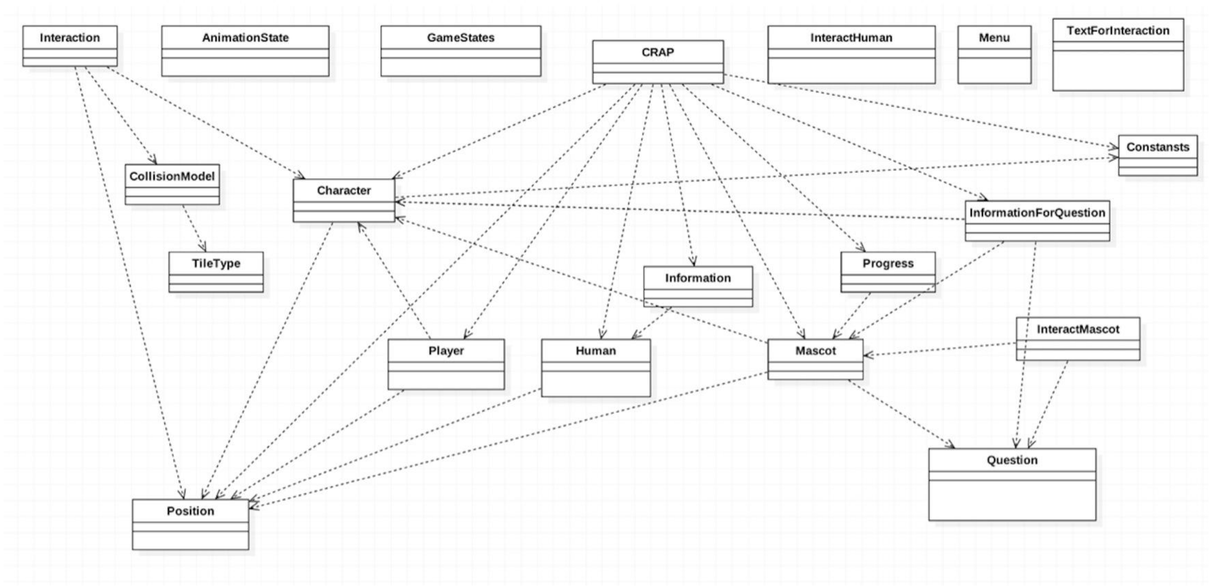
Class diagrams for model, view, controller and all of them together.



