DUNGEON REALMS,
design and implementation

Óscar Domínguez Corbalán

Facultat Informàtica de Barcelona (FIB)
Universitat Politécnica de Catalunya (UPC)
BarcelonaTech

June, 2013

Director: Miquel Barceló

Department: ESSI

Project submitted in partial fulfillmentof the
requirements for the degree of Computer Engineering.

A mi hermano; nuestro país funcionará el día que a los genios como él se les dé una oportunidad en lugar de excluirlos del sistema educativo. Algo que, por cierto, no parece que vaya a ocurrir.

Al resto de mi familia, que me ha enseñado a forjar mis propias opiniones en vez de, como es uso común, introducir sutilmente sus opiniones, creencias, partidos políticos, equipos de fútbol, etc.

A mis chicas, Águeda y Noa, que llegaron a mitad de la aventura y desde entonces me han traído la alegría necesaria día tras día.

Y, por último, a los grandes amigos que he conocido en el campus: Charlie, Marc, Héctor, Javi, María, Manel, Hilario, Pau, Sergi, Cristian, y muchos otros. Sólo Dios sabe cuántos bares hemos cerrado; si Dios no existe, entonces no lo sabe nadie…

Contents

1 Abstract

The final goal of this thesis is to develop a digital version of the card game Dungeon Realms [1], originally created by the Spaniard game author José Carlos de Diego Guerrero [2]. To achieve it we will pursue an incremental, iterative and agile development cycle that will allow dividing the complete game into smaller milestones that can be taken by a single person relatively quickly.

The mere inception of Dungeon Realms: the Final Career Project comes with (director) client pre-requisites: we need a game that allows for multiple languages –at least English, Spanish and Catalan-, that can be played versus an AI or between two humans through Internet, that such AI has different difficulty levels and that is multi-platform.

The project's first step will be to create an initial design with the major components of the game. As in any iterative process, after each separate development goal the design will need to be reviewed and modified accordingly to pursue the next milestone of the project. As in agile methodologies, every iteration will divide the sub-goal into separate, smaller, tasks. This combination will allow a single developer to produce a steady amount of progress in implementation, documentation and actual thinking.

After the initial design, we'll need to research and compare existing languages and frameworks in which to make the actual implementation of the game: it is important to select a target environment that has good tools and standard libraries to help ease the development –having to program every 40 year old data structure is out of the scope of this thesis. This selection can and will affect such important things as how the server and the interface can or can't be implemented.

After these initial steps we will follow with the major project iterations:

1. The first development goal will be as simple as displaying a hard-coded card on the screen.
2. After that we will create a program that is able to display all the existing cards, previously stored in digital format.
3. Follow building an interactive User Interface that tracks user clicks on the cards.
4. Develop a server that communicates with the UI, at which point initial tests with two human players can start;
5. Implement the game rules, so a game can actually be played between two human players.
6. Create an AI that, using previously developed game rules, can play versus a human player.
7. (Optional) If not out of project time, extra optional iterations will consist in adding one or some of Dungeon Realms' currently existing expansions –so we need a reusable and extensible game with that prevision.

2 History of video game developers

Video game development is the process of creating a video game and its development is undertaken by a game developer, which may range from a single person to a large business [3].

2.1 The first stone, 1940-70

In the 1940s, physicists Thomas T. Goldsmith Jr. and Estle Ray Mann came up with the idea of creating a simple electronic game inspired by World War II radar displays. By connecting a cathode ray tube to an oscilloscope and devising knobs that controlled the angle and trajectory of the light traces displayed on the oscilloscope, they were able to invent a missile game that, when using screen overlays, created the effect of firing missiles at various targets [4]. In 1947 they filled the patent for the invention they described as a "Cathode-ray Tube Amusement Device" was filled and a year later it was accepted [5]. However, due to the high cost, it wasn't released to the market.

The next three decades introduced several novelties by diverse authors such as using a digital display [6], the (pretty well-known) *Spacewar* developed by students at the MIT [6] or the announcement, in 1969, of the first home console by Sanders, a military contractor. It was at that time when Alan Turing developed his idea of a computer chess program.

Most of the progress was done on university mainframe computers **as a hobby** and the fact that hardware was hard to find and compatibility among systems inexistent, means that many of the games were just forgotten.

2.2 Creating a market, 1970-80

It was in the 70s when video games really became an affordable leisure activity thanks to the availability of the first arcade machines that could be found in businesses such as bars or amusement arcades. Anyone could play such as Pong or Spacewar for a coin. At the same time, a first generation of consoles based on the game Pong flooded the market.

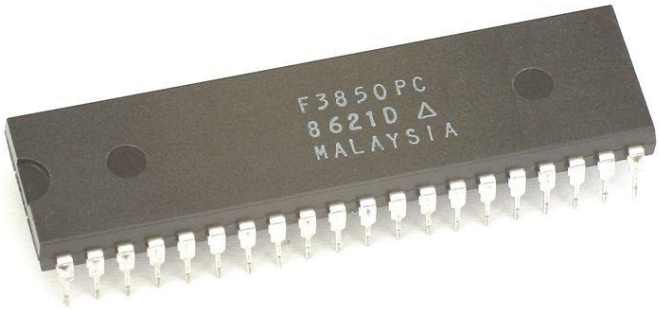
It's also important to note that, during that time, a lot of research on video games was lost because students and researchers of universities illicitly made use of university mainframes to develop their ideas – using such expensive equipment for that wasn't something they wanted to be known.

The fact that both arcades and consoles were based on hardware specifically built for it meant that developers were no longer single or few-man independent teams doing research, but well-sized programming teams directly employed by the big companies that had to develop consoles from scratch that had embedded one or a few games.

2.3 The Golden age, 1980-85

Meanwhile, in the 70s, the company Fairchild was investing in developing the MOSFET technology to create integrated circuits. Due to business management decisions, Fairchild's R&D head, Gordon Moore -the same dude that created the *Moore's law*-, along with other team members, decided to leave the company and create Intel. This spawned a technology race between both companies that would bring a new concept of video game –as well as revolutionizing the world.

Shortly after the Intel 8080 market launch, in 1974 and a year later, the company Fairchild launched a rival processor –the F8- while rival companies did the same –NEC, Motorola… Many developers started rewriting games to port them into the new paradigm of microprocessors that flooded the market and, in 1976, Fairchild launched the first console with a microprocessor, soon followed by rival models. Due to the huge technology leap, 1977 saw the extinction of most of the companies that marketed the first generation of Pong consoles we mentioned earlier.



And then followed what is known as the Golden Age of arcades: thanks to the relative low cost of integrating microprocessors with other chips compared to developing specific hardware, many companies launched video game arcades: Space Invaders, Pac-Man, Asteroids, Tron, Mario Bros… The huge success saw the creation of a lot of dedicated leisure businesses that had lots of arcades where people would go to play. Collaterally, arcades also found a place in the corner of almost any place: groceries, super-markets, bars, restaurants… even some food franchises were known for having an arcade at each premise.

A powerful market had been created: in 1982, the arcade video game industry reached its peak, generating $8, surpassing the annual gross revenue of both pop music ($4 billion) and Hollywood films ($3 billion) together [7].

How did it affect developers? The effect was that they fragmented into studios that created their own games. Developers were no longer the lucky dudes that had both the knowledge and access to the very expensive hardware only big companies could afford.

Separating the game implementation from the hardware, even if not entirely, caused that anyone mastering a microprocessor assembly language could write a game and sell it to a company or even found its own company. The biggest impact to developers was the creation of ROM cartridges, allowing developers to be greatly independent from the console companies.

2.4 Developers grow in number, 1985-1995

Consoles saw another market crash in the early 80s because the market got flooded with poor quality games as lots of new developers tried to enter the business. This caused again the bankrupt of most

companies, but allowed Nintendo to dominate with the NES (launched during 1985, sold 62 million units) and later the SNES (1990, 50 million), with SEGA (50 million adding Mega Drive and Master System sales) as the main competitor and Atari, the third sales-wise (less than 4 million) [8], way behind. The console market had become a battle of two as many smaller companies tried to find a place.

At the same time the home computer sales boomed, what had a huge impact for **developers**: home PCs allowed their owners to program simple games and hobbyist groups soon formed and PC game software followed.

At first many of the games were just clones of the arcades and, soon after, authors started publishing source code on books [9] and magazines. Early game developers would find the computer code for their games -which they had never thought to copyright- published without their names. This meant that amateurs could learn and develop their own games with a very low entry barrier to the knowledge, and the stories of teenagers that did teach themselves to create video games are more than common.

Due to it, the second half of the 80s saw the decline, too, of arcades because developers for the different computer systems were more numerous and they could do it much cheaper –only a computer was needed as investment.

2.5 Oligarchy, 1995-2005

Microsoft's launch of Windows 95 and the DirectX libraries meant that now developers had to access hardware –network, mice, screen…- through an operative system API. However a limiting factor it may seem, the years demonstrated it allowed saving lots of efforts programming low-level features and, at the same time, the hardware dependencieswith graphic cards meant that manufacturers had to build it with DirectX needs in mind or it wouldn't be compatible and games would not work, hence people wouldn't buy it.

As Direct3D made 3D games grow and OpenGL tried to catch up, developers had each time better tools and hardware to create games. The big expectations turned games into millionaire super productions that many times had the next hardware generation as target platform, which allowed for 2 or 3 years production span times –tendency that is still followed. Consoles tried to follow the rhythm, but their big life cycle, compared to computer hardware, meant graphics became obsolete and only Nintendo, Microsoft and Sony survived, with SEGA leaving –to become a game developer and publisher later on.

Companies such as Activision, Microsoft and Electronic Arts bought any and every successful developer studio or good-selling franchise, what meant PC games had also become an oligarchy with independent studios not having the tools to compete or the money to advertise at the same level, what in the end meant selling all the games' rights in exchange of publishing and marketing.

As Windows became, definitely, the mainstream OS, CPU and graphic card manufacturers crashed one after the other as they couldn't maintain the quick increase in requirements that Microsoft standardized through Windows, leaving only two card companies, ATI and nVidia, and two CPU companies, AMD and Intel, competing. Everything in the market, hardware and software, was now in the hands of a few.

2.6 Indie developers fight back, 2005-13

The model of huge companies with world-spread studios proved inefficient due to hundreds of people having to participate on game decisions, what caused a progressive crash of companies -the latest example, as of the writing of this thesis, is THQ, that had got up to $1 billion yearly revenue. The lack of funding for advertising meant that most gaming paperback magazines and many reference web sites –the last being

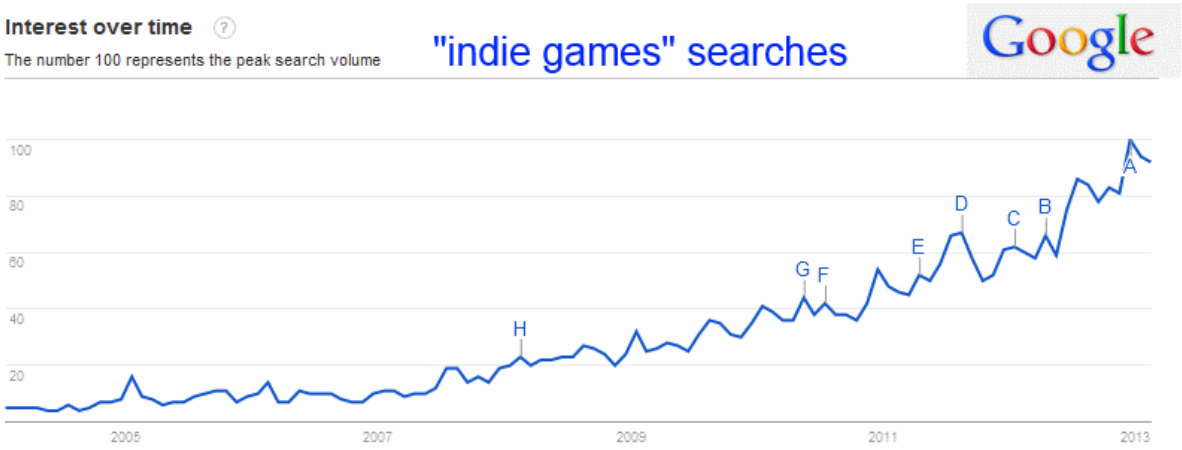
IGN, the #1- progressively crashed with them. The bubble had exploded as every big company tried to survive by selling game franchises, studios and firing developers.

At the same time, powerful tools such as Flash and game engines like Ogre allowed many new, young, developers to learn thanks to the free knowledge in Internet forums and create games from scratch: new blood was here like in the 80s.

As many experienced developers found themselves in the street after being fired from big companies, Internet access prices got smaller more and more people connected to the network. There was no longer the need to compete for expensive advertising in magazine pages since small studios could choose among hundreds of web pages –some of them, like Indie Game Mag or Bit Maiden, specialized on the niche. The result: lots of publisher-independent studios, as in the 80s, were created.

Mobile communication apps, social networks and online game stores, offering game downloads instead of physical disks, added to the turmoil giving the chance to anyone, anywhere, to create a viral ad that would reach millions of clients at virtually no cost.

The terms indie –short form of independent- developers and indie games saw their use increase in time as they found their place even in console online stores. As of 2013, many indie games, developed by small studios, already rival with the creations of productions with hundreds people and millions to back them.



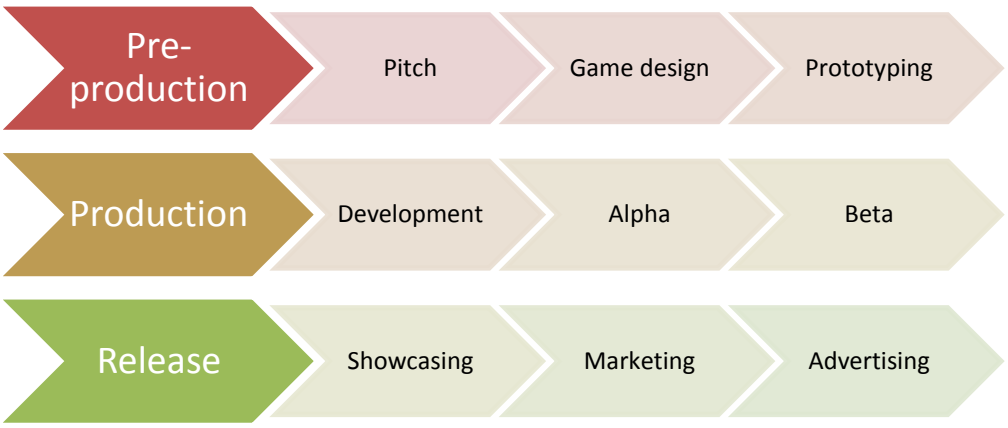
As an example of an indie game, Minecraft, developed by only 3 guys, has sold over 9.5 million copies [10] since its January 2011 release without spending a single dollar in advertisement -word of mouth did the job. The only PC game in history that has sold faster is Activision's –the most profitable video game company in the world with thousands of employees- Diablo 3, with 12 million sales [11] since May 2012. However, Minecraft's mobile and console versions have sold over 10 million copies, making a total of 20 million copies [12]… and we leave out of the equation many more millions that played the free, limited, PC version. So, even selling faster, the indie one definitely sold more copies.

Will indie developers take the market again? We'll see: keep plugged to the present.

3 How is a video game created?

3.1 The process

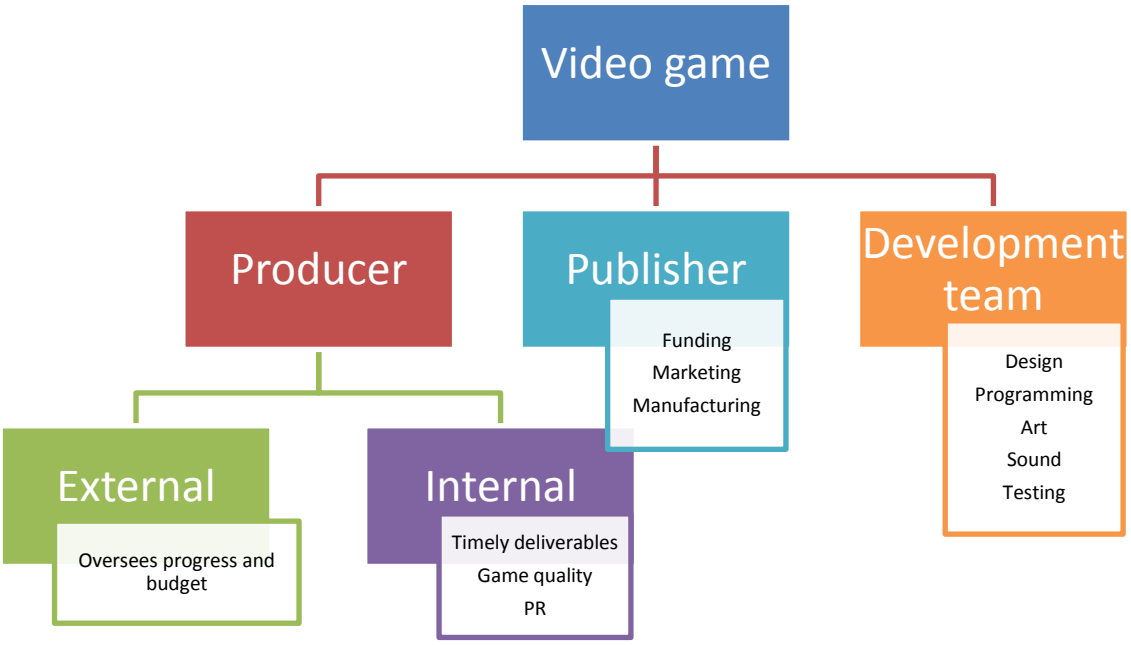
Mainstream games are generally developed in phases. First, game designer usually produces initial game proposal document, that contains the concept, gameplay, feature list, setting and story, target audience, requirements and schedule, staff and budget estimates.



If the idea is approved and the developer receives funding, a full-scale development begins. This usually involves a 20–100 person team of various responsibilities: designers, artists, programmers, testers, etc. The games go through development, alpha, and beta stages until finally being released. Modern games are advertised, marketed, and showcased at trade show demos. Even so, many games do not turn a profit.

3.2 The roles

Development is overseen by (working for the developer) internal and (working for the publisher external producers. The former manages the development team, schedules, reports progress, hires and assigns staff, and so on. The external producer oversees developer progress and budget.

