NTNU
Det skapende universitet

AnyBoard spillplattform

Et JavaScript rammeverk for å støtte utviklingingen av digital brettspill

Tomas Albertsen Fagerbekk

Master i datateknologi
Innlevert: september 2015
Hovedveileder: Monica Divitini, IDI

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

Tomas Albertsen Fagerbekk

AnyBoard game platform

A JavaScript software platform supporting the development of hybrid board games.

Trondheim, September 2015

Norwegian University of Science and Technology
Faculty of Information Technology, Mathematics and Electrical Engineering
Department of Computer and Information Science

NTNU – Trondheim
Norwegian University of
Science and Technology

Abstract

Previous work has sparked interest in hybrid games, games that combine video games and traditional board games by using digital surfaces or tangible, digital tokens. Whilst video games provide a rich, dynamic and interactive environment, board games has the advantage of a social face-to-face interaction. Hybrid board games attempts to combine the best of both, providing the dynamic interactivity of digital games with the social aspect of traditional games.

In this thesis a software platform has been developed for creating hybrid board games. Its goal is to simplify the challenges with creating and implementing game concepts digitally, in combination with integrating tangible digital devices.

The work has resulted in AnyBoard, a JavaScript based platform that is openly available at github.com/tomfa/anyboardjs. A common communication protocol as well as firmware implementations for two different Arduino-based chipsets has been developed and tested on mobile surfaces. With the current state of the AnyBoard platform, developers are able to create games using pawns that detect location, display colors and print cards from a mobile-surface game hub, without knowing anything but common web technology. The platform supports integration with any other JavaScript-based game library available.

AnyBoard can be used as is to create hybrid board games, but also has a great potential to become a richer and more potent tool with further development.

Keywords: hybrid games, board game, tangible interfaces, arduino, javascript

Preface

This project was carried out as a Master's thesis at NTNU as part of the MSc programme in Computer Science during the summer of 2015. The work has been supervised by Monica Divitini and co-supervised by Simone Mora.

The work is in large part motivated by challenges with previous projects done by students of supervisor Monica Divitini when creating hybrid board games. The contributions from in this thesis include the development and design of the AnyBoard library, token drivers and firmware that is located at github.com/tomfa/anyboardjs. Available open source tools have been used where applicable.

The design and assembly of the Arduino-based tokens has been done by co-supervisor Simone Mora.

Thanks to my supervisors for guidance, input and motivation throughout the thesis. You've been great!

Contents

Chapter 1

Introduction

The aim of this thesis is to look at how one can simplify the development of hybrid board games. The next sections of this chapter describe the context and motivation of the thesis. The research questions and research method are then described, before the final section (1.6) which gives an outline of the report.

1.1 Motivation

Much of the motivation is based on experiences from previous work on hybrid board games[1] at NTNU. Don't Panic(1) was a hybrid board game developed for training and play for emergency personnel, where the goal was to minimize the level and spreading of panic. The players used small digital pawns representing each player to move about a non-digital board. Details around Don't Panic can be seen in chapter 2.2. The lessons from this work is much of the background for the motivation in the following subsections of this chapter.

Hybrid board games are more immersive than traditional board games

The limitations of traditional board games lie in the static tokens and constrains. It is up to the players to understand the concepts and rules, as well as imagine or enforce the consequences of player actions. A digital game on the other hand is more dynamic with a richer interface that can be more absorbing. Consequences of a players actions can be enforced and accompanied by sound, vibration and graphical simulation. But digital games often take focus away from the social aspect of their traditional counterparts. The hope for hybrid board games is an

[1] Hybrid (board) games: We will use this term in the thesis to refer to the type of games we are targeting: A game where players interact with digital pawns, such as Arduino devices on a physical non-digital board

immersive, dynamic game play that also keep the social aspect of face-to-face interaction.

Both Don't Panic and other similar projects(2, 3) show good results: The players find them engaging and fun. One comparison between digital tabletop[2] and traditional board games even indicate that "senior citizens found the tabletop version of the game to be more immersive and absorbing [than regular board games]"(2).

Existing tools can be difficult to set up

In the testing of Don't Panic(1), the users enjoyed the game play and showed enough interest in it to request to keep a version of the game for themselves. The challenge with this was the complexity of setting the game up. It required turning on an off-site server that held a database and worked as a sort of game hub, as well as manually starting scripts in on-site components. The result was that it wasn't feasible for the players to initialize the setup of the board game by themselves.

In addition to this, much time had been put into programming the devices. Creating a replica of the game required much more than ordering a new set of hardware.

Existing tools can be expensive

In other more established tools for hybrid board games, the interaction is typically done via digital tabletop devices(2), such as Microsoft Surface Hub[3]. These devices, similar to a table with a large touch screen on top, restrict the mobility of the game, require a dedicated physical space and remain a large investment in terms of money.

Creating hybrid board games is time consuming

While there are plenty tools for creating online games, and a few for creating digital tabletop-based games, we have been unsuccessful in finding existing tools for creating hybrid board games that uses small hardware tokens.

The result when implementing Don't Panic was creating custom scripts and communication for the different pawns and implementation of game logic. Needless to say, this was very time consuming.

---

[2] Digital tabletop: A term we will use to refer to large horizontal touch screen devices with a built in computer, such as Microsoft Surface Hub

[3] www.microsoft.com/microsoft-surface-hub/en-us

1.2 Problem definition

Based on the observations made in the previous section, we see that hybrid board games pose several problems.

The cost for developers to create new hybrid board games is very big in terms of time spent. They are required to create most of their game from scratch, as there are few suitable existing tools. Due to the different types of components in a hybrid game, they will also have to know several kinds of technology.

Also in terms of money, the cost for developers is big. Due to the lack of standard tools, creating a game that makes use of a certain token[4] will require acquiring that token, since one cannot be certain it works without actual testing.

For players, the acquisition cost for the required hardware is huge if we consider the digital tabletop solutions. If we on the other hand consider hybrid games with tokens, the lack of mature software solution require players to have time and technical knowledge to set up and use.

1.3 Research questions

Below are our research questions. The first being a main question, that embraces those that follow.

RQ1: How can we lower the barrier for developers to start creating hybrid board games? We do wish for as many as possible to start creating hybrid board games. In order to get there, we need to lower the barrier so that more developers play around with the technology and get interested.

RQ2: How can we lower the investment required by developers, both in time and money, in order to create hybrid board games? One of the main obstacles we have seen from previous work is the amount of time involved. If creating hybrid board games was quick and cheap, more developers would do the same.

RQ3: How can we facilitate developers so that they are able to simplify the setup of such board games, in order to make it easy for players to acquire and play hybrid games? We believe in the importance of involving non-technical people in these games. If developers are able to create hybrid board games that *everyone* is able to set up and use, this opens a range of new possibilities, including creating a business around it.

[4] *Token*: A term we will use in this thesis a lot to note a tangible piece of hardware used in a hybrid game. E.g. a digital pawn.

1.4 Research method

This thesis takes an experimental approach to developing a platform for creating hybrid board games. Due to practical reasons, adjustments to the development process and requirements has come by evaluation from ourselves, without feedback from outside user groups.

The process led to the creating of a platform, which address challenges from previous work done with hybrid games. First by reviewing relevant literature and current workflow for creating these games. This was followed by a design process and implementation phase, before we evaluated how the resulting platform addressed the identified challenges.

1.5 Contribution

This thesis has resulted in a AnyBoard, a JavaScript-based platform for creating hybrid board games. It is openly available on github.com/tomfa/anyboardjs.

The location also includes extensive documentation, several examples, and drivers as well as firmware for three different tokens. These are also described in appendix I, C and D respectively. For the source code itself, we refer to the Github repository in the previous paragraph.

The thesis has already formed the basis for a published article from our institute, *Making interactive board games to learn: Reflections on AnyBoard*(4). This article is included in appendix G

In addition, we'd like to point to the discussion chapter (7) where we suggest different future directions and features for AnyBoard.

1.5.1 Limitations

AnyBoard itself is compatible with any JavaScript-environment. As such, it can be integrated with any other JS-framework or game engine without issues. For communication with tokens, AnyBoard depends on plugin-drivers with a firmware compatible with that driver. These drivers can pose their own limitations.

The drivers and firmware for the three types of tokens we have supported, can be seen in appendix D. These require Evothings Bluetooth libraries and Cordova libraries. This limits their use to Android and iOS environments.

1.6 Thesis outline

The report is split into seven section.

The first section (chapter 1) consists of the introduction to and motivation for this thesis. We explain the why, what and how of the thesis, providing a problem definition, research questions and method.

Section two (chapter 2) contains a problem elaboration. We take a look at related previous work, and characteristics of common board games as well as what concepts they include. A description of the life cycle of developing hybrid board games and which roles are involved follows, and thereafter we identify common challenges AnyBoard should address. We then illustrate a rough outline of the platform and the parts it should consist of, before we finally present a set of high level requirements for AnyBoard.

In the third section (chapter 3), we evaluate different tools and platforms for AnyBoard to make use of or build upon. Creating a hybrid digital game is a complex process, and here we look at existing tools that can be used in collaboration to simplify the complexity and make use of existing communities.

The fourth section (chapter 4-5) involves the design and implementation of AnyBoard. We present non-functional and functional requirements in section 4.1, followed by a more specific architecture for AnyBoard. In chapter 5 we present the implementation: details on the parts that was developed, the game entities and token communication protocol.

Evaluating the result is done in the fifth section (chapter 6). We implement a quiz game controlled with physical tokens, both with and without the use of AnyBoard. We evaluate how AnyBoard made this easier, and compare it to the implementation without the use of AnyBoard.

Lastly (chapter 7-8), we summarize and discuss the what could've been done better. Which parts of the thesis gave value and not. We explain what this mean for our research questions, before we provide our suggestions and thoughts around future work.

Chapter 2

Problem Elaboration

2.1 Previous work

This thesis and its motivation is derived from previous work with hybrid games at NTNU, as mention in chapter 1. In particular, Don't Panic was well received, but implementation was time consuming and setup during play proved difficult for players(1).

Other work done regarding hybrid games are often centered around digital tabletops as the central component of the game. Weathergods(3), KnightMage(5) and False Prophets(6) are examples of these. Most of these report a positive feedback from users, inclining that hybrid games indeed can utilize the best from both regular and digital games. An article evaluating tabletop game experience for seniors even claim they enjoy hybrid games and get more immersed than with regular games (2).

Two frameworks has been previously developed for creating digital tabletop board games: ToyVision(7) and ReacTIVision(8). However, their relevance might be dwindling, as digital tabletops for private use has near to disappeared. Most of the existing work regarding digital tabletop games was done in the period 2004-2010. Since then, there has been little development around digital tabletop games and digital tabletops in general, at least that target to the private consumer marked. Microsoft PixelSense[1], and their accompanied hardware, Samsung SUR40 is retired, according to their own websites[2]. We speculate that the high

[1] Microsoft PixelSense: Microsofts initative for tabletops.

[2] www.samsung.com/uk/business/business-products/smart-signage/specialised-display/LH40SFWTGC/EN displays Samsung SUR40 as no longer available.

cost of aqcuiring tabletop devices[3] has played a large role.

Hybrid games without digital tabletops has also been created. The game *"In Search of the Amulet"*(9)is an 8x8 tiled board implemented with rfid tags was used together with rfid readers inside digital player pawns. The pawns detected its movement and location on the board, and reported it to a computer. Each player here had each their computer functioning as a hidden private space[4] that allowing them to do actions without the other player knowing. In addition, a "public" screen showed information visible for both players. One of the challenges here was the potential intruding effects of the displays, and it was deliberately made to require as little as possible interaction with the computer in order to uphold the social aspect. The article does not mention any existing hybrid game platform being used.

2.1.1 Summary

Hybrid games of various kinds has been tested, prototyped and developed previously. From those articles we've seen, there has been nothing but positive feedback from users, who have found such games entertaining and immersive.

Platforms for developing hybrid games more easily has been made, but these are centered around digital tabletop environments, which is not suitable for our task, due to their high acquisition cost.

With the exception of *In Search of the Amulet*(9), we've been unable to find previous work of hybrid games with simple digital pawns independent of a digital tabletop device. We have been completely unable to find hybrid game platforms similar to the one we've planned to develop.

2.2 Characteristics of board games

In this section we will look at three different board games in order to identify typical components of the board games. The purpose of this is to identify classic types of interactions and components in board games, so that we can ensure including the most central elements of board games in our platform.

Note that this is not an exhaustive list of all board game characteristics in these games, but rather a superficial view over the most common and easily visible features. We have not used a previously designed approach for this analysis, but rather attempted to categorize the elements we've seen in our own words.

A summary of the characteristical concepts of board games are described in

[3] Samsung SUR40 goes for about 5000 USD on eBay market (August 2015)

[4] Hidden private space: A part of the game that is only visible to one player

subsection 2.2.4, while the physical elements common for board games can be found in subsection 2.2.5.

The three board games we've chosen is Lords of Waterdeep, Monopoly and Don't Panic. These are chosen to cover common, but different types of board games. Lords of Waterdeep is a classic Dungeons and Dragons game, making use of typical "advanced" board game dynamics, while Monopoly is a simpler common type of board game, employing many standard actions and elements. Don't Panic has been chosen in order to include an existing hybrid board game.

2.2.1 Monopoly

Figure 2.1: A version of Monopoly from Parker Brothers. Player pawns move from one to the next of the 40 tiles clockwise using two six sided dice. *[Picture by Horst Frank at the German language Wikipedia]*

Monopoly is a widely popular property trading game. Players acquire properties and money through chance cards, landing on property tiles and completing "laps" on the board[5]. They charge each other money for the "use" (landing on) their land. Players lose by going "bankrupt", which is when their assets amount to less than 0 amount of money. The **winner is declared when** all other players are bankrupt.

The Monopoly **board** consists of roughly 40 discrete **tiles** as shown in figure 2.1. Each player has a **pawn** that represents them, which they move about the board

[5] We've based our description on Monopoly rules from http://www.hasbro.com/common/instruct/00009.pdf

with. The movement is determined by a roll of **dice**. Monopoly is **turn-based**, which means players take turn to throw the dice, moving their pawns, and completing a set of possible **actions** they can choose from, determined from the tile they land on.

Each action usually involves buying, paying or exchanging **resources** in form of money or stocks – which are placed in an inventory of each player. The inventory is a **private space** which contains the resources belonging to that player. Players starts the game with some initial resources in their inventory, but most are acquired during the game play. Resources are represented in form of two sorts of **informational tokens**: stocks and monopoly money.

In short, the goal of the game is to maximize the acquisition of resources.

2.2.2 Lords of Waterdeep

Lords of Waterdeep (LoW) is a **turn-based** game from the Dungeons and Dragons series, where players fight as a Lord in order to control a city. The goal of the game is to maximize the number of **points**. which are obtained mainly by completing quests. Quests are completed by obtaining a specific combination of different adventurers in addition to a quest-card[6].

In addition to quests and resources held by a player (marked red in figure 2.2), each player also is a specific Lord (**role**) which gives bonus points to certain quests for that player. LoW also include a type of cards called Intrigue, which can be used in various advantageous ways.

The Lord and Intrigue-cards are held in a **hidden private space**, a space that is not visible to other players other than the holder. Compared to games that doesn't have such dynamic, this hidden space introduces a new dimension to the game, as players hold different information.

LoW also make use of **rounds**. After all players have placed their pawns on the board, and actions has been completed, a round is over. All pawns are being put back into the inventory of each player, and points given by acquiring new tiles (buildings) are increased. After the fourth round, players are given an extra pawn, and the game ends after the eight round.

Another interesting component of LoW is its **dynamic board**. One of the initial tiles can be used by players that wish to buy a new tile. The new tile will be an available in that and all the remaining rounds, and will reward the buyer upon usage.

[6] We have based the description of LoW rules on rules from http://media.wizards.com/downloads/dnd/DnD_LOW_Rulebook_EN.pdf

Figure 2.2: Lords of Waterdeep. Players have multiple pawns and place them on available tiles as they please. New tiles (yellow) can be bought into the game allowing for a greater set of possible actions. Each player has both a visible (red) and hidden (blue) private space. *[Picture by the author]*

2.2.3 Don't Panic

Don't Panic (DP) is a **cooperative** board game with different zones, where panic can emerge among "people" and spread to nearby zones. The players, unlike in Monopoly or LoW, work together in an effort to reduce panic. In DP, the players attempt to calm a situation down, and loses if the panic level goes beyond a certain threshold.

Each player is presented by a pawn (colored circles in figure 2.3) located on tiles (black nodes), and takes turn to draw two informational cards, two event cards and complete four "actions". Event cards are challenges which the players have to deal with on their turn, and can be resolved by "actions" or the use of informational cards. The informational cards are a form of resource that can (optionally) be used on a players turn.

The four "actions" performed by players on each turn can be chosen from a set of actions containing movement of pawn, movement of people between zones, set

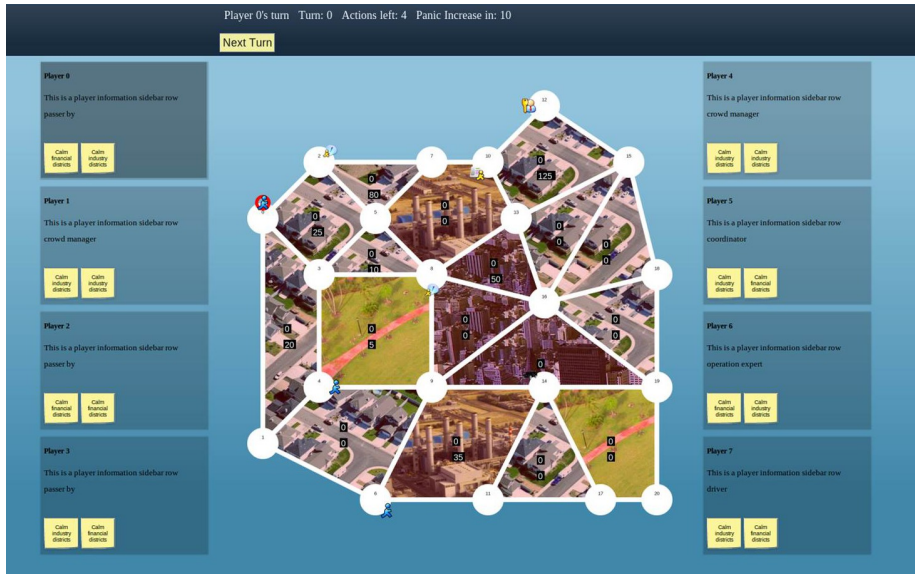

Figure 2.3: Digital version of Don't Panic. The map is divided into zones with a respective panic level. Player pawns locate themselves on the different nodes, and attempt to lower and contain the panic. *[Picture by Ines Di Loreto]*

up or remove road blocks to contain panic, as well as building an information center. Each player is also randomly given a role at the start of the game, which gives them an advantage on a specific type of action.

Once the game starts, a **timer** is initialized, counting down from a certain amount of minutes. Once the timer rings, panic levels are increased, and can spread between zones. This introduces a dimension of stress or hurry to the game, as playing faster gives an advantage.

2.2.4 Summary: concepts

Players and Teams: Board games are typically played in social settings with two or more players. In many board games such as Monopoly, each player is represented with a physical object called "pawn". In LoW, we saw that this is not the case, but rather had several agents that acted on behalf of the player.

Commonly, every player play against each other and is as such on team with only themselves, such as in Monopoly or Low. In other games, such as DP, several or all players can also play on teams with each other, fighting common enemies.

Roles and skills are player-specific properties that deviate from the standard rules of the game, and can give advantages or disadvantages in different aspects of the

game. In DP, each player is assigned a Role by random, which gives them advantages in performing certain actions. For example, a driver can move more people out of paniced areas. Similarly in LoW, each player is assigned a Lord card (character) by random, which increases the points granted upon completing certain quests. LoW also has special quests that provide an advantage for the rest of the game (skill). For example, completing the quest "Quell Merchenary Uprising" will provide the player with two extra points for every other quest of the same type he or she completes.

Public and private spaces: Public and private spaces are the conceptual areas containing parts of the game that is interacted with by all or one of the players. Public spaces can be interacted with by all players, while each player has a private space that only that player can interact with. In Monopoly, question cards, dice and cards are a part of the public space, while money belong to the private spaces of each player.

Hidden private space, is the a private space that is not visible for other players. For example a hidden hand contains cards that are visible only to the holder of the cards. For example in LoW your Intrigue cards (uncommon advantageous actions you can perform) is hidden from other players.

Events are special parts of the game that often go outside the normal flow of the game. In Monopoly, one of the tiles will send you to "jail". This includes a movement of the pawn that is not based on dice like the rest of the game, and gives an exception to the dice rolling and movement in the next turns. An event is also triggered when drawing question cards: winning the lottery or having to repair your houses go outside the normal flow of the game. In LoW, playing Intrigue cards will trigger events, allowing for players to take resources from others or imposing a mandatory quest upon other players. And in DP, Event Cards and playing Informational Cards can be categorized as events. Events are typically unpredictable, either by being in a shuffled deck of cards or played from a hidden private space.

Resources are assets belonging to a player. In Monopoly, the resources are Stocks, Money and Houses. Player controlled events, such as the Get Out of Jail Free Card (Monopoly) or Intrigue card (LoW) can also be considered resources. Resources are often made physical through Indicator Tokens (see 2.2.5).

Turns and actions: In turn-based games each player takes turn to complete some actions, before the next person plays his turn. Each turn usually involves a scripted set of actions. In Monopoly, you must always roll two dice and move your pawn that amount of tiles. This is a mandatory action. The tile determines

the next set of actions you can perform. Either 1) pay the owner rent (mandatory), if the tile is currently owned by anther player, 2) buy the stock (optional) if the tile is for sale, 3) draw a card and complete it's action (mandatory) if landing on a question-tile. In other words, turns are sets of actions, where each action can either be mandatory or optional, that can have a precondition, i.e. do this if that, and can be ordered, i.e. done in a speific order.

Rounds is a concept that holds some event will occur, after some criterion is met. In LoW, actions is performed via agents, which is "spent" once put on the board. When all agent actions are exhausted, the round is over, and agents are put back into their respective players private space so that they can perform actions on behalf of the player again. In DP, one can interpret the time between each alarm as a round. Once the round is over, panic is increased and potentially spread.

Rules tell us the possible interactions between game components: How can your pawn move around the board? When is a winner declared, or a player out of the game? Rules explicitly define the starting conditions upon starting the game, and are a central part of all games, as they dictate the flow of the game and define the boundaries of them.

2.2.5 Summary: Physical components

Board is the surface used to play a board game. Most games have a unchanging board (Monopoly), while some have a modular board with varying layout in each session or while the game is played (LoW). The board consists of tiles and placeholders for other assets.

Tiles are discrete locations on the board. Which tile a player is located on, usually determines which set of *actions* the player can perform. In Monopoly for example, a stock can only be bought when the the pawn is located on that specific tile. In Don't Panic, it didn't determine which actions one could perform, but rather which part of the city the player could perform his actions upon. In some games, such as Monopoly and DP, movement is only allowed over adjacent tiles.

Indicator tokens, e.g. Gold, Wood, Stone, Gas, Money, Points, Stocks and Houses help keeping track of an informational aspect in the game. In Monopoly, the paper money is simply an indicator of how much resources, and stocks an indicator of the lots owner. It serves its purpose by holding some information, and as such simplifying the amount of information players have to remember and keep track of themselves.

These are simple tokens, as they are only representational of some game information, and does not change the game state or provide interactive

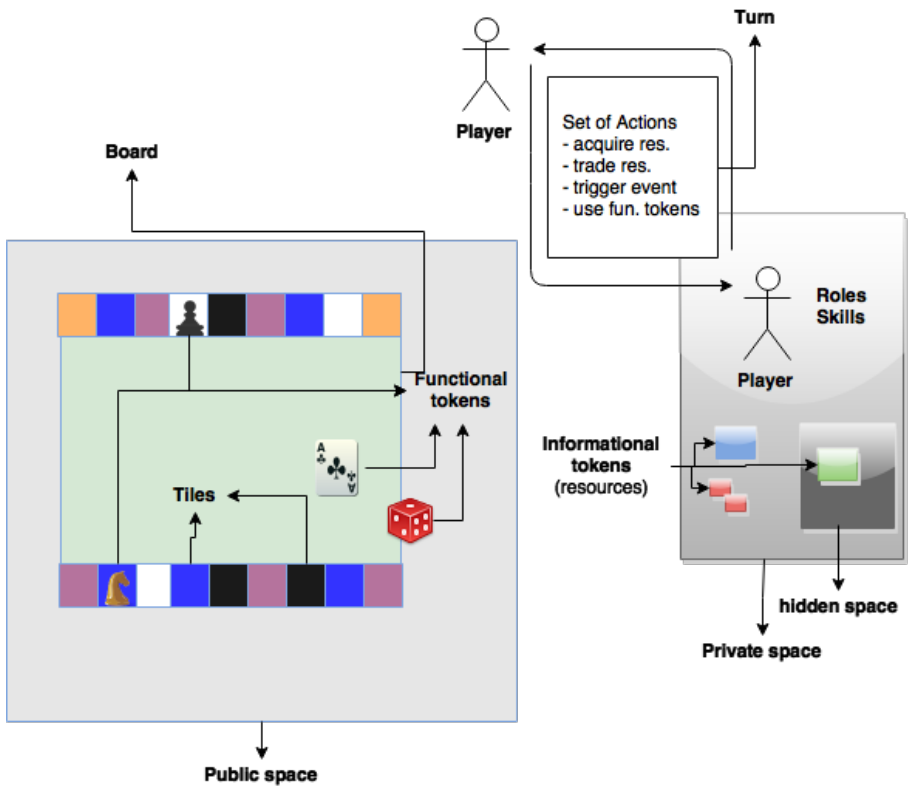

Figure 2.4: Common components of board games. Board with tiles and functional tokens as a part of the public space, while informational tokens (resources) as part of each players private space, potentially hidden from view from other players. The game is progressed by players taking turn to acquire and trade resources, triggering events and interacting with functional tokens.

functionality like randomness. Such tokens can easily be replaced by a number value or position on a screen, as they are not interacted with other than changing owner.

Functional tokens, e.g. dice, cards and pawns play an active part in how the game unfolds. For example, a dice in provides randomness as a functionality to the player, while the timer in DP provide an aspect of time or hurry. Cards are the most common functional tokens, which can provide many types of events and exceptions to the standard flow of the game, as well as provide randomness through being shuffled and used in a drawable deck.

These tokens change the state of the game by opening or closing certain actions

and aspects for players. Each of these tokens have a specialized functionality, and can not be implemented as generally as indicator tokens.

2.3 Life cycle and roles

In this section we go through the life cycle of a game, from idea through creation of a playable hybrid board game. We focus on the roles and their contributions and actions in creating and using the game, as was the case with the previous hybrid board game created at NTNU, Don't Panic . In the next section we will look at the challenges linked with these activities.

We have identified three typical roles: a **Game Designer**, who formalizes and defines the game. The end result from a game designers production is a playable paper prototype. A **Developer** takes this prototype and identifies and acquires suitable hardware pieces, designs the user interface to be used in the game controller and implements the game through the controller and the hardware tokens. A **Player** should then able to acquire, set up and play the game. These roles and their typical activities are illustrated in figure 2.5, 2.6 and 2.7. The activites that we will focus on in this thesis is marked in green.

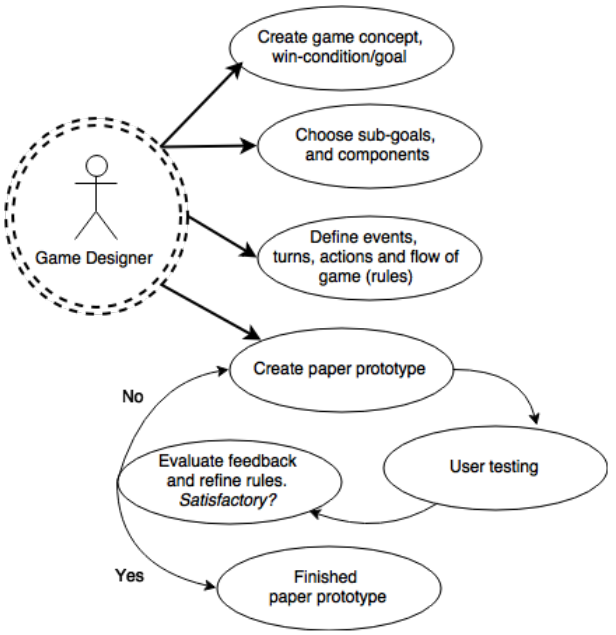

Figure 2.5: An overview over the responsibilities of the Game Designer in implementing hybrid board games.

2.3.1 Designing

The Game Designer starts with an idea, and details it through defining concepts, providing a motivation/goal, and giving the game clear rules and boundaries. The

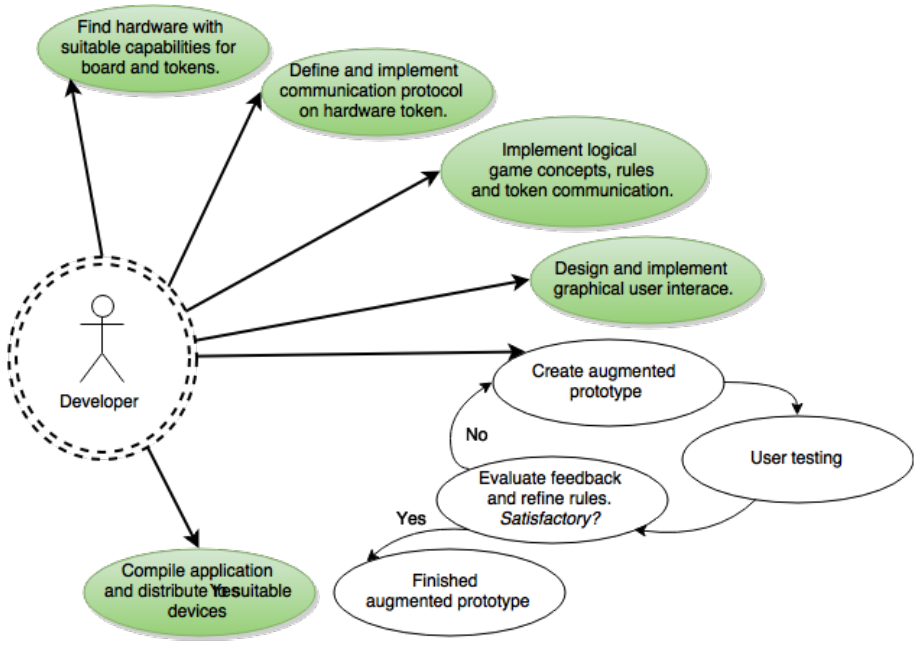

Figure 2.6: An overview over the responsibilities of the Developer in implementing hybrid board games, after a finished paper prototype is provided by the game designer. The responsibilites marked with green are areas where we think AnyBoard can assist.

end result from the game designer should be a playable paper prototype of the game. The following questions are the main questions that a game designer should be able to provide answers for (short Monopoly example answer in emphasized text):

1. How are the teams - who is playing against who? *Every player for themselves*
2. What is the game concept? What are the the goal of players? *Achieving monopoly by making your opponents bankrupt*
3. What resources and game objects does the game consist of? Which is manifested in physical tokens? *Stock, Money, Pawns, Houses, Hotel, Board, Question Cards*
4. What are the definite win- and lose conditions? *You lose when you have no valuables left. Winner is the last player standing*
5. Through which actions does the game progress? *Roll dice to change tiles. Tiles can be bought, traded, trigger event, paid rent for staying at. Each player takes it turn to complete a set of actions*
6. What are the initial set-up conditions of the game? *Every player start at go*

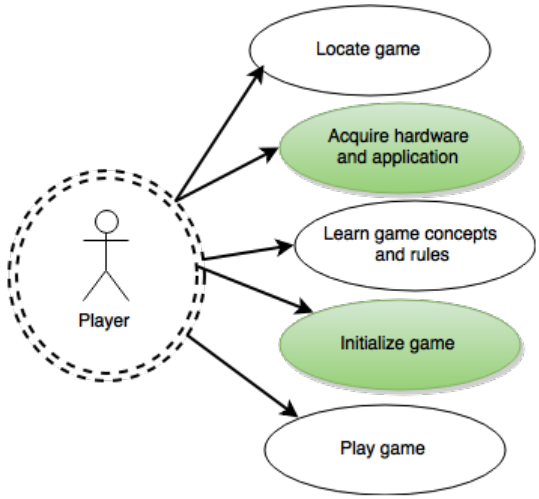

Figure 2.7: Main interactions from the view of a player in using hybrid board games. From previous experience in the testing of Don't Panic, replacing hardware tokens and game initialization (marked green) was too hard for non-technical players to receive a copy of the game.

tile with the same initial amount of resources.