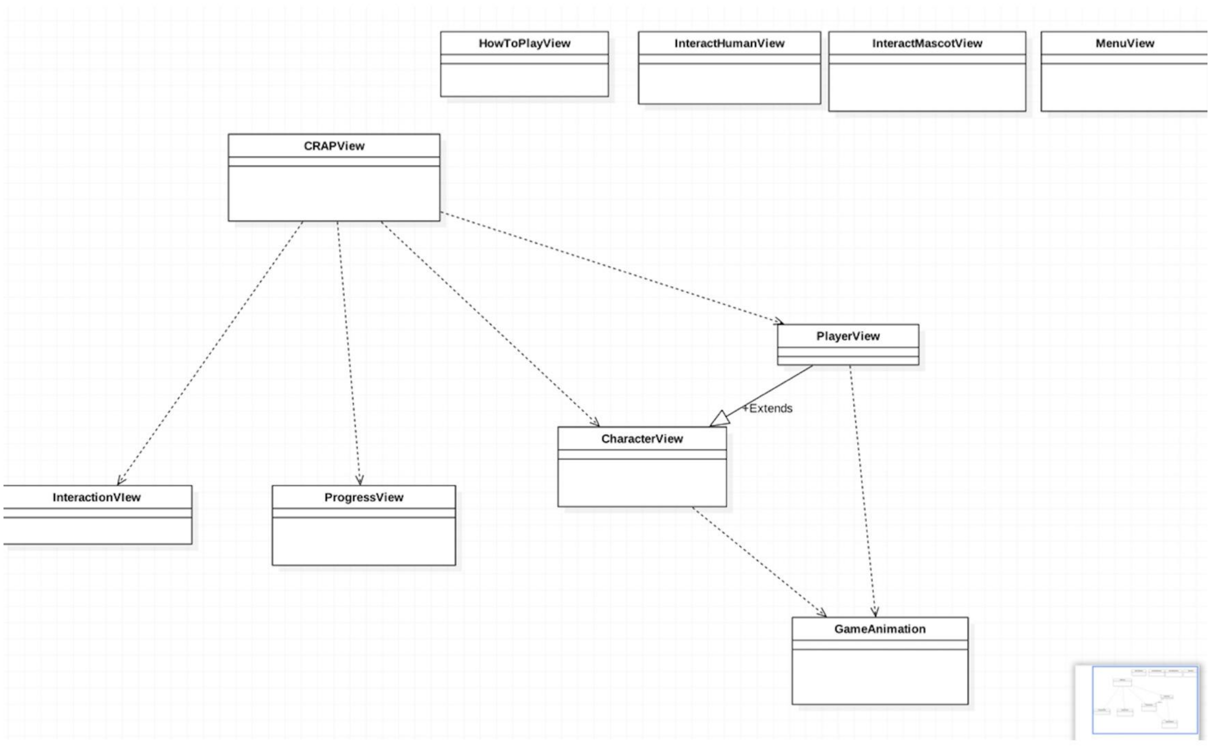
The connection between the main packages.



Connection between all the controller classes.



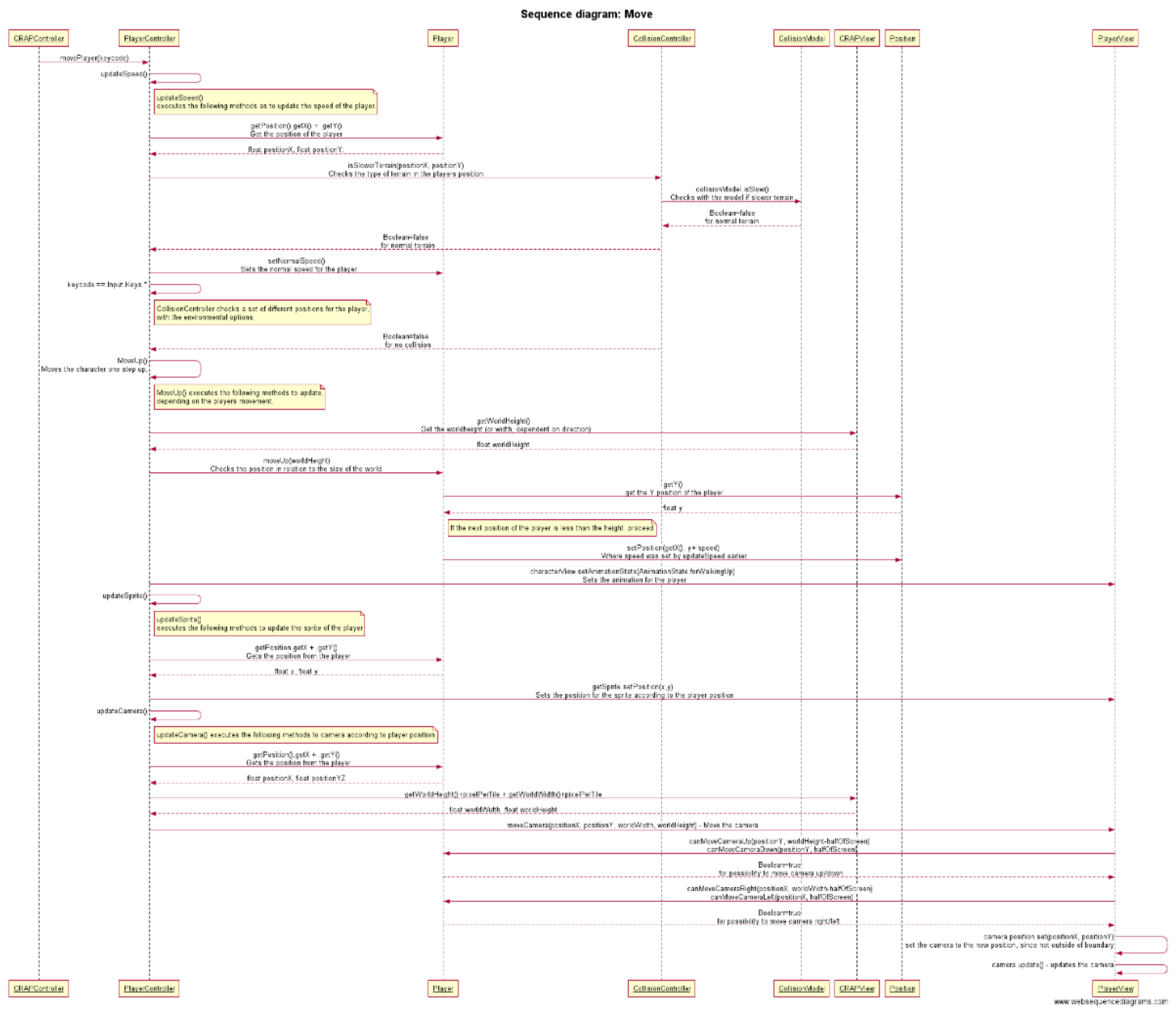
Connection between all the model classes.