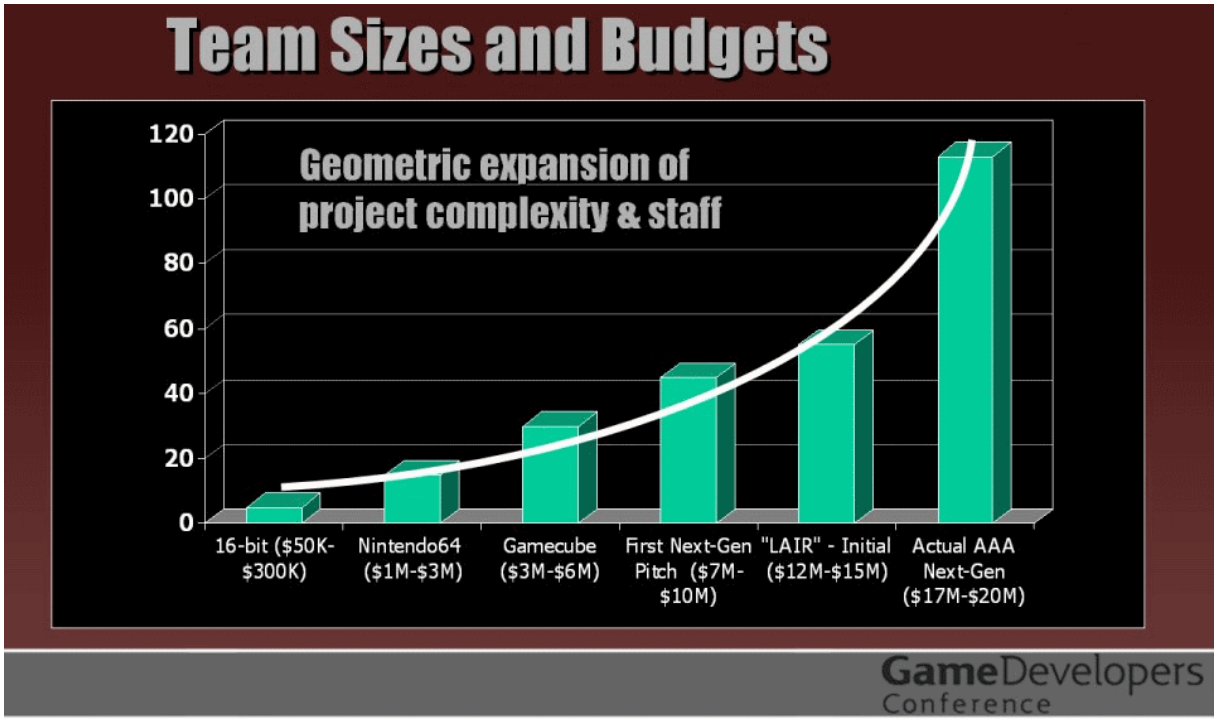


A video game publisher is a company that publishes video games that they have either developed internally or have had developed by a video game developer. They are responsible for product's manufacturing and marketing, including market research and all aspects of advertising.

Developing companies divide their subtasks of game's development. Individual job titles may vary, being artists the most represented, followed by programmers, then designers, and finally, audio specialists.

3.3 The cost

Due to low costs and low capabilities of the first computers, a lone programmer could develop a full game. However, approaching the 21st century, ever-increasing computer processing power and heightened consumer expectations made it impossible for a single developer to produce a mainstream game. The following graph shows the average price of game production slowly rose from $2M in 2000 to over 5M in 2006 to over 20M in 2010.



However, companies' flagship games are way over those quantities. Games like (Bethesda's) Elder Scrolls: Skyrim or (Microsoft's) Halo 4 production costs are around $100M [13]; (Electronic Arts') Star Wars: The Old Republic is around $150M [14].

Those are the costs of production: after investing huge quantities in marketing and administration, the numbers usually double or even triple. Developers of (Ubisoft's) TERA reported the game total costs ascended to $400M. The most expensive Hollywood movies costs are under $300M.

4 Developing a game as a thesis

It's clear after what has been exposed so far that video games are one of the most expensive products to produce mainly due to the wide array of different skills needed, the large number of people involved and the long time needed to produce a single game.

How can a single man develop a game in the reduced time frame (4 to 6 months) that a thesis takes?

4.1 The process

4.1.1 Pre-production

This project's scope is to develop the game Dungeon Realms, created by José Carlos [1]. What implications does it have? He, as a single man, has already done the pre-production step by designing a game and prototyping different versions of the game, which is actually playable with printable paper cards.

Since the moment this thesis exists, it means the producers have accepted José's game design: director Miquel Barceló takes the role of the external producer, agreeing with me, acting as the publisher, to publish a PC game.

4.1.2 Production, the real scope

And, from now on, full game development is taken as the thesis scope: a game engine, as described in the Abstract, has to be programmed from scratch to reproduce a digital version of Dungeon Realms. Interestingly I, the publisher, choose develop in-house, all by myself, the project.

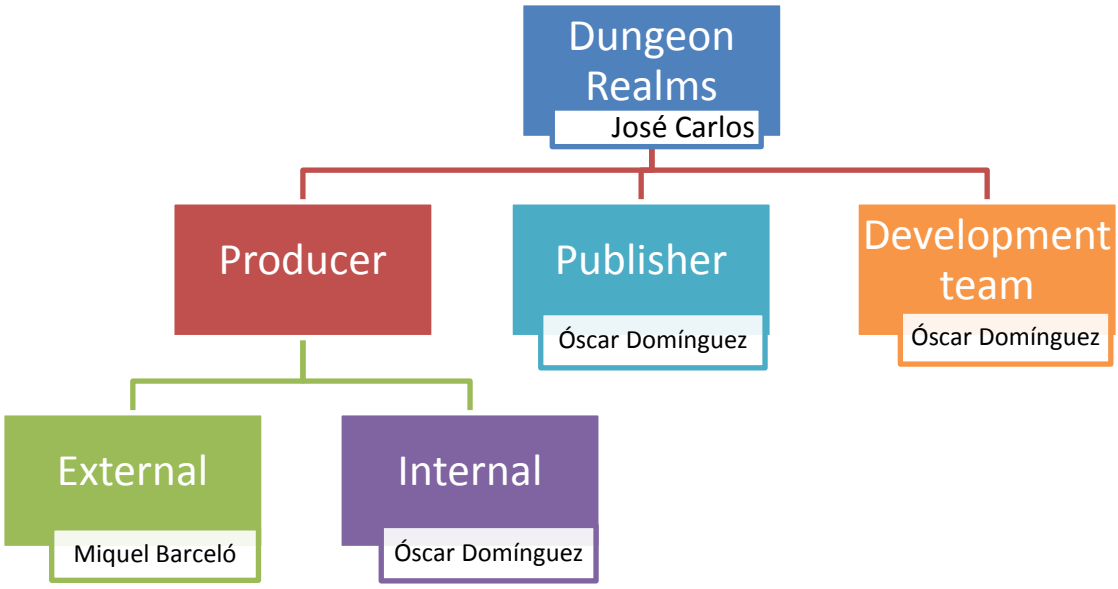
However, it's important to remark this thesis final objective is to obtain a Computing Engineer title. Thus, we can omit art and sound creation as it isn't, in any way, related to the skills needed to be demonstrated. Being picky, we could further argue that, in a card game, sound is the least important aspect and that (in-card) art has already some José's pen drawings that can be utilized.

4.1.3 Release

The game release will be limited to showcasing the master (final) version to the jury that has to value this thesis –as well as any extra people attending. It's the jury who will take, more or less, the role of potential buyers of the final product. Hence, it's me who will take a publisher role by showing the implementation of Dungeon Realms.

4.2 The roles

We can now just tap together everything exposed in the previous point.



5 Dungeon Realms: FCP

To achieve it we will pursue an agile development cycle that will allow dividing the complete game into smaller milestones that can be taken by a single person relatively quickly.

5.1 About agile development

Agile software development, introduced in 2001 [15], is a group of iterative, incremental software development methods with several things in common:

- Adaptive planning
- Time-box iterations
- Flexible iterations

Agile disciplines are mainly thought and suited to be applied by small teams of experts [16], that is a point suited perfectly for this project: the team is small (single developer) and everyone (only me) acts as an expert on everything being developed.

We reproduce here a chart of suitability of different development methods found at Wikipedia:

Methodology	Agile	Plan-driven	Formal methods
Criticality	Low	High	Extreme
Developer XP	Senior	Junior	Senior
Project req.	Change often	Change	Limited
Team size	Small	Large	Variable through time
Culture	Responds to change	Demands order	Extreme quality

Let's defend why we think using an agile approach is best suited to this project:

- **Criticality**: while it is true that the thesis has an inherent criticality because its objective is to obtain an Engineering title –failing means paying around 2.000 € in taxes to try again-, the actual project criticality is low. This kind of game, per se, is not an important system for anything, nor does anything bad happen if it crashed due to a bug –we just patch it. Have in mind that other kind of games, like an online poker where people uses real money is probably considered of an extreme criticality by the producer –what if a player crashes due to a bug in the mid of a game?.
    - **Choice**: agile
- **Experience**: the choice for every other thesis I've read until now has been plan-driven, or defended as such even if developed otherwise, probably because a student, inexperienced like me, feels more secure with a previously thought time plan that meets the thesis deadline. However, it must be argued that I have equal experience with both agile and plan-driven development, so it's a draw.
    - **Choice**: agile or plan-drive
- **Project requirements**: this implementation of Dungeon Realms is limited by the version 1 rules; however, we don't know how many hours we will take to implement it. That means we can either fall into cutting requirements (e.g. card images) or adding new ones, like game expansions, as is considered from start. Hence, the requirements are considered to be going to

change, even if not for the main developing of the game, and we don't know either when, how much, or in which sense.
    - **Choice**: agile or plan-driven
- **Team Size**: one single lonely dude.
    - **Choice**: agile
- **Culture**: this could also be called personal bias. A very ordered, non-proactive person is never going to feel comfortable using methods like Scrum where he can' follow an agenda with detailed steps because half the steps may change every week; neither is a team looking for the ultra-awesome definitive game formed by disperse-minded people –those actually exists and may take up to 10 years developing a game, even totally discarding full-fledged implemented games and engines for not meeting the extreme quality.
    - **Choice**: my culture? Respond to change; hence, agile.

We meet every point for an agile project development, 2/5 for the usual plan-driven and none for formal, documentation-heavy methodologies.

5.1.1 Agile guidelines

But! There's always some big but behind the corner. Agile methodologies are actually minded for small teams, not single developers, hence practices like "pair" programming are hardly possible to implement by "one" man that does not have any multiple personality disorder. How do we take into it? The answer is to drop specific best practices that cannot be followed and use general agile practices instead:

- When implementing every functionality *do the simplest thing that could possibly work* [17]. Note that simple != easy.
- Implement everything in a continuous integration. There's no use for a single programmer to develop in parallel separate big subsystems to later integrate them.
- Test every new functionality added. Bugs in code should be found as sound as possible.
- Work in small bites, optimally the code should be never left in a state that does not compile. Big chunks of work should be divided into smaller tasks.
- Identify tasks, estimates and track the performance against estimates. This will help identify things that slow down the progress, time-wasters, etc.
- Since working solo in this scenario means the developer is acting as business and developer at the same time, there's no sense in separate and translating user stories from/to engineering tasks, hence we'll just work with more general backlogs items.

5.2 Project's iterative methodology

We will follow an iterative development cycle with marked steps. In general terms, an iteration objective will mark a backlog with separated things that need to be done to reach it that, finally, will the base for concrete tasks. This will allow us to don't overdesign unneeded elements or to program in provision of the future. We follow explaining the steps.

5.2.1 Iteration objective

This will contain the final objective that must be met to consider the iteration done. A summary of the major steps has already been stated on the abstract, but each iteration will contain a more elaborated explanation taking into account the current development state.

5.2.2 Iteration backlog

Taking off from the objective, in the backlog we'll list the tasks –research, documenting, programming…- that need to be done to reach it. As we previously explained in our guidelines we'll limit more or less to general entries, since we will already provide detailed engineering tasks on the effort chart.

Backlog entries are numbered following the format *Bin*, where:

- *B* stands for Backlog
- *i* is the iteration number
- *n* the number assigned in that iteration.

5.2.3 Effort chart

We'll provide a chart with every task done and the time actually needed to do it as well as the time that had been expected. This will help to take self-awareness of where the time is gone and analyze it to prevent further time wasting or bad forecasting. We'll also try to summarize the reason of the major time divergences.

Effort chart entries are numbered B*in.*T*m*, where:

- B*in* is the backlog concept that spawns task T, and,
- *m* is the number assigned to the task related to other work done for Bin, such that usually- but not always- T*(m-1)* must be done before T*m*

5.2.4 Documentation

Most if not all of the tasks in the effort charts should be backed up by the documentation that follows. We'll include from UML diagrams to defending implementation choices and, as such, every iteration will contain different items in separate points.

6 Iteration 0

6.1 Objective

The iteration ends when the project is in a state where development can start.

6.2 Backlog

B01 - Agree the project with Miquel and register the thesis.

B02 - Research about videogame development.

B03 - Learn to play Dungeon Realms.

B04 – Plan a work methodology for the project.

6.3 Effort chart

This iteration has taken quite longer than expected. The main reason is the underestimation of the huge amount of data found on the research about videogames development, that to be read, filtered and then resumed into a few pages. There were also a couple problems with the project inscription that made not possible to officially register it on due date and took some calls and time to solve.

Ref	Concept	Estimate	Real
B01.T1	Project agreement and bureaucracy	2	4
B01.T2	Write up a project motivation	3	3
B02.T1	Research video game development	5	10
B02.T2	Research video game developer history	5	7
B03.T1	Learn to play Dungeon Realms	5	8
B04.T1	Choose a methodology	3	3
	Total	23	36

This iteration documentation ends here: if you've read the previous sections, then you have already assimilated the content spawned from this initial iteration.

7 Iteration 1

7.1 Objective

Startup familiarization iteration with the mere objective of displaying a single card with hard-coded fields. Implies the selection a development environment.

7.2 Backlog

B11 – Choose an implementation language.

B12 – Display a card

7.3 Effort chart

Pretty straightforward iteration. It was decided to try building a prototype with each of the two best-suited languages. After a lot of time put into the Java prototype we decided to drop it because easy things were too hard to do; with the AS3 prototype, it was very quick.

Ref	Concept	Estimate	Real
B11.T1	Research options and choose the best programming language	5	6
B11.T2	Research IDEs for the selected PL	2	2
B12.T1	Discarded attempt Java card-stub plus program to display it	2	10
B12.T2	Program a stub card object using AS3/Flash	1	2
B12.T3	Create a simple program that displays a card object using AS3/Flash	1	1
	Total	11	21

7.4 Choosing an implementation language

The programming language we choose will accompany us during the rest of the project, and it will affect how easy and hard is to accomplish different parts of the development. We follow with an explanation of our requirements and choice.

7.4.1 Requirements

ID	Requirement	Why?	Fail to meet
Req-Imp-1	Portability	Design choice	The game does not work on a major OS
Req- Imp -2	Familiarity	Reduce time wasting	Slower implementation. More bugs.
Req- Imp -3	Network APIs	Play over the Internet	Network functionalities need to be developed.

7.4.2 Options

Language	Req- Imp -1	Req- Imp -2	Req- Imp -3
Flash (AS3.0)	Yes	Yes	Yes
C++	No	Yes	Yes
C#	Half	Yes	Yes
Java	Yes	Yes	Yes
HTML 5	Yes (on paper)	No	Yes

7.4.2.1 Flash

Adobe **Flash** is a powerful tool that uses ActionScript 3.0 and a multimedia and software platform used for authoring of vector graphics, animation, games and Rich Internet Applications (RIAs) which can be viewed, played and executed in Adobe Flash Player. The Player is a virtual machine that compiles ActionScript at run-time and has an implementation for every major OS.

The main shortcoming for this option is that the network possibilities are not as developed as options like Java or C#, what summed with the low efficiency of a virtual machine usually ends up with developers building servers in another language different than ActionScript.

7.4.2.2 C++

The C with APIs. While it's the language that has more options on what can be done –at the cost of having less high level helpers-, we don't want to have to compile different binaries for every platform having the risk of encountering incompatible calls or libraries. Discarded

7.4.2.3 C#

More and more game developers use it every day because it's simple to code and Microsoft's Visual Studio even allows using other languages, as C++, on any modules. Binaries are a semicompiled language that a virtual machine translates into processor-specific instructions. However, it's only assured to work seamlessly on Windows because the VMs for Linux and Mac are community-built.

7.4.2.4 HTML5

The portability is (internet) browser-dependent, so if it works on Chrome or Firefox then it works for any OS, or so they tell: the fact is that HTML5 is still not a stable standard and most things, specially interaction-related, require a lot of conditional clauses to change code snippets depending on the browser and OS where it's being executed. HTML5 provides ways to establish real-time connections between client and server, although it still isn't a standard (things change) and I wouldn't know where to start. Plus, for the moment, it's not really thought to implement games –even if there are developers researching it.

7.4.2.5 Java

Java provides full portability like Flash thanks to a virtual machine present for almost anything –have you ever heard about Java toasters?. It's a language with APIs for almost anything and the best option together with Flash.

7.4.2.6 Veredict

We'll use Flash and ActionScript 3 to develop Dungeon Realms. It offers almost anything that Java does but will reduce the overhead of programming interactivity on each UI object.

7.5 AS3/Flash roundup

Why choose Flash? Because:

- The install base is massive, even compared to HTML5. Virtually every computer with any graphic OS supports and has Flash installed, as well as any modern console and even current smartphones. It's simply impossible to target a wider audience.
- It's a fixed platform, we don't need to worry about different browsers, or different hardware, or different OS, or different anything. The code runs on a virtual machine that also acts as sandbox –we can't break the computer with dangerous code!
- There are very mature development environments available which support full debugging via the world class Visual Studio.
- If wanted, it's possible to distribute the game at no cost: internet is full of Flash portals that are an amazing resource for distributing Flash games.
- It has support for TCP sockets built right in –we can get Multiplayer working.
- There's a very wide developer base and a lot of API code.

7.6 Choosing an IDE

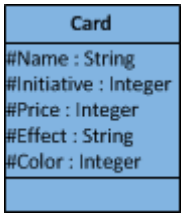
There are few options to consider here, mainly official Adobe Flash Professional and FlashDevelop.

- FlashDevelop: lacks an editor for graphical drawing and testing, although packs a more potent programming text editor. Requires building all graphic assets with another tool and embed them with binary-treatment code.
- Adobe Flash Professional: integrates tools to draw and create interactivity to reduce manual repetitive coding. The text editor has less helpers compared to FlashDevelop –e.g. imports are not added automatically.