These answers is described in detail and usually goes through several iterations (often with user feedback), before he or she comes to a final version that should be able to be played on a paper prototype. This iteration and refinement process could also be skipped, and rather done in the next phase, after implementing the game as an hybrid board game.

2.3.2 Implementing

The Developer should have a clear description of the board game from the Game Designer. He will convert the game into a hybrid version, by identifying suitable digital tokens and interfaces, and implement the game logic into the necessary tokens and devices. A user friendly interface is necessary to allow players to initiate and play the game, and the applications should be distributed in a way that makes it easy to acquire. In short, the Developer takes the game from a finished concept to a finished product. This is shown in figure 2.6.

Here we have assumed a case where a game controller (phone, Tablet, computer) and a set of digital hardware tokens has been used to create a hybrid version of the board game. The Developers main responsibilites, are (examples are shown in emphasized text):

1. Translate the required token expressions and interactions into a clear grammar. *Pawn can be placed on board and should notify of location. Pawn can show color and vibrate.*
2. Find or make digital hardware tokens that are capable of necessary token expressions. *"rfduino" device should be capable of our requirements.*
3. Define and implement a communication protocol between tokens. *Make "rfduino" accept "SET COLOR RED" over serial Bluetooth and execute the corresponding expression.*
4. Implement game concepts, rules and token communication. *Creating the logical representations of "Stock", "Money", "Board" etc, a token communicator object, and procedures to initiate the game and handle "paying rent".*
5. Designing and implementing the GUI. *Creating a menu with buttons to run initiate procedure, read rules, and exit game. Designing an choice screen to handle trades between players*
6. Compile the game so it is distributable. *Compile to iOS and Android files and distribute on App Store and Google Play*

Prior to the compiling and distribution the game, the Developer should go through an iteration and refinement process to ensure the quality of the game. The game should now be ready for players to acquire and play the game.

2.3.3 Playing

A Player of the game is not involved in the creation of the game, but is a consumer of the finished product, a player of the hybrid board game. In acquiring and playing the game, he or she will typically go through the following steps (examples in emphasized text, illustrated in figure 2.7):

1. Locate the game. *Hear of game from a friend, and go to its website.*
2. Acquire hardware and application. *Order the necessary tokens online, and download a corresponding application from App Store*
3. Learn game rules. *Read the game manual, and a FAQ on the applications website*
4. Initiate Game. *Turn on tokens, open phone application, establish communication between tokens, and set up initial game conditions*
5. Play game. *Interact with tokens (and phone application) according to the game rules*

2.4 Challenges

In this section, we will look at some of the challenges that each role face today when creating hybrid games, such as Don’t Panic. These will strongly influence

us when making high level requirements in the next section.

2.4.1 Game Designer

The work of the Game Designer is creating an entertaining experience. It's a creative process, with few rules that dictates how things ought to be. The challenges for a game designer is finding the inspiration and ideas necessary to create a good game concept. The designer should find volunteers to play the game and give adjustive feedback. Once a prototype is polished enough for the designer to be satisfied, the rules and concepts must be defined in such a way that the developer is able to translate it to a program.

- Finding inspiration for the game concept and ideas can be hard.
- The game designer must find volunteers that can provide user feedback to help refine the game.
- The game designer should preferably have knowledge of typical game concepts.
- Defining the game clearly enough to be computer translated.

2.4.2 Developer

There exists few suitable digital tokens for hybrid board games. As with the augmented version of Don't Panic, this can lead to a large amount of time being spent finding or making custom tokens (number 2), and establishing communication (number 3 and 4) between the tokens. This also requires knowledge of low level programming, as the tokens are based on low level programming languages such as C.

There are few existing game tools aimed for board games. Among them, there are no tools geared for using digital tokens. A developer might therefore have to build the board game concepts from scratch (4, 5), and modify the tools to support the tangible aspect (4).

- Developers must know both high level and low level code
- Large amount of time is used creating custom hardware tokens
- Communication between tokens must be built from scratch.
- Existing game tools is likely to require modification to support the tangible aspect of hybrid board games.
- End-users (Players) use various devices, operating systems and screen sizes. It can be time consuming to support the different devices.
- Licenses for game development tools can be costly.

2.4.3 Player

Challenges of the Player: First, the lack of mature components make it hard to initiate the game (number 4 in). In Don't Panic, set up of the game required technical knowledge and was time consuming. It required starting an off-site server, and starting scripts manually on some of the game tokens. Second, the equipment used was custom made, and could as such not be easily replaced (number 2). Since the equipment was custom made, it could not be reused for other purposes without reprogramming. If a Player wish to acquire a second similar game, he must anticipate to buy another set of hardware (number 2). Having played and enjoyed one hybrid board game, the lack of a an community or platform around hybrid board games can make it difficult to find new such games (number 1)

- In existing prototypes it has been hard for players to initiate a game, as the setup have required technical knowledge.
- Hardware purchased for one game has not been suitable for reuse in other games without reprogramming, which is both time consuming and requires technical knowledge.
- Lack of community around hybrid board games, makes it hard to acquire and locate such games.

2.5 Components and high level requirements

From the challenges we've identified, we find that the lack of mature tools and active community is basis for much of the challenges. We therefore pose that a platform for creating hybrid games could assist greatly in the process. The role of a game designer could be assited by a community that provides resources on concepts that a board game typically consist of. A game developer could be assistted through suitable hardware tokens and tools for integrating them with a user friendly mobile surface. From testing Don't Panic, we know that acquisition of hardware tokens should be cheap, and games easier to set up. A platform can assist by making tokens reusable between games. We also think that a platform could make games more easily discoverable for players.

2.5.1 High level requirements

We have chosen to focus mainly on the challenges of the developer, and assist him or her in the implementation process of hybrid board games.

Table 2.1: High level requirements

	Functionality
D1	A **game developer** should be able to extend or remove parts of the platform, in order to suit his or her needs that are not originally covered by the platform.
D2	A game developer should not be required to pay for creating games through the platform, in order to lower the barrier for using it.
D3	A game developer should not be required to rewrite his or her game in order to be supported on different platforms or screen sizes, so to minimize the development effort necessary to reach a broad audience.
D4	A game developer should be assisted with guidance on how to compile and deploy the applications to Google Play (Android) and App Store (iOS), in order to simplify the development and distribution of games made with the platform.
D5	A game developer should be provided with an API for communicating with supported digital tokens, and not be required to have knowledge of low-level code, in order to lower the barrier for using the AnyBoard platform.
D6	A game developer should be able to extend digital token support to new tokens with new features, in order not to restrict the developer to a fixed set of digital tokens.
D7	A game developer should be provided with classes and abstractions for typical board game entities, such as boards, pawns, dice, cards and decks, in order to lower the development time.
D8	A game developer should be provided with a basic set of visual elements for screen display, such as menus, cards, boards, pawns, timers and buttons, in order to lower the development time.
D9	A game developer should be provided with signals and event handlers to simplify implementing responses to a players actions, and game events.
P1	**A player** should not be required to have technical competence in order to initialize a game made with the platform, in order to lower the barrier for players to acquire an hybrid board game.
P2	A player should be able to reuse board game hardware to play other games with ease, in order to lower the barrier for players to try other hybrid board games once having acquired one.

2.5.2 Main components

We present here a broad idea for how we can fulfill these requirements and create tools to accomodate and help developers and players to create and use hybrid board games. A sketch of this can be seen in figure 2.8. We don't expect to implement all components in this thesis. The parts for implementation will be narrowed down in chapter 4

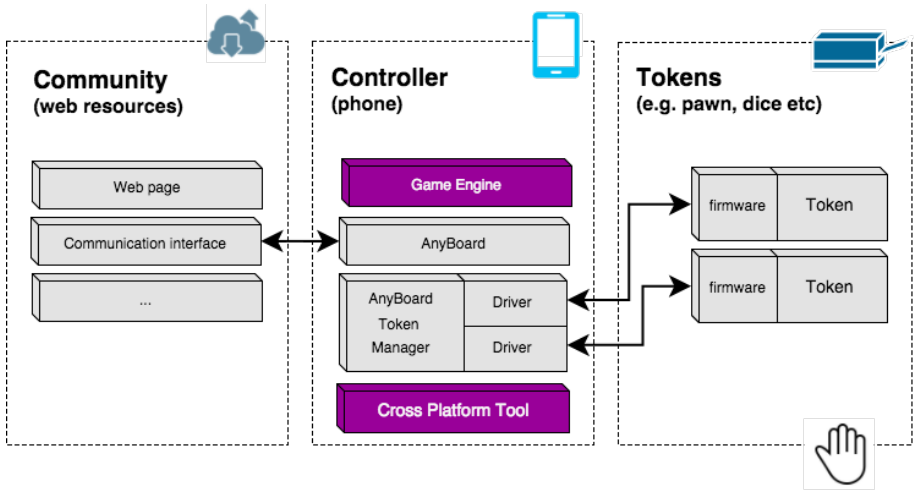

Figure 2.8: High level components of AnyBoard.

Game development tools and communities already exists, and hence the main part that makes the AnyBoard platform unique, is helping integrate the digital tokens as a part of games. This is therefore the area of greatest importance.

Example tokens, with low level code implementing typical token capabilities, will be provided for developers that wish to create games with general token requirements. This token will communicate with a *Token manager* on the application side that handles the communication between the game logic and physical devices. This component will provide a token API on the software side, so developers can listen to token-events and send commands without the knowledge of the low level code, as well as assist developers to create easy-to-use interfaces for connecting and initiating the game and game tokens. *(P1, D5)*

The Token manager is separated from any specific token, and communicates through a *device specific driver*. A generic extendable driver will be provided to assist developers that wish to create their own tokens with other capabilities. *(D1, D6)*

A *Game engine* will provide tools that help the developer quickly create the components of his or her game. We wish to provide base components specifically suited for hybrid board games, such as Board, Tile, Pawn, Dice etc, both the logical and visual *UI* part. *(D7, D8, D9)*

The AnyBoard software platform should be based on a *cross platform tool* that enable games made with the AnyBoard to compile to different operating systems. *(D3, D4)*

Several of these components exists already, and we aim to use open source, free-to-use, modular and well documented tools, so that a developer can pick apart the AnyBoard system and add capabilities where need be. *(D1, D2)*

Lastly, a web-based home for AnyBoard can grow a community and provide information for all roles involved with hybrid board games. The AnyBoard platform could be downloaded from here, and tokens sold from a web store. It can also provide a knowledge base and tools for developers to assist each other. Through a *Game Store* or an overview of hybrid board games, we hope to assist users with finding other games they can play, and an assistive IDE for game developers could help lower the knowledge barrier for new developers even further. *(P2)*[7].

[7] A game store, or a web based IDE is a later stage than the AnyBoard platform presented in this thesis.

Chapter 3

Preliminary studies

In this chapter we'll look at two different types of tools: Cross platform tools and game engines. These are natural tools to use in the development of mobile games, because they drastically simplify the development process. It is therefore also important that AnyBoard is compatible with some of them; since the tools differ in design and features, we must determine which of them we aim to support and which we can ignore.

In this chapter we will first look at cross platform tools, and evaluate candidate tools. The choice of cross platform tool will limit the set of game engine candidates, which is the evaluated in section 3.2.

3.1 Cross platform tools

One of the main goals of the software platform is that it should be easily accessible to use and develop on. Since we're in the beginning of developing the platform, the tools we choose should restrict us in the least amount of way, regarding technical capabilities. The purpose of using a cross platform tool is being able to develop for multiple platforms simultaneously, and hence lower the development time for both us and developers making use of AnyBoard.

In order to lower barrier for new developers or other users to use the platform, AnyBoard should ideally be built with the use of free tools, and written in popular languages. Code written in AnyBoard should be deployable to multiple platforms with none or only minor modifications. Support for deploying to iOS and Android platforms is chosen as a minimum requirement, due to them covering the largest audience[1].

[1] According to http://en.wikipedia.org/wiki/List_of_mobile_software_distribution_platforms (Ac-

3.1.1 Criteria

We've judged options in regard to the following criteria:

- Popularity of programming language – Projects based on languages popular in open source communities have a larger pool of potential developers to contribute to the platform
- Knowledge of programming language with regards to team – The programming languages chosen should be of some familiarity to the people involved in this project
- License and cost to use – Cost is a barrier for developers to contribute to the project. Like with hardware, required software should be as cheap as possible to acquire. An open source framework would be preferable, not to create restrictions on possible functionality of the software.
- Existing Bluetooth capabilities - Bluetooth functionality is a requirement for the basic functionality of AnyBoard. A tool with good Bluetooth support would be preferable.

3.1.2 Candidate cross-platform tools

The candidates were chosen from comparisons and benchmarks of cross platform tools(10, 11, 12, 13). Some were excluded immediately if they clearly didn't meet our criteria, while a couple were added due to our previous experience with them.

cessed 2015-05-13)

Name	Language	Bluetooth	License	Free	Popularity
PhoneGap[a]	JS	Yes	Apache	Yes	Very high (> 60%)
Appcelerator [b]	JS	possible (not via Appcelerator API)	Apache	Yes	High (> 40%)
Cocos2d[c]	C++/JS	possible (not via Cocos API)	MiT	Yes	Medium (> 20%)
Unity3d[d]	C#/JS	possible (not via Unity API)	Proprietary	Yes*	Very high (> 60%)
Corona[e]	Lua	possible (not via Corona API)	Proprietary	Yes*	High (> 40%)
Qt[f]	C++	Yes	LGPL	Yes*	High (> 40%)
Xamarin[g]	C#	possible (not via Xamarin API)	Proprietary	Yes*	High (> 40%)
Kivy[h]	Python	possible (not via Kivy API)	MiT	Yes	Unknown
Evothings[i]	JS	Yes	Apache	Yes	Unknown

Table 3.1: Overview over initial candidates for cross-platform compatibility. Preferable properties of the platforms are marked in light green, less preferable in orange, while undesirable properties are marked in dark red. Popularity based on developers awareness in Cross-Platform Tool Benchmarking 2014(13).

[a] phonegap.com - The original name for the Apache Cordova framework. PhoneGap is essentially Cordova with additional but optional pay-to-use services.

[b] appcelerator.com – Compiles JS to native code.

[c] cocos2d.org - Cross-platform game engine available in both JS, Lua and C++. Compiles to Mac OSx, Windows, Android and iOS.

[d] unity3d.com - Cross-platform framework built on an advanced game engine. Free Personal edition is limited.

[e] coronalabs.com – *Limited version without access to native calls are free to use. Due to the limitations of the API and license, Bluetooth communication is unavailable for free version.

[f] qt.io. *Free to use for open-source, non-commercial applications

[g] xamarin.com – compiles C# code to native applications. Free version has limited features.

[h] kivy.org - Python-based framework that compiles to mobile (iOS and Android) as well as Windows, OSx and Linux. Popularity unknown, but presumed to be low compared to other alternatives.

[i] evothings.com - Light framework based on Cordova, with libraries centered around communication with small hardware objects (Internet of Things) and simplifying testing. Popularity unknown, but benefits from being based on Cordova.

3.1.3 Evaluation

Cross-platform tools (CPTs) are increasing in popularity. Some of them compiles to a native application on each platform (Unity, Cocos, Appcelerator), while the others run in an "in-app browser" (PhoneGap, Evothings), essentially acting as web pages. The latter will in most cases simplify the development and make an easier transition for existing web-developers, at the price of performance and functionality. Such tools might therefore not be an alternative for graphic-intensive games or where access to certain parts of the phones functionality.

With the exception of Evothings, Bluetooth capabilities is not an area of focus for any of the tools. They do however support writing own libraries or plugins or give developers the opportunity to write native code that can access Bluetooth capabilities on the phone. We could also find support or plugins for PhoneGap and Qt to simplify Bluetooth access for us.

We chose three different frameworks to further investigate based on our initial findings. PhoneGap, due to it's licensing and popularity; Evothings, for being closely related to PhoneGap and adding functionality suited for our purpose, and lastly Appcelerator Titanium for the comparison of in-browser vs native app platforms.

PhoneGap

PhoneGap by Adobe/Nitobi was the original creator of Apache Cordova, which is today the most popular engine for creating mobile cross-platform in-browser applications. The Cordova engine provides basic phone functionality such as Camera, GPS and vibration to a web based environment for creating apps. In 2011, Adobe donated the Cordova code-base to the Apache Foundation, and PhoneGap instead focused on providing services on top of the engine, such as marketing, analysis, building, support and training. PhoneGap consists today of both an Open Source fork of Apache Cordova in addition to these services. The services are optional, and most of them are pay-to-use.

- + Based on well known Cordova platform
- + No or little additional code necessary to support different platforms
- - Compiles to in browser apps, giving reduced performance

Evothings

Evothings is a toolkit based on the Cordova platform, as PhoneGap. Evothings is centered around the idea of "Internet of things" or "ubiquitous" computing, and it provides simplifications for communicating with several different types of tokens,

as well as code examples.

While the Evothings toolkit itself is not very popular, valid Cordova code will be valid in Evothings. An app developed with Evothings could also be ported to other Cordova-based frameworks with no or only small changes in the code base. This has the added benefit of making existing Cordova/PhoneGap community relevant for development on this platform.

The additional features included in Evothings compared to Apache Cordova is a simplification of testing by allowing instant deployment to your phone in the testing phase, using a Evothings app. In addition, libraries to communicate using Bluetooth with external hardware is included.

- + Toolkit allows instant deployment to phones through Evothings test suite app
- + Extra support and examples for relevant communication with external hardware through Bluetooth
- + Based on well known Cordova platform, making a large existing community relevant for this platform
- + No or little additional code necessary to support different platforms
- - Compiles to in browser apps, giving reduced performance

Appcelerator Titanium

Appcelerator Titanium has a different approach than the Cordova-engine on how to create cross-platform applications. Appcelerator themselves compare Appcelerator Titanium vs PhoneGap and explain their differences. Some of their main points[2] is that:

The barrier to entry in using PhoneGap to package web pages as native apps is extremely low. - Starting developing with PhoneGap requires very little knowledge or experience with creating mobile applications, or knowing the difference between platforms. Knowledge of creating web applications is sufficient for starting to create PhoneGap applications for different platforms.

Very few native APIs are exposed to PhoneGap applications by default - While PhoneGap by default only provides functionality that are common among different phones, Titanium has a richer set of functionality available from the phone. A Titanium-based app can be better tailored to fit and use all of the capabilities of a phone.

The quality of the user interface in a PhoneGap application will vary based on

[2] According to http://www.appcelerator.com/blog/2012/05/comparing-titanium-and-phonegap/ - "Comparing Titanium and PhoneGap" (May, 2012)

the quality of the web view and rendering engine on the platform. - Since the user interface of PhoneGap is based on an in-browser app, it will not look like a native app with native controls and buttons. The performance of a browser interface will also be slower than interfaces made up of the phones native controls. In addition, due to browser differences, the interface might look different between platforms.

The scope of the Titanium API makes the addition of new platforms difficult – implementing the Titanium API on a new native platform is a massive undertaking. The price to pay for being able to create tailored apps for each platform is a lower code re-usage. Considerably more time must be spent for porting an application from one platform to another.

- + Compiles to native code, providing better performance than in-browser apps
- + Can be better tailored by use of native UI of each OS.
- + Provides a richer set of phone functionality
- + Has optional pay-to-use platforms
- - Higher barrier for new developers
- - Lower code re-usage (60-90% according to them selves) than in-browser-apps, requiring effort to port app to different platforms

3.1.4 Conclusion

From our findings we find that the Cordova-platform has a lower barrier for entry than Titanium. We believe that this is of greater importance than the performance advantage and access to native UI. This is large part due to our low need for a high performance app, and our requirement of low barrier to contribute to the platform 2.5.

Between PhoneGap and Evothings, the difference seems to be small due to both of them being based on a Cordova engine. Applications made with Evothings are designed to be built as regular Cordova apps[3], which are compatible with PhoneGap. Hence, valid Phonegap code will run with Evothings and visa versa. Therefore, we have chosen to develop and test in the Evothings framework. Its suitability in our project with regards to Bluetooth devices and communication with tangible hardware is unmatched. It also provides a few very handy development features, such as instantaneous deployment to the phone. This allows for rapid testing and development . If the services PhoneGap provides is of interest later, a transition to using that is expected to go swiftly.

[3] As shown in http://evothings.com/doc/build/Cordova-guide.html

3.2 Game engines

The purpose of a game engine (GE) is to structure the game components and allow reuse of its code in other games. It also provides features for common components of games, such as file loading, audio playing and graphics handling. In large part, the AnyBoard platform is a GE for hybrid board games, albeit a simple one. It also differs in that AnyBoard provides board game entities, and communication with digital tokens. This is unlike typical GEs, that rather focus on timing, sound and graphics. Since creating a GE is a massive task, we would like to reuse existing open source game engines to include these features in AnyBoard or find game engines that can complement the functionality of AnyBoard when used side by side.

Common features of a GE include:

1. **Graphics engine/Renderer** – Assisting the rendering of graphical elements on the screen.
2. **Map creation tools** -Providing an easy way to create maps (or in our case boards), and structures suited for abstracting locations and its properties.
3. **Asset handler** - Early loading of game assets can put unnecessary load on memory, while late loading can lead to a low performance. An asset handler loads the game elements at an appropriate time.
4. **Physics Engine and collision detection** - Giving the game a realistic feeling of physics.
5. **Artificial Intelligence** - Some game engines provide components to support creating computer-based opponents.
6. **Signals and event handler** - With the exception of certain genres, games are commonly based on events initiated by the player, such as clicking buttons or interacting with elements in the game. A game engine often includes mechanisms that helps notify parts of the system that have registered to such events.
7. **Other** - Game engines can also provide simplification for networking in order to support multiplayer games or interactions with internet servers.

The purpose of finding a suitable GE is to enable the development of more complex games in collaboration with AnyBoard. When using an established GE, one can create game with complex graphics, scale graphics, create maps or play sounds more easily than without. If examples of AnyBoard is displayed in collaboration with popular GEs, we might also be able to more quickly recruit developers from that community.

3.2.1 Criteria

We've judged our options for a GE in regards to the following criteria:

- Licensing - As with other parts of the software platform, we prefer an open source, free-to-use game engine.
- Compatibility – As we have chosen Evothings (Cordova-based) as cross platform tool, a game engine based on the same programming language (JavaScript) is required.
- Popularity and activity - We want the software platform to be based on tools that are popular and are being actively developed. We've judged this on 1) Number of developers that have marked the repository as a favorite. 2) Number of code-changes done to the repository the last 12 months
- Performance - As one of the drawbacks of Cordova is performance, we wish for the game engine to be as light and efficient as possible. This can be hard to judge without testing, so we have used library size as a proximity to this. We have also looked at whether or not mobile platforms seems to be a target audience for the platform.

3.2.2 Candidate game engines

There is a large community around JavaScript, and there are a lot of options for a JavaScript-based game engine. Our candidates has been chosen from overview and comparisons by several community resources such as HTML5GameEngine[4] and Github[5]. Several were excluded immediately if they clearly didn't meet our criteria. Our candidates can be seen in table 3.2.

3.2.3 Evaluation

We have chosen CraftyJS, Phaser and Quintus to evaluate further.

All of these have compatibility with with Tiled[6] - a visual game map editor for creating maps. All three also support our other basic technical requirements, such as signal/event handlers, and have examples that indicate performance beyond our requirements.

Quintus is some way just what we want. It's a simple library that covers our basic technical requirements, and seems to support extendability well. It was originally made as an example for a book on game development, HTML5 Game Development(14), and is therefore well documented. They say on their own

[4] html5gameengine.com

[5] github.com/showcases/javascript-game-engines

[6] Tiled (mapeditor.org) - an commonly used editor for creating game maps.

Framework	License	Size[a]	Activity[b]	Community[c]
CraftyJS	MiT	99KB	81/7k/3k	1772/97
LimeJS	Apache	200KB*	5/54/21	1320/30
ImpactJS	Propr.			
Ludei	Propr.			
EnchantJS	MiT	225KB*	19/202/282	1300/25
MelonJS	MiT	168KB	333/14k/8k	1367/33
Quintus	MiT	20KB	17/793/509	1000/29
Phaser	MiT	692KB	1.1k/230k/117k	8694/140

Table 3.2: Overview over candidates for game-engine. Preferable properties of the platforms are marked in light green, less preferable in orange, while undesirable properties are marked in dark red.

[a] Size of minified JS libraries. Libraries marked with * are unminified size
[b] Files changed/additions/deletions in the 12 months between June 2014 to June 2015. k = thousand
[c] Number of favorites/Number of contributors to repository pr. June 2015

website[7] that it *does very little game-wise by itself and provides little more than a backbone for the other modules to build around*.

Its downside is a low level of activity and small community. Its lower amount of activity can to some extent be contributed to lower code base size, but the difference in community from Crafty and Phaser still shows that Quintus is not as well established and polished. On their Github page[8] it states *"Warning: Quintus is at a very early stage of development, use at your own risk.*"*.

CraftyJS is a larger library than Quintus, and provides a richer set of built in functionality. Documentation seems to leave us with the impressions of a modular and easily extendable architecture. One comparison of Phaser and Crafty(15) has listed this as the main advantage of Crafty. It supports mobile, even though that's not a focus from the looks of their webpage.

Phaser is clearly the most mature and popular of the alternatives we have investigated. Phaser have more game examples and thorough guides than our other candidates. The framework has support for TypeScript and JavaScript in addition to providing an in-browser game editor. Its graphics engine is PixiJS, a standalone webGL renderer which claims performance to be their strength.

Phaser seems suited for this project in many ways. It's focus is mobile, and provides a scaling manager component for handling various screen sizes. It has

[7] html5quintus.com/guide/core.md
[8] github.com/cykod/Quintus

the community and guides to make it easily understood and learnt, and feels like a solid component to base our project upon.

Our only concern is the suitability for such a polished product to be integrated in AnyBoard. While it does provide a plugin system, it could be difficult to encapsulate Phaser in a new framework. Users have reported that Phaser can feel clumsy and not modular enough(15, 16), which can make it hard to break apart and reuse relevant parts alone.

3.2.4 Conclusion

Both Quintus, Crafty and Phaser covers our basic criteria. Their differences lies mainly in size, amount of functionality and popularity. The more popular, the more solid it seems, and the less functionality will we be required to make from scratch. On the other hand, it can be harder to encapsulate and reuse only parts of the GE, and the game engine part can end up being unnecessarily heavy.

Quinitus is a great choice for learning purposes, and accompanied by its book(14) developers could learn about game engines, HTML5 game development, and then reuse that knowledge and parts or whole of Quintius in collaboration with AnyBoard.

Phaser on the other hand, seems to have a great community behind it, and has an appealing mobile-first focus. We believe creating examples with Phaser could spark interest among its large community and existing developers. This GE could also be used in collaboration with AnyBoard to simplify the making of professional and polished games. However, its size and complexity might prove too difficult to dissect and integrate with AnyBoard.

Due to these observations, and the very dynamic and active environment around game development in JavaScript, we will attempt to create AnyBoard as a *standalone* platform, independent of other GEs. This way, AnyBoard could be used in different settings, from educational to professional purposes. This will allow developers to use their own preferred library in collaboration with AnyBoard. *When providing examples where a GE is needed, we will go for Phaser* due to it having the largest community.

Chapter 4

System Design

4.1 Requirement specification

The requirements are written as stories inspired from agile methodologies. Instead of attempting to write a complete and detailed description of the system, where all requirements is to be implemented, we have written down a backlog of conceivable requirements and wishes, and implemented them with what we assert to be the most important ones first. Requirements are subject to change, and some have not been implemented either due to time or proving unnecessary during the development phase.

4.1.1 Non-functional requirements

Considering that our goal is for the AnyBoard library to be used by different developers in creating games, as well as further developed and maintained, we wish to lower the threshold and difficulty of using the library, as well as contributing to it.

We believe this is done by creating the library and environment in the following ways:

NFR-1, Setup should be simple, for both game developers and developers of AnyBoard - The first barrier of development is setup of the environment. To minimize this, a developer should ideally have to go through as few as possible steps before he or she can use the library, or contribute to it. This is relevant both to the game developer using AnyBoard as a library to create their own game, as well as a developer that wish to contribute or change the AnyBoard library itself.

We believe a simple setup is important for curious developers to test the library,

and for more dedicated developers to take the step and modify it for their own needs.

NFR-2, AnyBoard should be provided with a rich set of examples - A common way of understanding the usefulness and workings of any programming library is looking at simple examples demonstrating the code syntax, function and the effects of using that library. They can work both as a way of marketing the library to game developers, and a way of assisting developers in need of help or a starting point for their own game.

This can be done through publishing the code of complete games built with AnyBoard platform, as well as through small code snippets demonstrating some concrete functionality or even test code.

We believe examples are very important for the adoption of the AnyBoard library by game developers.

NFR-3, The features and functionality of AnyBoard should be tested, and easily available - Tests should be implemented, so that one can confidently change parts of the code without unknowingly breaking part of the code, as well as providing small examples of library usage.

As a code-base grows in size, the barrier to contribute to the code base increases. There are more classes and functions to be understood before one feels confident. One can often feel uncertain of how changing one part of the code affects the rest of the code base, and in fear of breaking some other functionality unknowingly, refrain from changing anything.

Having written tests for the functionality of the software, can dramatically lower the fear of unknowingly breaking parts of the code-base, and as such lower the barrier for contributing to the AnyBoard library. It also provides examples of how features are used and expected to work, which complements rich examples as well as documentation.

We believe tests are very important for the further development of the AnyBoard library.

NFR-4, AnyBoard must be consistently documented, in an accessible and understandable matter, both for new and experienced developers - Non-internal elements of the framework (Classes and methods intended to be public) should be well documented. Names, parameters and types of parameters should be easily attainable.

The larger a code base is, the more difficult is it often to understand. By providing

good documentation, in form of both in-line coding, auto-complete functionality for editors and online documentation, we lower the time used by both game developers and developers of the library to understand and use the it. We believe that providing good documentation is vital for the chances for the library to be used and developed further.

NFR-5, AnyBoard should strive to have decoupled code, as to maximize re-usability of different parts, and be independent of other libraries - Code should be decoupled, so parts can easily be replaced without having cascading effects on other parts of the code base.

An example of this would be using dependency injection: The token manager for instance will be using some sort of Bluetooth communicating capabilities in order to communicate with the token. By passing the Bluetooth driver to the manager class upon creation, instead of directly calling the Bluetooth drivers we know of, the Bluetooth communication libraries will be decoupled from the AnyBoard library.

Decoupling of code allows changing or replacing parts of the our library more easily.

4.1.2 Functional requirements

Here we present stories that relate to the hardware tokens and the interaction between those and the software platform.