Connection between all the view classes.

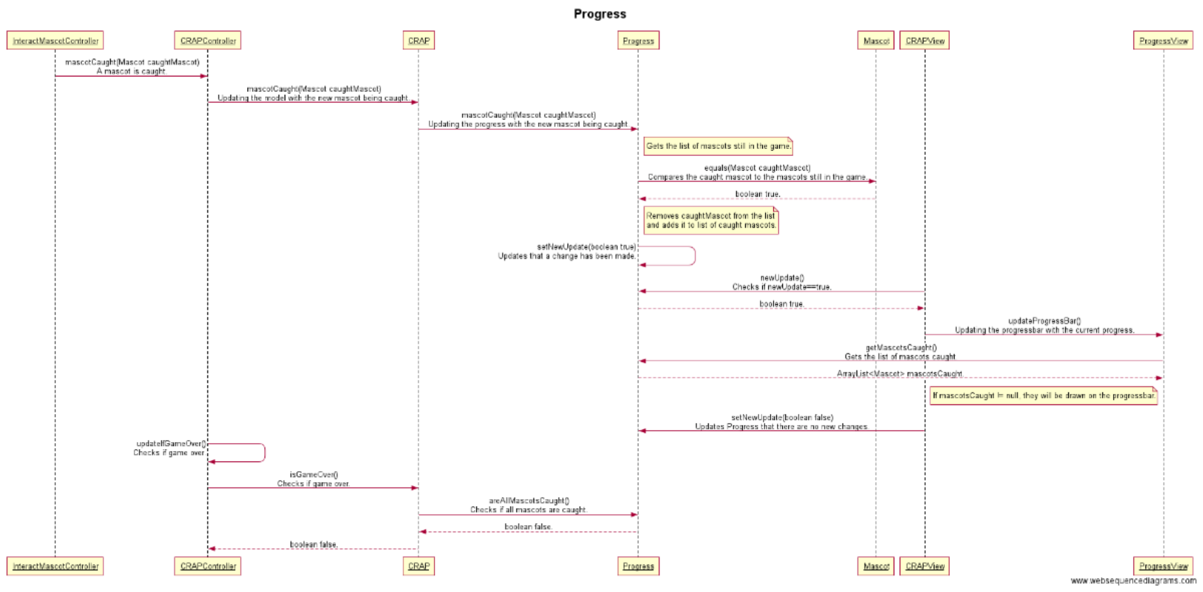
Appendix B

The two sequence diagrams from the main use cases.