Since the software to develop is a video game focused on interaction with the UI, we'll use Adobe Flash Professional, version CS5.5, which corresponds to the one released on 2011.

7.7 Design

Let's hope the (UML) design part keeps as easy as this for the rest of the project:

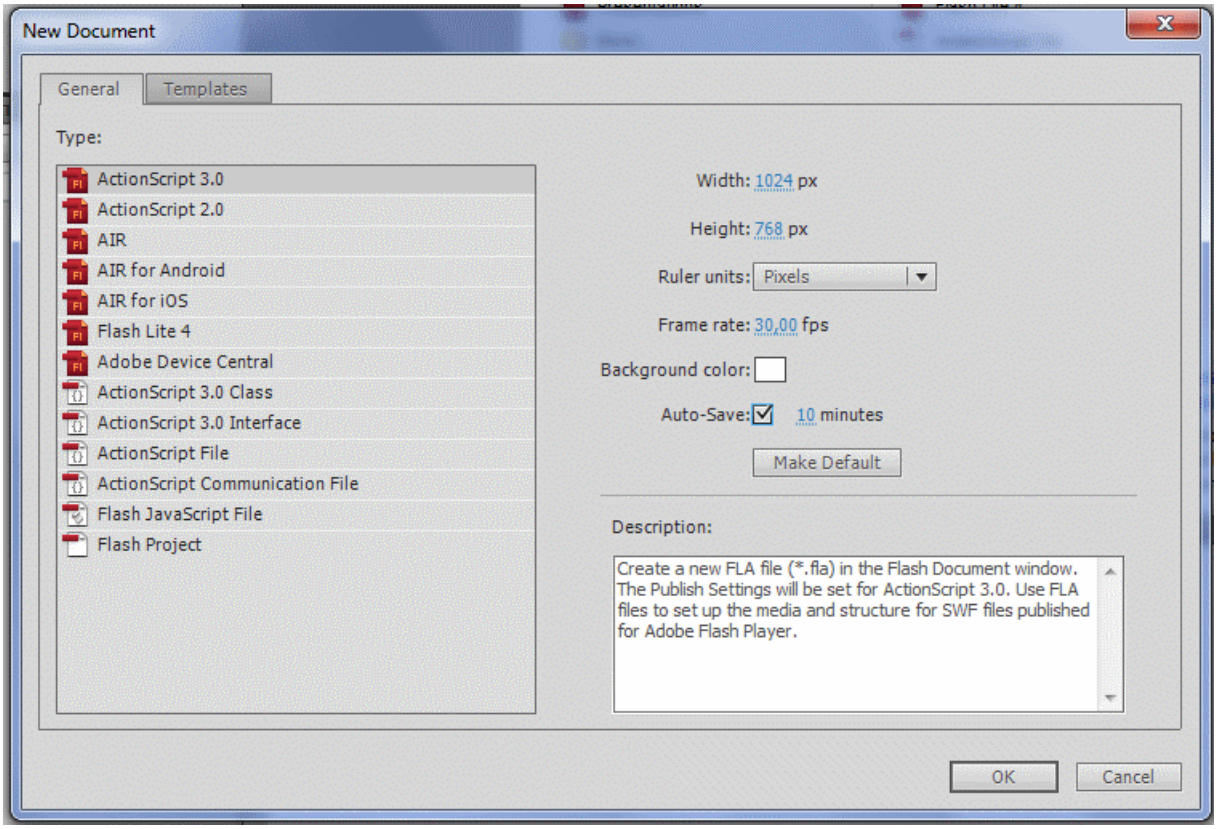


7.8 "Hello Card!"

This is a small how-to for Flash/AS3 neophytes: how to create objects that can be displayed on the screen?

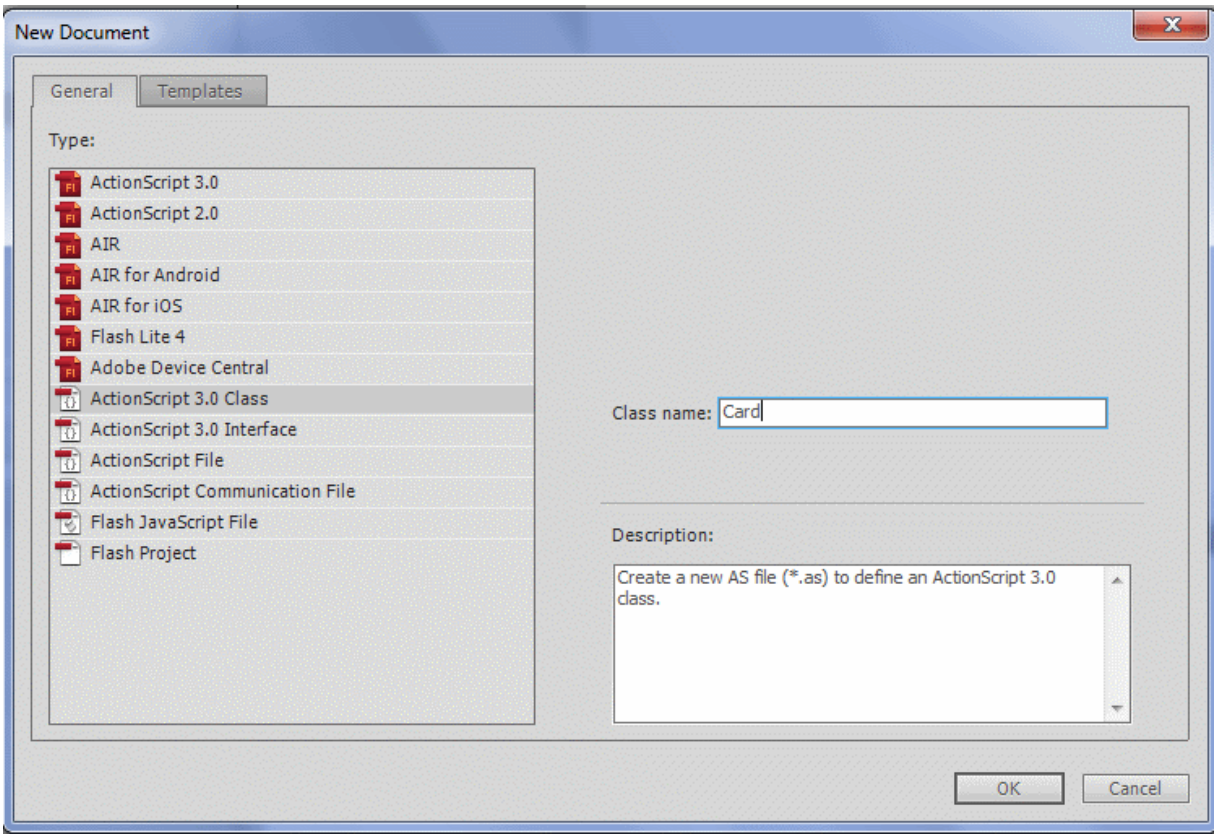
7.8.1 New project

We create a new ActionScript 3 project and setup a default resolution –can be changed later- and autosaving.



7.8.2 Create a Card object

Since this is the first object, we show how it is created. We have to choose to create an ActionScript 3.0 Class and we name it Card.



Then, in the text editor we write (we don't include here imports and variable declarations):

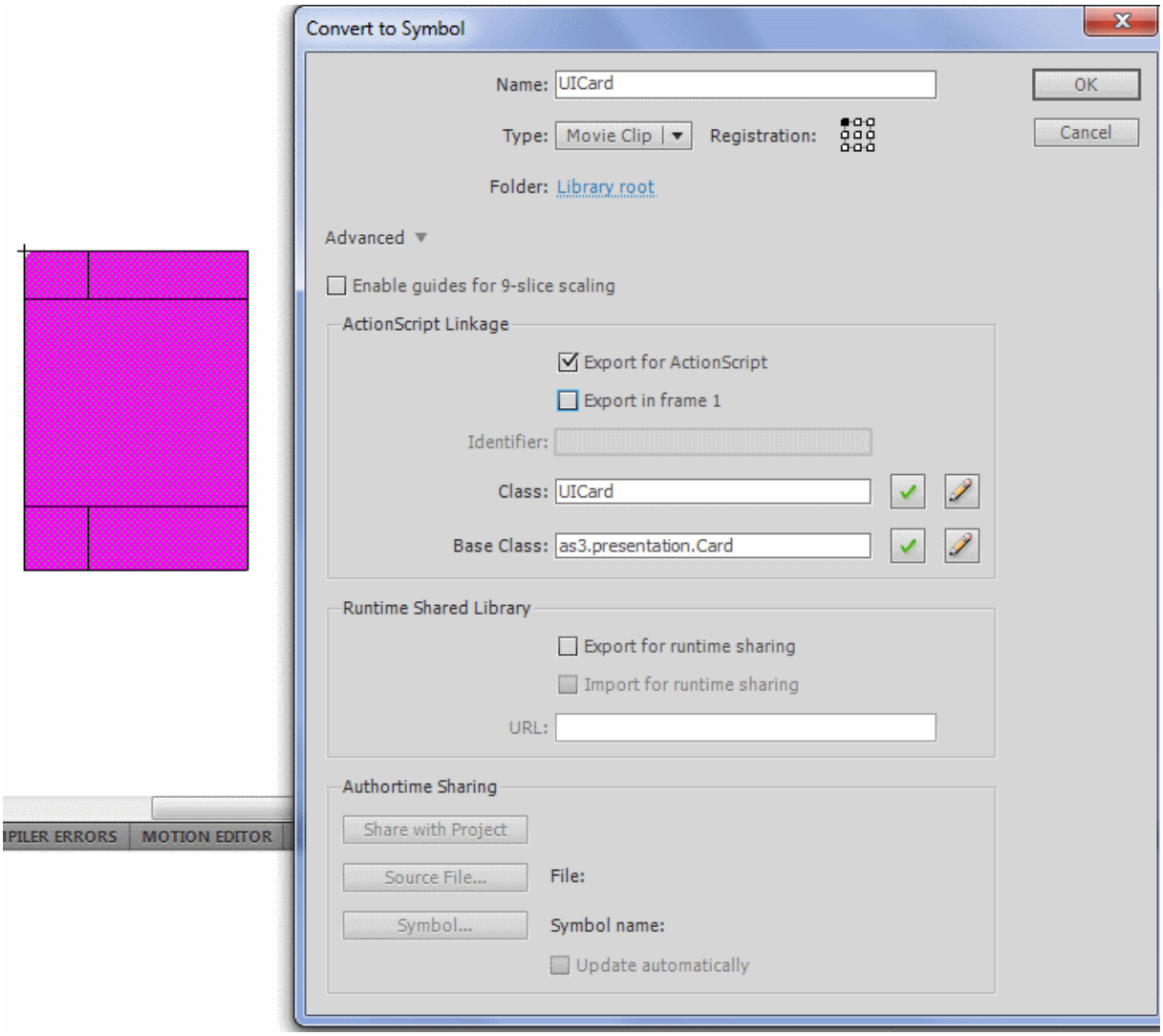
```
public class Card extends MovieClip {
        public function init(_id:int,_nm:String,_price:uint,xx:int,yy:int,_color:uint){
                //set background color
                graphics.beginFill(_color);
                graphics.drawRect( 0 , 0 , _CardWidth, _CardHeight );
                id = _id;
                nm = _nm;
                color = _color;
                price = _price;
                x = xx;
                y = yy;
                //add texts with help of an external helper function we've written; see Appendix
                initTextField(tfName,nm,          //textField and text
                              45,5,               //coordinates
                              90,20,              //size
                              tFormatSmall);      //format

                initTextField(tfID, id.toString(),0,2,40,26,tFormatBig);
                initTextField(tfPrice, price.toString(),0,168,40,24,tFormatBig);
                initTextField(tfEffect,"4 Gold",45,172,90,24,tFormatSmall);
                }
        }
}
```

We leave an empty class creator function because, in ActionScript 3, many behind-the-scenes instance linkings are done at the end of the creator and we cannot access methods from classes such as SimpleButton or MovieClip until the creator is finished. Note that we could actually add most of everything done until now there, but it's also useful to do like us as a programming pattern because we can then reuse

already instanced objects by just calling the init function –this AS3 pattern/good-practice is done by all experienced developers because instancing new objects in the VM is a very, very costly operation.

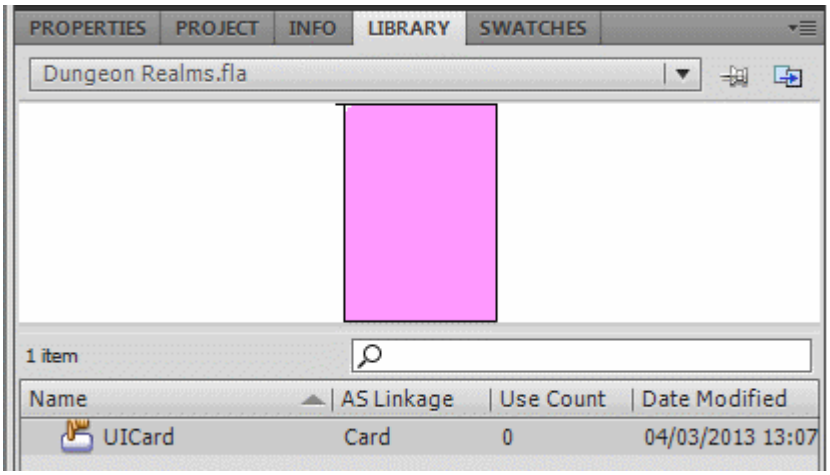
7.8.3 Add the card's graphic

Then we draw a rectangle on the screen with inner boxes for the card's content, right-click and choose Convert to Symbol. We can choose among three classes: Graphic, Button, MovieClip. We choose the last because we'll want it to display things inside (graphic, text) and it's the class meant for that.



We write our previously written ActionScript class as the Base Class. It means that every instance of Card will have this drawing as a background layer.

After pressing accept the new asset is added to the Library.



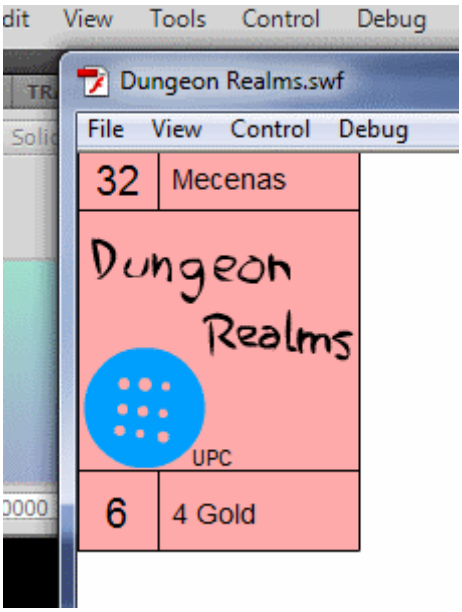
7.8.4 Display it

In a similar way, we open the project's properties window and write its class as as3.Launcher. Then we create a new ActionScript 3 file that contains such Launcher class:

```
public class Launcher extends MovieClip{
        public function Launcher() {
                trace("hello");
                var c:Card = new Card();
                c.init(32,"Mecenas", 6, 0, 0, 0xFFaaaa);
                addChild(c);
        }
```

When the virtual machine launches our executable file, it starts the main MovieClip and, if containing a base class, it is instanced through its creator function. In the code can be seen how we just instance a Card, initialize it with some parameters, and add it to the MovieClip's display tree.

After some drawing messing to make it cooler we obtain this:



8 Iteration 2

8.1 Objective

Display a prototype interface that can display town cards, the hand of the players and the decks.

8.2 Backlog

B21 - Design a way to store and retrieve cards

B22 - Implement a way to load the previously stored cards.

B23 - Create an interface that displays the loaded cards.

B24 - Implement card shuffling.

8.3 Effort Chart

With iteration 2 we've improved our estimation accuracy. The main drawback has been testing different XML approaches, what has required introducing data in various XML schemes to get a final approach.

We also found the AS3 API does not provide to get seed-based random numbers, so we had to find and implement an algorithm for that purpose.

Ref	Concept	Estimate	Real
B21.T1	Research options	5	5
B21.T2	Define card XML scheme	15	15
B21.T3	Design card effects' syntax representation	15	15
B22.T1	Create card loader class (Deck)	4	4
B22.T2	Manually introduce card data	5	8
B22.T3	Process effect data to show it graphically into the game card's	5	4
B22.T4	Draw graphic symbols for every effect	7	8
B23.T1	Design a UI	5	8
B23.T2	Implement the designed UI	20	22
B23.T3	Implement a way to load card ambient images	1	1
B23.T4	Process images for all cards and introduce routes in the XML file	3	3
B24.T1	Implement an algorithm that can provide seed-based pseudorandom numbers.	0	2
	Total	85	95

8.4 Defining how to store cards

8.4.1 Representing data in the era of information

Dungeon Realms: the Final Career Project, is developed in 2013; this means we are on the era of information. But, contrary to popular belief, information is not only the awesome possibility of reading a newspaper with a 600€ gadget and a 50€ monthly fee. The era of information is the era of engineering around information: nowadays, there are countless ways, standard or not, of storing and retrieving data.

This is good because the engineer of the 21st century does not -usually or if we don't want to- need to implement complex ways to store and retrieve data. It would even be possible for us, without too much extra hassle, to have a dedicated server send card information on demand, while playing a game, in whatever way we'd liked to, be it data-wise or just already rendered cards. It could even be served through a Content Delivery Network, such as Akamai or Amazon's Cloudfront, with a couple clicks and a few code changes in the program so we could scale the game to tens of thousands of concurrent players.

As the yet-not-sleeping reader is thinking right now, implementing these example options may be just too much for this thesis. Too much probably not in the meaning of extra effort required, but in what the project actually requires. And this is what we wanted to express with this introduction: in 2013 there are so many ways to store and retrieve data, that the problem for an engineer is not to implement a storing system, but choosing what's the best for a project among a million possibilities, all of which promise diverse things, that might or might not be for real.

8.4.2 Card data to represent

The data the Dungeon Realms software will need to actually work are the cards. Without its information there's no game: the player couldn't see the cards, the AI couldn't understand something that doesn't exist, and the rules couldn't be implemented. We could even state that it is the single most important factor in the game because, no matter what other choices we make (whether to program an AI, playable over Internet or not, etc), this one will always be present in every single implementation possible.