Tokens

The ability for simple interaction with tokens is the main feature of AnyBoard that sets it apart from other game engines. Hence, the requirements relating to this is the main part of the platform. Our requirements are shown in table 4.1, and is related to simple scanning, connecting and listening to tokens and its interactions as well as decoupling tokens from game code.

Graphics

Board games will include common features, such as menus or rules for games. We wish to provide common entities for quick development of these graphical elements. The requirements relating to this are shown in table 4.2.

Entities

Table 4.3 shows the requirements relating to logical board game entities. Hybrid board games will include many of the same entities, which was described in section 2.2. We believe it could be useful for developers to have these available.

Table 4.1: Functional requirements for token capabilities

	Functionality	**Priority**
FT-1	*As a developer*, I need a token manager that holds all tokens, so I easily can obtain the individual tokens instances from different parts of the code.	Low
FT-2	*As a developer*, I want the token manager to be able to scan for and return active tokens nearby, so I'm not required to have intimate Bluetooth knowledge.	High
FT-3	*As a developer*, I want to be able to replace the token driver without changing my game code, so I can efficiently test and try different tokens.	High
FT-4	*As a developer*, I want tokens to trigger events upon token-token events or token-constrain events, and allow me to subscribe to this and other events, so I can respond to token interaction.	High
FT-5	*As a developer*, I want to be able to obtain standard tokens and drivers, so I can quickly create a game without having intimate knowledge of hardware or low-level code.	High
FT-6	*As a developer*, I need to be able to test how my game is using tokens and responding to token interactions within a digital interface, and not have to use physical tokens for testing, so I can test more efficiently and develop games with tokens I have not acquired.	Medium
FT-7	*As a developer*, I need to be able to connect simultaneously to at least four different tokens.	High

Table 4.2: Requirements for graphical interface entities

	Functionality	**Priority**
FG-1	*As a developer*, I want to be able to create menus and menu items that link to new screens or internet web, so I can create a menu-feature quickly	Low
FG-2	*As a developer*, I want to be able to quickly be able to create a rule-screen from a set of written rules and a FAQ from a set of questions, so I can create this feature quickly.	Low
FG-3	*As a developer*, I wish to be able to retrieve rules and FAQ from a web page, so I don't have to have duplicate information in case I have a web page.	Low

Table 4.3: Functional requirements for game entities

	Functionality	**Priority**
FE-1	*As a developer*, I want to be able to build on a generic Resource model, so I don't have to spend time implementing interactions such as paying, trading or giving resources between Players and/or the Game.	High
FE-2	*As a developer*, I want to be able to build on a generic Card and Deck model, so I don't have to implement standard usages such as shuffling, drawing, playing.	High
FE-3	*As a developer*, I want to be able to build on a generic Player model, that holds a Pawn, has a set of resources, a set of cards, a set of missions, points and similar, so I don't have to implement those	High
FE-4	*As a developer*, I want to be able to build on a generic Dice model, so I don't have to spend time implementing interactions such as rolling and returning values from a set of dice.	High
FE-5	*As a developer*, I want to be able to build on a generic Timer model, so I can create events that happens at a set amount time into the game.	Low
FE-6	*As a developer*, I want to be able to build on a generic Turn model, so I can quickly specify what goes into each turn of a player.	Medium
FE-7	*As a developer*, I want to be able to build on a generic Board and Tile model with listeners and triggers, so I can quickly specify where one can go from which tile, their distance, what happens when one steps on a tile without having to create the models from scratch.	High
FE-8	*As a developer*, I want to be able to log events of different severity, and be able to see them, so I can debug my code and see where something fails	High
FE-9	*As a developer*, I want to be able to assign properties and fields to all my models, so I can create my own concepts and define my own properties on top of the existing models.	High

4.2 Application architecture

AnyBoard will exist over three types main parts, as first shown in figure 2.8. **Tokens**, such as the AnyBoard pawn, or printer, are the digital tokens that users can interact with. Tokens talk with the second part, the **Controller**. The controller is run on a smart device, such as a mobile phone or a tablet. Here, all the logic lies for the game, including the rendering of graphics. The third part is a web based community. This part is conceptual, and will not be implemented in this thesis. The application architecture that we will work on in this thesis, consists therefore of the controller and token part, which is shown in figure 4.1.

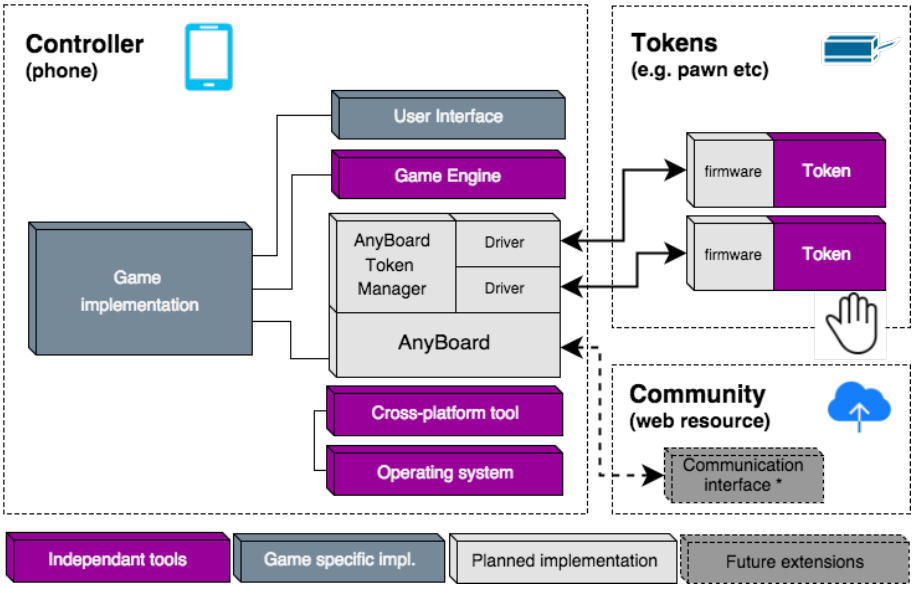

Figure 4.1: Architecture overview. AnyBoard will exist over three components. Digital tokens such as tangible pawns, a game controller/hub running on a smartphone, and an online application assisting the a community of both developers and players. My contribution in this thesis consists of the light grey boxes, denoted "Planned implementation".

4.3 Phone-Token communication

The means of communicating easily with different tokens is the most central component of AnyBoard. We aim for decoupled token-specific code, to address the requirements in table 4.1.

TokenManager is a static component that handles the scanning and connecting/disconnect to tokens. Its purpose is to identify nearby tokens, and determine an appropriate driver for the token to use. TokenManager has an own

driver for this purpose.

In figure 4.2 we see this illustrated. Upon finding suitable drivers for detected tokens, TokenManager instantiates Token A1, Token A2 and Token B. These are Token Class instances, which – seen from the developers point of view – work as a sort of API on top of the tokens. These abstract away the low level communication and commands between AnyBoard and the specific tokens, and provide affordances such as ledOn(), ledOf() and print() functions.

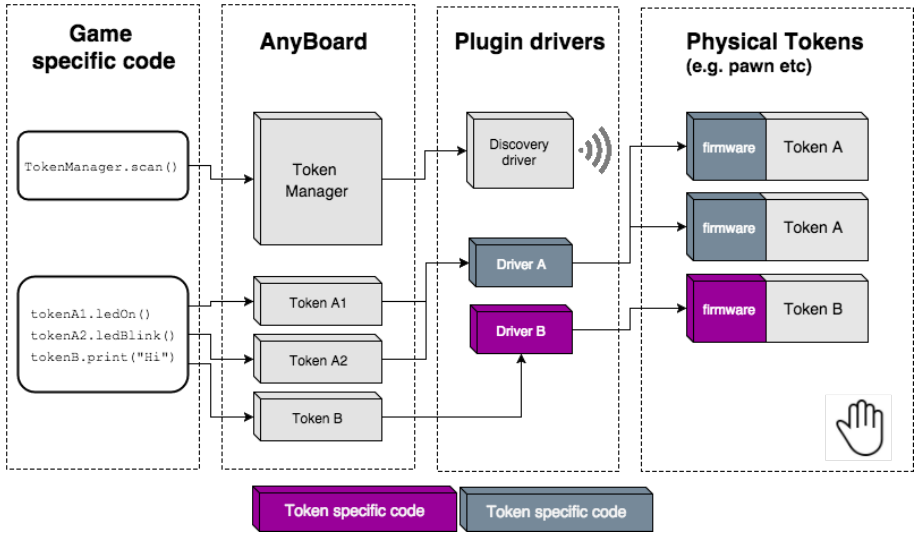

Figure 4.2: Communication overview between phone and token. An example where three tokens are discovered by the discovery driver and mapped to two different drivers, which handles the communication with their compatible tokens.

4.4 Web-Phone communication

Web is a component of AnyBoard that will not be implemented in this project, and will hence not be discussed or designed in any detail here. However, it's worth to note some thoughts about this important capability.

We imagine to use HTTP-communication where the phone sends requests to a API on the server. Suggested uses for this communication are:

1. Retrieving updated FAQ for the game.
2. Sending of events and actions in the game to enable multiplayer functionality, or viewers to watch a game.
3. Sending of events and actions in the game for history/logging.
4. Gathering statistics of usage.

5. Download new games from a game store
6. Retrieve updates for a current games

The basic tool necessary to enable this sort of communication is capability of communication over the HTTP-protocol. This capability could also be a useful feature for developers that wish to encapsulate web services into their board game or use physical devices that communicate using web APIs.

This capability is already built into the JavaScript language, with the class XMLRequest. Abstracted HTTP calls using Ajax is also available in a variety of other JavaScript libraries. We therefore see no obstacles for this capability, even without designing for it in the current phase.

Chapter 5

System implementation

This chapter describes the implementation of AnyBoard library. We follow the chronological implementation of the library, by starting with the setup of the environment, then the entities of the board game, followed by Bluetooth communication and token implementation. We'll describe what technical choices were made, our motivation for them, and what part they play in the project.

5.1 Development environment

During the development of AnyBoard, we have used some tools to simplify certain tasks for us. This section explains the different elements and tools that we have used during development of the AnyBoard library. These tools are not required, nor related to the use of the AnyBoard framework in creating board games, only in the development of the library itself. However, they have been of great assistance in accomplishing the non-functional requirements from chapter 4.1 regarding documentation, testing and quick setup. More information regarding these tools can be found in appendix A.

For the implementation of the AnyBoard platform itself, see section 5.2

All tools used here are based upon a NodeJS platform. Developing AnyBoard in this environment is therefore supported by those operating systems that support NodeJS (Linux, Mac OSX, Windows, SunOS).

5.1.1 Platform and dependency handler: NodeJS

NodeJS[1] is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. It is used in the project as a dependency

[1] nodejs.org

handler and platform for our other tools, i.e. it gives us the opportunity to use the other .

We have chosen to use NodeJS as a tool in this project due to the active community around it, which has resulted in it being a reliable platform with a rich variety of tools that can be found and used with it. In this project, it provides us with a large set of possible tools and libraries to simplify development. It also gathers all dependencies and specifies them in one file, and provides a *simple setup* (requirement NFR-1) of dependencies for the environment.

Being the by far most dominant JavaScript-based platform and something that we have previous experience with, we have not considered other options.

5.1.2 Test framework: MochaJS

Mocha [2] is a JavaScript-based test framework. It is used in the project in providing an easy to understand syntax in writiting tests, and reports the results of tests in a readable manner (see figure A.3 in the appendix).

Testing frameworks provides a clear syntax for testing, as well as methods and constructs for asserting whether or not function work as intended. Using a test framework and *writing tests* (requirement NFR-3) for our library provides confidence to developers that they don't break the code unknowingly when changing the code. It also *provides examples* (requirement NFR-2) of how to use the library.

We have chosen MochaJS over alternatives like UnitJS*unitjs.com* and VowsJS due to our familiarity with it, and great reporting format.

5.1.3 Task runner: Grunt

Grunt[3] is a JavaScript-based Task Runner. It is used in the project to simplify building the AnyBoard library, generating documentation and running tests.

Task runners simplify common tasks, such as concatenation of files, compilations etc. by allowing us to create shorthand commands for doing larger tasks. Our motivation is simplifying the creating of documentation files, concatenation and minifying our library from smaller modules, as well as test running. This contributes to the requirement NFR-1.

Grunt was chosen over alternatives such as GulpJS[4] and BroccoliJS[5] due to our

[2] mochajs.org
[3] gruntjs.com
[4] gulpjs.com
[5] broccolijs.com

prior experience with it, as well as larger community.

5.1.4 Documentation generation: JSDoc and grunt-jsdoc-to-markdown

JSDoc[6] is a standardized way of documenting JavaScript code. It allows IDEs[7] to give assisting information to a developer about the code he's using, which classes and methods are available, what they return, take as parameters etc.

In addition, software plugins for our development platform (NodeJS + Grunt) exist that allows us to generate automatic documentation based on JSDoc syntax.

Documenting code usually makes the further development and maintenance of code simpler. Doing it in certain ways, will also allow developers that create games with the AnyBoard libraries to be informed of which AnyBoard classes and methods that are available in his environment, as well as providing parameter list and types, provided they use a compatible IDE.

A third reason is the automatic generation of human readable documentation files, such as HTML or MarkDown files, that can be used to inform new developers of their available tools in the AnyBoard platform. Generating this documentation automatically instead of writing it manually saves us time, and gives us *consistency between different sources of documentation* (requirement NFR-4), namely from the IDE, inline code and online documentation.

We have chosen to document our code with JSDoc-oriented syntax over YUIDoc and Doxx due to JSDoc doing source parsing[8], and high extensability.

For the plugin that runs the documentation generation, we have used grunt-jsdoc-to-markdown[9]), over jsdox[10] due to its very readable output designed for Github pages (which the project already uses).

5.1.5 Summary

A lot of thought and effort has been put into the development environment, in order to ensure simple setup and deployment, quick generation of usable files and updated documentation. We feel that this has been well spent time, and goes far in addressing NFR-1: *Setup should be simple, for both game developers and developers of AnyBoard.*

---

[6] usejsdoc.org - a JavaScript documentation syntax and parser

[7] IDE is shorthand for Integrated Development Environment and is simply put a rich featured editor.

[8] JSDoc considers source code in addition to comments in order to generate documentation

[9] grunt-jsdoc-to-markdown - a grunt package generating Github friendly documentation generator of JavaScript files commented with JSDoc syntax, that we decided to use

[10] grunt-jsdox - a another documentation generator of Javascript files using JSDoc syntax

```
/** Represents a set of game dices that can be rolled to retrieve a random result.
 * @constructor
 * @param {number} [eyes=6] *(default: 6)* number of max eyes on a roll with this dice
 * @param {number} [numOfDice=1] *(default: 1)* number of dices
 * @example
 * // will create 1 dice, with 6 eyes
 * var dice = new AnyBoard.Dices();
 *
 * // will create 2 dice, with 6 eyes
 * var dice = new AnyBoard.Dices(2, 6);
 */
AnyBoard.Dices = function (eyes, numOfDice) {
    this.eyes = eyes || 6;
    this.numOfDice = numOfDice || 1;
};
```

Figure 5.1: Example of jsdoc commenting in source code.

Game developers are able to use AnyBoard by including the following line in their code:

```
<script src="path/to/anyboard.js"></script>
```

While developers of the AnyBoard platform will only be required to install node and its dependencies. For an elaboration on this, see appendix A.1.

new AnyBoard.Dices([eyes], [numOfDice])

Represents a set of game dices that can be rolled to retrieve a random result.

Param	Type	Default	Description
[eyes]	`number`	`6`	number of max eyes on a roll with this dice
[numOfDice]	`number`	`1`	number of dices

Example

```
// will create 1 dice, with 6 eyes
var dice = new AnyBoard.Dices();

// will create 2 dice, with 6 eyes
var dice = new AnyBoard.Dices(2, 6);
```

Figure 5.2: Example of generated documentation by grunt-jsdoc-to-markdown.

5.2 Logical Entities

The requirements of logical game entities were listed in table 4.3. With the exception of requirement FE-5, FE-6 and FE-7, the requirements were met. We here go through the the entities available in the current version of AnyBoard, and how they relate to the functional and non- functional requirements.

Examples of use and thorough documentation is available in appendix C and I for all entities, with regards to non-functional requirements NFR-2 and NFR-3. The entities has been subject to automatic testing (see appendix E) with regards to requirement NFR-4.

5.2.1 Decks of cards

AnyBoard provides two classes *AnyBoard.Deck* and *AnyBoard.Card* that combined addresses requirement FE-2: *As a developer, I want to be able to build on a generic Card and Deck model, so I don't have to implement standard usages such as shuffling, drawing, playing*.

The explicit part of the requirement was straight forward. A Deck in AnyBoard reads a JSON object[11] in order to create a Deck instance. The Deck consists of a *pile* and a *used pile* that initially is full/ empty of instances of *Card*.

[11] See Deck example in appendix C for how this object is structured

The affordances to draw, shuffle and play cards are also implemented, and the deck can optionally restock itself from the used pile or a brand new stock of cards.

However, we saw that this requirement is actually more complex than first expected. In board games, cards can have a wide range of consequences. Some might provide resources, while others can disrupt the game and go far beyond the regular flow of the game. Some cards have to be played at once, while others at will. In order to accommodate this large variety of consequences for drawing and playing cards, both Deck and Card entity has been given the affordance to execute custom code upon these events, through the methods *onPlay* and *onDraw*.

The Card model also allows for custom properties to be added, as part of addressing requirement FE-9.

5.2.2 Resources and transactions

The classes *AnyBoard.ResourceSet* and *AnyBoard.Resource* has been implemented to address requirement FE-1: *As a developer, I want to be able to build on a generic Resource model, so I don't have to spend time implementing interactions such as paying, trading or giving resources between Players and/or the Game*. The player model (described in the next subsection) complements these classes in addressing this requirement as well.

A Resource defines a valid type of game resource, and is simply a representation of a string, e.g. "Gold" or "Silver", but with the possibility to attach custom properties, as a part of addressing requirement FE-9.

A ResourceSet provides helpful functions to handle sets of resources. A ResourceSet could for example contain 4 Gold and 3 Silver. It provides the functionality of adding and subtracting resources to the set, as well as seeing the similarities between two ResourceSets, or checking whether or not a ResourceSet is contained by another. A ResourceSet can either allow or not allow containing a negative amount of Resource.

5.2.3 Player

The AnyBoard.Player class addresses requirement FE-3: *As a developer, I want to be able to build on a generic Player model, that holds a Pawn, has a set of resources, a set of cards, a set of missions, points and similar, so I don't have to implement those*.

An instance of the Player class holds a set of Cards and a ResourceSet. It provides the affordance to *pay()*, *receive()* and *trade()* a ResourceSet, which complements requirement FE-1. It also provides functions to *draw()* from a Deck

and *play()* a Card, which complements requirement FE-2.

The set of cards belonging to a Player exists in its own class, called Hand. This provides functions to *discard()* a card or the whole hand, as well as *has()* to see if the Hand contains a certain Card.

We have not implemented any connection between a Player and a Pawn, nor have we implemented missions or points. We still think this connection is a good idea, but under the time constraints of this thesis, these elements were not prioritized. However, we have addressed the requirement for custom properties (FE-9), which would allow a developer to attach a token or other property to the Player class as he or she sees fit.

5.2.4 Dice

AnyBoard.Dice is a generic implementation of game dice. It allows the creation of one or more dice in a set, with any amount of eyes. This addresses the requirement FE-4: *As a developer, I want to be able to build on a generic Dice model, so I don't have to spend time implementing interactions such as rolling and returning values from a set of dice*

5.2.5 Logging

AnyBoard.Logger is a static log class that addresses requirement FE-8: *As a developer, I want to be able to log events of different severity, and be able to see them, so I can debug my code and see where something fails*.

Logger is a simple class that affords methods for logging of different severity: *debug()*, *log()*, *warn()* and *error()*. In addition, the threshold for which severity of logging it should perform can be set through *setThreshold()*. This can be useful for enabling full logging output during development and testing.

The actual output of the logger, whether to a file, or a console is decoupled from the logger itself. AnyBoard.Logger contains a log handler. By default, it uses the console object available in most JavaScript environments. If used with Evothings, the log events with also be output through the logging tool available in Evothings, hyper. The log handler can be switched with one suitable for the needs of the developer. This is related to requirement NFR-5 regarding decoupled code.

5.2.6 Summary

The implemented entities provide a good starting point for creating hybrid board games, and we're pleased with the resulting components. No notable problems arose during development of the AnyBoard entities. FE-5 (Timer), FE-6 (Turn) and FE-7 (Board, Tile) were not implemented, due to time limitations. We still

see a purpose for these entities, and discuss their potential implementation in Discussion (section 7.

5.3 Token Communication

Implementing capabilites for Token communication was without doubt the hardest, most time consuming and trying part of the development, but also the central and unique component of AnyBoard.

In this section we will take a look at the implementation of the AnyBoard components relevant to this, the TokenManager, BaseToken and Driver. The first two classes are in large part containers of information, with APIs for game developers to access, while the Drivers handles the communication with the physical tokens.

Examples of use and thorough documentation is available in appendix C and I for all entities, with regards to non-functional requirements NFR-2 and NFR-3. The entities has been subject to automatic testing (see appendix E) with regards to requirement NFR-4.

5.3.1 Token Manager

AnyBoard.TokenManager is a static class of AnyBoard. Its purpose is to handle, scan for and connect to tokens. For this purpose it has a plugin "discovery" driver that enables this communication. Such a driver has been provided during the development. The TokenManager class itself is a "shell" around this driver, and provides the following affordances to the game developer:

scan() probes for tokens nearby. Which types of communication, whether it is Bluetooth, HTTP or both, is up to the plugin driver. Upon finding tokens, an uninitialized BaseToken (see the following subsection) will be created for each of them, containing address information and the ability to connect to the Tokens. As such, this addresses requirement FT-2: *As a developer, I want the token manager to be able to scan for and return active tokens nearby, so I'm not required to have intimate Bluetooth knowledge.*

The details of this procedure is shown in figure 5.3. Some code is executed from a game implementation, which calls the scan() function (1). TokenManager in turn calls upon its plugin driver to carry out the scanning (2). In our case, the driver makes use of Evothings and Cordova libraries (3) to create a Bluetooth signal that communicates with all nearby Bluetooth devices (4). For each token that responds, the driver will create instances of BaseToken, which it returns back to the original caller.

setDriver() replaces the driver used for discovering tokens. The driver must implement methods for scanning for and (dis-)connecting to tokens, and will be rejected otherwise. All the communication logic and methods retain in the driver.

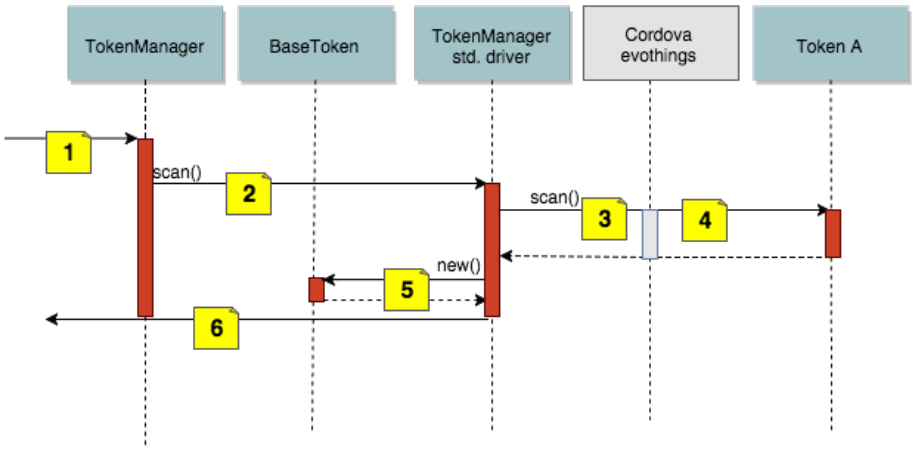

Figure 5.3: Flow chart for executing scan() functionality on TokenManager class

As such, one could change the method of connecting to tokens by simply exchanging the driver. This has been done to address requirement NFR-5, regarding decoupled code.

onTokenEvent() are one of six methods in the same category. We have categorized token interactions in three different types based on the article in appendix H, namely Token event, Token-Token event and Token-Constraint Event. TokenManager provides two methods for executing custom code upon each of these three types of events. One method is provided for inserting custom code to be executed every time, onToken...(), and another method is provided to be executed the next time, onceToken...(). This addresses requirement FT-4: *As a developer, I want tokens to trigger events upon token-token events or token-constrain events, and allow me to subscribe to this and other events, so I can respond to token interactions.*

The flow of these events are identical to that shown in figure 5.5, with the exception that the insertion and triggering of code (marked 1 and 6) goes via TokenManager instead of BaseToken.

get() allows retrieval of any Token that the TokenManager has scanned. This addresses FT-1: *As a developer, I need a token manager that holds all tokens, so I easily can obtain the individual tokens instances from different parts of the code.*

5.3.2 BaseToken

Connecting and disconnecting: Immediately after being scanned by the TokenManager, the BaseToken instance is not connected. For this to occur, one

has to use the method connect().

The flow of connecting when using the current Bluetooth plugin-driver is shown in figure 5.4. Upon calling the connect function on a BaseToken (1), the token will use the TokenManager plugin driver to connect to the token (2). It will use this instead of an own driver due to that necessary information to determine a suitable driver is unavailable before connection is established. The driver will then connect to the token (3) and read available information (4) from the token. Based on the services and characteristics a token responds with, the driver will find a compatible driver from the Drivers class (5). It will set this as a new driver for the token (6), and (if implemented) call the initialize method on the new driver (7). In our developed Bean and Token driver, this will in turn subscribe to any events and updates from the token (8).

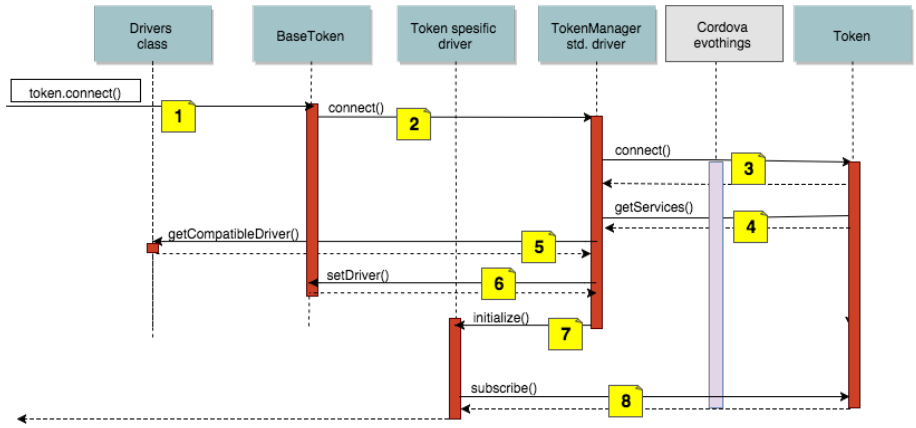

Figure 5.4: Flow chart for connecting to a token, after discovery.

Event listeners are implemented via the functions on(), once() and trigger(). In addition to the token-, token-token-, and token-constraint events, these events can be manually initiated by code, and also include reporting of minor changes, such as a change of LED color.

The methods on and once inserts custom code to be run always or once on certain events, while trigger() executes the inserted code for that event. Figure 5.5 illustrates how this works. First, code from the game developer states that it wish to insert custom code to be run upon a certain event (1), here the event is 'EVENT'). Then, a player interacts with a token (2). The token registers being acted upon (3), and sends a code (4) describing what sort of interaction has occurred. The token specific driver interprets the signal, and triggers the event (5) on the BaseToken instance that represents the token. If this event is the same type

as the developer has stated that he wish to listen to, his code is being run (6).

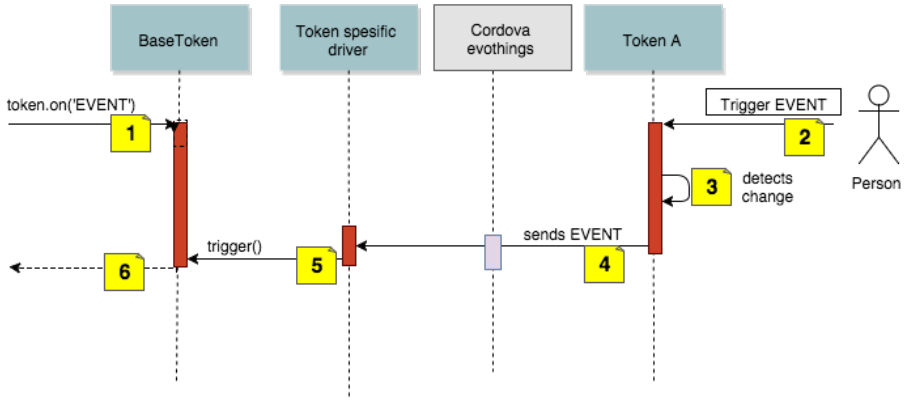

Figure 5.5: Flow chart for adding event listener, and its executing on BaseToken class

Token functionality calls are methods that communicate with the physical token, such as turning on LED or printing a message. There are many such functions implemented – an exhaustive list can be seen in the documentation in appendix I. Common for them all is that they pass two parameters, *win* and *fail*, which themselves are functions. The reasoning for this is that the communication with the token can take considerable computing time, and the calls are therefore done asynchronous, allowing the program to go on executing code instead of stalling while waiting for a reply.

This concept is still unfamiliar for some developers, even though it is (now) common in JavaScript development. Several full examples has therefore been provided alongside the library (See appendix C).

The flow of executing a token functionality call is illustrated by a ledOn command in figure 5.6. The game code calls ledOn function (1) on the BaseToken, which in turn calls the same method on the Token driver (2). This results in a generic send() command (3), which adds a once() listener back on the BaseToken, telling it to trigger the *win* function next time ledOn has been triggered on it (4). If the driver doesn't have a send queue, or after the earlier commands have gone through, the driver goes on to call a rawSend command, which sends the command to the token (5). Once the token has executed its command, it responds back to the driver that the command was completed (6), where upon the driver tells the BaseToken that the event has occurred (7). The BaseToken in turn executes the stored *win* function.

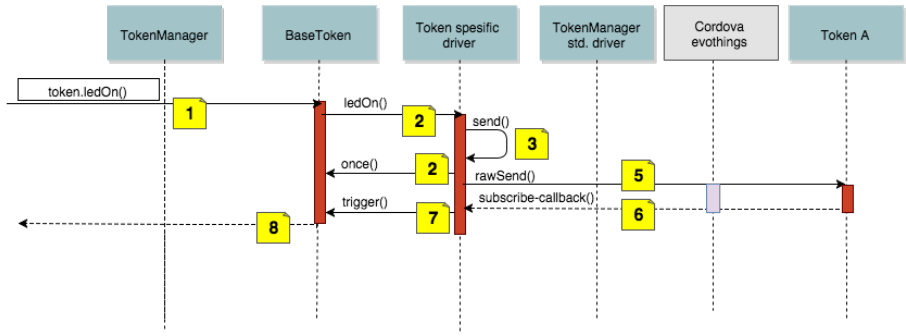

Figure 5.6: Flow chart for executing a token functionality call on BaseToken class

5.3.3 Drivers

Currently, two types of drivers has been made. The first one is a discovery driver, meant for the TokenManager class. Its purpose is to implement scanning for and connecting to tokens. Our implementation uses Bluetooth for this purpose.

The other type of driver is a Token driver. The purpose of the Token drivers is to translate the affordances of the BaseToken class to be compatible with a certain token model or set of models. As such, programming towards the BaseToken API, allows us to exchange the token and the driver without causing the game code to be invalid. This meets requirement FT-3 that states: *As a developer, I want to be able to replace the token driver without changing my game code, so I can efficiently test and try different tokens*. It's also relevant with regards to NFR-5, concerning re-usability and decoupled code.