Sequence diagram: Move
CRAPController
PlayerController
Player
CollisionController
CollisionModel
CRAPView
Position
PlayerView
movePlayer(keycode)
updateSpeed()
updateSpeed()
executes the folowing methods as to update the speed of the player.
getPosition().getX() + .getY()
Get the position of the player
float positionX, float positionY.
isSlowerTerrain(positionX, positionY)
Checks the type of terrain in the players position
collisionModel.isSlow()
Checks with the model if slower terrain
Boolean=false
for normal terrain
Boolean=false
for normal terrain
setNormalSpeed()
Sets the normal speed for the player
keycode == Input.Keys.*
CollisionController checks a set of different positions for the player, with the environmental options.
Boolean=false
for no collision
MoveUp()
Moves the character one step up.
MoveUp() executes the following methods to update, depending on the players movement.
getWorldHeight()
Get the worldheight (or width, dependent on direction)
float worldHeight
moveUp(worldHeight)
Checks the position in relation to the size of the world.
getY()
get the Y position of the player
float y
If the next position of the player is less than the height, proceed
setPosition(getX(), y+ speed)
Where speed was set by updateSpeed earlier.
characterView.setAnimationState(AnimationState.forWalkingUp)
Sets the animation for the player
updateSprite()
updateSprite()
executes the following methods to update the sprite of the player
getPosition.getX + .getY()
Gets the position from the player
float x, float y
getSprite.setPosition(x,y)
Sets the position for the sprite according to the player position
updateCamera()
updateCamera() executes the following methods to camera according to player position
getPosition().getX + .getY()
Gets the position from the player
float positionX, float positionYZ
getWorldHeight()+pixelPerTile + getWorldWidth()+pixelPerTile
float worldWidth, float worldHeight
moveCamera(positionX, positionY, worldWidth, worldHeight) - Move the camera
canMoveCameraUp(positionY, worldHeight-halfOfScreen)
canMoveCameraDown(positionY, halfOfScreen)
Boolean=true
for possibility to move camera up/down
canMoveCameraRight(positionX, worldWidth-halfOfScreen)
canMoveCameraLeft(positionX, halfOfScreen)
Boolean=true
for possibility to move camera right/left
camera.position.set(positionX, positionY)
set the camera to the new position, since not outside of boundary
camera.update() - updates the camera
www.websequencediagrams.com

The sequence diagram for the use case move.


Progress
InteractMascotController
CRAPController
CRAP
Progress
Mascot
CRAPView
ProgressView
mascotCaught(Mascot caughtMascot)
A mascot is caught.
mascotCaught(Mascot caughtMascot)
Updating the model with the new mascot being caught.
mascotCaught(Mascot caughtMascot)
Updating the progress with the new mascot being caught.
Gets the list of mascots still in the game.
equals(Mascot caughtMascot)
Compares the caught mascot to the mascots still in the game.
boolean true.
Removes caughtMascot from the list
and adds it to list of caught mascots.
setNewUpdate(boolean true)
Updates that a change has been made.
newUpdate()
Checks if newUpdate==true.
boolean true.
updateProgressBar()
Updating the progressbar with the current progress.
getMascotsCaught()
Gets the list of mascots caught.
ArrayList<Mascot> mascotsCaught.
If mascotsCaught != null, they will be drawn on the progressbar.
setNewUpdate(boolean false)
Updates Progress that there are no new changes.
updateIfGameOver()
Checks if game over.
isGameOver()
Checks if game over.
areAllMascotsCaught()
Checks if all mascots are caught.
boolean false.
boolean false.
InteractMascotController
CRAPController
CRAP
Progress
Mascot
CRAPView
ProgressView
www.websequencediagrams.com

The sequence diagram for the use case progress.