8.4.2.1 What IS a card?

A card, in any card game, is a real-life object that contains the most important information for the player: what effect does it have in the game according to the rules.

These last 5 words are very important and we'll demonstrate it with an example: a poker deck can be used to play a lot of different games. While it is a truth that, in most rules that apply, the higher numbers beat the lower ones and that the A owns the King and so on, it is not so by the cards. If such a game with poker cards where the lower numbers beat the higher ones did not exist, we could invent one in a few seconds –i.e. regular poker just changing that.

Hence, it is important to understand that the cards do not define the game or the rules, because if it were so we could just implement rules coding into the cards representation, thus having some kind of a distributed rule implementation. However fun it could be for someone –not me- to divagate into all that crazy coding, it would be neither practical nor an implementation of something that reflects the real world.

8.4.2.2 Project requirements

This project will need to have the information of each card available for it to function.

1. The implemented rules code will need to be able to retrieve some of the information on any card to know which movements are possible and the status of the game at any given moment.
2. The AI of the game will also need information on each individual card to decide and value every possible play possibility.
3. The UI of the game will need information to display it to the human player.

At this design point, 1 and 2 are the same: card type, initiative, buy cost and effect(s); and a unique id will also be required because there are cards identical except by their initiative. Point 3 is a superset of them that adds the card name and an illustration, with the exception that the information is needed for the user, not for a data processing algorithm; i.e. painting a scanned image would do the job for the user.

The conclusion here is: we can either use two representations or one. The former option means to use a data-oriented, for the game engine, and a rendering-oriented, for the UI, separate representations; the second option implies using a system or hybrid that is good enough for both goals.

ID	Requirement	Why?	Fail to meet
Req-Data-1	Fast queries	The engine, and especially the AI, needs to be able to make lots of data requests.	Moves become very slow to evaluate. Bad user experience.
Req-Data-2	Easy queries	Less code means better code.	Invest time to code retrieving data. Invest time debugging code.
Req-Data-3	Maintainable	It's a lot of data that needs manual entry, so errors or changes are expected.	Making a change is slow and expensive.
Req-Data-4	Easy extensibility	The game has expansions coming; a prevision for adding them means it's possible to add cards or extra information.	Adding a card is slow or a mess. The system may need to be changed.
Req-Data-5	Easy renderization	Every game action requires changes in the screen.	Painting cards is slow. Painting cards requires a lot of code.
Req-Data-6	Easy documentation	Extending the game may be needed in the future.	Barrier to extend the game. Prior training or experience needed.

	Req-Data-1	Req-Data-2	Req-Data-3	Req-Data-4	Req-Data-5	Req-Data-6
Plain Text	Half	No	No	Half	No	No
Database	Half	No	Depends	Depends	Depends	No
Hard-coded	Yes	Yes	No	No	Half	Yes
Images	Queries are hard or impossible		No	No	Yes	Yes
XML	Half	Yes	Yes	Yes	No	Yes
Hybrid	Yes	Yes	Yes	Yes	Yes	Yes

8.4.2.2.1 Plain Text

If memory footprint was the most important requirement –and it's not even a bit important for us-, then the best solution would be embedding a file –that is the minimal representation- into the executable: hence the reason it complies with first requirement. We could even study the balance between how much file compression loses how many processor cycles. It could also be extended as we wish, even making internal pointers or references to rows or columns –would stop o be a minimal representation- although with some complexity –half requirement 4-.

Hence, using plain text in the era of information is the worse option: lot of code needed to retrieve it for processing and for rendering, everything needs documenting and you can turn nuts looking for a field that needs a change.

8.4.2.2.2 Database

Using a Database, such as MySQL, means extending and maintaining the system is easy… as long as you know how that particular database system works. That's a barrier to modify the system for anyone that's not the original developer. With the particular knowledge, it's easy to add tables and update fields without much hassle. By the way it needs a local -for each client) database engine to run queries fast: waiting 100ms for each single query on a remote server is not an option.

Renderization can be easy if we store card images directly in the DB and request them with queries, which would need either that the database supports it or some hacking such as outputting the images as text in game code and reconstructing them to render. And, never forget, coding is a bad business.

Documenting a database is hard even if explaining single fields is easy –in most database fields even have a name related to its content-. Anyone contrary to this assertion should just try to see a chart with some dozens of tables or, worse, many charts with some tables each, and try to relate them just by looking. And arrows do not help.

8.4.2.2.3 Hard-coded

Geeks argue hard-coding data is the way to go. The geekest option here is to hard-code some dozens of cards as a single object each, and documenting their abstract class would be easy and enough for other people to understand it. However, it means a lot of time to construct it even if it allows for the fastest queries –e.g. numbers turns constants-.

It's also a loss of the power of object-oriented programming languages to code the data of many objects that are, by definition, the same.

8.4.2.2.4 Images

Even if we could use an image-recognition text-processing algorithm for retrieving the data of the card, its complexity is out of scope specially when we enter the marvelous world of debugging why it doesn't output the number present on the card. Concluding, using images for anything else than the UI is a horrible decision. Even for the UI, a problem spawns: we need to preprocess –manually and programmatically- every card so we have them available for the game, so the actual developing gets delayed until all of the art/card assets are ready –using stubs would mean constructing stubs images, hence throwing time/money to trash.

For the UI, rendering the card is as easy as telling an API to paint it, and that call would even come already documented. Changing a card would be as easy as changing the image, but adding, for example, a field to every card means pre-processing every card again, which should be an avoidable work.

8.4.2.2.5 Extensible Markup Language (XML)

Unless the reader has been living at the seabed –Sponge Bob…-, or is a computer pagan, he or she should know XML is a standard markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. XML is used from Microsoft Office document-storing to communication protocols because it is easy and parsers are available on virtually any programming language.

With XML documenting is easy because we name fields like in a database or a programming-language object, retrieving the data is trivial because we just choose an API –already debugged- that does it for us, and we could even embed the file in memory like the plain text example so queries turn as fastest as possible.

8.4.2.2.6 Hybrid

The best option so far is XML: let's use it. But, what are its drawbacks and how can we avoid them? If we take ideas from the other options we can construct a good hybrid.

- Req-Data-5: renderization is hard.
    - Solution: we reference images from the XML file.
- Req-Data-1: query optimization.
    - Solution: at program start, we instance an object for each card.
- Using images adds a problem of using images: we need to process images for them to be available for the UI.
    - Solution: use images only for the card internal drawing. Render the text and numbers from the XML info.

This last point will also allow for the project development to be independent from art assets: it is possible to construct first a 100% working software with just the central inner images missing. To state with a parallelism, it's the same that constructing a 3D engine while an artist constructs textures for the environment and characters: the engine is just tested with blank 3D models meanwhile.

8.4.2.3 Card information to store

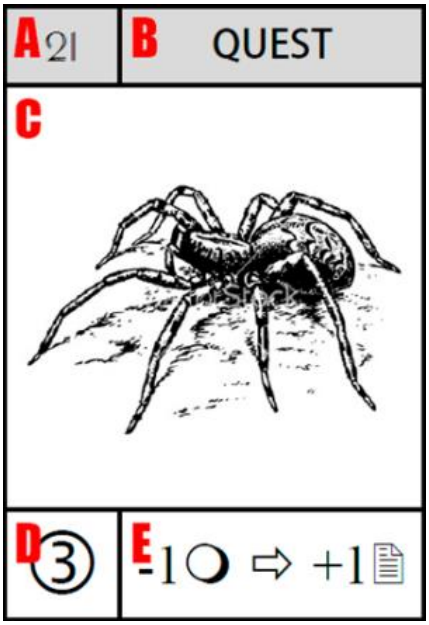
In Dungeon Realms, the data each card represents is varied in nature and meaning: while some of it is a word, other is numbers. Each of these numbers has different grammatical or rule significance: initiative of the card for action order, buy cost to acquire the card and, last, the effect it has. This effect is not only the single one that is not representable through only a single number or word, but has a syntax that allows for some deep meanings to the game. Let's sketch it:

A) Card initiative = number

B) Card name = word

C) Card image = icon

D) Card cost = number



Observations:

- B and C are only needed for the user interface.
- A and D are trivial to represent.
- E needs some study to be represented as XML.
- Town places do not have A, but an icon.

8.4.2.4 Card metainformation

Each card does also have metainformation in diverse ways: it has a background color and a type. These two have some degree of correlation between them and also with the card initiative and name. For example:

- Cards with initiative in range [37, 45] are yellow and are of type quest of level 3. And yellow cards are always quests of level 3 -absolute or bidirectional correlation-. However:
    - Cards in that range may or may not have or include the name Quest. 5 of 9 are called Quest but the other 4 do not even include it.
    - Many cards named Quest are of another color and type –i.e. quest of level 2.
- Cards in range [10, 18] are always of type player card, but can be either blue or red depending on the owner player.

8.4.2.5 To derive or not to derive: representing data correlation.

And here comes another design choice of the project. How and why should we represent these (some of them fully) correlated fields? Let's look at the options:

Design choice	Benefit(s)	Loss(es)
Represent derived data	With XML a kid can make it. All data is instantly available for the engine. Data can be easily maintained. No recompiling. Game-expansion friendliest.	Introducing data for the first time requires extra time. Changing or maintaining data means looking correlated fields.
Calculate derived data	Save time by not introducing redundancies. If sending cards over Internet: less bandwidth usage.	Coding fuzzy correlations is not easy. Changing data needs coding knowledge. Loss engine time calculating data. Adding an expansion may mean sending a code to trash.
Represent some **Derive some**	Don't code hard correlations. Don't write everything.	Introducing data for the first time requires extra time. Selective code for each data. Changing data needs coding knowledge. Loss engine time calculating data. Adding an expansion may mean sending a code to trash.

The conclusions here are:

- **Hybrid option**: it is luring because it is the quickest solution at first. However, having two ways to consult data means having two places to look at when changes have to be made and problems from the two ways.
- **Calculate**: it provides the smallest data entry in exchange of programming rules to derivate data. The problem is that any correlation that is not bidirectional translates into non-trivial code, and most of the data falls into it. Expansions can also mean to recode or tweak all the code, which is unacceptable.
- **Represent**: it's slow and boring to introduce all the data at first, but at least it's really easy once we have a design –if it was orders of magnitude bigger it could just be subcontracted to nonqualified workers. Also, with XML, we can just modify existing information or add extra nodes wherever if an expansion requires to add data.

We could induce a corollary from Okham's *lex parsimoniae* to defend the position that "the simplest implementation tends to be the correct one". However, I honestly think that any engineer should reach the FCP with that knowledge learnt by experience: if your mother can't understand it, then you're doing it too

complicated. Most people could understand how the XML representation works with a little explanation and even introduce the data for me –and not every engineer would be able to calculate all the fuzzy correlations straight away.

Conclusion? The winner is to represent all data. The only untied rope that remains is: player's cards are exactly the same, but color –red or blue- depends on assignment at the start of the game. We choose to not code its colors in the XML, but to assign it at game start.

8.4.3 A card XML

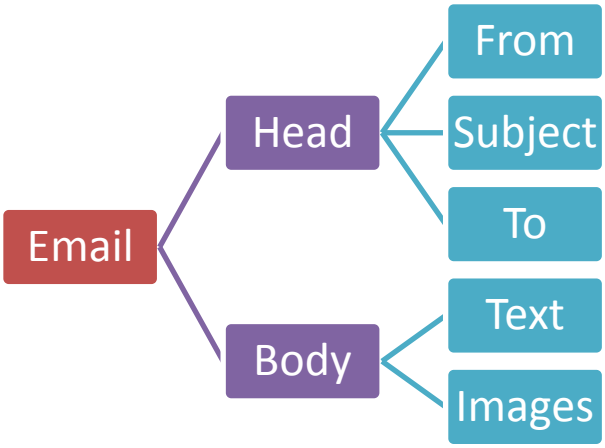
8.4.3.1 A short introduction to XML

After all this babble, any reader unfriendly with it is probably at the point of dropping this document unless he/she starts understanding something: what is XML?

- XML stands for eXtensible Markup Language, is designed to transport and store data and is important to know and very easy to learn. [18]

Everyone in the contemporary age should –but not everyone does- know how to send an email. What is an email? When we write an email, we have to fill the mail header and the body. In the header we make sure it has our reply address, a destination and a , while in the body we can introduce text and attach files.

In XML we can imagine data as a hierarchic tree: how would be an email tree?



XML just represents that as text using tags: a word encompassed by the symbols < and >. The Email tag would be <Email>.

```
<email>
  <header>
    <from>
      Oscar@mail.com
    </from>
    <to>
      FIB@mail.com
    </to>
    <subject>
      XML example
    </subject>
```

```
  </header>
  <body>
    <text>
      I am writing an XML example
    </text>
  </body>
</email>
```

For readability purposes I've marked the tags as bold text and colored the content green. What are the tags with a "/" bar? It's a closing tag: we put the "from" field content between the tag <from> and the closing tag </from>.

And that's it. We can change it in any way: we can change email for letter and it would still be correct – representing a letter and writing streets and postal codes instead of mails. We could also add a child inside the <body>, for example <image> and write an URL, or <attached_file>. If we preferred so, we could change <text> for multiple <paragraph> tags.

Just note most modern mail manager programs use XML representations to store your mails, although no with representations as simple as that –i.e. an email header has a lot of extra information such as info of the servers it came from and security signatures from them to avoid spam.

8.4.4 Embedding XML in ActionScript

Embedding XML files in ActionScript is really easy… when you hack your way around to parse the content correctly. First, to embed an arbitrary file, we need to declare a variable of type Class and put [Embed] metadata on it, using the MIME type application/octet-stream:

```
[Embed(source="/assets/Cards.xml", mimeType="application/octet-stream")]
protected const EmbeddedXML:Class;
```

With that, the compiler autogenerates a subclass of the ByteArrayAsset class and sets our variable to be a reference to this autogenerated class. We can then use this class reference to create instances of the ByteArrayAsset using the new operator, and we can extract information from the byte array using methods of the ByteArray class:

```
private function loadFromXML(){
        var contentfile:ByteArrayAsset = new EmbeddedXML();
        var contentstr:String = contentfile.readUTFBytes( contentfile.length );
        xml =  new XML( contentstr );
}
```

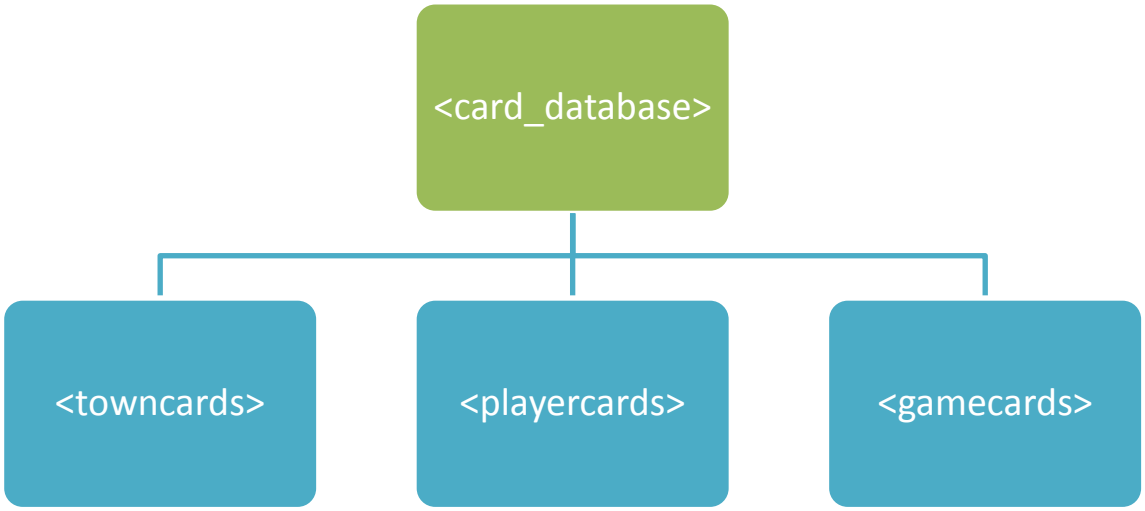
Concretely, we used the readUTFBytes method [19], which reads a sequence of UTF-8 bytes specified by the length parameter from the byte stream and returns a String. Lastly we parsed that String into an actual XML by using the built-in method [20] for that effect.

You must specify that the MIME type for the embedding is application/octet-stream, which causes the byte data to be embedded "as is", with no interpretation. It also causes the autogenerated class to extend ByteArrayAsset rather than another asset class.

var contentfile:ByteArray = new EmbeddedXML();

8.5 Cards.xml

Now that we know what information to store and how to do it, let's define our file that will store our cards database. We will consider three different types cards from the storing point of view:



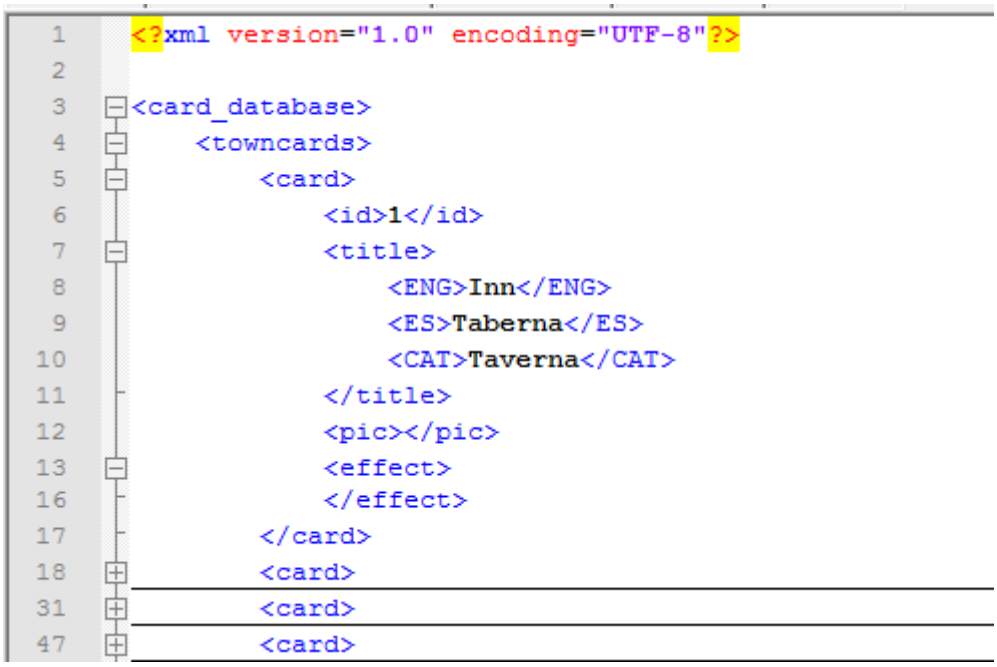
- Town cards are the ones with the book symbol in the top left corner. They don't have initiative or buy cost.
- Player cards are the starting cards of the players. Both players have the same ones and they don't have a buy cost.
- Game cards are ones that come to the game through the event line. These have an initiative and a buy cost –except for the Tether ones, although the buy cost can be considered to be infinite.