For implementation details on the drivers, see appendix D.

Driver communication

During the initial development of drivers, firmware and their communication, we've met several challenges. The development was done with two different tokens, the LightBlue Bean and a customly assembled RFduino (more information on these can be found in appendix D). We first drafted a Bluetooth communication protocol to suit our needs, where we considered the benefits and drawbacks of using a JSON based data stream compared with a binary one. This is shown in figure 5.7.

Initially, we went with the JSON based protocol. Tests using USB-communication with RFduino worked well, while Bluetooth communication with Bean had some instabilities, but worked most of the time.

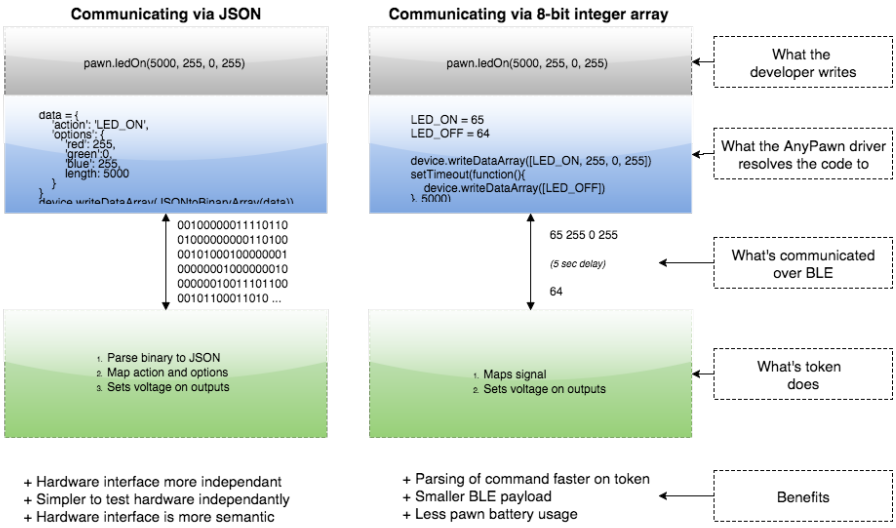

Figure 5.7: Considering the benefits and drawback of JSON vs byte communication protocol.

What we didn't know at the time was that Bluetooth low energy allowed packets for no more than 20 bytes. In order to allow larger than 20 byte packets, one would have to implement packet handling to divide and recombine packets at each end. This was not a problem for the RFduino while using USB to communicate, but over Bluetooth, our firmware attempted to parse partial packets unsuccessfully. Our code for Bean, which at the time was copy-pasted from what we could find online, proved to have a bug in it that broke every 64th byte that was sent. Which meant it worked fine sometimes, and sometimes not at all.

We also spent much time getting reading and writing serial data to work. Each token has a large list of so called services, characteristics and descriptors that notes which capabilities it has. For example, the Bean has a unique service identifier `a495ff10-c5b1-4b44-b512-1370f02d74de`, under which the characteristic `a495ff11-c5b1-4b44-b512-1370f02d74de` specifies that it can be written to. In order to write to the Bean, one has to specify both the service identifier and characteristic before sending data. Identifying these UUIDs took what felt like an unnecessarily large amount of time.

We ended up after identifying the correct UUIDs for different services, and changed our protocol to a binary one. This puts a limitation of maximum 12 byte commands to the Bean, and 20 byte commands to the RFduino. The smaller Bean packets size is due to their own protocol implemented on top of the Bluetooth

protocol, that has a 8 bit overhead. The final draft of the binary Bluetooth protocol that is currently used can be seen in appendix F.

5.3.4 Summary

With the exception of FT-6, all requirements regarding functional requirements for tokens has been fulfilled.

Requirement FT-1 through FT-4 has been addressed in the previous subsections. Requirement FT-5 says *As a developer, I want to be able to obtain standard tokens and drivers, so I can quickly create a game without having intimate knowledge of hardware or low-level code*. We have created two different token drivers, one for RFduino and one for Bean, as well as a discovery driver for the TokenManager. In addition, we've created firmware for three different token setups. Details on these can be seen in appendix D. The Bean driver and Bean Token can be bought and used without writing any line of firmware, nor adjusting drivers, nor solder any components. We regard this requirement as satisfied.

Requirement FT-6 has not been completed. A dummy token for testing purposes has not been developed. The reasons for this is the time constraints in this thesis, and being lowered in priority. When dropping the requirements regarding graphical elements, and without creating full games, there was no need for a Dummy token for testing. We still believe that this functionality is important for fast development, and allowing development to be done without the need of possessing tokens physically.

Requirement FT-7 states that *As a developer, I need to be able to connect simultaneously to at least four different tokens*. We regard this requirement as satisfied. There has not been tests performed with more than three tokens, but we have no reason to believe four connected tokens would cause a problem.

5.4 Graphical Elements

No efforts have been made to complete requirements FG-1, FG-2 or FG-3.

Graphical elements and the manipulation of these are the main element of near to all game engines. Efforts made to create a set of standard graphical elements, would therefore quickly be incompatible with most game engines. If we created a standard menu with Phaser, this would not be compatible with Quintius or visa versa. Our best option to create a widely compatible graphical elements, was considered to be by creating plain HTML elements. With the changes in browser speed and web technologies in the recent years (CSS3, HTML5, WebGL), this could also provide rich and quick graphical environments in games. However, these requirements were still downprioritized and none was implemented.

We believe that if such capabilities should be provided by AnyBoard, it would be best served as a standalone UI-library next to AnyBoard. This is discussed further in chapter (7).

Chapter 6

Evaluation

The research questions stated in chapter 1.3, in plain terms, asks how we can simplify the development of hybrid board games by limiting the usage of time, money and required knowledge. In problem definition (1.2), we pose that creating a tool – AnyBoard – to assist developers can assist in this.

In this chapter we will evaluate to what extent AnyBoard accomplishes these goals, more specifically by:

1. Lowering the money cost to developers, by providing assistance in creating board games without having acquired the hardware you develop for.
2. Lowering the knowledge requirements needed to create hybrid board games, by providing useful examples and abstractions of token communication and firmware.
3. Lowering the time spent development, by providing helpful abstractions of hybrid game concepts, and token entities.

We first present how we will evaluate this, followed by a description of the process before we in 6.3 present results. Relevant code for the evaluation is found in appendix B.

6.1 Evaluation method

We have evaluated this by implementing a quiz game that uses two RFduino pawns (see D.1) to control the game flow and answering of alternatives. The implementation has been done without having physical tokens present in order to evaluate goal 1. After having implemented it with the use of AnyBoard, the game was reimplemented without it. We then look at the differences between the implementations and evaluate how much more time and knowledge the

implementation without AnyBoard requires. This is done to evaluate goals 2 and 3.

6.1.1 Criteria

The implementations are evaluation against each other on the following basis

- C1: Number of implementation code lines.
- C2: Code readability
- C3: Number of different libraries used.
- C4: Number of different sources of knowledge
- C5: Ability to test game flow without a physical token.

Number of code lines is an indicator of time spent developing. Parts can be copy pasted, but most code lines will have to be written by the developer deliberately. We believe, more specifically, that number of instructions are closer correlated to time spent than number of lines. We will therefore use this to give us a rough estimate of the time used.

Code readability is our second criterion. Much of the time spent developing is used reading code rather than writing it. That the resulting implementation is easily understood, will lower the time spent interpreting it at a later stage, and simplify development for those that use this implementation as inspiration or learning example. We also believe it to be likely that a library with an interface that is short and easy to understand, will lower the developing time for developers that make use of that interface.

Number of libraries has been chosen as a third criterion. Each library has its own interface, and will require some time getting to know. Hence, the fewer libraries used, the shorter time is needed for research. Derived libraries, i.e. libraries that only act as a dependency for other libraries, are not considered as the developer doesn't have to interact with these.

Different sources of knowledge is included because a considerable part of the time spent developing is used finding information. We think that the more different sources you will have to go through, the more time is spent finding information. Leaning on (and finding) five web resources to write code, takes more time than leaning on one web resource.

The ability to test the game and play it, is necessary in order to be reassured that a game is finished and works as intended. Developed programs where one is unsure of what works or not, often ends up in much time spent fixing bugs.

6.2 Evaluation process

The evaluation process have been done by designing a small quiz game that includes some of the common token interactions that AnyBoard will support. This includes discovering and connecting to token, as well as responding to token-token events and token-constraint events (see appendix H).

The game has then been implemented using the AnyBoard platform. AnyBoard has then been removed from the implementation and replaced with plain JavaScript. The Evothings Bluetooth library that has been used in the development of AnyBoard has also been used in the JS-implementation[1]. This way we get to isolate the difference of only AnyBoard library itself.

Both implementations has been done without physical access to the relevant token, in order to simulate a situation where a developer can't afford, or don't currently possess the token he or she is developing for. We have done this evaluation ourselves, without the use of volunteering developers.

6.2.1 Implemented game

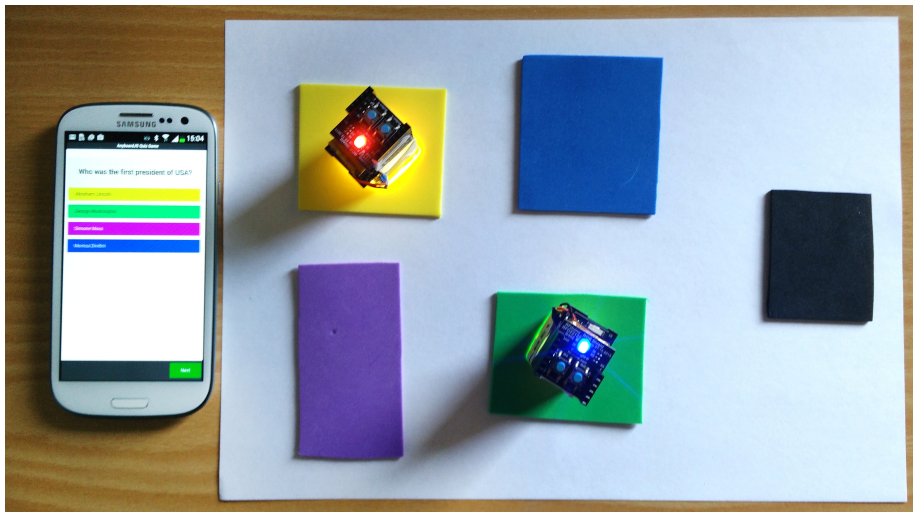

Figure 6.1: The quiz game implemented for evaluation. Each player places his or her token on the colored tile corresponding to the alternative.

The game we have created for evaluation is a simple quiz game for mobile phones. The game itself, all of its screens, and code structure can be seen in appendix B along with differences between the two implementations. The code

[1] JS: A common abbreviation for JavaScript

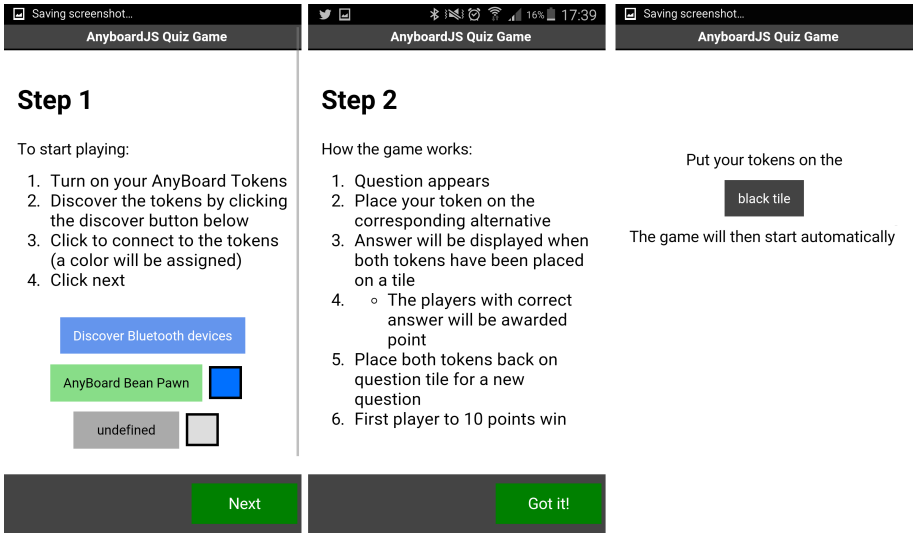

Figure 6.2: The setup phase of the quiz game, assisting the player in connecting to tokens and place the tokens on the starting point.

for it can be seen in full and downloaded at github.com/tomfa/anyboardjs-evaluation.

The game consists of a short setup phase where the tokens are being connected to, assigned a player color and the player places the tokens on the board. This phase it shown in figure 6.2. This allows us to evaluate the difficulty for players to set up the game. Placing all the tokens on a question-field will both trigger Token-Constraint events as well as Token-Token events as two or more tokens are placed on the same location.

After having set up the game, the players will iterate between answering questions and reading solutions as shown in figure 6.1. A player answers the question by placing his or her token on the colored tile that corresponds to that alternative. This triggers a Token-Constraint response event in AnyBoard, which checks whether or not all players has answered yet. If all players has answered, the solution will then appear. A new question will appear once all players has placed their tokens back on the question-field.

6.2.2 AnyBoard implementation

The AnyBoard implementation of the game was done deliberately by separating code into the following parts:

- index.html, marking up the elements of the game.

- style.css, defining styling of elements.
- data.js, declaring data elements, such as questions.
- ui.js, manipulates graphical elements.
- logic.js, holds all game logic and token interaction.

In addition, it contains firmware and drivers. This structure allowed us to put all code interacting with AnyBoard to be placed in one file (logic.js), which simplified the comparison between the two implementations.

The code size is kept to a minimum. We have not attempted creating a polished game, but a simple and functional implementation.

6.2.3 JavaScript implementation

We have based the JS-implementation on the AnyBoard-implementation. The code that uses AnyBoard (logic.js) has been replaced with plain JS in addition to using Evothings Bluetooth library.

In short, the functionality that was replaced is:

- Logging to console instead of through the AnyBoard Logger entity
- Replacing token and token-event listeners with naive listeners
- Re-implementing Bluetooth discovering, and connecting to tokens
- Re-implementing functions for sending and reading data from token

AnyBoard provides some nice functionality such as message throttling to prevent overwhelming the token with Bluetooth messages, but such features are secondary. We have not re-implemented these in the JS-implementation, but rather listed these missing features below. The reason for this choice is that we consider it unlikely to be implemented in a real scenario. These features take too much time to implement compared with the value for a single game.

The replacement of AnyBoard logic with plain JavaScript can be seen on Github[2] or in appendix B.

The features that have been ignored, and hence is unavailable in the JS-implemention are:

1) Message throttling: The JS-implementation lacks message throttling, and can therefore in rare cases overwhelm the token with too many messages, leading to loss of data packets and hence integrity. This would be necessary in games including high data transfer to the token, such as with a print token. However in this case, traffic to the token is very limited, and this should therefore not be an

[2] Git commit replacing AnyBoard logic with Javascript: github.com/tomfa/anyboardjs-evaluation/commit/64d335c3b606bcf0a1d2f89f0f0be3fbba27a7e0

issue.

2) Error handling: The AnyBoard platform provides built in error handling in most cases, as well as extensive logging. This makes the AnyBoard platform more robust, and easier to debug. Since the JS-implementation is a concrete game, it does not have the same need for generic methods and validating data and parameters. There is therefore much less error handling in the JS-implementation.

3) Plugin drivers: In AnyBoard, we have strived to separate game logic from token communication. The driver used for communicating with a certain token is therefore separated from the AnyBoard-implementation, and the implementation will work even though the token is replaced with a completely new one, or firmware is updated, given it has a compatible driver. This is not the case for the JS-implementation. This is targeted to the specific token this demo was intended for.

6.3 Results

6.3.1 Number of code lines

Code	AnyBoard	JavaScript	Increase
All lines	186	333	+80%
Different lines	35	183	+423%
Different instructions	18	137	+661%

Table 6.1: Different amount of code logic between the two implementations, in number of lines

The full difference in source code between the two implementations can be seen in appendix B.

The logic of the game, placed in `logic.js`, grew from 186 to 333 lines (80%). This was done by replacing 35 lines with 183. If we count the number of instruction, and ignore comments, empty lines and lines used for decent code readability, we can lower these numbers to 18 lines for 137. When these lines of instruction was considered, each line of AnyBoard code was on average replaced by more than seven (7,61) lines of JavaScript.

The implementation have zero differences in regards to HTML, CSS, manipulation of DOM or other aspects of the game.

6.3.2 Code readability

As the previous section indicates, AnyBoard can significantly lower the necessary code with the regards to game logic in hybrid board games. It does this through a higher expressiveness, as evident when 18 code lines replaces 137.

Below are an example from the game implementations. The code will send command to the token for it to change it's LED color.

Listing 6.1: Turning on LED (with AnyBoard)

```
token.ledOn(token.color);
```

Listing 6.2: Turning on LED (without AnyBoard)

```
var supportedColors = {
  'red': [255, 0, 0],
  'green': [0, 255, 0],
  'blue': [0, 0, 255],
  'white': [255, 255, 255]
};
var ledOnBitCommand = 129;
var sendData = supportedColors[token.color]
sendData.unshift(ledOnBitCommand)
logic._sendOutgoingData(token, sendData);

[...]

_sendOutgoingData: function(device, uint8data, win, fail) {
  evothings.ble.writeCharacteristic(
    device.deviceHandle,
    device.serialChar,
    uint8data,
    function(){ win && win()},
    function(){ fail && fail()}
  );
}
```

The AnyBoard implementation is pretty unambiguous. It's one line, calling a method on an object, sending a single parameter. Number of parameters is a commonly cited evaluation for how readable code is(17, 18). This as opposed to `_sendOutgoingData`, which accepts four arguments.

We also have code on different abstract levels. This is generally acknowledged to be bad coding practice(17). Handling token operations throughout the whole JS implementation involves low level code to some extend. Such as here, where we shift 8-bit integers to form a valid byte-array to be sent via Bluetooth, preceding

sending of data between different hardware components. In a small game such as this one, where we're creating graphical mobile games communicating with a low level protocol, this is inevitable without a library such as AnyBoard.

We see the same difference in any part of the code base related to token communication, including connecting, discovering and subscribing to token events. This can be seen in appendix B.

6.3.3 Number of different libraries

Both implementations exposes two libraries to the developer: jQuery[3] plus one more – AnyBoard in the first implementation, and Evothings Bluetooth[4] library for the JS-implementation.

6.3.4 Number of different sources of knowledge

Impl.	Location	Comment
AnyBoard	Github AnyBoard[5]	AnyBoard library documentation
AnyBoard	jQuery webpage[6]	jQuery library documentation
JavaScript	jQuery webpage[7]	jQuery library documentation
JavaScript	Github Evothings[8]	Evothings Bluetooth library documentation
JavaScript	Example of html identifers[9]	Token Bluetooth-identifiers
JavaScript	Firmware doc[10]	The concret pawn arduino implementation

Table 6.2: Different sources of knowledge for each implementations

The number of resources is lower for the AnyBoard implementation. This is not surprising, as one avoids to deal directly with the specific pawn and its implementation when using AnyBoard. From the development of the RFduino and Bean drivers in creating AnyBoard, we know one can spend *considerable* time and effort for small details. Especially finding the correct UUIDs[11] for Bluetooth characteristics and services. These UUIDs are often not documented anywhere, and in our case was found on a small user-created repository after attempting many other alternatives.

6.3.5 Ability to test game

The implementation and testing was done without the required token. The required token was instead placed with colleagues, who could (with delay, over Skype and without intimate knowledge of the source code) give feedback on what worked and not. The challenge was then how to make sure the game responded as

[3] jQuery, a JavaScript library designed to simplify the client-side scripting of HTML. jquery.com
[4] github.com/evothings/Cordova-ble
[11] UUID - Universally unique identifier. A string to give some object or function an unique ID

intended upon actually being used with the token.

We did encounter a few bugs in our implementation that was first discovered upon testing with the physical token. Three rounds of tests were done before we discovered a very helpful functionality of Evothings. Evothings allowed for us to manually insert javascript code into the Cordova application during runtime. That made us able to create virtual tokens and simulate a connect or movement to a board. The code needed for the different implementations to accomplish this was more or less equal, as shown in the examples below. The test done after this was discovered, was fully functional.

Listing 6.3: Code run in evothings workbench to simulate a movement of physical token on a constraint (AnyBoard-implementation)

```
var token = d.players[0].properties.token
var currentTile = 2;
token.trigger(
  "MOVE_TO",
  {
    "meta-eventType": "token-constraint",
    "constraint": currentTile
  });
```

Listing 6.4: Code run in Evothings workbench to simulate a movement of physical token on a constraint (JS-implementation)

```
var token = d.players[0].token;
var currentTile = 2;
logic.trigger(
  "MOVE_TO",
  {
    "token": token,
    "constraint": currentTile
  });
```

Chapter 7

Discussion

In this chapter, we'll discuss choices done at the different stages in the thesis. Looking back we can find both misplaced and well placed effort, bad as well as good priorities. We also have a range of ideas for what entities and features AnyBoard could have benefited from as well as thought for the further development of the platform.

We'll first look at the process, followed by the implementation and lastly reflections on our evaluation.

7.1 Development process

7.1.1 Time spent researching game engines

In the start of the development process, quite some time was spent both evaluating different game engines, and (later) existing games made with the Phaser engine. Phaser provides a plugin API, and we suspected that we could create a plugin that exposed a Bluetooth API to the Phaser engine.

We eventually decided to go away from Phaser, and create a standalone library. This in itself is a decision we're very happy with, as the library is accessible without having to learn Phaser or any other library in particular. But the decision should have come sooner. Too much time was spent researching and looking at Phaser related work that in the end gave no value to the thesis.

7.1.2 Selected AnyBoard entities

The set of implemented entities, could have had better synergy. For example, Deck, Card and Dice was implemented, alongside Tokens. But Tokens interact in

many more ways with the concept of Board and Tile. However, these entities were not prioritized.

The way entities were chosen for implementation was based on priority in the design chapter, but also a gut feeling on how much time the implementations would take vs value and time remaining. Rather than basing this on gut feeling, the development phase could have benefited from estimating the amount of time required for each entity at the start, and then developed iteratively, achieving set minimally viable improvements on each iteration. This way, we would probably had end up with better synergy between the entities, e.g. by creating Board, Tile and Token. On the other hand, this extra planning could have created more overhead and been demotivating. It might not have had much positive effect in a project with only one developer, and such a short development phase.

7.2 Implementation

7.2.1 Independent from other libraries

We feel confident in the decision of implementing AnyBoard decoupled from other libraries or frameworks. During the research phase, we found a huge amount of many relevant libraries and options for game development in JavaScript. The choice to remain decoupled from any one library in particular, removes the risk of AnyBoard being outdated and dependant on something that is no longer maintained.

7.2.2 A good development environment

We have benefited by the way the development environment was set up. First, the tests provides great value in different ways. First, AnyBoard has been developed by a single developer that won't be working with AnyBoard after the delivery of this thesis. But the library will continue to be developed upon at the institute. In order for new developers to feel confident that they don't break the functionality, tests are important. Tests also give value by providing examples for how different entities can be used. With the usage of Grunt Task runner, tests are also easily run with a simple command.

Unfortunately, the tests were neglected in the later stages of the development process, during the implementation of token functionality. This came back to us during the development of examples, where several bugs were discovered as demos didn't work as expected. Even after everything worked OK, some new bugs were introduced during refactoring[1] of code. We strongly encourage these

[1] *Code refactoring is the process of restructuring existing computer code – changing the factoring – without changing its external behavior.* – Wikipedia

tests to be written before doing much more development.

Secondly, we're very satisfied with the documentation. A new student took over the development of AnyBoard only days before the end of this thesis. The first feedback from the student was that it was so well documented, that he felt confident this would work out fine even though he had no previous experience with JavaScript, he thought this would work out well. This is just what we hoped to hear when we set documentation as a high priority. We're also very happy about the automatically generated docs based on inline JSDoc. Not only does it keep the source of documentation to one place, but it provides both inline explanation of methods, autocomplete in most IDEs as well as auto-generated online documentation.

7.2.3 Missing a DummyToken

Requirement FT-6 (see table 4.1) asks for a simplified way of testing without having acquired the physical token. Our idea for this was by creating a Dummy Token, but we failed to fulfill this requirement (see section 5.3).

The result is that AnyBoard doesn't assist in developing games for tokens you don't possess. In the evaluation, we found no benefit from debugging with or without AnyBoard. We figured out a way that worked well, but that was made possible by having intimate knowledge of the code base and tools. Even then, the solution didn't immediately come to mind.

We regret not putting a higher priority on creating a solution for Dummy Tokens, and encourage this to be done in near future. Luckily, the decoupling of code should make this a relatively straight forward task. The only implementation required would be replacing the driver `discovery.rfduino.Bluetooth.js` with a "dummy driver", that pretends to find tokens, send and respond to data. Since replacing the discovery driver is done by the game developer by replacing the import of one driver with another, changing from testing to production environment will not require any knowledge nor code change other than this one line.

7.2.4 Make Logger logs easier to read

The Logger has several minor areas of improvement. First of all, it's not intuitive to see debug log with numbers, for example:

```
LOG: AnyBoard (0): Connected to Token: AnyBoard Printer
LOG: AnyBoard (10): Somethingelsehappened
```

Instead of numbers, "DEBUG", "WARN" etc. would be more readable.

Secondly, we have NORMAL as a level of severity. This is an uncommon level that might throw of some developers. Instead, one could use Apache Commons Logging labels: *[FATAL ERROR WARN INFO DEBUG TRACE]*, or Python logging terms[2]: *[critical error warn info debug]*.

Lastly, the Logger functionality to print to hyper (Evothings logging tool) automatically was very useful during debugging. An improvement to this, would making threshold level of the Logger to also apply to hyper logging. When it indiscriminately logs everything, the errors can drown in a the amount of logging.

7.2.5 Ensure stability in Bluetooth communication

During the late stages of implementation, we discovered that the Bluetooth drivers for AnyBoard Token, AnyBoard Bean Token and AnyBoard Printer were causing packet losses when several commands were sent at the same time. The reason for this was a naive sending of data to the Bluetooth devices. There was no throttling, which caused an overload on the tokens.

Throttling was implemented to increase stability, with great success. However, there are some limitations to the implementation, which we think should be addressed.

If a token does not receive a command, as illustrated in figure 5.6, or does not confirm the received command, or this confirmation is lost, AnyBoard will not be able to determine whether or not the executed command was received or completed by a token. In the current implementation, AnyBoard will timeout after 2 seconds, and simply go on to the next command, ignoring the possibly lost one.

This can lead to inconsistent outcomes, and unreliability issues and annoyances.

A solution could be to give each packet an identifier so that both client and token can confirm received commands and respond in a reliable fashion, so that every packet is delivered in order.

It's worth noting that the source code that holds the Bluetooth communication handling, all resides in the individual drivers for each chip (pr now one for RFduino and one for the bean). As these draw on very much of the same functionality, we believe it would be beneficial to subtract the common logic in these two drivers into a common base class that they instead inherit from. This could also make it serve as a template for developers who wish to create their own drivers.

[2] docs.python.org/2/library/logging.html

7.2.6 Implement remaining AnyBoard entities

Several game entities that were planned were not implemented. We prioritized creating fully functional, tested and documented entities rather than completing all requirements within the time frame of the thesis. We believe this choice to be the correct one, with regards to the library being developed by new personnel after the delivery of this thesis. Easily understandable and well documented code makes it easier for these people to take over.

A short description of the entities that hasn't been implemented yet, but were a part of the original requirements are shown in the list below.

- **Board and Tiles** - Originally planned to be implemented (requirement FE-7 in 4.3), but not prioritized. In the evaluation, instead of using Board and Tiles, we kept track of location by adding a simple integer property to the tokens, telling where on the board it is located. AnyBoard does not simplify any logic on a board or movement between them other than a change event on the token and this token property. We believe AnyBoard.Board and AnyBoard.Tile classes with methods for (1) calculating distance between tiles, (2) whether or not movement between tiles are legal and (3) special properties belonging to tiles would simplify the game development.
- **Turn** - was a part of the original requirement (FE-6 in 4.3). It was postponed until we could test an actual case where we needed it. We didn't arrive at a point where this would be useful, which is not surprising seeing that a Turn is more abstract that the physical entities we've worked with. We're not too sorry about this. At a later stage, we believe this concept could show up as being useful having entities for. At that point it would make sense implementing this entity.
- **Timer** - requirement FE-5 (see 4.3) was not implemented due to it's low priority. This entity would be useful during implementation of quiz games, Don't Panic and other games that uses time as a game concept.

7.2.7 Do minor implementation changes

"MOVE" as a token-constraint event

The driver `rfduino.evothings.Bluetooth.js` triggers two events when a token is moved, `MOVE_TO` and `MOVE_FROM`. We believe we would have benefited with a single `MOVE` event that tells both where the token was and went. This could be useful in cases where some action is to be performed based on both your current and previous location.

Get ”other” tokens

Quite it would’ve been useful to get all other tokens except the one you’re holding. For example by implementing `token.getOtherTokens()`. Same with getting all *connected* tokens (not including those that are simply discovered). Today’s implementation is a bit verbose, and makes the code less readable:

```
connectedTokens = []
for (var key in AnyBoard.TokenManager.tokens) {
  var aToken = AnyBoard.TokenManager.tokens[key];
  if (key !== token.address && aToken.isConnected()) {
    connectedTokens.push(aToken);
  }
}
```

Remove complexity in firmware

We’ve been too quick in implementing some of the token functionality. In particular, lets take a short look at hasLed functionality.

If the driver hasn’t implemented `hasLed`, AnyBoard will respond with an error instead of doing the reasonable deduction that drivers that don’t implement hasLed (or ledOn), neither has the functionality.

Perhaps more importantly is the allowing firmware not to implement such methods. In case of Bean firmware, the Bean Token implements ledOn, while the Bean Printer does not. When executing hasLed, the firmware on Bean Printer has to handle the command "hasLed" and respond to this for the AnyBoard.BaseToken method `hasLed` to correctly return `false`. Instead of this being necessary, we should allow it to respond 0 (unknown command) and from the AnyBoard library correctly respond `false`.

7.3 Evaluation

7.3.1 Should have tested with target audience

Instead of testing the library ourselves, we should have preferably tested the library against potential users. The developer of AnyBoard is not a typical user of AnyBoard, as he naturally sits with considerable knowledge of the library. In short, we lack feedback from our target audience.

7.3.2 Lacks evaluation of AnyBoard entities

The game implemented makes use of interactions with pawn that can be considered to be common in board games, namely token-constraint and token-token interactions, as well as sending commands to the token. But the implementation lack use of logical entities such as dice, decks or cards. This part of the AnyBoard is therefore basically unevaluated.

7.4 Potentially interesting new features

Here we go through some features which was not a part of the design for AnyBoard, but was imagined or felt need for during the implementation. These can also serve as ideas for future work.

7.4.1 Facilitate persistent storage

AnyBoard could facilitate for developers to get easy access to persistent storage. A small library could provide access for developers to a simple database or other persistent storage, for storing player names, previously connected devices etc. These capabilities could, among other things, reduce setup time for players.

This functionality might be considered unnecessary, though, as there is access to persistent storage in HTML5. Phaser, which is the game engine we've looked to for ensuring compatibility has opted to not provide such a functionality[3] due to the simplicity of plain HTML5 LocalStorage. HTML5 LocalStorage provides a simple key-value storage, which is restricted in that values can only be strings (conversion can in practice use this for integer and JSON), with a recommended limit of 5 MB[4]. Should this capability not be sufficient, AnyBoard could make use of the Cordova Storage[5] plugin, which is based on SQLite 3, that could provide file storage and more.

7.4.2 HTTP-based tokens

Currently, AnyBoard is tested with a driver that connects to tokens which communicate over Bluetooth. HTTP-based interfaces are also popular, and we believe that providing examples with HTTP-based drivers could open up for new exiting features.

---

[3] Phaser game engine recommends using HTML5 Localstorage for persistent storage according to html5gamedevs.com/topic/3765-preferred-way-of-saving-data/ - Phaser forums

[4] http://www.w3.org/TR/webstorage/

[5] Cordova Storage Documentation, on use of Storage and HTML5 LocalStorage - Cordova.apache.org/docs/en/3.0.0/Cordova_storage_storage.md.html

One example is a driver for Phillips Hue[6], which is a light bulb with an HTTP-interface which is able to change colors and dim upon HTTP-requests. We imagine Hue could be incorporated with a game, to provide both relevant mood to a situation (imagine more red light signaling a threat), or as an informational token of other sorts.

7.4.3 Hidden private space

Currently, the intended implementation of hidden private space in AnyBoard-based games are by printing cards that a player keeps to himself. This requires players playing games with private spaces to have a AnyBoard compatible printer, as well as refilling paper and ink.

Another way of implementing hidden private spaces would be by using several mobile devices, where one acts as the main game hub, and the other acts as a private space to each player. This would be more time consuming to set up, and shift the focus further towards a digital focus during play, which we consider a drawback. On the other side, it lowers the barrier for acquiring and playing games that uses this characteristic.

7.4.4 Private actions and communication

Using the cell phones of each player as a device to interact with could also allow us the capabilities of performing actions or communicating in secret.

In traditional board games, all actions and all communication between players is usually publicly visible. This is a natural restriction by the fact that the players sit next to each other, being able to see and hear what other players do and say. In digital games on the other hand, people are some times able to do actions in secret, or team up to plot against other players.

The fact that traditional board games are so open, and nothing can be performed without other players taking notice, might be one of the main components that make these more social than digital games. Allowing for these features can in other words lower the benefit hybrid games have over digital games. It suffers from the same potential problem as the suggestion for hidden private space, in that it diverts the focus from the physical aspect to the digital.

7.4.5 QR-codes and barcodes

A Bean Printer driver was developed during this thesis, and can be found at github.com/tomfa/anyboardjs. This demonstrates the capabilities of printing text,

[6] meethue.com - home for Phillips Hue, a digital light bulb with an HTTP-interface

using an AdaFruit Mini Thermal Printer[7].

A relatively simple change in the Bean driver and firmware would allow us to print QR- and barcodes as this is a feature of the AdaFruit printer[8]. Printing QR and barcodes allows for cellphone based interactivity, such as providing a phone number, opening a SMS with predefined text or URLs. This is provided by Android and iOS by default and does not require the development of a separate standalone mobile application.

The functionality of reading QR and bar codes from the game hub could be done via Cordova plugins[9] or via HTML5 GetUserMedia[10] which would work synergistically with this feature.

7.4.6 AnyBoardUI library

AnyBoard does currently not provide any means to simplify creating graphical components. This is in part an intentional choice, as we during the development realized how much it would narrow the compatibility with other libraries and JavaScript tools available.

However, creating GUI is a sizeable job that can be cumbersome for inexperienced developers. The existing example games attached with this report are simple, HTML-based. By the very least, we think more examples should be provided in order to lower the barrier for new developers.

Seeing that board games have a set of pretty standard interaction and events, we propose that a UI-module of AnyBoard that implements standard graphical elements is developed and delivered next to the AnyBoard library. We advice against integrating the UI library into the standard library. That way, the standard library could be used standalone and independent of other tools, while the UI-library is directed at developers that are inexperienced with UI-programming. This should implement the following elements:

- Displaying boards created with Tiled
- Menu with buttons and labels.
- Buttons and text holders for starting game, reading rules and connecting to tokens
- Displaying pawns and their position

[7] AdaFruit Mini Thermal Printer: https://learn.adafruit.com/mini-thermal-receipt-printer

[8] AdaFruit printer QR capabilities: https://learn.adafruit.com/downloads/pdf/mini-thermal-receipt-printer.pdf

[9] QR code capabilities in Cordova-based applications: phonegap.com/blog/build/barcodescanner-plugin

[10] GetUserMedia is not supported by all mobile browsers yet – caniuse.com/#search=getusermedia

- Displaying cards, and providing action buttons
- Displaying players and their resources

7.4.7 Chrome Cast

Chrome Cast[11] is a small device similar to Apple TV, allowing for iPhone, Android and Chrome (browser) applications to display their working PC/mobile/tablet screen on a TV screen.

Allowing for games to be cast to a larger screen introduces another digital device to the hybrid board game, and could act as a diversion from the tangible parts. However, using chrome casting is an optional feature, and could be skipped by users if it proves distracting. Also, we believe that in games where physical interaction with the phone is kept to minimal and the game hub screen is purely informational, casting to a bigger screen will keep focus on the physical parts, and avoid potentially having to pass around a mobile phone.

Whether or not any new feature in AnyBoard could simplify adding support for Chrome Cast, we have not investigated in. But even an example showcasing how it could be done, might attract potential game developers.

[11] www.chromecast.com, developers.google.com/cast/docs/developers

Chapter 8

Conclusion

The objective of this thesis was to simplify the development of hybrid board games.

In our research questions, we asked how we can facilitate developers to create hybrid board games that are easy to set up (RQ-3). Our approach was to simplify the discovery, connection and communication with the tokens. This was done through creating highly abstract methods for this functionality. This has enabled us in all examples and the quiz game to provide a very simple setup for players: Given that the tokens are attached to a power source and the mobile application is open, all they need to do is click one button to search for tokens, which appear as button on the screen. They then click on this button to connect to the token.

We also ask how we can lower the investment required for developers to implement hybrid board games (RQ-2). A hybrid board game will necessarily use hardware components, so in order to keep it low-cost, we believe using mobiles and tablets where feasible is the best move, as the players are likely to already own these devices. In addition, Arduino and similar digital devices are becoming ubiquitous as their price drops, and these can replace previously used, expensive digital tabletop devices. On the software side, a free to use, well documented and purposed tool, will lower the time investment. This has not previously been available, but is what we've tried to address with AnyBoard.

Our main research question is how we can lower the barrier for developers to start creating hybrid board games (RQ-1). We believe this is can be accomplished through better tools for simplifying the interaction with Arduino and similar devices. AnyBoard is to our knowledge the first tool made specifically for this purpose. We think that the thoroughness of the documentation and the examples

provided will provide a gentle learning curve and quick results for developers who decide to use AnyBoard.

8.1 Future Work

We've addressed many suggestions for potential future work of AnyBoard in the discussion chapter. But the perhaps most interesting one is the one we addressed in the article "Making interactive board games to learn: Reflections on AnyBoard"(4). By drastically shortening the distance from computer code to visual responses, AnyBoard could be a relevant tool for learning settings. One way of doing this could be to implement a Blockly[1] or Scratch module on top of the AnyBoard code. These are visual "drag-and-drop" programming language, which makes programming a suitable learning activity for non-programming students or pupils in high an middle school. Alternatively, one use AnyBoard in collaboration with Quintius (see chapter 3.1)

Another potential use of AnyBoard is location based games. AnyBoard provides a simplified connection and communication over Bluetooth, while in a JavaScript (or web) environment on mobile devices. Relatively small changes could also allow us to print and read bar- and QR codes. This gives us a great starting point for creating games where the players move in an outdoor environment from post to post, where each post involve some sort of interaction. For example, the interaction could involve getting in proximity to, shake, move or change the temperature of a token. QR codes could link to web pages, or provide phone numbers or messages. In other words, AnyBoard could provide the basis for an new type of interactive "treasure hunt" involving tokens with different capabilities, smart phones and internet connectivity.

[1] developers.google.com/blockly

Bibliography

[1] I. Di Loreto, S. Mora, and M. Divitini, "Don't panic: Enhancing soft skills for civil protection workers," in *Serious Games Development and Applications*, pp. 1–12, Springer, 2012.

[2] A. Al Mahmud, O. Mubin, S. Shahid, and J.-B. Martens, "Designing and evaluating the tabletop game experience for senior citizens," in *Proceedings of the 5th Nordic conference on Human-computer interaction: building bridges*, pp. 403–406, ACM, 2008.

[3] S. Bakker, D. Vorstenbosch, E. van den Hoven, G. Hollemans, and T. Bergman, "Weathergods: tangible interaction in a digital tabletop game," in *Proceedings of the 1st international conference on Tangible and embedded interaction*, ACM, 2007.

[4] S. Mora, T. Fagerbekk, I. Di Loreto, and M. Divitini, "Making interactive board games to learn: Reflections on anyboard," in *Proceedings of the Workshop of Making as a Pathway to Foster Joyful Engagement and Creativity in Learning (Make2Learn 2015), CEUR-WS, Vol. 1450*, pp. 29–36, 2015.

[5] C. Magerkurth, M. Memisoglu, T. Engelke, and N. Streitz, "Towards the next generation of tabletop gaming experiences," in *Proceedings of Graphics interface 2004*, pp. 73–80, Canadian Human-Computer Communications Society, 2004.

[6] R. L. Mandryk and D. S. Maranan, "False prophets: exploring hybrid board/video games," in *CHI'02 extended abstracts on Human factors in computing systems*, pp. 640–641, ACM, 2002.

[7] J. Marco, E. Cerezo, and S. Baldassarri, "Toyvision: a toolkit for prototyping tabletop tangible games," in *Proceedings of the 4th ACM SIGCHI symposium on Engineering interactive computing systems*, pp. 71–80, ACM, 2012.

[8] M. Kaltenbrunner and R. Bencina, "reactivision: a computer-vision framework for table-based tangible interaction," in *Proceedings of the 1st international conference on Tangible and embedded interaction*, pp. 69–74, ACM, 2007.

[9] C. Magerkurth, "Hybrid gaming environments: keeping the human in the loop within the internet of things," *Universal Access in the Information Society*, vol. 11, no. 3, pp. 273–283, 2012.

[10] C. Reynolds, "Ten of the best cross-platform mobile development tools for enterprises." http://appindex.com/blog/ten-best-cross-platform-development-mobile-enterprises/, 2014. [Online, accessed 2015-04-21].

[11] J. Cowart, "Pros and cons of the top 5 cross-platform tools." http://www.developereconomics.com/pros-cons-top-5-cross-platform-tools/, 2015. [Online, accessed 2015-04-21].

[12] C. Baldwin, "Should i develop for ios or android? 10 cross-platform tools that make both possible." http://thinkapps.com/blog/development/develop-for-ios-v-android-cross-platform-tools/, 2014. [Online, accessed 2015-04-21].

[13] research2guidance, "Cross-platform tool benchmarking 2014." http://research2guidance.com/cross-platform-tool-benchmarking-2014, 2014. [Online, accessed 2015-04-21].

[14] P. Rettig, *Professional HTML5 mobile game development*. John Wiley & Sons, 2012.

[15] ashes999 (Github user), "Phaser vs craftyjs." http://ashes999.github.io/javascript-blog/2014/10/06/phaser-vs-craftyjs/, 2014. [Online, aksessert 2015-06-05].

[16] A. Holden, "The search for a javascript game engine." http://adam-holden.com/blog/category/game-development/crafty/, 2014. [Online, aksessert 2015-06-05].

[17] R. C. Martin, *Clean code: a handbook of agile software craftsmanship*. Pearson Education, 2009.

[18] K. Henney, “Seven ineffective coding habits.” https://vimeo.com/97329157, 2014. [Online, accessed 2015-09-21].

Appendix A

Details on development environment

A.1 Setup

For updated and more thorough setup details, we refer to github.com/tomfa/anyboardjs

A.1.1 For developing AnyBoard further

The following three steps is required to set up the development environment for AnyBoard. In addition, we strongly recommend installing Evothings from evothings.com.

1. Download the latest version of the repo from github.
2. Install NodeJS from nodejs.org
3. Install dependencies with `npm install`

You should now be ready to develop. Open the repository with your favorite IDE (mine is WebStorm).

A.1.2 For developing games using AnyBoard

Locate your Cordova application. If you don't have this yet, you can download Evothings and copy one of its examples from evothings.com.

The following code must be inserted in the `head` tag of the index.html file.

1) Will you develop for Android and iOS devices? If yes, include the following: `<script src="Cordova.js"></script>`

2) Will you use the Bean Printer, Bean Token and AnyBoard Token as a part of this game? If yes, include the following:

```
<script src="dist/AnyBoard.js"></script>
<script src="libs/evothings/evothings.js"></script>
<script src="libs/evothings/easyble/easyble.js"></script>
<script src="drivers/discovery.evothings.Bluetooth.js"></script>
<script src="drivers/rfduino.evothings.Bluetooth.js"></script>
<script src="drivers/bean.evothings.Bluetooth.js"></script>
```

3) Have you updated the token with the newest firmware? If no, upload the firmware located in the firmware folder at github.com/tomfa/anyboardjs.

That's it. Look at the document (I) to get started with creating your game.

A.2 Using Evothings in development

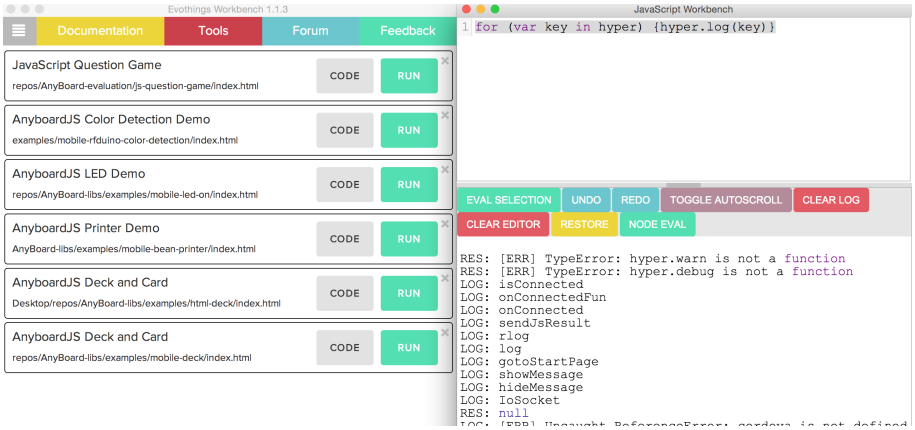

Figure A.1: Screen shot of Evothings workbench. Log output can be seen in the lower right, while code insertion can be done in the upper right. An overview over applications are seen on the left

We recommend using Evothings during development. This is due to it's quick debugging and setup. A mobile phone connects to the computer on which development is done, and provides several nice features:

- **Instant logging from phone to computer during testing**: The application running on the phone will log instantly to the computer.
- **Allows insertion of code**: You're allowed to insert code on the fly into the application

- **Automatic deployment of code to connected devices**: The computer watches on source files of the currently running application. If you change any file on the computer while the app is open on the phone, the phone is provided with the newest version.

This is done without requiring any code changes on the Cordova application. No libraries or commands are required to be run on the source files. The only requirement is that the Evothings mobile application is installed on the mobile phone used for testing.

A.3 Deploying apps

The apps and demos we've developed, have been Cordova based. Since Evothings doesn't require any particular code change, the deployment of apps and making them available from app stores or .apks is not really related to Evothings. However, they have published a nice guide at evothings.com/doc/build/build-overview.html.

In order to publish your applications as a standalone app for others to download, or want to publish an app on App Store or Play Store, this would be a good place to start.

A.4 Details on modules

All tools used here are based upon a NodeJS platform. Developing AnyBoard in this environment is therefore supported by those operating systems that support NodeJS (Linux, Mac OSX, Windows, SunOS).

A.4.1 Platform and dependency handler: NodeJS

NodeJS[1] is a platform built on Chrome's JavaScript runtime for easily building fast, scalable network applications. It is used in the project as a dependency handler and platform for our other tools, i.e. it gives us the opportunity to use the other .

Being the by far most dominant JavaScript-based platform and something that we have previous experience with, we have not considered other options.

Usage

In figure A.2 *package.json* is a node package file for AnyBoard. It gives meta data about AnyBoard, such as location of online repository and version number, in addition to specifying all dependencies that should be installed in order to set up

[1] nodejs.org

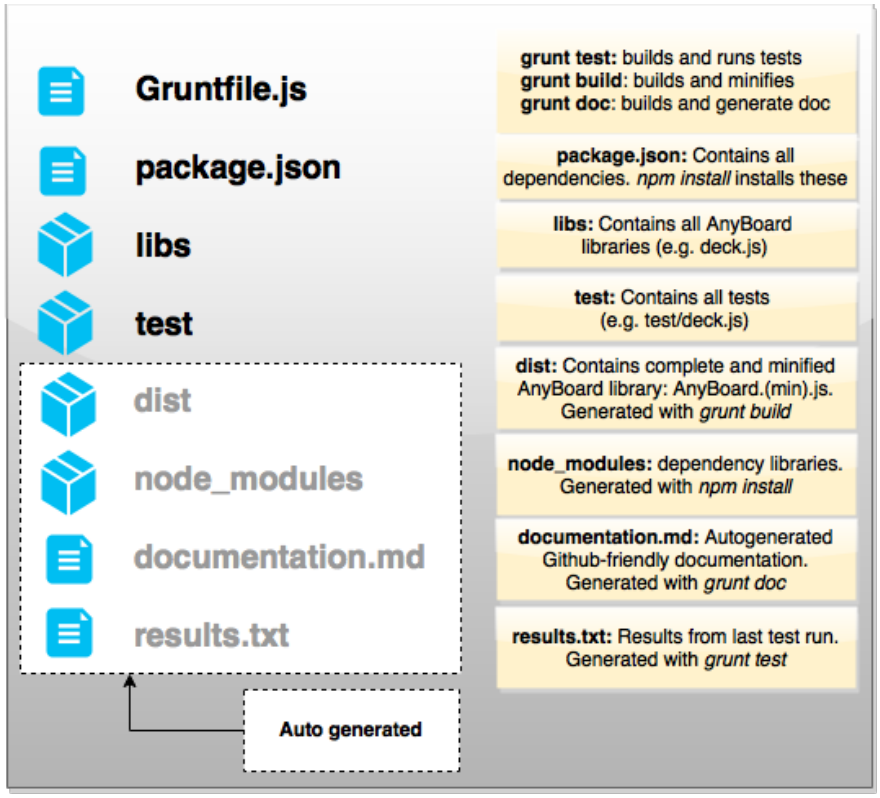

Figure A.2: Overview over the AnyBoard development files and folder structure.

the development environment. Installing all dependencies is as easy as installing NodeJS, followed by navigation to the folder in a terminal and typing *npm install*.

Upon installing dependencies, those are automatically installed under the *node_modules* folder, shown in the auto generated section of figure A.2.

Replacing NodeJS

Not using or replacing NodeJS will require changing most of the development environment, including test framework and task runner, but has no effect on the functionality of the AnyBoard library.

A.4.2 Test framework: MochaJS

Mocha [2] is a JavaScript-based test framework. It is used in the project in providing an easy to understand syntax in writiting tests, and reports the results of

[2] mochajs.org

tests in a readable manner (see figure A.3).

Motivation

Testing frameworks provides a clear syntax for testing, as well as methods and constructs for asserting whether or not function work as intended. Using a test framework and *writing tests* (requirement NFR-3) for our library provides confidence to developers that they don't break the code unknowingly when changing the code. It also *provides examples* (requirement NFR-2) of how to use the library.

We have chosen MochaJS over alternatives like UnitJS*unitjs.com* and VowsJS due to our familiarity with it, and great reporting format.

Usage

Our tests are located within the *test* library shown in figure A.2. They are named with the same name as the library in the *libs* folder that they test. All .js files placed in this folder will automatically be run by our MochaJS installation upon running *grunt test* from the console when located in this folder.

Replacing MochaJS

Changing the choice of test framework will require rewriting at least parts of the tests, but has no effect on the functionality or build of the AnyBoard library, nor documentation.

```
[tomas:~/Desktop/repos/AnyBoard-libs]$ grunt test                    (master*)
Running "mochaTest:test" (mochaTest) task


  AnyBoard.Dices
    when initated without parameters
      ✓ is initiated "new AnyBoard.Dices()"
      ✓ has 1 dice
      ✓ that dice has 6 eyes
    when initated with one parameter
      ✓ is initiated with e.g. "new AnyBoard.Dices(10)"
      ✓ have 1 dice
      ✓ that dice have equal amount of eyes as the parameter (10)
    when initated with two parameters
      ✓ is initiated with e.g. "new AnyBoard.Dices(12, 6)"
      ✓ have the second parameter amount of dices (6)
      ✓ each dice has amount of eyes equal to the first parameter
    when rolled with .roll()
      ✓ returns a number [amount of dices] <= x <= [amount of dices] times [eyes]
    when rolled with .rollEach()
      ✓ returns an array of results equal in length to number of dices
      ✓ returns an array where each element is between 1 and number of eyes
```

Figure A.3: Human readable test results from unit tests made with Mocha test framework.

A.4.3 Task runner: Grunt

Grunt[3] is a JavaScript-based Task Runner. It is used in the project to simplify building the AnyBoard library, generating documentation and running tests.

Grunt was chosen over alternatives such as GulpJS[4] and BroccoliJS[5] due to our prior experience with it, as well as larger community.

Usage

Gruntfile.js as shown on top in figure A.2. The file specifies 7 different tasks, that is run by *grunt "taskName"*, where the different "taskName" are specified in bold below. The first four tasks are simple tasks that have names dictated by a corresponding grunt module, whereas the last three are combinded tasks of which we have selected the name.

- **concat** - concatenates all javascript files located in *lib* folder into one file, that is then put into *dist/AnyBoard.js*.
- **uglify** - Takes *dist/AnyBoard.js* and compresses/minifies it, and stores it in *dist/AnyBoard.min.js*
- **mochaTest** - runs all javascript files located in the *test* folder, and report their successes to the console. The result is also (over-) written to *results.txt*
- **jsdoc2md** - reads through *dist/AnyBoard.js* and interprets the commenting. A markdown-file is generated and (over-) written to *documentation.md*
- **build** - runs *concat* followed by *uglify*
- **test** - runs *build* followed by *mochaTest*
- **doc** - runs *build* followed by *jsdoc2md*

Replacing Grunt

Replacing Grunt with Gulp, Broccoli or another task runner will involve replacing grunt modules specified in *package.json* with modules designed for the new task runner, as well as replacing the *Gruntfile.js* file with a new configuration file. It has no effect on the functionality of the AnyBoard library.

A.4.4 Documentation generation: JSDoc and grunt-jsdoc-to-markdown

JSDoc[6] is a standardized way of documenting JavaScript code. It allows IDEs[7] to give assisting information to a developer about the code he's using, which classes and methods are available, what they return, take as parameters etc.

[3] gruntjs.com
[4] gulpjs.com
[5] broccolijs.com
[6] usejsdoc.org - a JavaScript documentation syntax and parser
[7] IDE is shorthand for Integrated Development Environment and is simply put a rich featured editor.

In addition, software plugins for our development platform (NodeJS + Grunt) exist that allows us to generate automatic documentation based on JSDoc syntax.

```
/** Represents a set of game dices that can be rolled to retrieve a random result.
 * @constructor
 * @param {number} [eyes=6] *(default: 6)* number of max eyes on a roll with this dice
 * @param {number} [numOfDice=1] *(default: 1)* number of dices
 * @example
 * // will create 1 dice, with 6 eyes
 * var dice = new AnyBoard.Dices();
 *
 * // will create 2 dice, with 6 eyes
 * var dice = new AnyBoard.Dices(2, 6);
 */
AnyBoard.Dices = function (eyes, numOfDice) {
    this.eyes = eyes || 6;
    this.numOfDice = numOfDice || 1;
};
```

Figure A.4: Example of jsdoc commenting in source code.

new AnyBoard.Dices([eyes], [numOfDice])

Represents a set of game dices that can be rolled to retrieve a random result.

Param	Type	Default	Description
[eyes]	number	6	number of max eyes on a roll with this dice
[numOfDice]	number	1	number of dices

Example

```
// will create 1 dice, with 6 eyes
var dice = new AnyBoard.Dices();

// will create 2 dice, with 6 eyes
var dice = new AnyBoard.Dices(2, 6);
```

Figure A.5: Example of generated documentation by grunt-jsdoc-to-markdown.

Usage

All source files are commented with JSDoc-syntax, see figure A.4. Most IDEs and some regular editors will automatically pick up on these comments and assist developers with writing code.

Our grunt command *grunt doc*, autogenerates the *documentation.md* documentation file, see A.5

Replacing JSDoc

Minor changes would be required to all source file comments in order to change to a different syntax for commenting. In addition, one would naturally have to replace grunt-jsdoc-to-markdown.

Replacing grunt-jsdoc-to-markdown

Replacing grunt-jsdoc-to-markdown will require finding a suitable grunt-compatible jsdoc compatible documentation generator, and replacing *grunt-jsdoc-to-markdown* in the *package.json* file.

Appendix B

AnyBoard Quiz Game

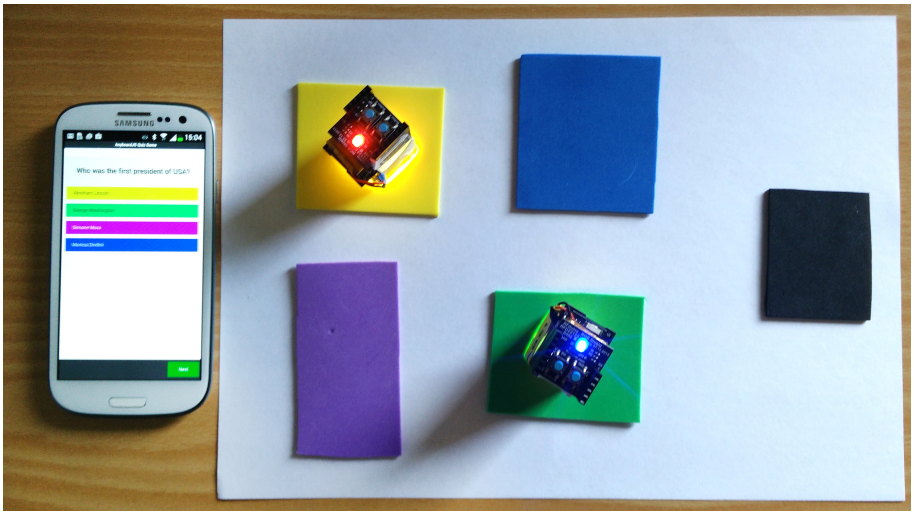

Figure B.1: The quiz game implemented for evaluation. Each player places his or her token on the colored tile corresponding to the alternative.

AnyBoard Quiz Game was a game implemented for the purpose of evaluation (see chapter 6). It's complete code base can be seen and downloaded from github.com/tomfa/anyboardjs-evaluation.

It can be seen in play in figure B.1.

B.1 Graphical user interface

The graphical user interface of AnyBoard Quiz Game is shown in figures B.2, B.3 and B.4.

It was designed for android and iOS mobile devices in mind, but should also be compatible with other Android and iOS devices, such as iPad.

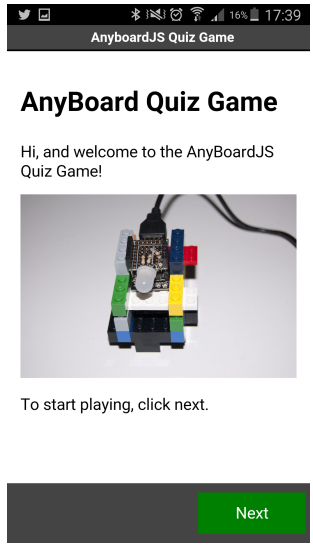

Figure B.2: Starting screen for the evaluation game.

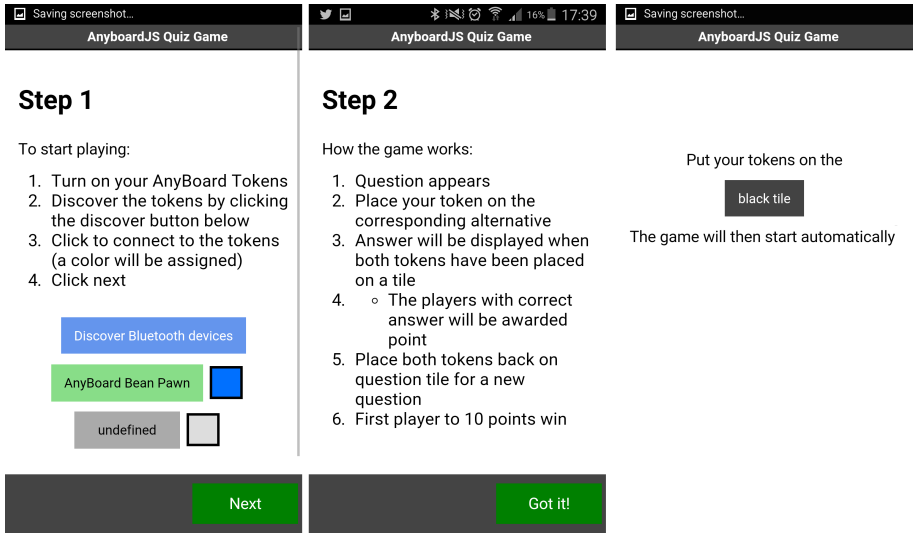

Figure B.3: Screen 2 (left), 3 (middle) and 4 (right) of AnyBoard Quiz, assisting the player in connecting to tokens and place the tokens on the starting point.

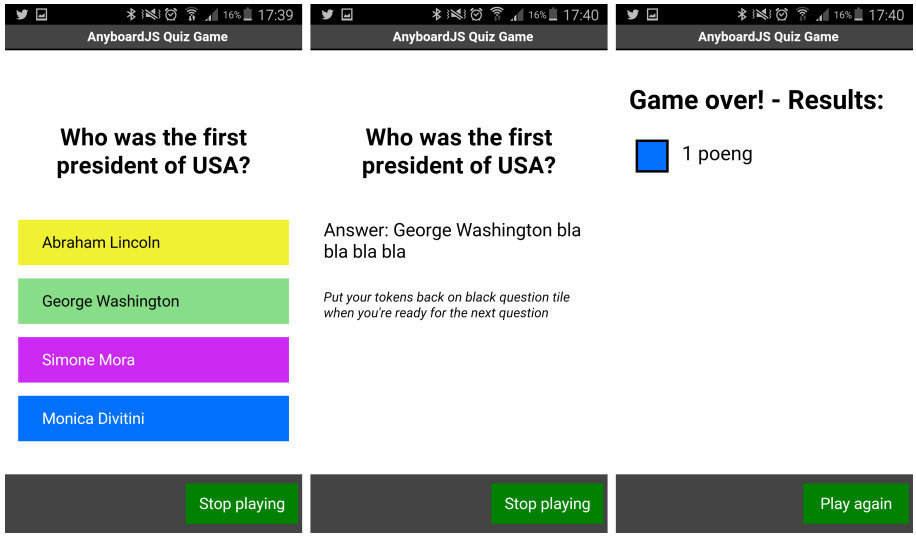

Figure B.4: Screen 5 (left), 6 (middle) and 7 (right) of AnyBoard Quiz. The player answers the questions by moving his or her token to the tile that corresponds to the color of the alternative. An answer will show when all tokens are placed. Once all questions has been answered, a summary will be displayed.

B.2 File structure

The file structure of AnyBoard Quiz game is shown in figure B.6. The game itself is divided in *index.html* (markup), *style.css* (styling), *js/data.js* (data), *js/logic.js* (logic) and *js/ui.js* (manipulation of the graphical elements).

`dist` - folder contains the AnyBoard library, `drivers` the AnyBoard drivers, `firmware` the firmware for the AnyBoard token. Libs contains dependencies for the AnyBoard drivers and implementation. jQuery is used for the game development.