We will give a unique id to every card following this pattern:

- Town cards: numbers from 1 to 9
- Player cards: numbers from 10 to 18
- Rest of the cards: XYY, where X is the card's power level and YY is its initiative. For example, the palace is level X=3 and has initiative YY=41, so it's unique ID is 341.

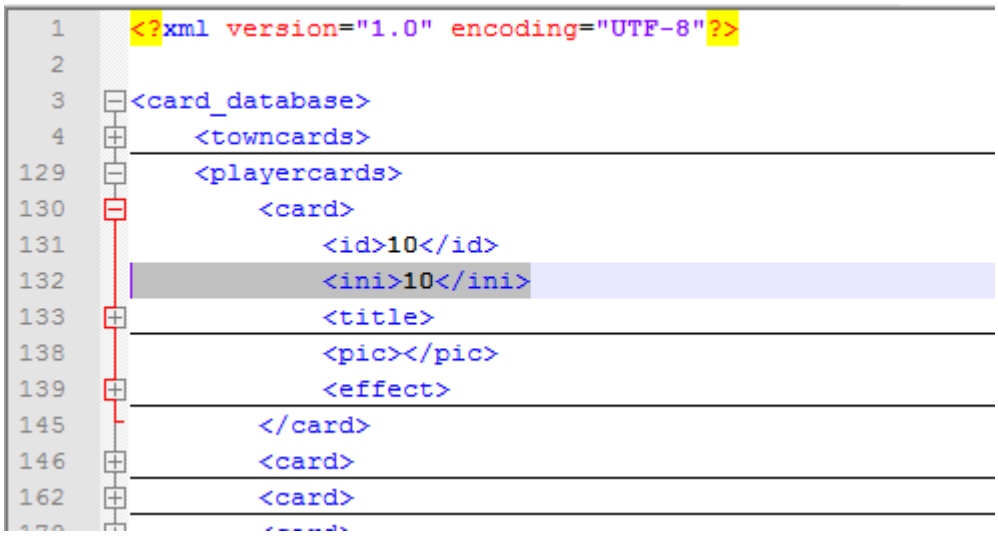
8.5.1 Town cards

Our town cards consist of a unique identification, the title, which is in three different languages as requested, and an image and effect nodes, that we'll get into later.



8.5.2 Player cards

These ones do also include a node that contains the initiative of the cards, that is used to decide who starts at planification phases.



8.5.3 Game cards

We include here both quest and tether cards. We separate them in four different subnodes, one for each power level tier. This way we can easily know the level of the cards, since it's common to many, without adding an xml node for it.

```
<card_database>
    <towncards>
    <playercards>
    <gamecards>
        <tier1>
        <!--level2-->
        <tier2>
        <!--level3-->
        <tier3>
        <!--level4-->
        <tier4>
    </gamecards>
</card_database>
```

As we said, these cards do need to include their buying cost in the game:

```
        <tier2>
            <card>
                <id>228</id>
                <type>Quest</type>
                <ini>28</ini>
                <title>
                    <ENG>Quest</ENG>
                    <ES>Búsqueda</ES>
                    <CAT>Cerca</CAT>
                </title>
                <pic></pic>
                <buycost>5</buycost>
                <effect>
            </card>
```

Tether cards are quite special because they don't need to have an effect or a buying cost.

```
            <card>
            <card>
                <id>101</id>
                <type>Tether</type>
                <ini>1</ini>
                <title>
                    <ENG>Tether</ENG>
                    <ES>Traba</ES>
                    <CAT>Trava</CAT>
                </title>
                <pic></pic>
            </card>
            <card>
        </tier1>
```

8.6 Card's effects XML syntax

In Dungeon Realms there's quite a bunch of effects and combinations of them in the cards. From gaining money and experience to having cards saved in your hose to recover them in the next game journey. To that, it adds that some cards have 'or' and 'and' logic operators and cards that need prerequisites to be played, such as discarding N cards to then draw N.

8.6.1 Dungeon Realm effects

The different card effects in the game are:

- Money: gain a coin.
- Experience: gain a point.
- Draw: get a card from the deck.
- Discard: put a card on the cemetery.
- Garbage: put a card out of the game. It can't return.
- Acquire: get a free quest card from the event line.
- Follower: gain an extra follower.
- Null: flip a town backwards.
- Hose: save a card of the current hand and add it in the next journey's hand.
- Replay (deed): use again a non-deed card.
- Double gold (deed): double the amount of gold gained during a turn.
- Double experience (deed): double the amount of experience gained during a turn.

Except the deeds, hose and null, all of them can be preceded by an integer. A 2 before an experience symbol means to gain two points instead of one. A -1 before a follower means to lose one of them. The combinations are:

- Positive or negative numbers: money, experience, follower.
- Positive numbers: draw and acquire.
- Negative numbers: discard and garbage.
- No numbers (always "one"): null, hose and deeds.

Since it's pretty disperse we won't make differentiations from the XML coding point of view. We would need to code effects in four different ways just to get a couple different effects coded in each.

There are also these operators:

- And: when it appears, both effects of the card take effect. For example, gain 3 coins and acquire a free card.
- Or: when it appears, the player must choose one of the options. For example, gain 3 coins or 5 experience points.
- Arrow (pointing right): do the prerequisite effect at the left side of the arrow and apply the right effect. For example, garbage up to two cards and gain up to two coins, one for each garbaged card.

8.6.2 Coding them in XML

Remember you saw empty effect nodes in the previous Cards.xml point? We'll work them here. To start, we define a different XML node for each possible effect:

- Money: <money>
- Experience: <px>
- Draw: <drawcard>
- Discard: <discard>
- Garbage: <garbage>
- Acquire: <cart>
- Follower: <follower>
- Null: <nulltown>
- Hose: <savehand>
- Replay: <doubleuse>
- Double gold: <doublegold>
- Double experience: <doublepx>

So, for example, a node…

```
<garbage>2</garbage>
```

…means to garbage two cards. Which is very intuitive to anyone that looks into the XML. The five last effects in the previous list –from null to double px-, do always have the value 1 inside the node. We could write them just empty, like <savehand/>, but then our AS3 code would need to check for different nodes –empty ones and non-empty ones.

Once we've solved the effect and integer multiplicators coding into XML, we then have to step into the operators. We'll start with the arrow, which is somewhat a logical prerequisite: if you do A then you can do B. If we generalize, we could view cards without the arrow as cards with an empty prerequisite. And XML does allow for empty nodes. Do bells ring into your ears, my dear reader?

We'll insert a prerequisite node and a post node into each effect node. For example, the previous "garbage two cards" would be:

```
<effect>
    <pre/>
    <post>
        <garbage>2</garbage>
    </post>
</effect>
```

It says that the card does not have any explicit prerequisite –empty pre node- to be played, and that the effect –post- is to garbage two cards. Note we say it doesn't have an explicit requirement: to be able to garbage two cards from the hand, the player will need to actually have two hands in his or her hand to be garbaged! But that will be a job for the game engine.

The next step is to represent the 'and' operator. We've decided to don't introduce it explicitly, but make it implicit when there are multiple nodes under the post effect:

```
<effect>
        <pre/>
        <post>
            <cart>1</cart>
            <money>2</money>
        </post>
</effect>
```

This one means to get one free card from the event line and also gain two coins. When there is an 'or' operator, we add it as a node under the post node:

```
<effect>
	<post>
		<or/>
		<money>1</money>
		<px>2</px>
	</post>
</effect>
```

It means the player has to choose to either get one coin or two experience points. Note we've not included the <pre/> node here: from an XML parsing point of view it won't make a difference whether the node is empty or it actually isn't there. Lastly, the arrow operator:

```
<effect>
	<pre>
		<discard>2</discard>
	</pre>
	<post>
		<money>2</money>
	</post>
</effect>
```

The effect represented here is to discard up to two cards and gain up to two coins.

8.7 Loading the cards in the game

We already know how we coded and stored cards in our Cards.xml file. How we access and interpret it? Let's explain it in small steps:

```
public class Deck {
	[Embed(source="/assets/Cards.xml", mimeType="application/octet-stream")]
	protected const EmbeddedXML:Class;
	protected var xml:XML = new XML();

	public function Deck() {
		Registry.screenLog.appendText("Deck - loading XML file\n");
		trace("Loading cards deck");
		loadFromXML();
		Registry.screenLog.appendText("");
	}

	private function loadFromXML(){
		var contentfile:ByteArrayAsset = new EmbeddedXML();
		var contentstr:String = contentfile.readUTFBytes( contentfile.length );
		xml = new XML( contentstr );
	}
}
```

First, we have to embed our XML file inside the flash executable. That is because (pseudo)compiled AS3 applications are self-contained in a single file in SWF format. These prorams cannot access the local file system due to the Virtual Machine security limitations –although it can actually be done through some hacks or through windows that let the user manually select a file, hence giving consent to read it.

In our game engine, we instantiate and initialize our Deck class with the desired language for the cards:

```
var d:Deck = new Deck(Constants.L_ENG);
```

Our deck is a parser for the XML file that has a method to get each of the types of cards. For example, to initialize the cards of a player's deck there's the initPlayerDeck call, which needs a destination Array where the card instances will be stored. It can also get an optional parameter to paint the cards in the deck with any color; by default it's blue, which is the local player's color, so to paint them red, the opponent's, we have the initOpponentDeck method, that calls the original one with the optional parameter set to red.

```
public function initPlayerDeck(t:Array, color:uint = 0xdddddff){
        Registry.screenLog.appendText("Deck - creating player cards...\n");
        var _pre:Vector.<int>,
            _post:Vector.<int>;
        for(var i:int = 0; i < 9; i++){
                t[i] = new GameCard();
                _pre = parseEffect(xml.playercards.card[i].effect.pre.children());
                _post = parseEffect(xml.playercards.card[i].effect.post.children());
                t[i].init2(xml.playercards.card[i].id,
                        parseTitle(card.title),
                         color,
                        xml.playercards.card[i].ini,
                         _pre,
                        _post);
        }
}
public function initOpponentDeck(t:Array){
        initPlayerDeck(t, 0xffdddd);
}
```

The algorithm first parses the pre –present when there is an arrow operator- and post effects and then initializes the GameCard instance with all the values needed for each card: id, title in the selected language, color, initiative and effects. Note how we can traverse XML nodes as if they were public variables of objects thanks to the integrated XML API present in AS3, and how we do also iterate all the cards as if they were an array. The rest of the cards perform similarly, but changing the type of parameters extracted and the subclass of Card instantiated.

```
        protected function parseTitle(e:XMLList):String{
                switch(language){
                        case Constants.L_ENG: return e.ENG.valueOf();
                        case Constants.L_ESP: return e.ES.valueOf();
                        case Constants.L_CAT: return e.CAT.valueOf();
                        default: return "ERROR";
                }
        }
```

To get the title in the correct language, we just switch with an instance variable, using predefined constants, that stores the correct selection

8.7.1 Choosing a data structure for the effects

Here we had to choose how to represent that as a data structure. An option would be to directly create a tree data structure that mimicked the XML structure, and then an algorithm that, at execution time,

would traverse the tree of the card instance looking for the leafs –the effect nodes- , much in the way of how a compiler interprets the programming language's syntax.

This approach was given a first try, but quickly discarded. It was a big overkill to create some kind of compiler to process a dozen different effects with a handful modifiers such as being under pre and post nodes and the operator –arrow, and, or. Maybe for a bigger ore more complex effect syntax it would have been worth.

The second option, the one actually implemented, is to store the effects in a plain data structure, which is either a table or a list. The first would seem to occupy more space than a list –due to empty table positions-, although in our VM that's untrue. The table is also easier to access and debug, so for it we go! Here's our definition for each table position and the range of values accepted:

Table index	Contains	Value range
0	No effects for the card	0-1
1	Has AND operator	0-1
2	Has OR operator	0-1
3	N/A - reserved	-----
4	PX effect	ℕ
5	Money effect	ℕ
6	Draw effect	ℕ
7	Discard effect	ℕ
8	Garbage effect	ℕ
9	Cart effect	ℕ
10	Follower effect	ℕ
11	Null town effect	ℕ
12	Double use effect	0-1
13	Double gold effect	0-1
14	Double PX effect	0-1
15	Save hand card effect	0-1

Putting some glue to all of it: position 0 indicates when there are no effects present, for example, when in the prerequisite effects of arrow operators; positions 1 and 2 exclude each other, as there are no cards with an 'and' and an 'or' operator at the same time; positions 5-10 contain effects that can have a value attached, as how many times to apply the effect; positions 12-15 contain single-use only effects, so it's again a 0 or 1 content.

For an easier iteration later we define these values in a Constants file:

```
public class Constants {
        public static const V_EMPTY = 0;
        public static const V_AND = 1;
        public static const V_OR = 2;
```

```
        public static const V_START = 4;

        public static const V_PX = 4;
        public static const V_MONEY = 5;
        /* the rest */
        public static const V_DOUBLE_PX = 14;
        public static const V_SAVE_HAND = 15;

        public static const V_END = 15;
        public static const VLENGTH = V_END+1;

        public static const V_CLASSES:Vector.<Class> = new <Class>
                [/* */ SymPX,SymGold,SymCard,/* */, SymHand];
}
```

These declarations relate semantically the indexes of the definition to easier to understand words when developing. We also added an start and end indexes to put in fors, so the algorithms won't need to be touched when adding more effects: we just add one and sum one to V_END; there's also a declared constant for effect table lengths.

Lastly, note also the Vector at the end. That vector is used to seamlessly reference our library movie clips that will graphically represent the effect in the screen. For example, SymPX is the class of the clip that shows a green square –the PX symbol in Dungeon Realms-, so V_CLASSES[V_PX] returns that class. That is, we can do mc:MovieClip = new V_CLASSES[V_PX] and will get an instance of the green square. Ahhh… the greatness of AS3 powerful constant definition.

So, retaking the last page, the parseEffect method gets an XMLList as a parameter, which is the list of subnodes under the pre and post nodes in a card's XML representation. For the language, an XMLList is an XML that has no root node.

```
protected function parseEffect(e:XMLList):Vector.<int>{
        var v:Vector.<int> = new Vector.<int>(Constants.VLENGTH);
        if(e.length() == 0){
                v[Constants.V_EMPTY] = 1;
        }
        else{
                if(e.length() > 1) v[Constants.V_AND] = 1;
                for each (var node:XML in e) {
                        //trace(node.name());
                        switch(node.name().toString()){
                                case "money":
                                        v[Constants.V_MONEY] = node.valueOf();
                                        break;
                                case "px":
                                        v[Constants.V_PX] = node.valueOf();
                                        break;
                                /*
                                        More cases
                                */
                                case "savehand":
                                        v[Constants.V_SAVE_HAND] = 1;
                                        break;
```

```
                    case "nulltown":
                        v[Constants.V_NULLTOWN] = 1;
                        break;
                    case "or":
                        v[Constants.V_OR] = 1;
                        v[Constants.V_AND] = 0;
                        break;
                }
            }
        }
```

First thing we do is to instantiate a Vector, of integers, of length equal to the total number of different effects, using our constant. We then iterate XML nodes with a for each, and switch accordingly, setting our vector easily thanks, again, to our constants file. Note how at some of the effects we extract the value in the tree with node.valueOf(), while in the later effects we just set a 1 –as we said when defining the XML file, it doesn't matter if one of those nodes is empty or contains a value. Lastly, if we found the <or/> node, we set off the AND operator and activate the OR position.

8.8 UI design

8.8.1 Are we starting development from the roof?

Most traditional developers will probably think we're nuts because we're starting developing from the roof. However, this view is biased because only a reduced number of those developers has had to dedicate himself to UI design, and of those only a very small fraction –mainly for the web- has had to create a software where one of the main ranks to measure it is the interface. On the other hand, if we picked specialists on UX and UI, they would argue designing and creating the interface is the first step to take the biggest step towards delimiting the project's reach.

Then, using breadth-first thinking, the next child option is: could we take a middle-man approach and start at multiple important points at the same time and integrate them later? That's the approach a team with multiple developers takes and the answer is that if you pursue your dreams with enough insistence you will get them. Having the dream of being able to multiply myself into many copies of me is not in my list, though. Taking this way would also drop continuous integration and taking small bites of coding at a time. So we prune this branch.

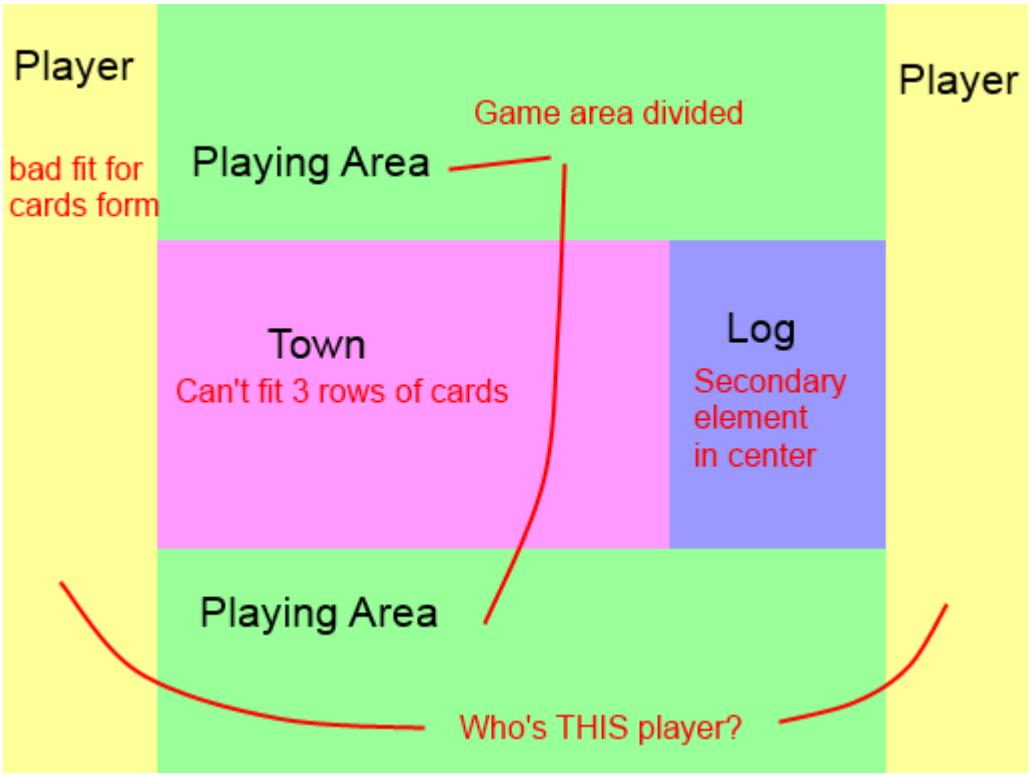
A videogame is a product where the interface experience is one of the most important assets the user will take into account when deciding whether to purchase –whatever meaning it has in the diverse economic models possible- the product. Interface responsiveness, easiness to learn and how does it feedbacks the game status to the user is called gameplay. Having to do more than one key press or mouse actions to give the game an instruction, having the game respond slow is bad or not having feedback of which are the actions possible are all bad and frustrating for the player.