The code amount in each implementation can be seen in figure B.5.

```
[tomas:...p/repos/AnyBoard-evaluation]$ wc -l anyboard-question-game/js/*
     112 anyboard-question-game/js/data.js
     186 anyboard-question-game/js/logic.js
      75 anyboard-question-game/js/ui.js
     373 total
[tomas:...p/repos/AnyBoard-evaluation]$ wc -l js-question-game/js/*      (
     112 js-question-game/js/data.js
     333 js-question-game/js/logic.js
      75 js-question-game/js/ui.js
     520 total
```

Figure B.5: Code lines in each implementation.

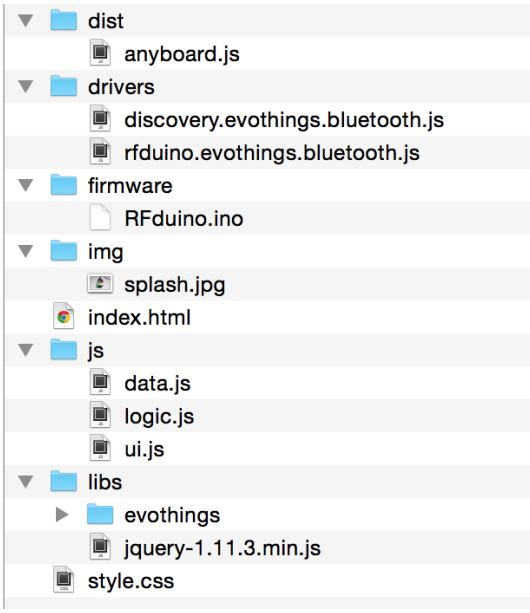

Figure B.6: File structure of the AnyBoard Quiz Game.

B.3 Implementation difference with and without AnyBoard

The code difference between the game implemented with and without AnyBoard is shown in the pdf-file on the next pages. Lines marked + with a green line denotes added lines, while − with red line shows deleted ones (when changing from the AnyBoard implementation to the non-AnyBoard implementation).

For comparison of length where blank lines, comments and unnecessarily "pretty" structures are ignored, we could remove/compress the following lines:

- AnyBoard implementation: 6, 38-47, 66-73
- JS implementatation: 43, 56, 76-82, 85, 86-91, 117, 129, 135, 140, 147, 151, 154, 162, 167, 170, 175, 178, 186-189, 195, 199-200, 204-208, 212, 218, 228, 230, 233-236

Replace AnyBoard logic with plain JS

master

Browse files

tomfa authored 3 hours ago

1 parent 27ade59 commit 64d335c3b606bcf0a1d2f89f0f0be3fbba27a7e0

Showing **2 changed files** with **183 additions** and **41 deletions**.

Unified Split

6 ■■■■■ js-question-game/index.html

View

```
@@ -14,15 +14,9 @@
      <!-- cordova.js based -->
         <script src="cordova.js"></script>

-    <!-- AnyBoard libraries -->
-    <script src="dist/anyboard.js"></script>
-
      <!-- Bluetooth driver and dependencies -->
      <script src="libs/evothings/evothings.js"></script>
      <script src="libs/evothings/easyble/easyble.js"></script>
-    <script src="drivers/discovery.evothings.bluetooth.js"></script>
-    <script src="drivers/rfduino.evothings.bluetooth.js"></script>
-    <script src="drivers/bean.evothings.bluetooth.js"></script>

      <!-- We've used jquery for quick development -->
      <script src="libs/jquery-1.11.3.min.js"></script>
```

218 ■■■■■ js-question-game/js/logic.js

View

```
@@ -1,9 +1,12 @@
+if (typeof hyper === 'undefined') hyper = console;
+
 var logic = {
     initiate: function() {
-        var handleTokenMove = function(token, constraint, options) {
-            AnyBoard.Logger.log(token.player + " moved to tile " + constraint);
+        var handleTokenMove = function(options) {
+            var token = options.token;
+            var constraint = options.constraint;
+            hyper.log(token.player.name + " moved to tile " + constraint);
             token.player.location = constraint;
-
             if (logic.everyOneHasAnswered()) {
                 ui.showAnswer();
                 return;
@@ -13,11 +16,13 @@ var logic = {
             // is placed on a tile. Therefore we check if the token is the only one, and trigger the TT-event manually
             // if it is
             if (logic.numberOfConnectedTokens() === 1 && token.player.location === 2) {
-                handleTokenTokenTouch(token, token);
+                handleTokenTokenTouch({"initiatingToken": token, "respondingToken": token});
             }
         };

-        var handleTokenTokenTouch = function(initiatingToken, respondingToken, options) {
+        var handleTokenTokenTouch = function(options) {
+            var initiatingToken = options.initiatingToken;
+            var respondingToken = options.respondingToken;
             if (respondingToken.player.location === 2) {
                 hyper.log("handleTokenTokenTouch");
                 if (typeof d.currentQuestionPos === "undefined") ui.activatePanel('game');
@@ -28,23 +33,28 @@ var logic = {
         logic.addListener("game", logic.startGame);
         logic.addListener("summary", ui.finishGame);

-        AnyBoard.TokenManager.onTokenConstraintEvent("MOVE_TO", handleTokenMove);
-        AnyBoard.TokenManager.onTokenTokenEvent("MOVE_NEXT_TO", handleTokenTokenTouch)
+        logic.addListener("MOVE_TO", handleTokenMove);
+        logic.addListener("MOVE_NEXT_TO", handleTokenTokenTouch)
     },

     // Discover bluetooth tokens in proximity
     discover: function() {
```

```
37  42          var self = this;
38              AnyBoard.TokenManager.scan(
39                  // success function to be executed upon _each_ token that is discovered
40                  function(token) {
41                      self.addDiscovered(token);
42                  },
43                  // function to be executed upon failure
44                  function(errorCode) {
45                      hyper.log(errorCode)
46                  }
47              );
    43  +
    44  +       evothings.easyble.reportDeviceOnce(true);
    45  +       evothings.easyble.startScan(function(device){
    46  +           hyper.log('Device found: ' + device.name + ' address: ' + device.address + ' rssi: ' + device.rssi);
    47  +           device.sendGtHeader = 0x80;
    48  +           device.gettingServices = false;
    49  +           device.serialChar = null; // Characteristic handle for serial write, set on getServices()
    50  +           device.serialDesc = null; // Description for characteristic handle, set on getServices()
    51  +           device.singlePacketWrite = true;
    52  +           self.addDiscovered(device);
    53  +       }, function(errorCode) {
    54  +           hyper.log(errorCode)
    55  +       });
    56  +
    57  +       setTimeout(function() {evothings.easyble.stopScan();}, 5000);
48  58      },
49  59
50  60      // Function to be executed upon having discovered a token
        @@ -59,25 +69,34 @@ var logic = {
59  69              '<button class="player-icon' + '"> </button>' + '</div>');
60  70
61  71              // Add listener to be executed if the token connects
62              -           token.on('connect', function() {
    72  +           logic.addListener('connect', function(token) {
63  73                  document.getElementById(token.address).className = 'green token';
64  74                  token.color = d.colors.pop();
65              -           token.player = new AnyBoard.Player(
66              -               token.color + "-player",
67              -               {
68              -                   "color": token.color,
69              -                   "points": 0,
70              -                   "locations": -1,
71              -                   "token": token
72              -               }
73              -           );
    75  +           token.isConnected = true;
    76  +           token.player = {
    77  +               "name": token.color + "-player",
    78  +               "color": token.color,
    79  +               "points": 0,
    80  +               "locations": -1,
    81  +               "token": token
    82  +           };
74  83                  d.players.push(token.player);
75  84                  $(document.getElementById(token.address)).next().addClass(token.color);
76              -           token.ledOn(token.color);
    85  +
    86  +           var supportedColors = {
    87  +               'red': [255, 0, 0],
    88  +               'green': [0, 255, 0],
    89  +               'blue': [0, 0, 255],
    90  +               'white': [255, 255, 255]
    91  +           };
    92  +           var ledOnBitCommand = 129;
    93  +           logic._sendOutgoingData(token, new Uint8Array(supportedColors[token.color]).unshift(ledOnBitCommand));
77  94              });
78  95
79  96              // Add listener to be executed if the token disconnects
80              -           token.on('disconnect', function() {
    97  +           listener.addListener('disconnect', function(options) {
    98  +               var token = options.token;
    99  +               token.isConnected = false;
81  100                 document.getElementById(token.address).className = 'grey token';
82  101                 $(document.getElementById(token.address)).next().removeClass(token.color);
83  102                 d.colors.push(token.color);
        @@ -86,6 +105,135 @@ var logic = {
86  105
```

```
87  106     },
88  107
    108 +   _sendOutgoingData: function(device, uint8data, win, fail) {
    109 +       evothings.ble.writeCharacteristic(
    110 +           device.deviceHandle,
    111 +           device.serialChar,
    112 +           uint8data,
    113 +           function(){ win && win()},
    114 +           function(){ fail && fail()}
    115 +       );
    116 +   },
    117 +
    118 +   _handleIncomingData: function(device, uint8data) {
    119 +       var moveBitCommand = 194;
    120 +       var cmd = uint8data[0];
    121 +       var currentTile = uint8array[1];
    122 +       var previousTile = uint8array[2];
    123 +       if (cmd === moveBitCommand) {
    124 +           logic.trigger("MOVE_TO", {"token": device, "constraint": currentTile});
    125 +           logic.addListener("MOVE_NEXT_TO", handleTokenTokenTouch)
    126 +       }
    127 +       hyper.log(device.address + " moved from " + previousTile + " to " + currentTile);
    128 +   },
    129 +
    130 +   _evothingsDisconnect: function(device) {
    131 +       device.close();
    132 +       device.haveServices = false;
    133 +       logic.trigger("disconnect", {"token": device });
    134 +   },
    135 +
    136 +   _evothingsConnect: function(device) {
    137 +       var requiredCharacteristic = 'a495ff11-c5b1-4b44-b512-1370f02d74de';
    138 +       var requiredService = 'a495ff10-c5b1-4b44-b512-1370f02d74de';
    139 +       var requiredDescriptor = '00002902-0000-1000-8000-00805f9b34fb';
    140 +
    141 +       device.connect(function() {
    142 +           getServices();
    143 +       }, function(errorCode) {
    144 +           device.haveServices = false;
    145 +           fail(errorCode);
    146 +       });
    147 +
    148 +       var getServices = function() {
    149 +           if (device.gettingServices)
    150 +               return;
    151 +
    152 +           var self = device;
    153 +           device.gettingServices = true;
    154 +
    155 +           hyper.log('Fetch services for ' + device.address);
    156 +           evothings.ble.readAllServiceData(
    157 +               device.deviceHandle,
    158 +               function(services) {
    159 +                   device.services = {};
    160 +                   device.characteristics = {};
    161 +                   device.descriptors = {};
    162 +
    163 +                   for (var si in services) {
    164 +                       var service = services[si];
    165 +                       if (service.uuid !== requiredService)
    166 +                           continue;
    167 +
    168 +                       device.services[service.uuid] = service;
    169 +                       hyper.log('Service: ' + service.uuid);
    170 +
    171 +                       for (var ci in service.characteristics) {
    172 +                           var characteristic = service.characteristics[ci];
    173 +                           if (characteristic.uuid !== requiredCharacteristic)
    174 +                               continue;
    175 +
    176 +                           hyper.log('Characteristic: ' + characteristic.uuid);
    177 +                           device.characteristics[characteristic.uuid] = characteristic;
    178 +
    179 +                           for (var di in characteristic.descriptors) {
    180 +                               var descriptor = characteristic.descriptors[di];
    181 +                               if (descriptor.uuid !== requiredDescriptor)
    182 +                                   continue;
    183 +                               device.descriptors[descriptor.uuid] = descriptor;
    184
```

```
+                                device.serialChar = characteristic.handle;
185 +                                device.serialDesc = descriptor.handle;
186 +                            }
187 +                        }
188 +                    }
189 +
190 +                    if (device.serialChar) {
191 +                        device.haveServices = true;
192 +                        device.gettingServices = false;
193 +                        logic._evothingsSubscribeToEvents(device, logic._handleIncomingData);
194 +                        logic.trigger('connect', device);
195 +                    }
196 +                    else {
197 +                        device.gettingServices = false;
198 +                        hyper.log('Could not find predefined services for token');
199 +                    }
200 +                },
201 +                function(errorCode) {
202 +                    device.gettingServices = false;
203 +                    hyper.log('Could not fetch services for token ' + device.name + '. ' + errorCode);
204 +                }
205 +            );
206 +        }
207 +    },
208 +
209 +    _evothingsSubscribeToEvents: function(device, callback) {
210 +        var enableSubscribeDescriptor = '00002902-0000-1000-8000-00805f9b34fb';
211 +        var notificationCharacteristic = '00002221-0000-1000-8000-00805f9b34fb';
212 +
213 +        evothings.ble.writeDescriptor(
214 +            device.deviceHandle,
215 +            device.descriptors[enableSubscribeDescriptor].handle,
216 +            new Uint8Array([1,0])
217 +        );
218 +
219 +        evothings.ble.enableNotification(
220 +            device.deviceHandle,
221 +            device.serialChar,
222 +            function(data){
223 +                data = new DataView(data);
224 +                var length = data.byteLength;
225 +                var uint8Data = [];
226 +                for (var i = 0; i < length; i++) {
227 +                    uint8Data.push(data.getUint8(i));
228 +                }
229 +                callback && callback(device, uint8Data);
230 +            },
231 +            function(errorCode){
232 +                hyper.log("Could not subscribe to notifications");
233 +            }
234 +        );
235 +    },
236 +
89  237     // Attempts to connect to token.
90  238     connect: function(tokenAddress) {
91  239         var token = d.devices[tokenAddress];
    @@ -96,14 +244,14 @@ var logic = {
96  244
97  245         // If already connected, disconnect
98  246         if (document.getElementById(tokenAddress).className.indexOf('green') !== -1) {
99      -            token.disconnect();
    247 +            logic._evothingsDisconnect(token);
100 248             return;
101 249         }
102 250         // Signal that we're attempting to connect
103 251         document.getElementById(tokenAddress).className = 'blue token';
104 252
105 253         // Send connect command.
106     -        token.connect();
    254 +        logic._evothingsConnect(token);
107 255     },
108 256
109 257     startGame: function(){
    @@ -165,9 +313,9 @@ var logic = {
165 313     },
166 314
167 315     everyOneHasAnswered: function() {
168     -        var tokenSet = AnyBoard.TokenManager.tokens;
```

```
316 +        var tokenSet = d.devices;
169 317      for (var key in tokenSet) {
170     -        if (tokenSet.hasOwnProperty(key) && tokenSet[key].isConnected()) {
    318 +        if (tokenSet.hasOwnProperty(key) && tokenSet[key].isConnected) {
171 319              if (tokenSet[key].player.location < 3) {
172 320                  return false;
173 321              }
    ❄   @@ -177,10 +325,10 @@ var logic = {
177 325      },
178 326
179 327      numberOfConnectedTokens: function() {
180     -        var tokenSet = AnyBoard.TokenManager.tokens;
    328 +        var tokenSet = d.devices;
181 329          var numOfConnected = 0;
182 330          for (var key in tokenSet) {
183     -            if (tokenSet.hasOwnProperty(key) && tokenSet[key].isConnected()) numOfConnected += 1;
    331 +            if (tokenSet.hasOwnProperty(key) && tokenSet[key].isConnected) numOfConnected += 1;
184 332          }
185 333          return numOfConnected;
186 334      }
    ❄
```

Appendix C

Provided examples

We have created four small examples that can be found at github.com/tomfa/anyboardjs. We will describe them in the following section.

C.1 Deck and Card

This example is an example of a player that draw cards from a deck to his hand. The cards on his hand can then be played to the board. The game also shows a log from the events of the game on the screen.

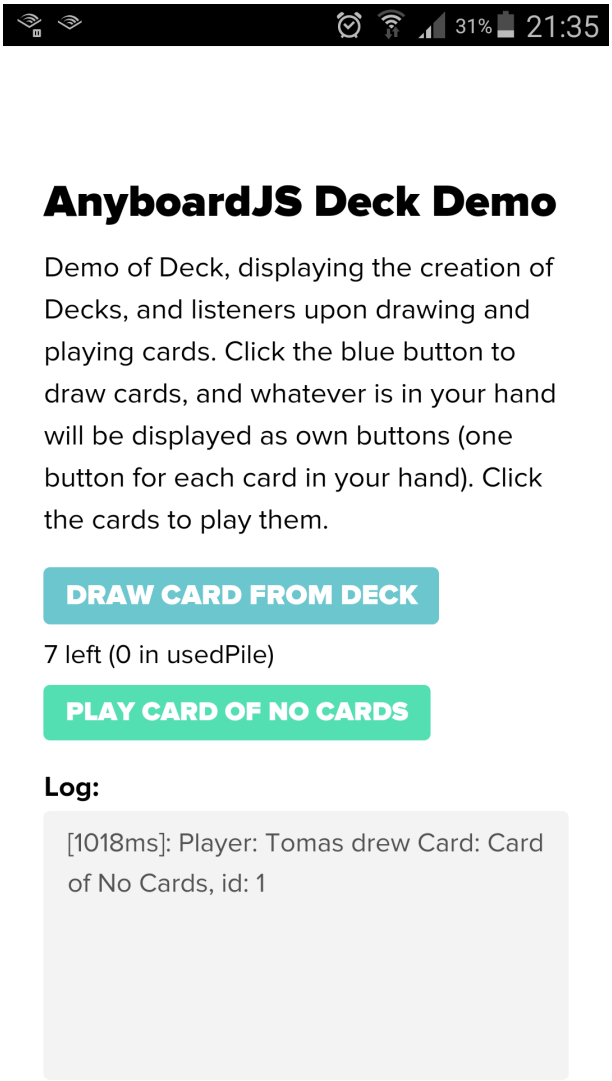

Figure C.1: The Deck demo. New cards are drawn by clicking "DRAW CARDS FROM DECK". Each card on hand is shown as a green button with the card title printed on top. Events happening to the token is listened to an printed in the log part at the bottom.

The example shows use of:

- AnyBoard.Logger, setting threshold.
- AnyBoard.Deck, its loading of data and callback on play and draw events.
- AnyBoard.Card, its callback on play events
- AnyBoard.Player, playing cards and discarding hand.

The code for this example can be found in `examples/mobile-deck/` and `examples/html-deck/`.

C.2 LED demo

Here we show how one can scan for devices, then select which of them to connect to, and send a simple LED command.

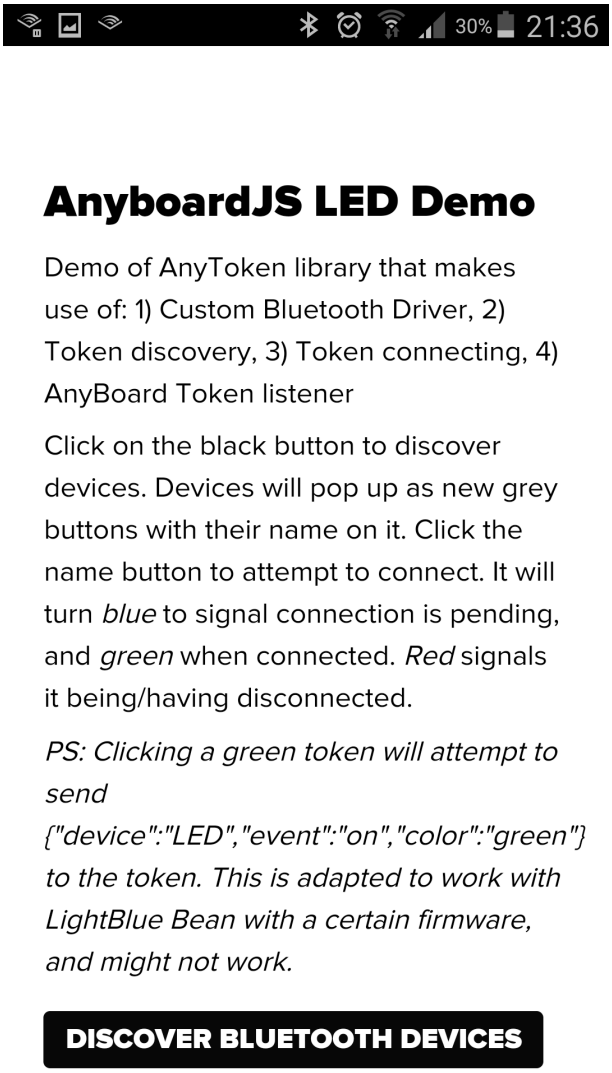

Figure C.2: Simple example of turning on LED. It provides a button for discovering nearby tokens, which then is shown is individual buttons. Once clicked, the example will connect to the corresponding button and send a command to turn on its LED. The same design is used in the print demo and color detection demo.

The example shows use of:

- AnyBoard.TokenManager, scanning for devices
- AnyBoard.BaseToken, listening to events, connecting, disconnecting and sending LED command

The code for this example can be found in `examples/mobile-led-on/`.

C.3 Printer demo

In this example we print some dummy text to the Bean Printer.

The example shows use of:

- AnyBoard.TokenManager, scanning for devices
- AnyBoard.BaseToken, listening to events, connecting, disconnecting and sending print command

The code for this example can be found in `examples/mobile-bean-printer/`.

C.4 Color detection demo

In this example we connect to a token, and register and logs its movements between pawns of different color.

The example shows use of:

- AnyBoard.TokenManager, scanning for devices
- AnyBoard.BaseToken, listening to events, connecting, disconnecting and responding to MOVE command sent from token.

The code for this example can be found in `examples/mobile-rfduino-color-detection/`.

Appendix D

Implemented tokens

D.1 AnyBoard RFduino Token

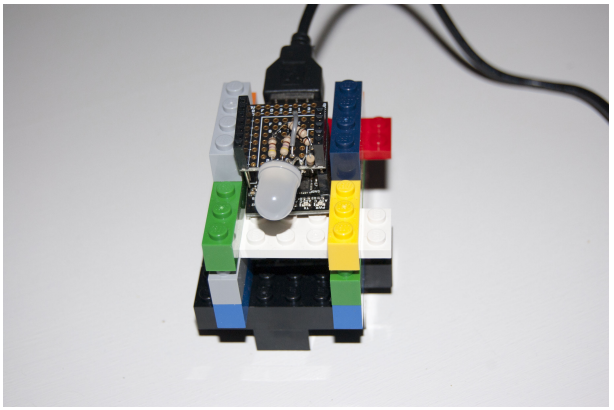

Figure D.1: The AnyBoard RFduino Pawn

The AnyBoard Token is based on RFduino components and was assembled by co-supervisor Simone Mora. It is an RFduino with Bluetooth and battery capabilities and uses Adafruit TCS34725www.adafruit.com/products/1334 for detecting colors. The cost of this is roughly 50 USD using manufactured shields. Homemade circuit boards can be created at roughly 10 USD.

Other than detecting colors, the token also provides the capability of displaying a color.

AnyBoard communicates with the token through `drivers/rfduino.evothings.Bluetooth.js`, and the token runs on

`firmware/RFduinoToken.ino`.

The token is under continued development by Mora.

D.2 AnyBoard Bean Token

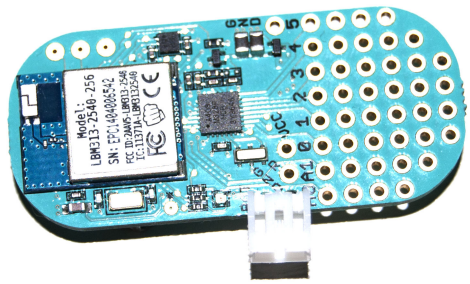

Figure D.2: The AnyBoard Bean

The AnyBoard Bean Token consists simply of a LightBlue Bean[1]. The cost of this is 30 USD.

The token provides capability of displaying a LED color, measure temperature and contains an accelerometer. Neither temperature or accelerometer reading has been implemented, but could perhaps spark some neat ideas if they were available. Including these features in the existing driver and firmware should be a simple job.

AnyBoard communicates with this setup through |drivers/bean.evothings.Bluetooth.js|, and the bean runs on |firmware/BeanToken.ino|.

[1] legacy.punchthrough.com/bean

D.3 AnyBoard Bean Printer

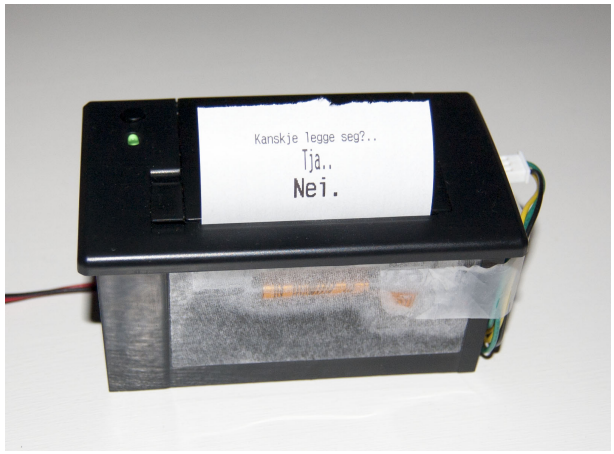

Figure D.3: The AnyBoard Bean Token

The AnyBoard Bean printer consists of an Adafruit Mini Thermal Receipt Printer[2] connected with a LightBlue Bean[3].The cost of this setup is roughly 95 USD.

Only text-size, printer feeding and text-alignment was implemented during this thesis. We described potential uses of its other capabilities (QR, barcode etc.) in chapter 7

AnyBoard communicates with this setup through |drivers/bean.evothings.Bluetooth.js|, and the bean runs on |firmware/BeanPrinter.ino|.

[2] https://learn.adafruit.com/mini-thermal-receipt-printer
[3] legacy.punchthrough.com/bean

Appendix E

AnyBoard Tests

Tests has been written for the AnyBoard entities. This excludes drivers and firmware. In addition, tests have regrettably only partially been implemented for the Token and TokenManager. We advice for these to be completed.

Tests are located inside the folder `test`. All files here are automatically run by navigating to the AnyBoard folder in terminal and running the command `grunt test`.

Upon running, the tests provide a human readable output as shown in figure E.1.

```
[tomas:~/Desktop/repos/AnyBoard-libs]$ grunt test                    (master*)
Running "mochaTest:test" (mochaTest) task


  AnyBoard.Dices
    when initated without parameters
      ✓ is initiated "new AnyBoard.Dices()"
      ✓ has 1 dice
      ✓ that dice has 6 eyes
    when initated with one parameter
      ✓ is initiated with e.g. "new AnyBoard.Dices(10)"
      ✓ have 1 dice
      ✓ that dice have equal amount of eyes as the parameter (10)
    when initated with two parameters
      ✓ is initiated with e.g. "new AnyBoard.Dices(12, 6)"
      ✓ have the second parameter amount of dices (6)
      ✓ each dice has amount of eyes equal to the first parameter
    when rolled with .roll()
      ✓ returns a number [amount of dices] <= x <= [amount of dices] times [eyes]
    when rolled with .rollEach()
      ✓ returns an array of results equal in length to number of dices
      ✓ returns an array where each element is between 1 and number of eyes
```

Figure E.1: Screen shot of the resulting output from running tests.

Appendix F

Bluetooth communication protocol

		Removed permanently		Not implemented (no use yet)			
		SEND DATA				**RETURN DATA**	
SENDER	**NICKNAME**	**FUNCTION (1B)**	**DATA (up to 11B)**			**DATA (up to 20B)**	**Comment**
Chip	INVALID_DATA_RECEIVE	0				[0]	Sent if token didn't support incoming data command
Cellphone	GET_NAME	32				[32, <less than 20 chars>]	Returns name of device, in ASCII (example: "AnyPawn")
Cellphone	GET_VERSION	33				[33, <less than 20 chars>]	Returns version of firmware, in ASCII (example: "0.1")
Cellphone	GET_UUID	34				[34, <less than 20 chars>]	Returns a unique identifier for that token
Cellphone	GET_BATTERY_STATUS	35					Returns the status of battery, in percentage
Cellphone	HAS_LED	64				[64, <1 if true, else 0>]	Whether or not it has a LED
Cellphone	HAS_LED_COLOR	65				[65, <1 if true, else 0>]	Whether or not it has LED with color
Cellphone	HAS_VIBRATION	66				[66, <1 if true, else 0>]	Whether or not it has vibration
Cellphone	HAS_COLOR_DETECTION	67				[67, <1 if true, else 0>]	Whether or not it can detect color of underlying board
Cellphone	HAS_LED_SCREEN	68				[68, <1 if true, else 0>]	Whether or not it has LED based screen
Cellphone	LED_SCREEN_WIDTH	69				[69, <2 byte integer width>]	Returns number of columns in LED "screen"
Cellphone	LED_SCREEN_HEIGHT	70				[70, <2 byte integer width>]	Returns number of rows in LED "screen"
Cellphone	HAS_RFID	71				[71, <1 if true, else 0>]	Whether or not it can read RFID chips
Cellphone	HAS_NFC	72				[72, <1 if true, else 0>]	Whether or not it can read NFC chips
Cellphone	HAS_ACCELEROMETER	73				[73, 0, 0, 0] to [73, 1, 1, 1] (xyz axis)	Whether or not it has accelerometer (X, Y, Z axis)
Cellphone	HAS_PRINT	74				[74, <1 if true, else 0>]	
Cellphone	LED_OFF	128				[128] to confirm action taken	Turns off LED (blink or stable)
Cellphone	LED_ON	129	[1B red, 1B green, 1B blue]			[129] to confirm action taken	Turns on LED (stable)
Cellphone	LED_BLINK	130	[1B red, 1B green, 1B blue]			[130] to confirm action taken	Turns on LED (blinking)
Cellphone	VIBRATE_OFF	131				[131] to confirm action taken	Cancels any vibration if still active
Cellphone	VIBRATE	132	[1B length, 1B mode, 1B strength]			[132] to confirm action taken	Turns on vibration for up to 25,6 seconds (length), up to 256 different modes, 256 different strengths)
Cellphone	SET_LED_SCREEN	133	??			[133] to confirm action taken	??
Cellphone	READ_NFC	134				Return raw NFC read data	
Cellphone	READ_RFID	135				Returns raw RFID read data	
Cellphone	READ_COLOR	136				Returns raw camera read color code	
Cellphone	PRINT_FEED	137				[137] to confirm action taken	
Cellphone	PRINT_JUSTIFY	138	[1B character "l"/"c"/"r"]			[138] to confirm action taken	
Cellphone	PRINT_SET_SIZE	139	[1B character "S"/"M"/"L"]			[139] to confirm action taken	
Cellphone	PRINT_WRITE	140	ASCII encoding for characters to write			[140] to confirm action taken	
Chip	LIFT	192	[192, 2B x-axis, 2B y-axis, 2B z-axis]				Indicates the pawn is being lifted
Chip	MOVE	194	[194, 1B previous sector, 1B new sector]				Indicates pawn has arrived at new tile

Appendix G

Article: Reflections on AnyBoard

Making interactive board games to learn: Reflections on AnyBoard

Simone Mora[*], Tomas Fagerbekk[*], Ines Di Loreto[**], Monica Divitini[*]

[*] Dept. of Information and Computer Science, NTNU, Trondheim, Norway
{simone.mora, divitini}@idi.ntnu.no
[**] TechCICO, ICD-Université de technologie de Troyes, France
ines.di_loreto@utt.fr

ABSTRACT. In this paper we discuss the making of interactive board games as a learning activity. We do this by presenting AnyBoard, a platform that we are currently developing to support the design and implementation of board games. In our approach we do not use a game board virtualised on an interactive surface, but rather achieve interactivity through technology-augmented game pieces. In this way, we aim at offering to game designers a broader design space and lower costs of the final product. In this paper we discuss the possible use of AnyBoard in the learning context.