Thus, having a well-defined interface from the start will limit very accurately what we need from the game's code and how do we need it. Why dry our brains thinking on it when the UI can do it for us?

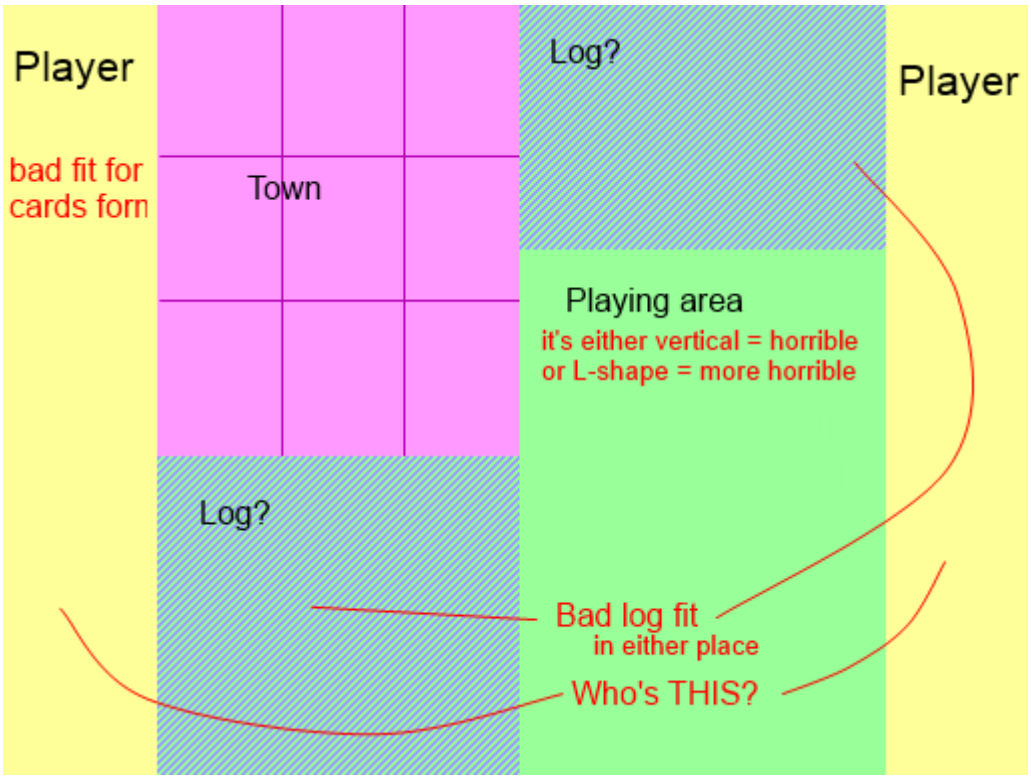
8.8.2 Interface design…

Dungeon Realms, the actual card game, has many elements that give info to the player at any given time. Both players have a hand of cards, a drawing deck, a discard pile and an amount of experience and money. On the table there are also nine cards in the town section and multiple cards owned by each player; and some card can have adventure markers.

The game interface should provide the same information for the player and it should also contain a text log with game actions. There are two approaches to the UI of a card game: vertically and horizontally. Let's take a look at why no one with a small bit of common sense has ever used a horizontal design on a computer card game:

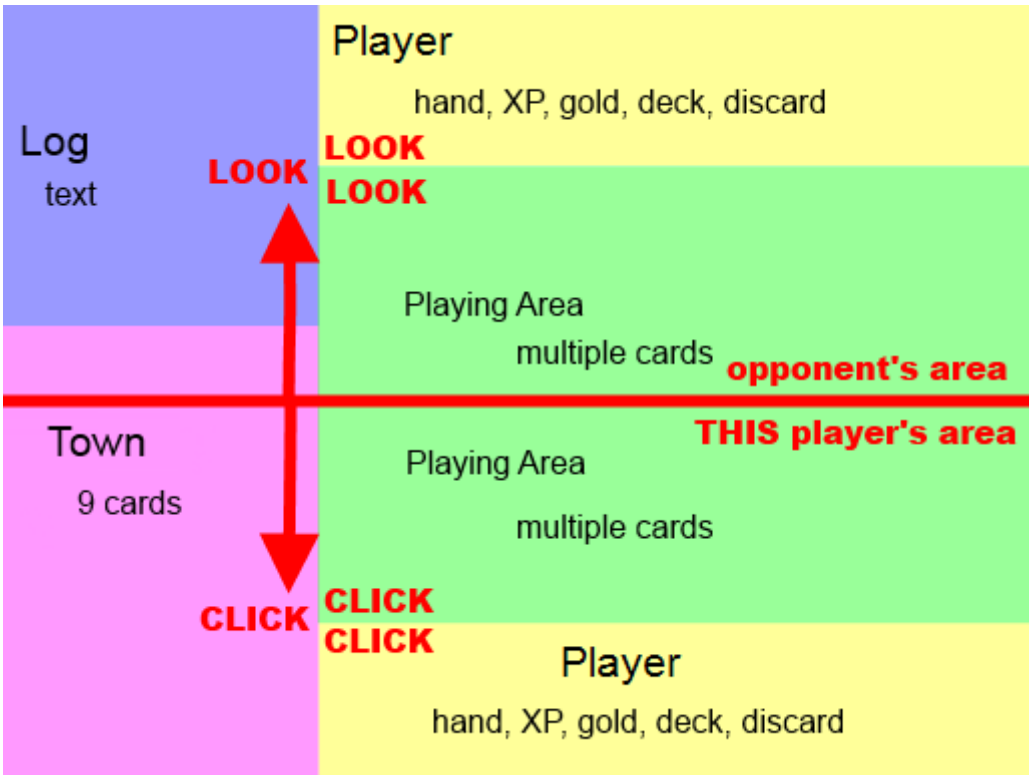


The main usability problem is the player expects to see his or her cards on the lower part of the screen, so putting them on a side is the first bad design choice. It's also not related to the real world where we see our cards looking down and the opponent's in front of us. Also in this disposition the town can't get a 3x3 size without taking a chunk of each player's playing area and we have to place a secondary element, the log, on the center of the screen.



Why can't something coherent be made? The answer is cards are vertical elements –wider than taller- and as such trying to dispose vertical elements horizontally implies the elements won't fit in a useful way. To make an UI we have to draw horizontal elements that can contain related cards disposed one at the side of the next. Since an image is worth more than a thousand words:

With a pattern like this we can also see how mouse movement is optimized because interaction is only needed with half of the screen:



8.8.3 … and polishing

The current player needs a big area to display the current cards in hand, but doing so for the opponent means showing the back side of the cards. While that may be useful for a couple test cases, it's not so from an interface perspective. Why show 5 unknown cards when we can show just a number?

That brings us to the next detail: shall we draw the deck and discard pile? The answer is that Dungeon Realms is an already complex game because there can be a lot of cards on the screen, so we'd be losing space for 4 cards counting both players area.

Taking it all together, we will only show actual cards –the ones that display its content. Gold, XP, deck cards, discard pile and cards in hand should be shown as a number. This helps achieve a UI approach called less is more [21] that synergies towards our agile methodology guideline of doing the simplest thing that may work.

The only redundancy we will have is having the current player cards also displayed –that inherently show the numbers. We'll do so to use the same component to show both player numbers.

So here's the reworked User Interface:

Since we're going to need cards to be displayed to be able to test the UI without the need of developing extra stub classes, we jump now to the opposite side of the project's design: loading cards.

8.8.4 …and yet more polishing

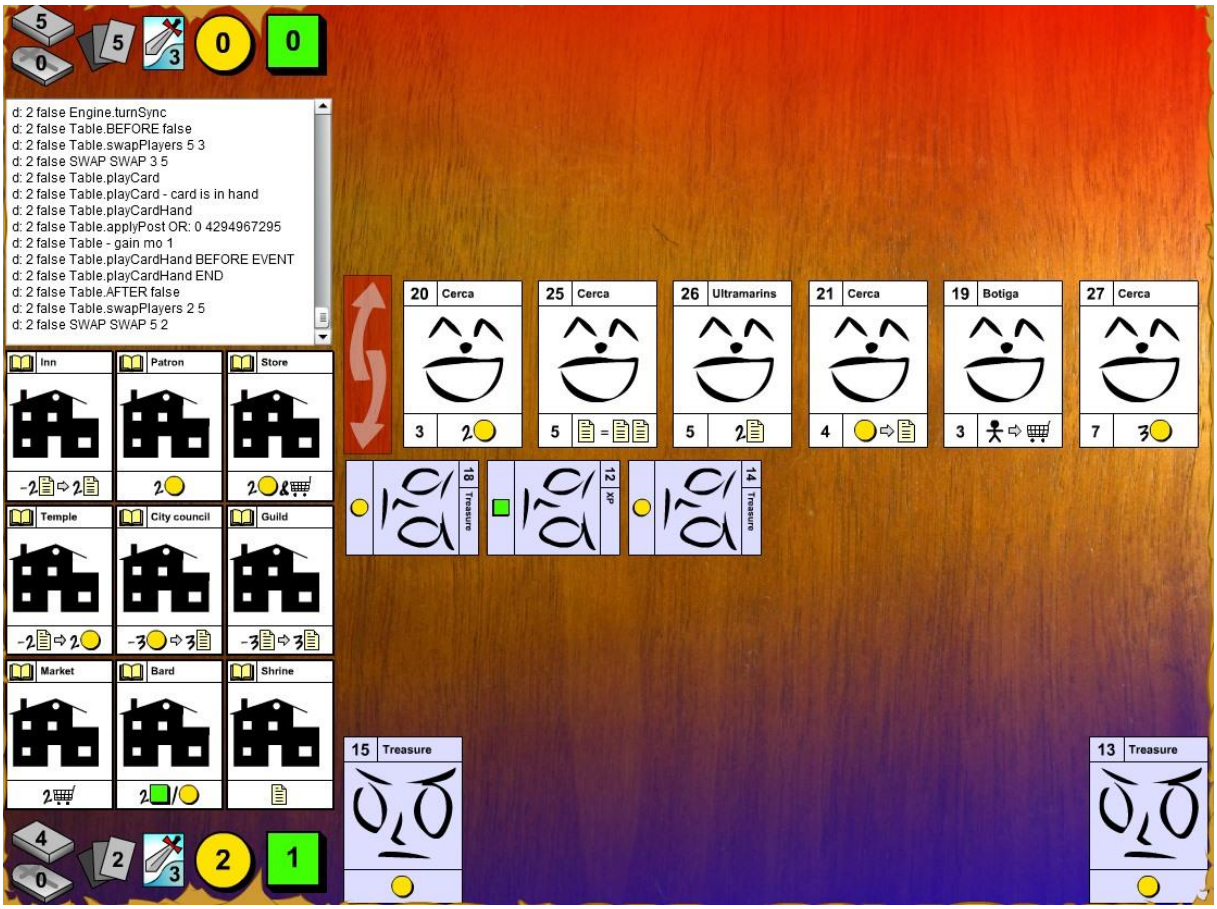
After initial tests we found the disposition of the Player information wasn't too fancy:

- The fact that we're used to read from left to right turned kind of jerky having to look right to read our data.
- The opponent's numbers are kind of floating in his playing area.
- The player hand gets somewhat shortened by having the data in that disposition

To solve it, we found the best solution was to displace the information to the left and show it horizontally instead of vertically:

Lastly, we put the event line that shows the quest cards between both playing areas and a button for the player to tell when he wants to end the turn:

Now, the actual game screen as it looks later in the development, exactly as we designed it:



8.9 The display list

The display list is the system by which objects are displayed on the screen using ActionScript 3.0. It is one of the main concepts to learn if you want to create anything visual using ActionScript. It is a list that contains all visible Flash content, it is used to control what can appear visually on the screen on the depths at which objects are placed over each other.

The basic usage of the display list involves displaying objects on the screen and removing others from the screen. These two tasks require using the .addChild() and .removeChild() methods.

Objects shown on the display list are placed in a hierarchy. If you add more than one object to the display list using the .addChild() method you will notice that objects you add later will be placed over previous objects (practically covering them).

For example, if you add to the display list an instance of a rectangle class, an instance of a circle class, and finally an instance of a star class, then the circle will be able (if a part is over it) to hide parts of the rectangle and the star will be able to hide both. Think of it as if you were adding layers of transparent paper, with a drawing in each sheet, one over the other.

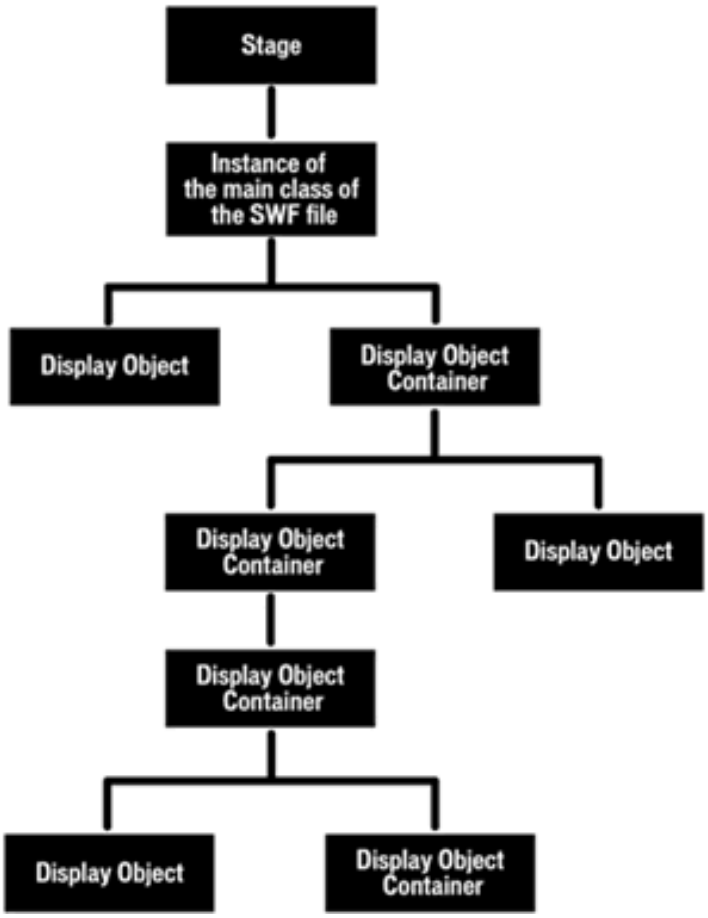
To avoid messing too with these lists AS3 does also provide the .addChildAt(parent, depth) method, that allows to add a child at the desired depth –being it the position at the list.

8.9.1 The typical display list

When an AS3 program runs, the VM that executes it is responsible to instantiate a Stage object, which is the base container of display objects. Each application has one and only one Stage object, which contains all on-screen display objects. The Stage is the top-level container and is at the top of the display list –which, be surprised when you see the following image, it's actually a tree- hierarchy. Each SWF file has an associated ActionScript class, known as the main class of the SWF file. As with any typical imperative programming language, the main class is the one that runs when we tell the computer to run our program.

When Flash Player opens a SWF file in an HTML page, Flash Player calls the constructor function for that class and the instance that is created (which is always a type of display object) is added as a child of the Stage object. The main class of a SWF file always extends the Sprite class. You can access the Stage through the .stage property of any DisplayObject instance. It's the Player VM that sets, behind the scene, the .stage pointer of every DisplayObject.

With any non-trivial game, though, the Stage pointer is kept in a global variable. When we say non-trivial we're mainly talking when non-DisplayObjects need to access it to manage the elements of the screen. For example, when the player is very low on life points in a shooting game, the engine could go to this global variable and find the correct place in the tree –with any algorithm to traverse it- to add blood drops to the screen to create an effect for the player to know he's about to die.