1 Introduction

Making games, either analogic or computer-based, has long been used as a learning activity in different educational contexts. Making games has proved useful in areas as diverse as engaging students with cultural heritage [1] and teaching university students about software architectures [2]. Teachers have used making games as a way for teaching programming in high or middle schools for many years, for example using RPG maker[1] or Scratch[2].

In this context, we want to discuss the making of interactive board games to promote learning. With the term interactive we mean board games that use tangible computer-augmented objects. There are different benefits that could be achieved, mainly combining the learning strengths of creating games with the strengths of making tangible and interactive objects for educational purposes [3, 4]. In addition, board games are popular also among the elderly, offering a cross-generational form of entertainment. This is an aspect that could be exploited to create cross-generational maker activities. Finally, by making board games the learner does not get distracted by the complex graphics that is common to many video games. In this way, it is easier for the learner to concentrate on the game concept.

To ease the development of digital board game we present *AnyBoard*, a framework for supporting the making of interactive board games. *AnyBoard* supports the design of digital board games by providing theoretical constructs, software tools and a set of augmented game pieces (all currently under development). The platform was not originally

[1] RPG Makers - https://en.wikipedia.org/wiki/RPG_Maker
[2] The Scratch project – http://scratch.mit.edu

designed to be used in the educational context, but mainly targeting maker communities. However, it could potentially be useful in the context of educational maker activities. In this paper, after presenting the *AnyBoard* approach, we discuss the challenges connected to the use of this platform for learning building on our experience on the creation of an interactive board game.

2 The AnyBoard framework

The dominant paradigm for creating digital board games consists in designing games for interactive surfaces such as smartphones, tablets or tabletop computers. In some cases, artefacts that resemble game tokens, yet augmented with markers (e.g. barcodes, RFIDs), can facilitate interaction with the interactive surface [5, 6].

We propose a different approach: the game pieces are the means to bring interactivity, rather than the game board virtualised on an interactive surface. Distributing interactivity across multiple components opens for a wider space of possibility in designing game experiences. For example, game pieces can influence the state of a game not only when they sit on an interactive surface, but also when they are manipulated over and around it. In this way, the board is mainly used to stage the game and set a context for the use of the pieces, as in traditional board games. In addition, the interactive area of the board is less limited by size, which also determines the portability of the game and costs.

2.1 A new perspective on digital board games

In our approach to digital board games the role of technology is twofold. On one side it brings interactivity by augmenting, not virtualizing, pieces' material representations, for example we aim at providing developers with tangible game pieces augmented with visual, audio or haptic feedbacks (e.g. by means of LEDs or displays). On the other side sensor technology is used to capture players' physical interaction with pieces aiming at preserving their traditional physical affordances; for example, to sense the result of a dice throw, or the movements of pieces onto the board.

Game pieces still preserve their traditional aspect, having a tangible representation that complements an intangible one provided by technology. For example, in a revisited version of Monopoly tokens might preserve their physical semblances to identify players but might embed a graphical representation of the number of property owned by the player (e.g. in icons or symbols on a LCD display). The intangible representation is kept updated by a computer game engine during the playtime, as a consequence of players' interaction with game pieces and activation of game rules. The interaction with pieces is based on a double loop [7].

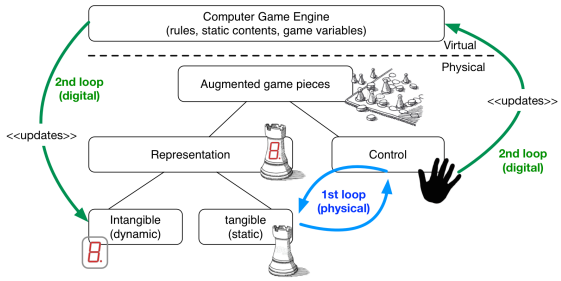

Fig. 1. Double interaction loop in interactive board games

A first interaction loop consists in the passive haptic and visual feedback the player perceives when manipulating pieces on the board, this loop is in common with traditional board games. A second loop adds interactivity by means of graphical and auditory feedbacks conveyed via the tokens' intangible representation (Figure 1). This loop requires technology for sensing tokens' manipulations as well as providing visual/audio feedbacks. The set of valid interactions with game pieces are defined by the affordances of pieces and by game rules. To formalize these rules we build on two theories: the T+C framework [8], providing a powerful descriptive language, and the MCRit model [9], addressing issues of representations and control in TUI.

2.2 Key design constructs

We define a game, which is composed by *game dynamics* (the sum of game logic and rules), as a sequence of player-initiated *interaction events* that modify *spatial configurations* of *tokens* with respect to board *constraints* and other tokens. In the following, building on the T+C framework, we describe key constructs required to develop an *Anyboard* game.

Tokens are technology-augmented artifacts capable of triggering digital operations that can activate game dynamics. They are an augmentation of traditional pawns, dice and cards. Tokens may be capable of sensing information (e.g. proximity with other tokens) and displaying computer graphic and sound.

Constraints are confining regions in the board space, for example checks in the Chess game and territories in Risk. The association or dissociation of a token within a constraint can be mapped to digital operations to activate game dynamic. Constrained regions are determined by a perimeter that could be visual, or physical.

Interaction events are player-triggered manipulations of tokens, that modify the (digital and physical) state of a game. We identified three types of events.

Solo-token event (T) - the manipulation of a single token over or on the board. For example, the action of rolling a dice or drawing a card.

Token-constraint event (T-C) - the operation of building transient token-constraint associations by adding or removing tokens to a constrained region of the board. T-C events can have different consequences depending on game rules.

Token-token (T-T) event - the operation of building transient token-token adjacency-relationships, achieved by moving tokens on the board. For example, approaching a token next to a different token to unlock special powers, or to exchange a resource between two players.

Sequence of *interaction events*, validated against game-specific rules, activate game dynamics and allow the game to evolve from a state to another. For example, we can model the act of capturing a piece in chess as a sequence of interaction events that modify proximity between two chess tokens within checkers constraints. For more details, see [10].

In the following section we describe how theoretical contracts have been implemented for the augmentation of an existing board game.

2.3 An example

Don't Panic, is a collaborative game inspired by Pandemic[3]. Four players start the game as member of a panic manager team that must work together to manage panicking crowds. A map representing a city map is displayed on the game board and the territory is divided in *sectors*. Each sector contains a number of people (PO) characterized by a panic level (PL). During the game randomly triggered panicking events (e.g. fires, explosions) increase PLs in determinate sectors. Each player is represented on the board space by a personal *pawn token* and gets a limited number of actions with the goal to lower the panic level in the city. Using the public *"Calm!" and "Move!" tokens* a player can either reduce the panic in a specific sector or move panicked people to an adjacent sector. Information *cards tokens* distributed in each turn can lower the panic in multiple sectors. Players collectively win the game when the PL in all sectors is zero. For a full description of game rules see [11].

Don't Panic is composed by a cardboard and a set of tokens:

The board (Figure 2-a) – is a cardboard that visualizes a map portraying a territory divided in nodes, sector and paths. Nodes feature physical *constraints* and no degree of freedoms for the hosted tokens; sector and paths provide visual *constraints* allowing tokens' translation and rotation, within the perimeter.

The card deck (Figure 2-c) – dynamically print information card *tokens*. Each card has a textual description of how it affects the game and a barcode that links the card to its digital representation. The top surface of the card deck can read the barcode on the card and trigger actions in the game (Figure 2-d).

Pawn tokens (Figure 2-b) – embody the players' presence on a node. Pawns can be moved from node to node and provide visual information via a LCD display. These include the role of the player, number of people present in sectors adjacent to each of the four pawn's sides; and their panic level.

The Calm! token – represent the field action of calming people talking to them, thus reducing the PL in a specific sector. This action is activated when Calm! is bumped towards a Pawn token (Figure 2-e).

[3] Pandemic board game - http://zmangames.com

The Move! token – simulates moving people across sectors, in this way people moved acquire the panic level of the recipient sector. This action is activated by dragging Move! across a border between two sectors (Figure 2-f).

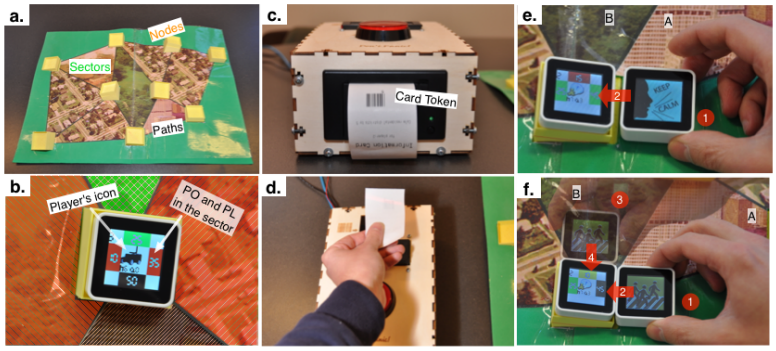

Fig. 2. Don´t Panic interactive tokens

3 The AnyBoard software library

We present here a library to bridge the gap between the theoretical constructs reported in section 2 and the making of interactive game pieces.

Game design communities, game developer environments and game engines already exist, and hence the main part that makes the AnyBoard framework unique is helping integrate the development interactive tokens interaction as a part of games (Figure 3). This role is performed by the *Token Manager* library.

The *token manager* provides a *Token API* to available game engines, so developers can listen to player-token events and send commands to the interactive tokens without the knowledge of the low level code or the tokens' hardware. The API provides primitives specifically suited for augmented board games, such as Token, Constraint and Interaction Events (Section 2.2). The API is generic enough to be used with popular game engines, both commercial and open source, such as Phaser[4] and Unity[5].

On the other end, the *token manager* is separated from any specific hardware implementation, and communicates with the physical tokens through *device-specific drivers*. Besides a set of tokens is provided as part of the framework, expert users can tinker them or build new tokens using popular toolkits such as Arduino and RaspberryPi. Drivers for the Arduino platform as well as a generic extendable driver will be provided to assist developers that wish to create their own tokens with specific technologies.

[4] Phaser game engine – http://phaser.io
[5] Unity game engine – http://unity3d.com

The AnyBoard library builds on the Apache Cordova[6] platform that enable games made for AnyBoard to compile to different operating systems, including mobile ones such as Android and iOS. We aim to use open source, free-to-use, modular and well documented tools, so that a developer can pick apart the AnyBoard system and add capabilities where need be.

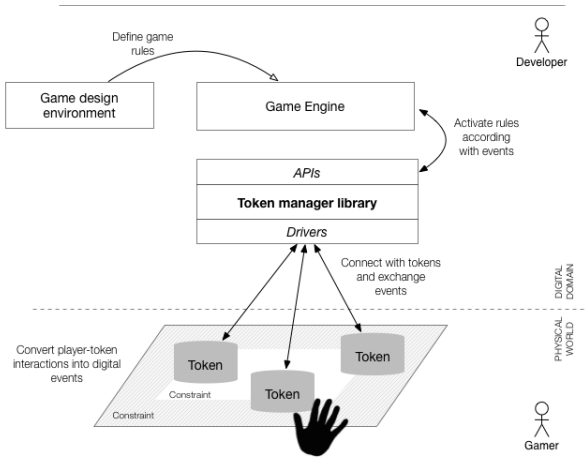

Fig. 3. High level components of AnyBoard

Standard example games, and templates implementing typical token capabilities, will be provided for developers that wish to create games with general token requirements.

Finally, a web-based community for AnyBoard is intended to grow a community and provide information for all roles involved with augmented board games. The AnyBoard platform will be available from there, and tokens sold from a third-party or made using prototyping techniques and open source schematics. The community will provide a knowledge base and tools for developers. Furthermore, it will feature a repository of ready to use *Anyboard games* and an assistive IDE for game.

4 AnyBoard for learning

Making an interactive board game with AnyBoard requires different competencies, varying from game design to software and hardware development skills, and it therefore opens for the design of learning activities with different and multiple learning objectives. In this section we reflect on how the platform, when fully developed, could be used for learning.

[6] Apache Cordova Platform - https://cordova.apache.org

The phases that are required for the full development of an AnyBoard game include:

- Game design, i.e. the definition of the game concept, logic and rules
- Interaction game design, i.e. the definition of the interactions of the players with the game tokens and the interaction among players, either directly or mediated by game elements
- Mapping of the game into the associated token+constraints system
- Implementation of hardware and software, this might range from implementation of the game engine to the development of the token interactions. This phase might also include the production of objects that are not computerized, like the board and cases to tailor the appearance of tokens, e.g. using 3D-printing.

The different phases allow to explore different learning goals through adequate activities. The design of the learning activity might focus on one or more of the following learning area:

- Specific subjects. If the game is designed as a serious game, i.e. by playing the game players are expected to learn X, it requires that students gain a knowledge of subject X to inform their design. In this perspective, the implementation part might be less relevant and the main focus is on the first phase of game design.

- Interaction design by designing the intended interactions among players and the interaction with the tokens. It should be noted that the actual design of the tokens appearance and interaction is strictly related to the game design. It can also become a way to learn about the game subject. For example, in developing a game for crisis management, one could work on tokens that resemble actual objects in the domain, mirroring their behavior in the real world.

- Abstract thinking/logic. To achieve this learning goal, in addition to the high level design, one should put focus on the translation of the high level rules into the framework constructs in terms of tokens and constraints.

- Coding. Learning to code can be achieved during the implementation phase. This might include both more traditional coding for the game engine and coding for embedded systems. In this way, different computational approaches, languages, and feedback systems might be explored.

- Tinkering. The design and implementation of the game requires to play around with different software and tangible components.

5 Conclusions: towards a revised framework

In this short paper we presented an innovative approach to design of interactive board games. The approach is based on the use of interactive tokens on an analogic surface, in alternative to current approaches that mainly rely on interactive surfaces. This approach, we claim, might be suitable to be used in the context of learning by making. The paper discusses the potential of the framework for learning.

To realize this vision there are a number of components that should be added to the framework, including:

- Graphical interface for coding, hiding if required by the design of the learning activity, the complexity of moving from high level rules to the token+constraint system.
- Templates for learning activities with different learning objectives. These templates should help the organizers of the learning activities to quick-start the design, choosing activities that reflect the intended learning objective.
- Scaffolding, possibly including support to hide complexity or irrelevant parts of the platform. This can be achieved in different ways at different level of complexity. It should be taken into account that board games might have a lot of objects and very complex rules that can be overwhelming for a non expert. At the same time, though learners are getting less distracted by developing graphics than in a traditional video games, still the development of the tangible parts might become very complex and distract the learner from other possible learning objectives.
- Community support oriented to education
- Analytics for reflection

As part of our future work we aim at developing these components following a learner-centered approach. This will require to identify more in detail the benefits of this approach compared to other types of game development for learning to focus the development of the component necessary for the more suitable learning activities.

REFERENCES

1. Yiannoutsou, N. & Avouris, N.: Game Design as a context for Learning in Cultural Institutions. In C. Karagiannidis, P. Politis, & I. Karasavvidis, eds. *Research on e-Learning and ICT in Education*. New York, NY: Springer New York, pp. 165–177, (2014).
2. Wang, A.I. & Wu, B.: Using Game Development to Teach Software Architecture. *Int. J. Comput. Games Technol.* (2011)
3. Horn, M.S., Crouser, R.J. & Bers, M.U.: Tangible interaction and learning: the case for a hybrid approach. *Personal and Ubiquitous Computing*, 16(4), pp.379–389 (2012).
4. Mellis, D.A. & Buechley, L.: Case studies in the personal fabrication of electronic products. In Proceedings of the Designing Interactive Systems Conference (DIS2012), ACM Press, pp. 268-277.
5. Haller, M., Forlines, C., Koeffel, C., Leitner, J., Shen, C.: Tabletop games: Platforms, experimental games and design recommendations. Art and Technology of Entertainment Computing and Communication. 271–297 (2010).
6. Bakker, S., Vorstenbosch, D., Van Den Hoven, E., Hollemans, G., Bergman, T.: Tangible interaction in tabletop games. In Proc. of ACE 2007. 163–170 (2007).
7. Ishii, H.: Tangible bits: beyond pixels. In Proc. of TEI 2008. (2008).
8. Ullmer, B., Ishii, H., Jacob, R.J.K.: Token+constraint systems for tangible interaction with digital information. ACM Transactions on Computer-Human Interaction (TOCHI). 12, (2005).
9. Ullmer, B., Ishii, H.: Emerging frameworks for tangible user interfaces. IBM systems journal. 39, 915–931 (2000).
10. Mora, S., Di Loreto, I., Divitini, M.: The interactive-token approach to board games. In Proceedings of AMI2015, LNCS, Springer (to appear).
11. Di Loreto, I., Mora, S., Divitini, M.: Don't Panic: Enhancing Soft Skills for Civil Protection Workers. In Proc of SGDA. 7528, 1–12 (2012).

Appendix H

A grammer for mapping token-based interaction to game dynamics

Note: This is a excerpt of a draft from an article by co-supervisor Simone Mora on token-based interaction. It has influenced the thesis in the implementation of token-events

Designing interactive game pieces, *game dynamics* can e mapped to sequences of *expressions* that describe players' interaction during the game.

Expressions are generated using *tokens*, *constraints* and *interaction events* according with a formal syntax and a grammar of rules derived from game-specific rules. Fore example expressions might encode static configurations of tokens on a board both with respect to constraints and to each other. Once validated by a game engine an expression can trigger digital operations that affect the (digital and physical) state of the game.

Tokens are technology-augmented artifacts capable players interact with. They may be capable of sensing information and displaying computer graphic and sound (active tokens) or they can be conventional object enhanced with electronic tags that act as triggers for game rules (passive tokens). Some tokens are personal, embodiment of the player on the board, while others are meant for shared use and can be passed around during the game. Examples of tokens are computer augmented pawns and dice, or barcode-tagged cards.

Constraints are physical or visual confining regions in the board space that can be

mapped with game mechanics. Once a token is placed within a constraint the two can act as a system that enable nested interaction with other token-constraint system. Examples of constraint are checked for chess pieces, territories for risk pawns or the card deck for cards.

Interaction events are player-triggered manipulations of tokens that modify the (digital or physical) state of the game. There are three types of events:

(i) solo-token events (t-event), the manipulation of a single token over on on the board. For example the action of rolling a dice or drawing a card

(ii) token-constraint events (tc-event), the operation of building transient token-constraint associations by adding or removing tokens on particular surface region of the board. For example adding a pawn to a determinate sector of the board.

(iii) token-token events (tt-event), the operation of building transient adjacency relationships between tokens, achieved by moving tokens on the board. For example approaching a pawn next to another token artifact.

We specify the multiple ways tokens, constraint and interaction event can be composed in expressions using the Extended Back-Naur Form (EBNF[1]). In Listing 1 we describe the grammer. In our formalization tokens and constraint resemble the use of *nouns* in natural language while interaction events stand as *verbs*. Sequences of expressions parsed by a game engine can activate game dynamics thus allowing the game to evolve from a state to another. As a consequence, the players are notified by a change of tangible and in tangible representations of tokens. The grammar has to be finalized with *terminal symbols* for `token` and `constraint` for the specific game design.

The proposed syntax doesn't eliminate the possibility of meaningless expressions. For example not all the interaction events we define might be affordable by any token or some configurations of token and constrain, although physically possible, might not be allowed by game rules. Hence, in our grammar syntax rules must be complemented by a set of game-specific rules that define what interaction events, and token-constraint plus token-token spatial relationship are valid to form expressions generated with the proposed syntax. These grammar rules might be derived from game rules.

Examples of expressions that can be mapped to game are: approaching two pawns to exchange resources among players, or associating a card to an area on

[1] BN Explained - http://www.princeton.edu/ãchaney/tmve/wiki100k/docs/Extended_Backus-%E2%80%93Naur_Form.html

the board to unlock special interactions between pawns.

Listing H.1: Grammar for mapping game dynamics to token-based interactions in EBNF form

```
gamedynamic ::= expression*
expression ::= t-expression | tc-expression | tt-expression
t-expression ::= token, t-event
tc-expression ::= token, tc-event, constraint
tt-expression ::= token, tt-event, token
t-event ::= "shake", "tilt", "roll", "rotate_clockwise", "rotate_counterclo
tc-event ::= "add_to", "remove_from"
tt-event ::= "approach_to", "distance_from", "stack_on", "stack_under"
token ::= utf-8*
constraint ::= utf-8*
```

Appendix I

AnyBoard Documentation

The remaining part of the appendix consists of the complete documentation of the AnyBoard library pr. September 2015. This excludes drivers and firmware which is not considered a part of the library. The documentation is also available in interactive format at http://github.com/tomfa/anyboardjs.

Members

***AnyBoard* :** `object`

Global variable AnyBoard.

Typedefs

***playDrawCallback* :** `function`

This type of callback will be called when card is drawn or played

***simpleTriggerCallback* :** `function`

Type of callback called upon triggering of events

***stdStringCallback* :** `function`

Generic callback returning a string param

***stdBoolCallback* :** `function`

Generic callback returning a bool param

***stdNoParamCallback* :** `function`

Generic callback without params

***onScanCallback* :** `function`

Type of callback called upon detecting a token

***stdErrorCallback* :** `function`

This type of callback will be called upon failure to complete a function

AnyBoard : `object`

Global variable AnyBoard.

Kind: global variable

- AnyBoard : `object`
  - .Driver
    - new AnyBoard.Driver(options)
    - .toString() ⇒ `string`
  - .Deck
    - new AnyBoard.Deck(name, jsonDeck)
    - *instance*
      - .shuffle()
      - .initiate(jsonDeck)
      - .refill([newDeck])
      - .onPlay(func)
      - .onDraw(callback)
      - .toString() ⇒ `string`
    - *static*
      - .get(name) ⇒ `Deck`
  - .Card
    - new AnyBoard.Card(deck, options)
    - *instance*
      - .onPlay(func)
      - .onDraw(callback)
      - .toString() ⇒ `string`

        - *static*
            - .get(cardTitleOrID) ⇒ `Card`
    - .Dices
        - new AnyBoard.Dices([eyes], [numOfDice])
        - .roll() ⇒ `number`
        - .rollEach() ⇒ `Array`
    - .Player
        - new AnyBoard.Player(name, [options])
        - *instance*
            - .pay(resources, [receivingPlayer]) ⇒ `boolean`
            - .trade(giveResources, receiveResources, [player]) ⇒ `boolean`
            - .recieve(resourceSet)
            - .draw(deck, [options]) ⇒ `Card`
            - .play(card, [customOptions]) ⇒ `boolean`
            - .toString() ⇒ `string`
        - *static*
            - .get(name) ⇒ `Player`
    - .Hand
        - new AnyBoard.Hand(player, [options])
        - .has(card, [amount]) ⇒ `boolean`
        - .discardHand()
        - .discardCard(card)
        - .toString() ⇒ `string`
    - .Resource
        - new AnyBoard.Resource(name, [properties])
        - .get(name) ⇒ `Resource`
    - .ResourceSet
        - new AnyBoard.ResourceSet([resources], [allowNegative])
        - .contains(reqResource) ⇒ `boolean`
        - .add(resourceSet)
        - .subtract(resourceSet) ⇒ `boolean`
        - .similarities(resourceSet) ⇒ `object`
    - .BaseToken
        - new AnyBoard.BaseToken(name, address, device, [driver])
        - *instance*
            - .isConnected() ⇒ `boolean`
            - .connect([win], [fail])
            - .disconnect()
            - .trigger(eventName, [eventOptions])
            - .on(eventName, callbackFunction)
            - .once(eventName, callbackFunction)
            - .send(data, [win], [fail])
            - .print(value, [win], [fail])
            - .getFirmwareName([win], [fail])
            - .getFirmwareVersion([win], [fail])
            - .getFirmwareUUID([win], [fail])
            - .hasLed([win], [fail])
            - .hasLedColor([win], [fail])
            - .hasVibration([win], [fail])
            - .hasColorDetection([win], [fail])
            - .hasLedScreen([win], [fail])
            - .hasRfid([win], [fail])
            - .hasNfc([win], [fail])
            - .hasAccelometer([win], [fail])
            - .hasTemperature([win], [fail])
            - .ledOn(value, [win], [fail])

        - .ledBlink(value, [win], [fail])
        - .ledOff([win], [fail])
        - .toString() ⇒ `string`
      - *static*
        - .setDefaultDriver(driver) ⇒ `boolean`
    - .Drivers : `Object`
      - .get(name) ⇒ `Driver` | `undefined`
      - .getCompatibleDriver(type, compatibility) ⇒ `Driver`
    - .TokenManager
      - .setDriver(driver)
      - .scan([win], [fail], [timeout])
      - .get(address) ⇒ `BaseToken`
    - .Logger
      - .warn(message, [sender])
      - .error(message, [sender])
      - .log(message, [sender])
      - .debug(message, [sender])
      - .setThreshold(severity)
    - .Utils
      - .isEqual(a, b, [aStack], [bStack]) ⇒ `boolean`

AnyBoard.Driver

Kind: static class of `AnyBoard`

Properties

Name	Type	Description
name	`string`	name of the driver
description	`string`	description of the driver
version	`string`	version of the driver
dependencies	`string`	Text describing what, if anything, the driver depends on.
date	`string`	Date upon release/last build.
type	`Array`	Array of string describing Type of driver, e.g. "bluetooth"
compatibility	`Array` | `object` | `string`	An object or string that can be used to deduce compatibiity, or an array of different compatibilies.
properties	`object`	dictionary that holds custom attributes

- .Driver
  - new AnyBoard.Driver(options)
  - .toString() ⇒ `string`

new AnyBoard.Driver(options)

Represents a single Driver, e.g. for spesific token or bluetooth discovery

Param	Type	Description
options	`object`	options for the driver
options.name	`string`	name of the driver
options.description	`string`	description of the driver
options.version	`string`	version of the driver

options.type	`string`	Type of driver, e.g. "bluetooth"
options.compatibility	`Array` | `object` | `string`	An object or string that can be used to deduce compatibiity, or an array of different compatibilies. How this is used is determined by the set standard driver on TokenManager that handles scanning for and connecting to tokens.
[options.dependencies]	`string`	*(optional)* What if anything the driver depends on.
[options.date]	`string`	*(optional)* Date upon release/last build.
options.yourAttributeHere	`any`	custom attributes, as well as specified ones, are all placed in driver.properties. E.g. 'heat' would be placed in driver.properties.heat.

driver.toString() ⇒ `string`

Returns a short description of the Driver instance

Kind: instance method of `Driver`

AnyBoard.Deck

Kind: static class of `AnyBoard`

Properties

Name	Type	Description
name	`string`	name of Deck.
cards	`Array.<Card>`	complete set of cards in the deck
pile	`Array.<Card>`	remaining cards in this pile
usedPile	`Array.<Card>`	cards played from this deck
autoUsedRefill	`boolean`	*(default: true)* whether or not to automatically refill pile from usedPile when empty. Is ignored if autoNewRefill is true.
autoNewRefill	`boolean`	*(default: false)* whether or not to automatically refill pile with a whole new deck when empty.
playListeners	`Array.<function()>`	holds functions to be called when cards in this deck are played
drawListeners	`Array.<function()>`	holds functions to be called when cards in this deck are drawn

new AnyBoard.Deck(name, jsonDeck)

Represents a Deck of Cards

Param	Type	Description
name	`string`	name of Deck. This name can be used to retrieve the deck via AnyBoard.Deck.all[name].
jsonDeck	`object`	loaded JSON file. See examples/deck-loading/ for JSON format and loading.

deck.shuffle()

Shuffles the pile of undrawn cards . Pile is automatically shuffled upon construction, and upon initiate(). New cards added upon refill() are also automatically shuffled.

Kind: instance method of `Deck`

deck.initiate(jsonDeck)

Reads Deck from jsonObject and provides a shuffled version in pile. Is automatically called upon constructing a deck.

Kind: instance method of `Deck`

Param	Type	Description
jsonDeck	`object`	loaded json file. See examples-folder for example of json file and loading

deck.refill([newDeck])

Manually refills the pile. This is not necessary if autoUsedRefill or autoNewRefill property of deck is true.

Kind: instance method of `Deck`

Param	Type	Default	Description
[newDeck]	`boolean`	`false`	*(default: false)* True if to refill with a new deck. False if to refill with played cards (from usedPile)

deck.onPlay(func)

Adds functions to be executed upon all Cards in this Deck.

Kind: instance method of `Deck`

Param	Type	Description
func	`playDrawCallback`	callback function to be executed upon play of card from this deck

deck.onDraw(callback)

Adds functions to be executed upon draw of Card from this Deck

Kind: instance method of `Deck`

Param	Type	Description
callback	`playDrawCallback`	function to be executed with the 3 parameters AnyBoard.Card, AnyBoard.Player, (options) when cards are drawn

deck.toString() ⇒ `string`

Sting representation of a deck

Kind: instance method of `Deck`

Deck.get(name) ⇒ `Deck`

Returns deck with given name

Kind: static method of `Deck`
Returns: `Deck` - deck with given name (or undefined if non-existent)

Param	Type	Description
name	`string`	name of deck

AnyBoard.Card

Kind: static class of `AnyBoard`
Properties

Name	Type	Description
title	`string`	title of the card.
description	`string`	description for the Card
color	`string`	color of the Card
category	`string`	category of the card, not used by AnyBoard FrameWork
value	`number`	value of the card, not used by AnyBoard FrameWork
type	`string`	type of the card, not used by AnyBoard FrameWork
amount	`number`	amount of this card its deck
deck	`Deck`	deck that this card belongs to
playListeneres	`Array`	holds functions to be called upon play of this spesific card (before potential playListeners on its belonging deck)
drawListeners	`Array`	holds functions to be called upon draw of this spesific card (before potential drawListeners on its belonging deck)
properties	`object`	dictionary that holds custom attributes

new AnyBoard.Card(deck, options)

Represents a single Card Should be instantiated in bulk by calling the deck constructor

Param	Type	Default	Description
deck	`Deck`		deck to which the card belongs
options	`object`		options for the card
options.title	`string`		title of the card.

options.description	`string`		description for the Card
[options.color]	`string`		*(optional)* color of the Card
[options.category]	`string`		*(optional)* category of the card, not used by AnyBoard FrameWork
[options.value]	`number`		*(optional)* value of the card, not used by AnyBoard FrameWork
[options.type]	`string`		*(optional)* type of the card, not used by AnyBoard FrameWork
[options.amount]	`number`	`1`	*(optional, default: 1)* amount of this card in the deck
[options.yourAttributeHere]	`any`		custom attributes, as well as specified ones, are all placed in card.properties. E.g. 'heat' would be placed in card.properties.heat.

card.onPlay(func)

Adds functions to be executed upon a play of this card

Kind: instance method of `Card`

Param	Type	Description
func	`playDrawCallback`	callback function to be executed upon play of card from this deck

card.onDraw(callback)

Adds functions to be executed upon a draw of this card

Kind: instance method of `Card`

Param	Type	Description
callback	`playDrawCallback`	function to be executed upon play of card from this deck

card.toString() ⇒ `string`

Returns a string representation of the card.

Kind: instance method of `Card`

Card.get(cardTitleOrID) ⇒ `Card`

Returns card with given id

Kind: static method of `Card`
Returns: `Card` - card with given id (or undefined if non-existent)

Param	Type	Description
cardTitleOrID	`number` | `string`	id or title of card

AnyBoard.Dices

Kind: static class of `AnyBoard`

- .Dices
  - new AnyBoard.Dices([eyes], [numOfDice])
  - .roll() ⇒ `number`
  - .rollEach() ⇒ `Array`

new AnyBoard.Dices([eyes], [numOfDice])

Represents a set of game dices that can be rolled to retrieve a random result.