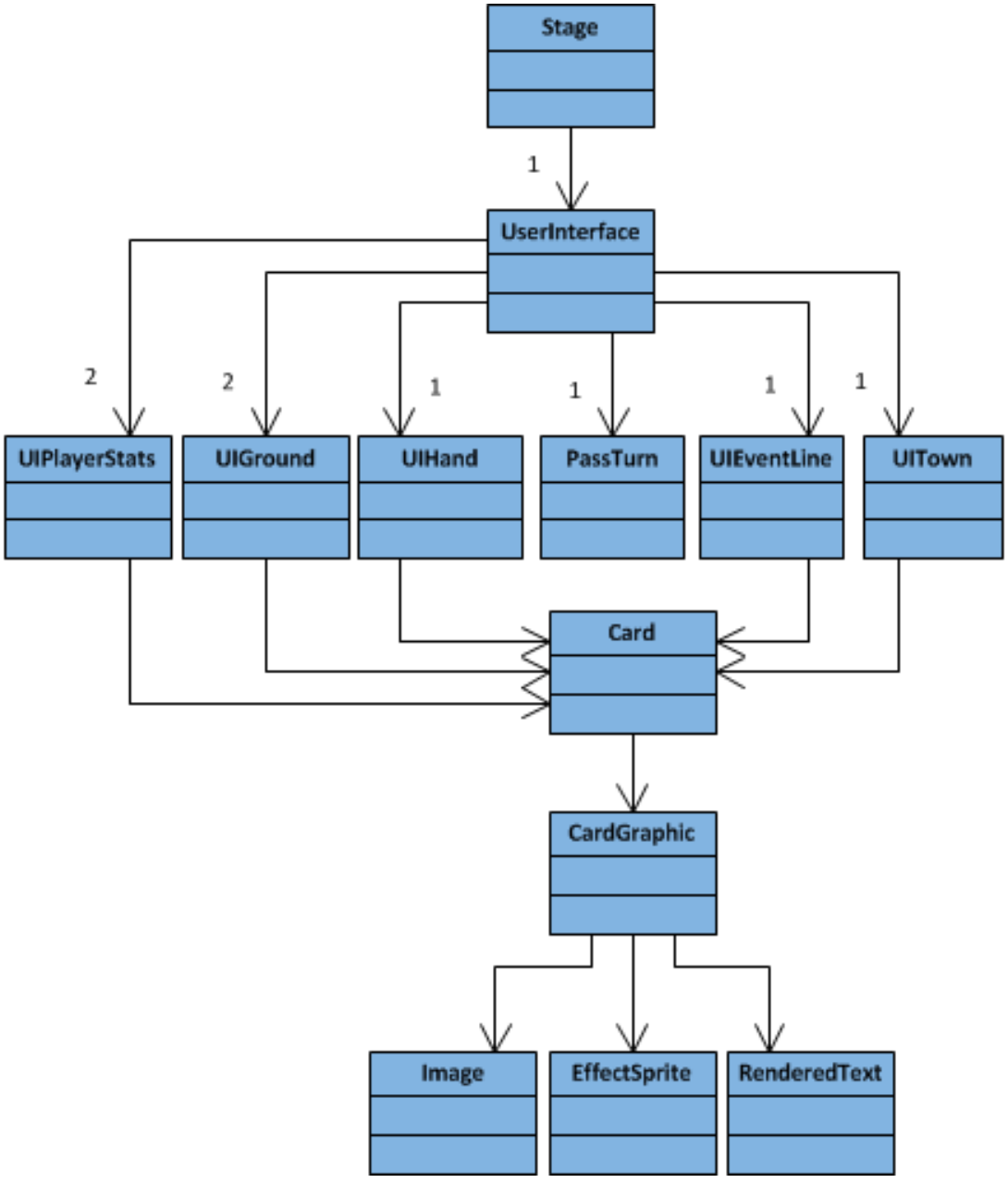
In ActionScript 3.0, all elements that appear on screen in an application are types of display objects. The flash.display package includes a DisplayObject class, which is a base class extended by a number of other classes. These different classes represent different types of display objects, such as vector shapes, movie clips, and text fields, to name a few.

Display object containers are special types of display objects that, in addition to having their own visual representation, can also contain child objects that are display objects. The DisplayObjectContainer class is a subclass of the DisplayObject class. A DisplayObjectContainer object can contain multiple display objects in its child list.

The typical DisplayObjectContainer used is the MovieClip (sub)class, which can be composed of several other MovieClip, Sprite, Graphic, etc. An example of a DisplayObject is the Shape class, that barely consists of a reference to the Graphics drawing interface, that has methods to draw lines from point to point, rectangles, ellipses and curves and filling them either with plain colors or gradients.

8.9.2 Our display list

Taking it into account, we've designed our display list for the game. Take into account every instance has a base clip of the class Stage that accommodates the rest of the interface elements. The Stage is the final responsible to make the hardware know what to render in the screen, and is something that cannot be massed with in AS3 –if we wanted that we would have taken C++!

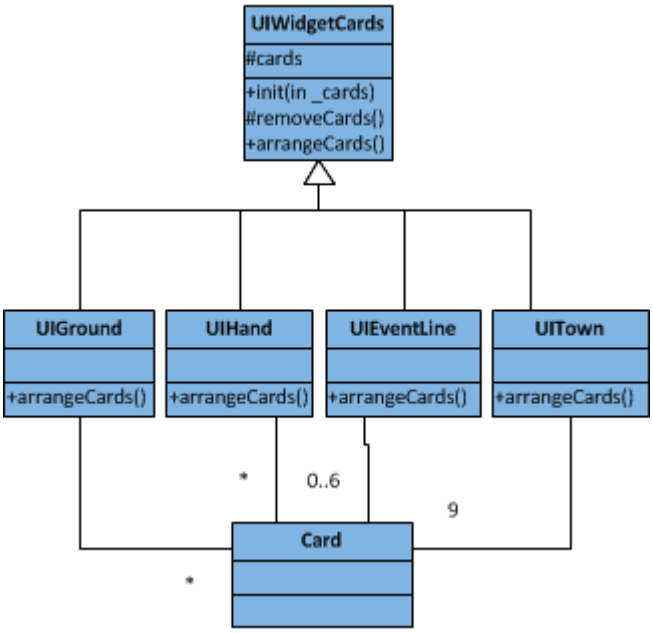


Note that we've drawn just one Card image for UML's well-practices sake, but at run-time we do actually have a bunch of different cards hanging for each of those nodes –as we saw in the typical display list graphic. Going back to the UML joke: actually we haven't separated Card boxes for each parent node because, after trying it, we didn't fancy too much having five Card boxes, five CardGraphic and fifteen other boxes hanging of it. We didn't want our reader's brains to explode.

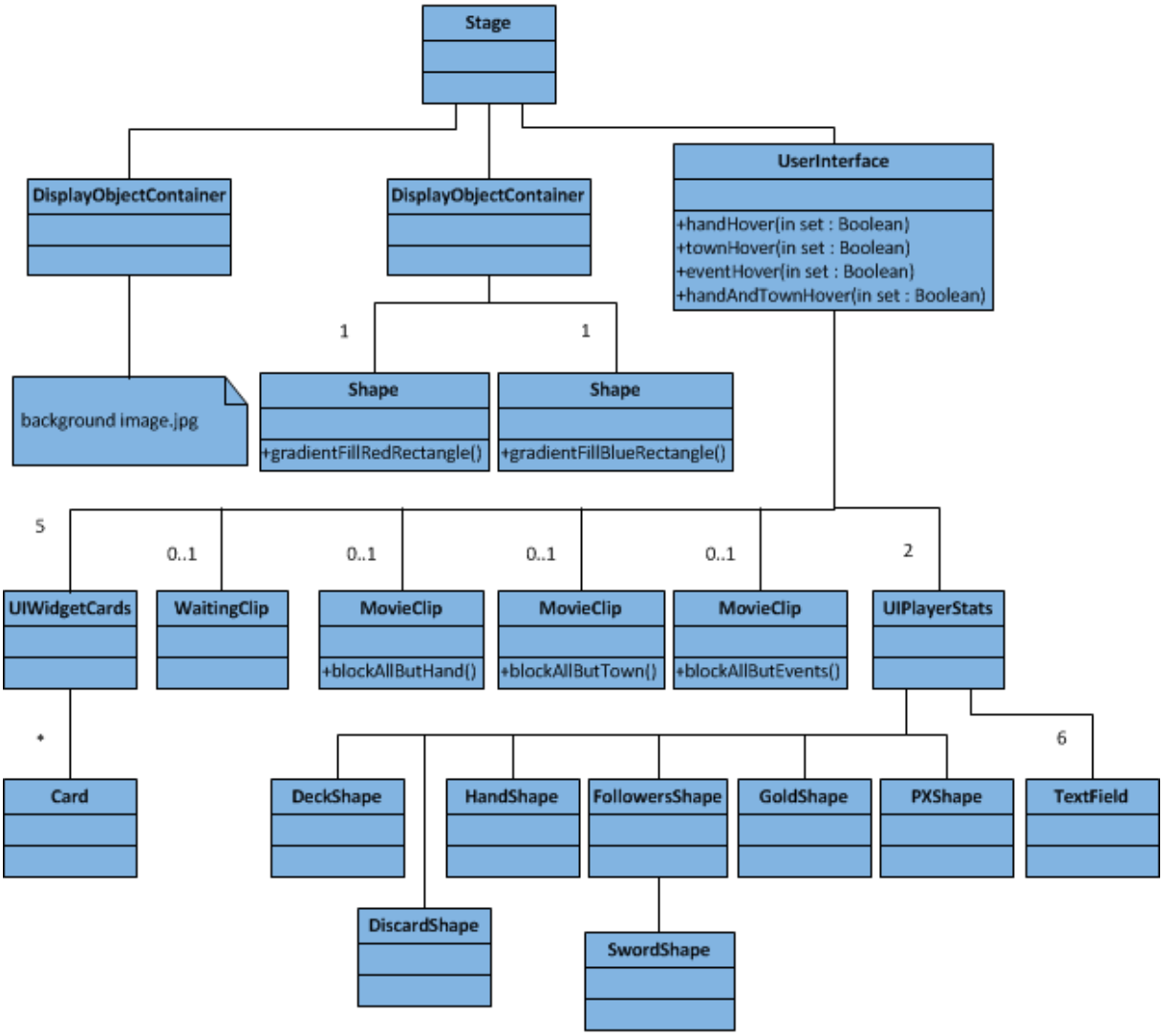
So, putting the concept together, the stage has a graphic interface added to it that consists of several different widgets that can be placed how we want in it. A handful of these elements do have cards that, at the same time, consist of several other –more concise- graphic elements: an image that goes in the center

of the card, sprites for the effect drawings at the foot, and a few parts of rendered text such as the card's initiative or cost to buy.

Our widgets to display cards have things in common: they do display cards. So we create a common class to all of them called UIWidgetCards, a class used to display cards. It's simple and provides a common method to clean the cards that are currently being displayed by the widget, as well as a common method to initialize. Each subclass will only need to implement an algorithm to arrange the cards in the correct position –grid, line, spacing, etc.



Note that our Stage is not only composed by the User Interface, but by other several elements as we show in the next graphic. Note that we've avoided to add to it the card's display subtree for spacing and understandgin reason. Just look at the card and imagine how other nodes hang from it: Shapes for possible effect symbols, various TextField for rendered text, an image for the background, PX symbols that mark the town cards that get extra PX after not being reserved for a turn…



8.10 UI implementation

Here comes the first real problem of our video game: there is still no existing book of design patterns for video games. We can utilize domain-related patterns but no pattern of interface design is useful for video games because those treat forms, data input, windows and so on.

So, the problem is implementing a user interface means displaying a lot of information and interactive elements at once that are usually highly correlated a few elements at a time. Controlling some dozens of cards, a dozen numbers being displayed, card markers, player's hand, etc, on a single typical –for normal software- window class would be highly hard an inefficient.

And at this point game development best practices come into play: widgetize the interface to solve the problem –I swear the first author to publish a pattern book for video games will include this. Widgetizing means dividing the single game window into different elements and implement them separately, so changing one does not affect other unrelated elements nor does the development of one require of the others. This way will also help us assure our continuous integration and small biting agile practices.

Hence, we'll have an interface class that will be composed of many different widgets or small interfaces.

8.10.1 Widget: Player info v1

This is our starting widget for displaying player information. If you read –you should have- how we built up the UI, this one is the widget we used before the final polishing of the interface disposition of elements.

So, the first and simplest correlated information is the player's info we decided to show together: number of cards, gold and PX. So we'll build it together in the UI as a widget.

We already learned on the first iteration, when displaying a card, how to subclass MovieClip in ActionScript and how to link it to a graphical element, so we won't show it again step by step.

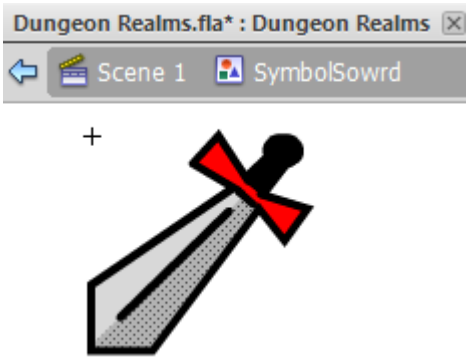
Hence, we use the Adobe Flash drawing tools and our Paint talent and draw a minimal widget:

We create a class that contains five text boxes, one for each number to display, and adjust the coordinates to display the numbers over the elements. Note we've used graphical meta-information –e.g. a gold circle for the gold- to let the player quickly know what means each number.

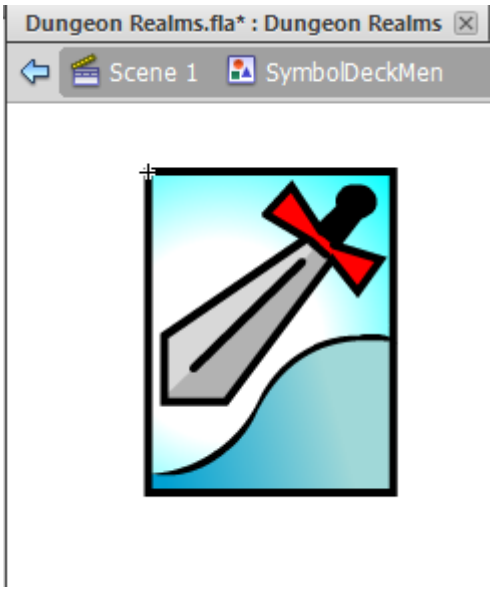
We initialize the text boxes in the same way we did our first example. Apart of that, the class itself is composed of a setter function for every one of the five numbers, so we don't copy any code since there's nothing really interesting to show.

8.10.2 Widget: Player info v2

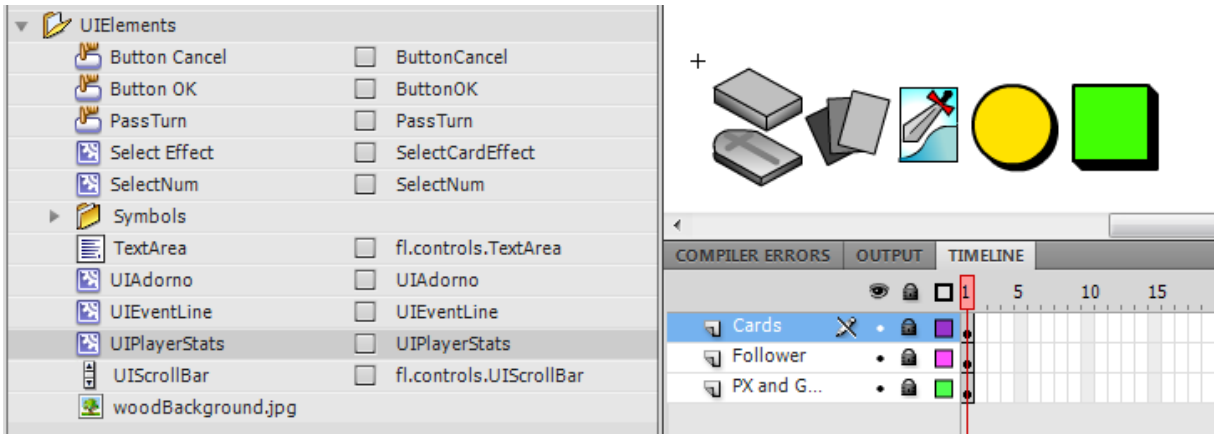
After knowing the final widget disposition, we decided to draw the elements directly with the Flash IDE integrated tools. For example:



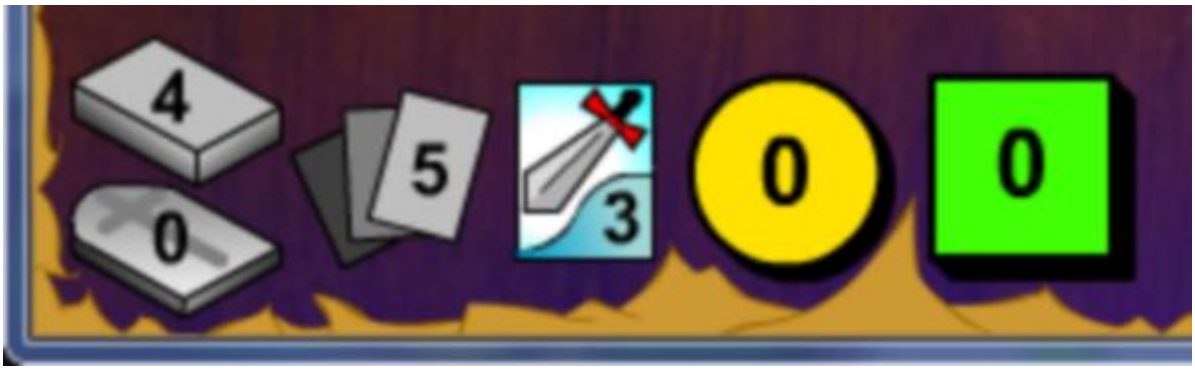
The sword is drawing by joining lines, similarly to CAD programs. Then we can fill gaps with colors and shades. We can use this sword into another composed symbol:



We keep putting hours into our limited drawing abilities to get all the needed icons, until we finally have something fancy and colorful for all the needed symbols we keep adding into our library. Finally we put them together into our UIPlayerStats class in a way decent enough to occupy few space and yet be communicative:



Once we place the TextFields via code in the correct places, this is what we get in-game:

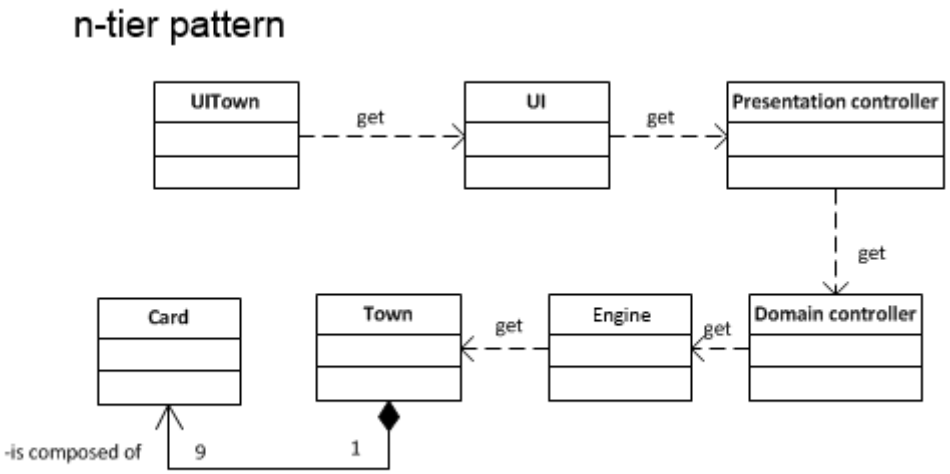


Not bad considering it was a red rectangle drawn in paint not so many pages back.