Param	Type	Default	Description
[eyes]	`number`	`6`	*(default: 6)* number of max eyes on a roll with this dice
[numOfDice]	`number`	`1`	*(default: 1)* number of dices

Example

```
// will create 1 dice, with 6 eyes
var dice = new AnyBoard.Dices();

// will create 2 dice, with 6 eyes
var dice = new AnyBoard.Dices(2, 6);
```

dices.roll() ⇒ `number`

Roll the dices and returns a the sum

Kind: instance method of `Dices`
Returns: `number` - combined result of rolls for all dices
Example

```
var dice = new AnyBoard.Dices();

// returns random number between 1 and 6
dice.roll()
```

Example

```
var dice = new AnyBoard.Dices(2, 6);

// returns random number between 1 and 12
dice.roll()
```

dices.rollEach() ⇒ `Array`

Roll the dices and returns an array of results for each dice

Kind: instance method of `Dices`
Returns: `Array` - list of results for each dice
Example

```
var dice = new AnyBoard.Dices(2, 6);

// returns an Array of numbers
var resultArray = dice.rollEach()

// result of first dice, between 1-6
resultArray[0]

// result of second dice, between 1-6
resultArray[1]
```

AnyBoard.Player

Kind: static class of `AnyBoard`

Properties

Name	Type	Description
hand	`Hand`	hand of cards (Quests)
faction	`string`	faction (Sp[ecial abilities or perks)
class	`string`	class (Special abilities or perks)
holds	`ResourceSet`	the resources belonging to this player
color	`string`	color representation of player

new AnyBoard.Player(name, [options])

Represents a Player (AnyBoard.Player)

Param	Type	Description
name	`string`	name of the player
[options]	`object`	*(optional)* options for the player
[options.color]	`string`	*(optional)* color representing the player
[options.faction]	`string`	*(optional)* faction representing the player
[options.class]	`string`	*(optional)* class representing the player
[options.yourAttributeHere]	`any`	*(optional)* custom attributes, as well as specified ones, are all placed in player.properties. E.g. 'age' would be placed in player.properties.age.

player.pay(resources, [receivingPlayer]) ⇒ `boolean`

Take resources from this player and give to receivingPlayer.

Kind: instance method of `Player`
Returns: `boolean` - whether or not transaction was completed (false if Player don't hold enough resources)

Param	Type	Description
resources	`ResourceSet`	dictionary of resources
[receivingPlayer]	`Player`	*(optional)* Who shall receive the resources. Omit if not to anyone (e.g. give to "the bank")

player.trade(giveResources, receiveResources, [player]) ⇒ `boolean`

Trade resources between players/game

Kind: instance method of `Player`

Returns: `boolean` - whether or not transaction was completed (false if Player don't hold enough resources)

Param	Type	Description
giveResources	`ResourceSet`	resources this player shall give
receiveResources	`ResourceSet`	resources this player receieves
[player]	`Player`	*(optional)* Who shall be traded with. Omit if not to a player, but to "the bank".

Example

```
new AnyBoard.Resource("gold");
new AnyBoard.Resource("silver");

var startTreasure = new AnyBoard.ResourceSet({"gold": 6, "silver": 42});
var goldTreasure = new AnyBoard.ResourceSet({"gold": 2});
var silverTreasure = new AnyBoard.ResourceSet({"silver": 12});

var dr1 = new AnyBoard.Player("firstDoctor");
var dr2 = new AnyBoard.Player("secondDoctor");

dr1.receive(startTreasure);
dr2.receive(startTreasure);

// returns true. dr1 will now own {"gold": 4, "silver": 54}. dr2 owns {"gold": 8, "silver": 30}
dr1.trade(goldTreasure, silverTreasure, dr2)
```

Example

```
// returns true. dr1 will now own {"gold": 2, "silver": 66}. dr2 still owns {"gold": 8, "silver": 30}
dr1.trade(goldTreasure, silverTreasure)
```

Example

```
var firstOverlappingTreasure = new AnyBoard.ResourceSet({"silver": 115, "gold": "6"});
var secondOverlappingTreasure= new AnyBoard.ResourceSet({"silver": 100, "gold": "7"});

// returns true. The trade nullifies the similarities, so that the trade can go through even though
//     dr1 has < 100 silver
dr1.trade(firstOverlappingTreasure, secondOverlappingTreasure)
```

player.recieve(resourceSet)

Receive resource from bank/game. Use pay() when receiving from players.

Kind: instance method of `Player`

Param	Type	Description
resourceSet	`ResourceSet`	resources to be added to this players bank

Example

```
new AnyBoard.Resource("gold");
new AnyBoard.Resource("silver");

var startTreasure = new AnyBoard.ResourceSet({"gold": 6, "silver": 42});
var secondTresure = new AnyBoard.ResourceSet({"silver": 12, "copper": 122});

var dr1 = new AnyBoard.Player("firstDoctor");  // player owns nothing initially
```

```
dr1.receive(startTreasure);  // owns {"gold": 6, "silver": 42}
dr1.receive(secondTresure);  // owns {"gold": 6, "silver": 54, "copper": 122}
```

player.draw(deck, [options]) ⇒ `Card`

Draws a card from a deck and puts it in the hand of the player

Kind: instance method of `Player`
Returns: `Card` - card that is drawn

Param	Type	Description
deck	`Deck`	deck to be drawn from
[options]	`object`	*(optional)* parameters to be sent to the drawListeners on the deck

Example

```
var dr1 = new AnyBoard.Player("firstDoctor");  // player has no cards initially

// Now has one card
dr1.draw(deck);

// Now has two cards. option parameter is being passed on to any drawListeners (See Deck/Card)
dr1.draw(deck, options);
```

player.play(card, [customOptions]) ⇒ `boolean`

Plays a card from the hand. If the hand does not contain the card, the card is not played and the hand unchanged.

Kind: instance method of `Player`
Returns: `boolean` - whether or not the card was played

Param	Type	Description
card	`Card`	card to be played
[customOptions]	`object`	*(optional)* custom options that the play should be played with

Example

```
var DrWho = new AnyBoard.Player("firstDoctor");  // player has no cards initially

// Store the card that was drawn
var card = DrWho.draw(existingDeck);

// Play that same card
DrWho.play(card)
```

player.toString() ⇒ `string`

Returns a string representation of the player

Kind: instance method of `Player`

Player.get(name) ⇒ `Player`

Returns player with given name

Kind: static method of `Player`
Returns: `Player` - player with given name (or undefined if non-existent)

Param	Type	Description
name	`string`	name of player

AnyBoard.Hand

Kind: static class of `AnyBoard`

new AnyBoard.Hand(player, [options])

Represents a Hand of a player, containing cards. Players are given one Hand in Person constructor.

Param	Type	Description
player	`Player`	player to which this hand belongs
[options]	`object`	*(optional)* custom properties added to this hand

hand.has(card, [amount]) ⇒ `boolean`

Checks whether or not a player has an amount card in this hand.

Kind: instance method of `Hand`
Returns: `boolean` - hasCard whether or not the player has that amount or more of that card in this hand

Param	Type	Default	Description
card	`Card`		card to be checked if is in hand
[amount]	`number`	`1`	*(default: 1)* amount of card to be checked if is in hand

Example

```
var DrWho = new AnyBoard.Player("firstDoctor");  // player has no cards initially

// Store the card that was drawn
var tardis = DrWho.draw(tardisDeck);

// returns true
DrWho.hand.has(card)

// returns false, as he has only one
DrWho.hand.has(card, 3)
```

hand.discardHand()

Discard the entire hand of the player, leaving him with no cards

Kind: instance method of `Hand`

hand.discardCard(card)

Discard a card from the hand of the player

Kind: instance method of `Hand`

Param	Type	Description
card	`Card`	card to be discarded.

hand.toString() ⇒ `string`

Returns a string representation of the hand

Kind: instance method of `Hand`

AnyBoard.Resource

Kind: static class of `AnyBoard`

Properties

Name	Type	Description
name	`string`	name of resource
properties	`any`	custom options added to resource

new AnyBoard.Resource(name, [properties])

Represents a simple resource (AnyBoard.Resource)

Param	Type	Description
name	`string`	name representing the resource
[properties]	`object`	*(optional)* custom properties of this resource

Example

```
var simpleGold = new AnyBoard.Resource("gold");

// The optional properties parameter can be of any type.
var advancedPowder = new AnyBoard.Resource("powder", {"value": 6, "color": "blue"});

// 6
advancedPowder.properties.value
```

Resource.get(name) ⇒ `Resource`

Returns resource with given name

Kind: static method of `Resource`

Returns: `Resource` - resource with given name (or undefined if non-existent)

Param	Type	Description
name	`string`	name of resource

Example

```
var simpleGold = new AnyBoard.Resource("gold");

// returns simpleGold
AnyBoard.Resource.get("gold");
```

AnyBoard.ResourceSet

Kind: static class of `AnyBoard`

Properties

Name	Type	Default	Description
resources	`object`		*(optional)* a set of initially contained resources
allowNegative	`boolean`	`false`	*(default: false)* whether or not to allow being subtracted resources to below 0 (dept)

new AnyBoard.ResourceSet([resources], [allowNegative])

Creates a ResourceSet

Param	Type	Default	Description
[resources]	`object`		*(optional)* a set of initially contained resources
[allowNegative]	`boolean`	`false`	*(default: false)* whether or not to allow being subtracted resources to below 0 (dept)

Example

```
// Returns a resourceset that can be deducted below 0
var debtBank = new AnyBoard.ResourceSet({}, true);
```

resourceSet.contains(reqResource) ⇒ `boolean`

Whether or not a ResourceSet contains another ResourceSet

Kind: instance method of `ResourceSet`

Returns: `boolean` - true if this ResourceSet contains reqResource, else false

Param	Type	Description
reqResource	`ResourceSet`	ResourceSet to be compared against

Example

```
new AnyBoard.Resource("gold");
new AnyBoard.Resource("silver");

var myTreasure = new AnyBoard.ResourceSet({"gold": 6, "silver": 42});
var minorDebt = new AnyBoard.ResourceSet({"gold": 1, "silver": 3});
var hugeDebt = new AnyBoard.ResourceSet({"gold": 12, "silver": 41});

// returns true
myTreasure.contains(minorDebt);

// returns false
myTreasure.contains(hugeDebt);
```

resourceSet.add(resourceSet)

Adds a ResourceSet to this one

Kind: instance method of `ResourceSet`

Param	Type	Description
resourceSet	`ResourceSet`	ResourceSet to be added to this one

Example

```
new AnyBoard.Resource("gold");
new AnyBoard.Resource("silver");

var myTreasure = new AnyBoard.ResourceSet({"gold": 6, "silver": 42});
var minorGift = new AnyBoard.ResourceSet({"silver": 2});

myTreasure.add(minorGift);
// myTreasure is now {"gold": 6, "silver": 45}
```

resourceSet.subtract(resourceSet) ⇒ `boolean`

Subtracts a dictionary of resources and amounts to a ResourceSet

Kind: instance method of `ResourceSet`
Returns: `boolean` - whether or not resources were subtracted successfully

Param	Type	Description
resourceSet	`ResourceSet`	set of resources to be subtracted

Example

```
new AnyBoard.Resource("gold");
new AnyBoard.Resource("silver");

var myTreasure = new AnyBoard.ResourceSet({"gold": 6, "silver": 42});
var minorGift = new AnyBoard.ResourceSet({"silver": 2});
var debtBank = new AnyBoard.ResourceSet({}, true);
var cosyBank = new AnyBoard.ResourceSet();

// returns true. myTreasure becomes {"gold": 6, "silver": 40}
myTreasure.subtract(minorGift);

// returns true. debtbank becomes {"silver": -2}
debtBank.subtract(minorGift);

// returns false and leaves cosyBank unchanged
cosyBank.subtract(minorGift);
```

resourceSet.similarities(resourceSet) ⇒ `object`

Returns the common resources and minimum amount between a dictionary of resources and amounts, and this ResourceSet

Kind: instance method of `ResourceSet`
Returns: `object` - similarities dictionary of common resources and amounts

Param	Type	Description
resourceSet	`ResourceSet`	dictionary of resources and amounts to be compared against

Example

```
new AnyBoard.Resource("gold");
new AnyBoard.Resource("silver");

var myTreasure = new AnyBoard.ResourceSet({"gold": 6, "silver": 42});
var otherTresure = new AnyBoard.ResourceSet({"silver": 2, "bacon": 12});

// returns {"silver": 2}
myTreasure.similarities(otherTresure);
```

AnyBoard.BaseToken

Kind: static class of `AnyBoard`

Properties

Name	Type	Description
name	`string`	name of the token
address	`string`	address of the token found when scanned
connected	`boolean`	whether or not the token is connected
device	`object`	driver spesific data.
listeners	`object`	functions to be execute upon certain triggered events
onceListeners	`object`	functions to be execute upon next triggering of certain events
sendQueue	`Array.<function()>`	queue for communicating with
cache	`object`	key-value store for caching certain communication calls
driver	`Driver`	driver that handles communication

- .BaseToken
  - new AnyBoard.BaseToken(name, address, device, [driver])
  - *instance*
    - .isConnected() ⇒ `boolean`
    - .connect([win], [fail])
    - .disconnect()
    - .trigger(eventName, [eventOptions])
    - .on(eventName, callbackFunction)
    - .once(eventName, callbackFunction)
    - .send(data, [win], [fail])
    - .print(value, [win], [fail])
    - .getFirmwareName([win], [fail])
    - .getFirmwareVersion([win], [fail])
    - .getFirmwareUUID([win], [fail])
    - .hasLed([win], [fail])
    - .hasLedColor([win], [fail])
    - .hasVibration([win], [fail])
    - .hasColorDetection([win], [fail])
    - .hasLedScreen([win], [fail])
    - .hasRfid([win], [fail])
    - .hasNfc([win], [fail])
    - .hasAccelometer([win], [fail])
    - .hasTemperature([win], [fail])
    - .ledOn(value, [win], [fail])
    - .ledBlink(value, [win], [fail])

        - .ledOff([win], [fail])
        - .toString() ⇒ `string`
    - *static*
        - .setDefaultDriver(driver) ⇒ `boolean`

new AnyBoard.BaseToken(name, address, device, [driver])

Base class for tokens. Should be used by communication driver upon AnyBoard.TokenManager.scan()

Param	Type	Default	Description
name	`string`		name of the token
address	`string`		address of the token found when scanned
device	`object`		device object used and handled by driver
[driver]	`Driver`	`AnyBoard.BaseToken._defaultDriver`	token driver for handling communication with it.

baseToken.isConnected() ⇒ `boolean`

Returns whether or not the token is connected

Kind: instance method of `BaseToken`
Returns: `boolean` - true if connected, else false

baseToken.connect([win], [fail])

Attempts to connect to token. Uses TokenManager driver, not its own, since connect needs to happen before determining suitable driver.

Kind: instance method of `BaseToken`

Param	Type	Description
[win]	`stdNoParamCallback`	*(optional)* function to be executed upon success
[fail]	`stdErrorCallback`	*(optional)* function to be executed upon failure

baseToken.disconnect()

Disconnects from the token.

Kind: instance method of `BaseToken`

baseToken.trigger(eventName, [eventOptions])

Trigger an event on a token

Kind: instance method of `BaseToken`

Param	Type	Description
eventName	`string`	name of event
[eventOptions]	`object`	*(optional)* dictionary of parameters and values

Example

```
var onTimeTravelCallback = function (options) {console.log("The tardis is great!")};
existingToken.on('timeTravelled', onTimeTravelCallback);

// Triggers the function, and prints praise for the tardis
existingToken.trigger('timeTravelled');
```

```
existingToken.trigger('timeTravelled');  // prints again
existingToken.trigger('timeTravelled');  // prints again
```

baseToken.on(eventName, callbackFunction)

Adds a callbackFunction to be executed always when event is triggered

Kind: instance method of `BaseToken`

Param	Type	Description
eventName	`string`	name of event to listen to
callbackFunction	`simpleTriggerCallback`	function to be executed

Example

```
var onTimeTravelCallback = function () {console.log("The tardis is great!")};
existingToken.on('timeTravelled', onTimeTravelCallback);

// Triggers the function, and prints praise for the tardis
existingToken.trigger('timeTravelled');

existingToken.trigger('timeTravelled');  // prints again
existingToken.trigger('timeTravelled');  // prints again
```

Example

```
var onTimeTravelCallback = function (options) {
    // Options can be left out of a trigger. You should therefore check
    // that input is as expected, throw an error or give a default value
    var name = (options && options.name ? options.name : "You're");

    console.log(options.name + " is great!");
};
existingToken.on('timeTravelled', onTimeTravelCallback);

// prints "Dr.Who is great!"
existingToken.trigger('timeTravelled', {"name": "Dr.Who"});

// prints "You're great!"
existingToken.trigger('timeTravelled');
```

baseToken.once(eventName, callbackFunction)

Adds a callbackFunction to be executed next time an event is triggered

Kind: instance method of `BaseToken`

Param	Type	Description
eventName	`string`	name of event to listen to
callbackFunction	`simpleTriggerCallback`	function to be executed

Example

```
var onTimeTravelCallback = function (options) {console.log("The tardis is great!")};
existingToken.once('timeTravelled', onTimeTravelCallback);

// Triggers the function, and prints praise for the tardis
existingToken.trigger('timeTravelled');
```

```
// No effect
existingToken.trigger('timeTravelled');
```

baseToken.send(data, [win], [fail])

Sends data to the token. Uses either own driver, or (if not set) TokenManager driver

Kind: instance method of `BaseToken`

Param	Type	Description
data	`Uint8Array` | `ArrayBuffer` | `String`	data to be sent
[win]	`stdNoParamCallback`	*(optional)* function to be executed upon success
[fail]	`stdErrorCallback`	*(optional)* function to be executed upon error

baseToken.print(value, [win], [fail])

Prints to Token

String can have special tokens to signify some printer command, e.g. ##n = newLine. Refer to the individual driver for token spesific implementation and capabilites

Kind: instance method of `BaseToken`

Param	Type	Description
value	`string`	
[win]	`stdNoParamCallback`	*(optional)* callback function to be called upon successful execution
[fail]	`stdErrorCallback`	*(optional)* callback function to be executed upon failure

baseToken.getFirmwareName([win], [fail])

Gets the name of the firmware type of the token

Kind: instance method of `BaseToken`

Param	Type	Description
[win]	`stdStringCallback`	*(optional)* callback function to be called upon successful execution
[fail]	`stdErrorCallback`	*(optional)* callback function to be executed upon failure

Example

```
// Function to be executed upon name retrieval
var getNameCallback = function (name) {console.log("Firmware name: " + name)};

// Function to be executed upon failure to retrieve name
var failGettingNameCallback = function (name) {console.log("Couldn't get name :(")};

existingToken.getFirmwareName(getNameCallback, failGettingNameCallback);

// Since it's asyncronous, this will be printed before the result
console.log("This comes first!")
```

baseToken.getFirmwareVersion([win], [fail])

Gets the version of the firmware type of the token

Kind: instance method of `BaseToken`

Param	Type	Description
[win]	`stdStringCallback`	*(optional)* callback function to be called upon successful execution
[fail]	`stdErrorCallback`	*(optional)* callback function to be executed upon failure

baseToken.getFirmwareUUID([win], [fail])

Gets a uniquie ID the firmware of the token

Kind: instance method of `BaseToken`

Param	Type	Description
[win]	`stdStringCallback`	*(optional)* callback function to be called upon successful execution
[fail]	`stdErrorCallback`	*(optional)* callback function to be executed upon failure

baseToken.hasLed([win], [fail])

Checks whether or not the token has simple LED

Kind: instance method of `BaseToken`

Param	Type	Description
[win]	`stdBoolCallback`	*(optional)* callback function to be called upon successful execution
[fail]	`stdErrorCallback`	*(optional)* callback function to be executed upon failure

baseToken.hasLedColor([win], [fail])

Checks whether or not the token has colored LEDs

Kind: instance method of `BaseToken`

Param	Type	Description
[win]	`stdBoolCallback`	*(optional)* callback function to be called upon successful execution
[fail]	`stdErrorCallback`	*(optional)* callback function to be executed upon failure

baseToken.hasVibration([win], [fail])

Checks whether or not the token has vibration

Kind: instance method of `BaseToken`

Param	Type	Description
[win]	`stdBoolCallback`	*(optional)* callback function to be called upon successful execution
[fail]	`stdErrorCallback`	*(optional)* callback function to be executed upon failure

baseToken.hasColorDetection([win], [fail])

Checks whether or not the token has ColorDetection

Kind: instance method of `BaseToken`

Param	Type	Description
[win]	`stdBoolCallback`	*(optional)* callback function to be called upon successful execution

[fail]	`stdErrorCallback`	*(optional)* callback function to be executed upon failure

baseToken.hasLedScreen([win], [fail])

Checks whether or not the token has LedSceen

Kind: instance method of `BaseToken`

Param	Type	Description
[win]	`stdBoolCallback`	*(optional)* callback function to be called upon successful execution
[fail]	`stdErrorCallback`	*(optional)* callback function to be executed upon failure

baseToken.hasRfid([win], [fail])

Checks whether or not the token has RFID reader

Kind: instance method of `BaseToken`

Param	Type	Description
[win]	`stdBoolCallback`	*(optional)* callback function to be called upon successful execution
[fail]	`stdErrorCallback`	*(optional)* callback function to be executed upon failure

baseToken.hasNfc([win], [fail])

Checks whether or not the token has NFC reader

Kind: instance method of `BaseToken`

Param	Type	Description
[win]	`stdBoolCallback`	*(optional)* callback function to be called upon successful execution
[fail]	`stdErrorCallback`	*(optional)* callback function to be executed upon failure

baseToken.hasAccelometer([win], [fail])

Checks whether or not the token has Accelometer

Kind: instance method of `BaseToken`

Param	Type	Description
[win]	`stdBoolCallback`	*(optional)* callback function to be called upon successful execution
[fail]	`stdErrorCallback`	*(optional)* callback function to be executed upon failure

baseToken.hasTemperature([win], [fail])

Checks whether or not the token has temperature measurement

Kind: instance method of `BaseToken`

Param	Type	Description
[win]	`stdBoolCallback`	*(optional)* callback function to be called upon successful execution
[fail]	`stdErrorCallback`	*(optional)* callback function to be executed upon failure

baseToken.ledOn(value, [win], [fail])

Sets color on token

Kind: instance method of `BaseToken`

Param	Type	Description
value	`string` | `Array`	string with color name or array of [red, green, blue] values 0-255
[win]	`stdNoParamCallback`	*(optional)* callback function to be called upon successful execution
[fail]	`stdErrorCallback`	*(optional)* callback function to be executed upon

Example

```
// sets Led to white
existingToken.ledOn([255, 255, 255]);

// sets Led to white (See driver implementation for what colors are supported)
existingToken.ledOn("white");
```

baseToken.ledBlink(value, [win], [fail])

tells token to blink its led

Kind: instance method of `BaseToken`

Param	Type	Description
value	`string` | `Array`	string with color name or array of [red, green, blue] values 0-255
[win]	`stdNoParamCallback`	*(optional)* callback function to be called upon successful execution
[fail]	`stdErrorCallback`	*(optional)* callback function to be executed upon

Example

```
// blinks red
existingToken.ledBlink([255, 0, 0]);

// blinks blue
existingToken.ledBlink("blue");
```

baseToken.ledOff([win], [fail])

Turns LED off

Kind: instance method of `BaseToken`

Param	Type	Description
[win]	`stdNoParamCallback`	*(optional)* callback function to be called upon successful execution
[fail]	`stdErrorCallback`	*(optional)* callback function to be executed upon

baseToken.toString() ⇒ `string`

Representational string of class instance.

Kind: instance method of `BaseToken`

BaseToken.setDefaultDriver(driver) ⇒ `boolean`

Sets a new default driver to handle communication for tokens without specified driver. The driver must have implement

a method *send(win, fail)* in order to discover tokens.

Kind: static method of `BaseToken`
Returns: `boolean` - whether or not driver was successfully set

Param	Type	Description
driver	`Driver`	driver to be used for communication

AnyBoard.Drivers : `Object`

Manager of drivers.

Kind: static property of `AnyBoard`

- .Drivers : `Object`
  - .get(name) ⇒ `Driver` | `undefined`
  - .getCompatibleDriver(type, compatibility) ⇒ `Driver`

Drivers.get(name) ⇒ `Driver` | `undefined`

Returns driver with given name

Kind: static method of `Drivers`
Returns: `Driver` | `undefined` - driver with given name (or undefined if non-existent)

Param	Type	Description
name	`string`	name of driver

Example

```
var discoveryBluetooth = new AnyBoard.Driver({
        name: 'theTardisMachine',
        description: 'bla bla',
        version: '1.0',
        type: ['bluetooth-discovery', 'bluetooth'],
        compatibility: ['tardis', 'pancakes']
    });

// Returns undefined
AnyBoard.Drivers.get("non-existant-driver")

// Returns driver
AnyBoard.Drivers.get("theTardisMachine")
```

Drivers.getCompatibleDriver(type, compatibility) ⇒ `Driver`

Returns first driver of certain type that matches the given compatibility.

Kind: static method of `Drivers`
Returns: `Driver` - compatible driver (or undefined if non-existent)

Param	Type	Description
type	`string`	name of driver
compatibility	`string`	`object`	name of driver

Example

```
var discoveryBluetooth = new AnyBoard.Driver({
```

```
        name: 'theTardisMachine',
        description: 'bla bla',
        version: '1.0',
        type: ['bluetooth-discovery', 'bluetooth'],
        compatibility: ['tardis', {"show": "Doctor Who"}]
    });

// Returns undefined (right type, wrong compatibility)
AnyBoard.Drivers.getCompatibleDriver('bluetooth', 'weirdCompatibility');

// Returns undefined (wrong type, right compatibility)
AnyBoard.Drivers.getCompatibleDriver('HTTP, {"service": "iCanTypeAnyThingHere"});

// Returns discoveryBluetooth driver
AnyBoard.Drivers.getCompatibleDriver('bluetooth', 'tardis');
```

AnyBoard.TokenManager

A token manager. Holds all tokens. Discovers and connects to them.

Kind: static property of `AnyBoard`
Properties

Name	Type	Description
tokens	`object`	dictionary of connect tokens that maps id to object
driver	`Driver`	driver for communication with tokens. Set with setDriver(driver);

TokenManager.setDriver(driver)

Sets a new default driver to handle communication for tokens without specified driver. The driver must have implemented methods *scan(win, fail, timeout) connect(token, win, fail) and disconnect(token, win, fail)*, in order to discover tokens.

Kind: static method of `TokenManager`

Param	Type	Description
driver	`Driver`	driver to be used for communication

TokenManager.scan([win], [fail], [timeout])

Scans for tokens nearby and stores in discoveredTokens property

Kind: static method of `TokenManager`

Param	Type	Description
[win]	`onScanCallback`	*(optional)* function to be executed when devices are found (called for each device found)
[fail]	`stdErrorCallback`	*(optional)* function to be executed upon failure
[timeout]	`number`	*(optional)* amount of milliseconds to scan before stopping. Driver has a default.

Example

```
var onDiscover = function(token) { console.log("I found " + token) };

// Scans for tokens. For every token found, it prints "I found ...")
TokenManager.scan(onDiscover);
```

TokenManager.get(address) ⇒ `BaseToken`

Returns a token handled by this TokenManager

Kind: static method of `TokenManager`
Returns: `BaseToken` - token if handled by this tokenManager, else undefined

Param	Type	Description
address	`string`	identifer of the token found when scanned

AnyBoard.Logger

Static logger object that handles logging. Will log using hyper.log if hyper is present (when using Evothings). Will then log all events, regardless of severity

Kind: static property of `AnyBoard`
Properties

Name	Type	Description
threshold	`number`	*(default: 10)* threshold on whether or not to log an event. Any message with level above or equal threshold will be logged
debugLevel	`number`	*(value: 0)* sets a threshold for when a log should be considered a debug log event.
normalLevel	`number`	*(value: 10)* sets a threshold for when a log should be considered a normal log event.
warningLevel	`number`	*(value: 20)* sets a threshold for when a log should be considered a warning.
errorLevel	`number`	*(value: 30)* sets a threshold for when a log should be considered a fatal error.
loggerObject	`object`	*(default: console)* logging method. Must have implemented .debug(), .log(), .warn() and .error()

Logger.warn(message, [sender])

logs a warning. Ignored if threshold > this.warningLevel (default: 20)

Kind: static method of `Logger`

Param	Type	Description
message	`string`	event to be logged
[sender]	`object`	*(optional)* sender of the message

Logger.error(message, [sender])

logs an error. Will never be ignored.

Kind: static method of `Logger`

Param	Type	Description
message	`string`	event to be logged
[sender]	`object`	*(optional)* sender of the message

Logger.log(message, [sender])

logs a normal event. Ignored if threshold > this.normalLevel (default: 10)

Kind: static method of `Logger`

Param	Type	Description
message	`string`	event to be logged
[sender]	`object`	*(optional)* sender of the message

Logger.debug(message, [sender])

logs debugging information. Ignored if threshold > this.debugLevel (default: 0)

Kind: static method of `Logger`

Param	Type	Description
message	`string`	event to be logged
[sender]	`object`	*(optional)* sender of the message

Logger.setThreshold(severity)

Sets threshold for logging

Kind: static method of `Logger`

Param	Type	Description
severity	`number`	a message has to have before being logged

Example

```
// By default, debug doesn't log
AnyBoard.debug("Hi")  // does not log
```

Example

```
// But you can lower the thresholdlevel
AnyBoard.Logger.setThreshold(AnyBoard.Logger.debugLevel)
AnyBoard.debug("I'm here afterall!")  // logs
```

Example

```
// Or increase it to avoid certain logging
AnyBoard.Logger.setThreshold(AnyBoard.Logger.errorLevel)
AnyBoard.warn("The tardis has arrived!")  // does not log
```

Example

```
// But you can never avoid errors
AnyBoard.Logger.setThreshold(AnyBoard.Logger.errorLevel+1)
AnyBoard.error("The Doctor is dead!!")  // logs
```

AnyBoard.Utils

Utility functions for AnyBoard

Kind: static property of `AnyBoard`

Utils.isEqual(a, b, [aStack], [bStack]) ⇒ `boolean`

Returns whether or not two objects are equal. Works with objects, dictionaries, and arrays as well.

Kind: static method of `Utils`
Returns: `boolean` - whether or not the items were equal

Param	Type	Description
a	`object` | `Array` | `String` | `number` | `boolean`	item to compare
b	`object` | `Array` | `String` | `number` | `boolean`	item to compare against a
[aStack]	`Array`	*(optional)* array of items to further compare
[bStack]	`Array`	*(optional)* array of items to further compare

Example

```
var tardis = {"quality": "awesome"}
var smardis = {"quality": "shabby"}
var drWhoCar = {"quality": "awesome"}

// Returns true
AnyBoard.Utils.isEqual(tardis, drWhoCar)

// Returns false
AnyBoard.Utils.isEqual(tardis, smardis)
```

playDrawCallback : `function`

This type of callback will be called when card is drawn or played

Kind: global typedef

Param	Type	Description
card	`Card`	that is played
player	`Player`	that played the card
[options]	`object`	*(optional)* custom options as extra parameter when AnyBoard.Player.play was called

simpleTriggerCallback : `function`

Type of callback called upon triggering of events

Kind: global typedef

Param	Type	Description

event	`string`	name of event
[options]	`object`	*(optional)* options called with the triggering of that event

stdStringCallback : `function`

Generic callback returning a string param

Kind: global typedef

Param	Type
string	`string`

stdBoolCallback : `function`

Generic callback returning a bool param

Kind: global typedef

Param	Type
boolean	`boolean`

stdNoParamCallback : `function`

Generic callback without params

Kind: global typedef

onScanCallback : `function`

Type of callback called upon detecting a token

Kind: global typedef

Param	Type	Description
token	`BaseToken`	discovered token

stdErrorCallback : `function`

This type of callback will be called upon failure to complete a function

Kind: global typedef

Param	Type
errorMessage	`string`