8.10.3 Widget: Town

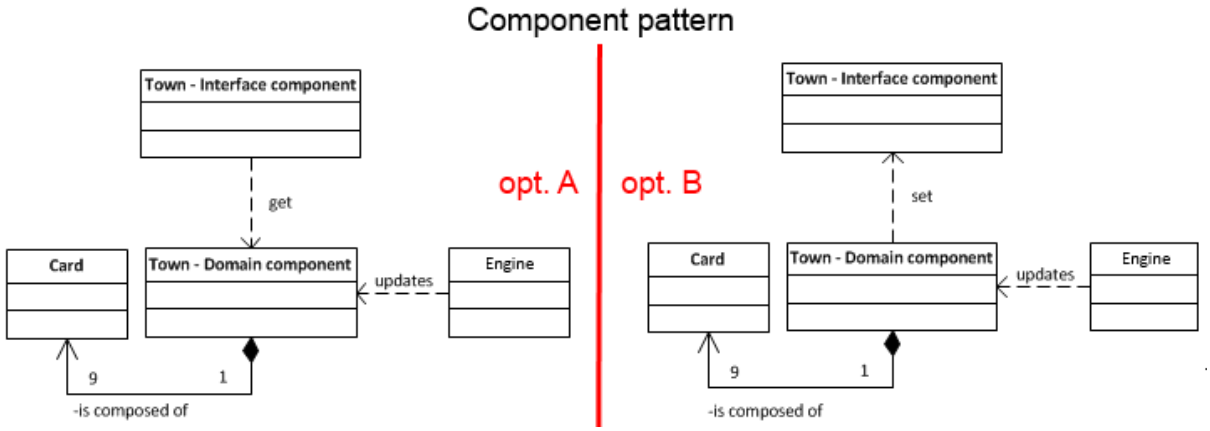
The town, in Dungeon Realms, is a composition of nine cards that are put together and follow different rules, like the player having to place adventurer markers on them, that the rest of cards in the table. So it feels like being converted in another widget of the interface.

Here comes another design choice. How do we tell which cards do the widget need to display? The very used n-tier design pattern would tell us to create two extra classes, the controllers, and to throw data requests through there:



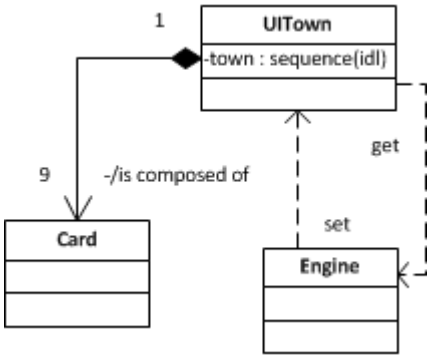
We're asking to launch five function calls to get a vector or an array, what really does make very little sense in our scenario –do the simplest thing that works and everything else. It's also important to have in mind that the tiered pattern is especially useful when transferring calls between software components that run on different servers –the communication is centered on the controllers.

The next option is to use the component pattern. This one is often used in video games when a class gets very complex. For example, a typical playable character class should contain code for capturing keyboard and mouse data for movement, setting the current graphic sprite photogram to show, physical engine calculations and so on, so that's divided into highly correlated components: character graphics, character actions capturing, character physics, etc.

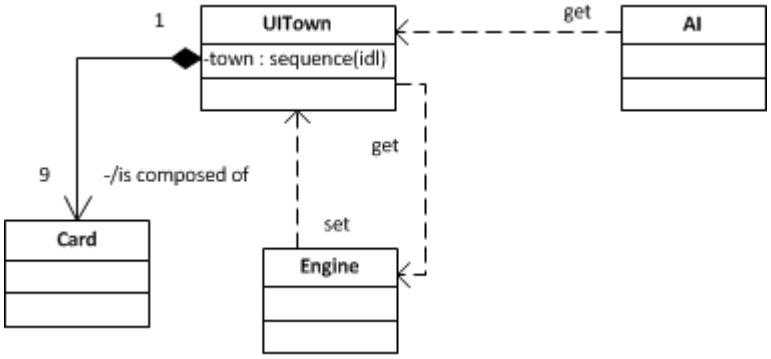


Sounds good, but we would be really overkilling by using this pattern since the Town is just a composition, an array, of nine cards.

Hence, being it just a variable we can think of storing it inside the UITown class such as this:

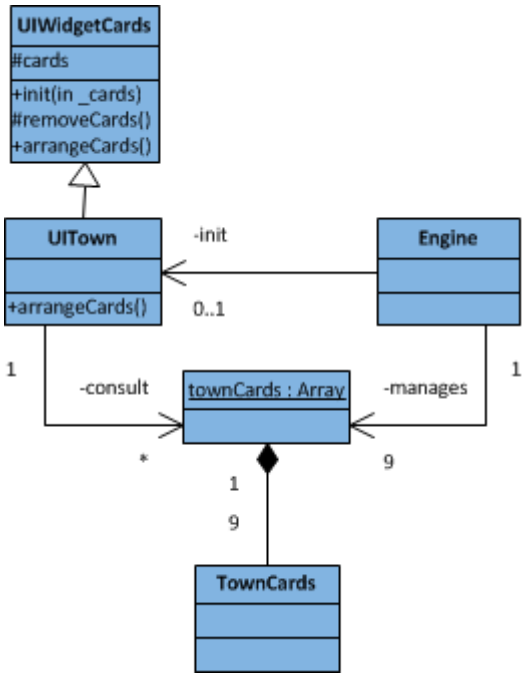


We could even choose either a pull or push –set or get- way of refreshing the data. This approach seems pretty fine but there's a problem once we think further. What happens when another class needs that data?



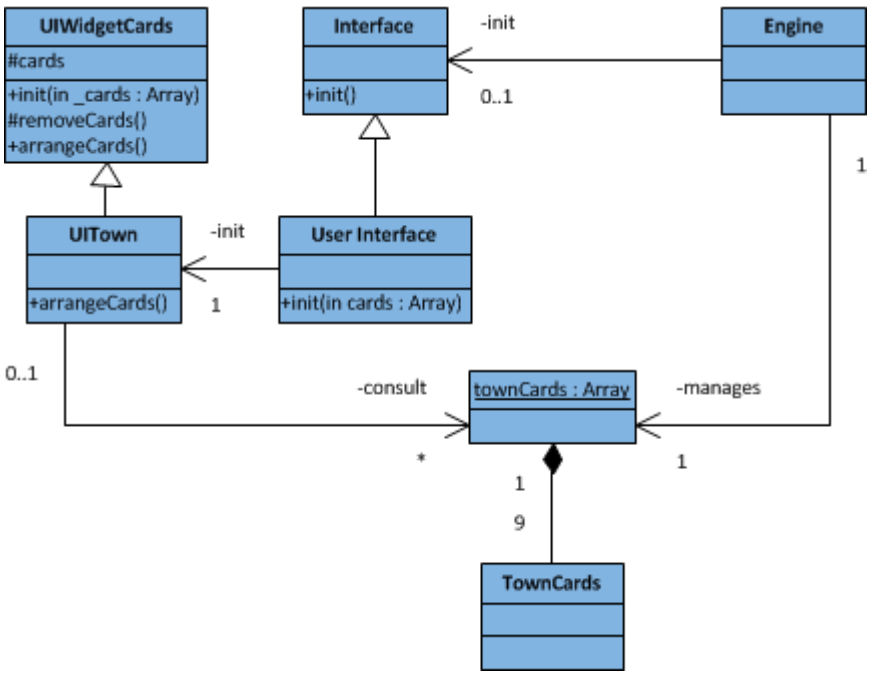
It's not that beautiful that classes not related to the interface need to access an interface element to request data. That scenario would also make the n-tier pattern overwhelming having to write a handful get calls for every data request –in the Flash virtual machine it means that once the code gets complex it can't run well even on high end system due to all the extra pointer and object calculations needed. The component pattern would be better suited, but then again we're just accessing just an array.

At the end of the day the UITown is accessing an array of 9 cards that is part of the engine. And we're using an OOP language. And we want to make the simplest ever thing that can work. And that's to pass the reference of the array to the UITown at initialization.



Since UITown then has a reference to the Town Cards array it will be able to ever know the cards it has to display. Since the reference points to the Array that the Engine is responsible to add and remove cards from, it will always contain refreshed data from the UITown's perspective without ever having to pass any extra pointers.

In concrete, the Engine instantiates an Interface at initialization, that might or might not be a UI. If it's the last case, the UI initializes the widget with the correct array. As follows:



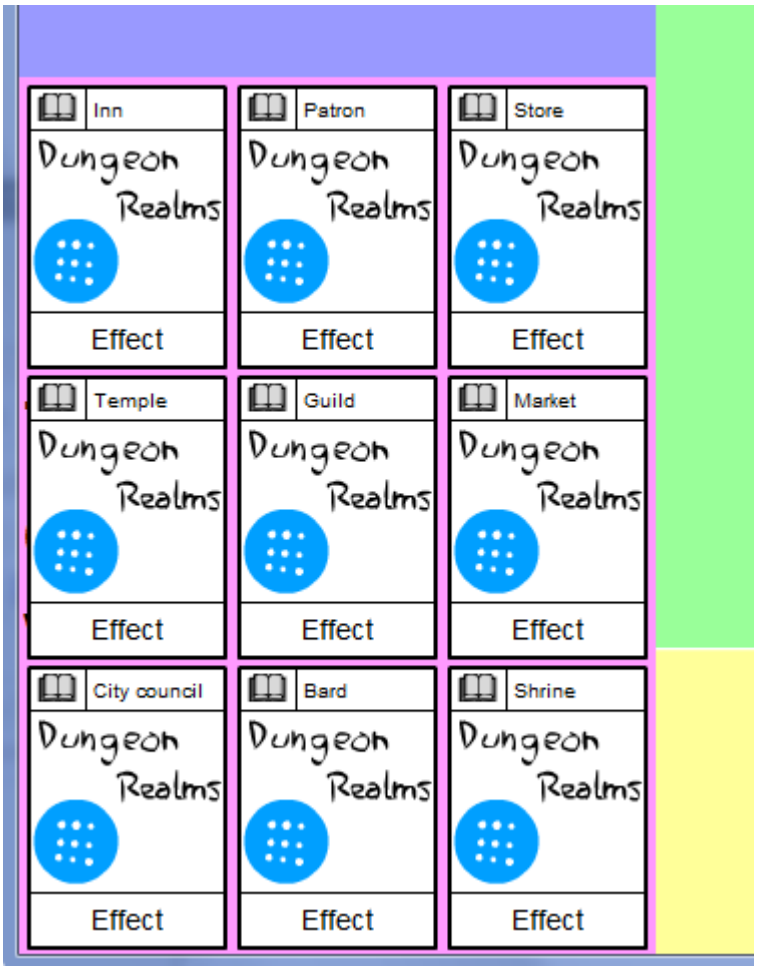
Our UITown just implements an algorithm to arrange the cards:

```
public function displayCards(){
    xx = 5;
    yy = 5;
    w = displayWidth * 1/Constants._CardWidth;
    removeCards();
```

```
        for(i = 0; i < 9; i++){
                addChild(cards[i]);
                cards[i].x = xx;
                cards[i].y = yy;
                cards[i].scaleX = w;
                cards[i].scaleY = w;
                if(xx < (displayWidth+5)*2)
                        xx += displayWidth+3;
                else{
                        xx = 5;
                        yy += cards[i].height + 3; ;
                }
        }
}
```

Note our code starts 5 pixels away (xx and yy variables) from the top left corner and proceeds setting the card's coordinates in a 3x3 matrix. The w variable is the factor by which we have to multiply the card MovieClip to show them in our desired displayWidth. Since Flash uses vectorial graphics, resizing is very highly preferred to creating assets in different resolutions. At run time, the VM does multiply the card height and width by the scale factor we set it to -our w.

So when we have >2 times the card's width, plus margin, it means we've already put three cards in that line and can start putting cards in the next line –the else- of the widget's grid.



8.10.4 Widget: Event line

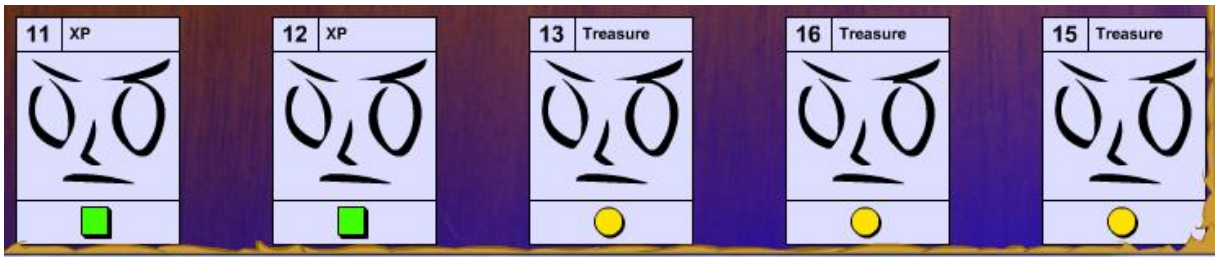
We follow the steps of the Town widget, only this time we display the cards in line:

```
public function displayCards(){
        xx = 5;
        yy = 5;
        w = displayWidth / Constants._CardWidth;
        removeCards();
        for(i = 0; i < cards.length; i++){
                addChild(cards[i]);
                cards[i].x = xx;
                cards[i].y = yy;
                cards[i].scaleX = w;
                cards[i].scaleY = w;
                xx += totalWidth/(cards.length-1);
        }
}
```

Note the xx+= line at the end does take the full width available for the event line to display all the cards instead adding a constant value.

8.10.5 Widget: Hand

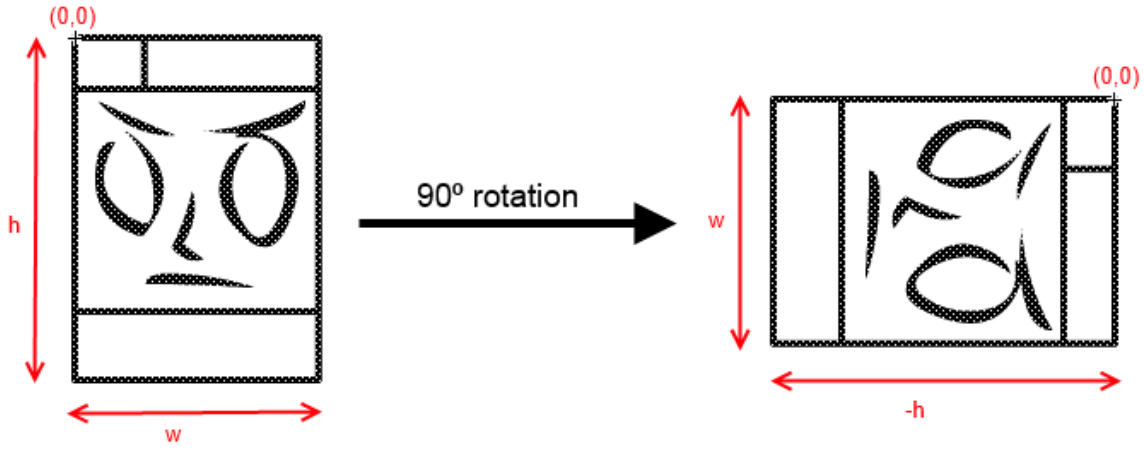
It's the same algorithm as in the Event Line, only that the totalWidth constant that tells the size of the widget is different.



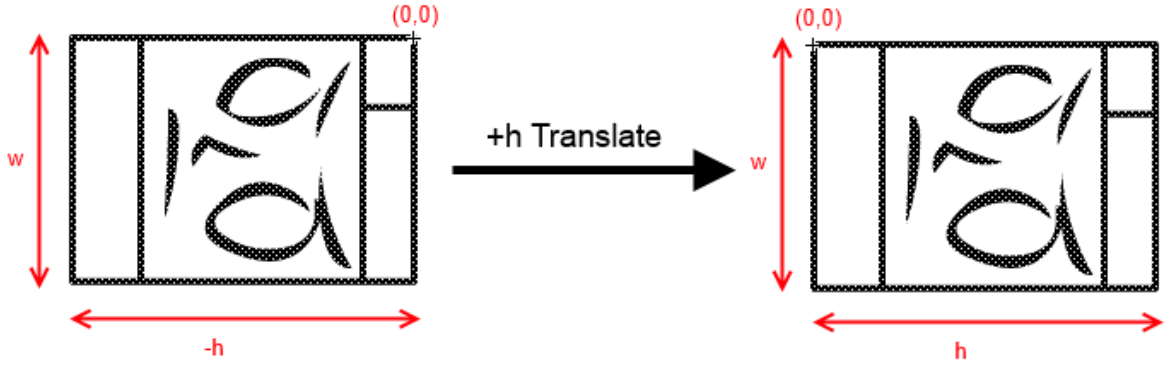
Note that at this point we're using stub images in the center of the cards. The Card class is also taking profit of the previously created PX and gold symbols.

8.10.6 Widget: Ground

This time the algorithm is similar to the Town widget's. However, the cards are rotated via code in the game when they are used, what causes them to change their reference point relatively to the card's visual appearance.



This fact became a tricky aspect that gave a lot of problems in the widget to correctly display the cards. Seeing the image you may assume that translating them like this:



Would work. Algebra says so. Even basic mathematics say so. But some cards kept not wanting to be translated so one cards overlapped others. Only after trying a few tweening libraries and losing a couple days with this simple task, we finally discovered Flash was randomly messing with our translations made via code. Solution? We don't translate the cards via code, we only rotate them.

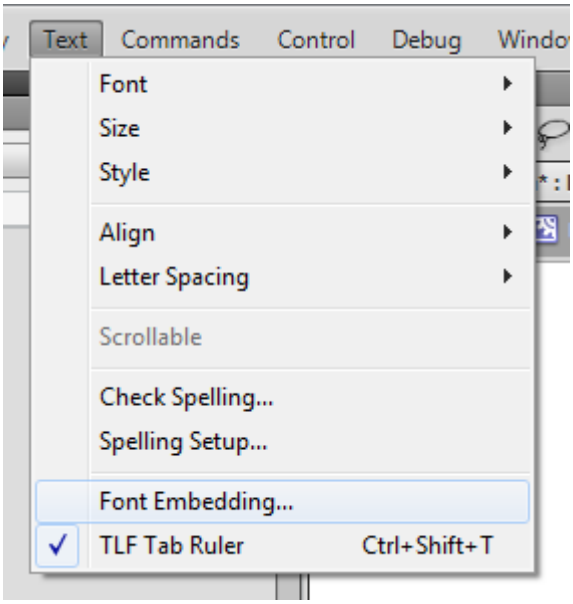
The implication of it is that we've to modify a bit the grid placing algorithm, of the Town widget to be useful here:

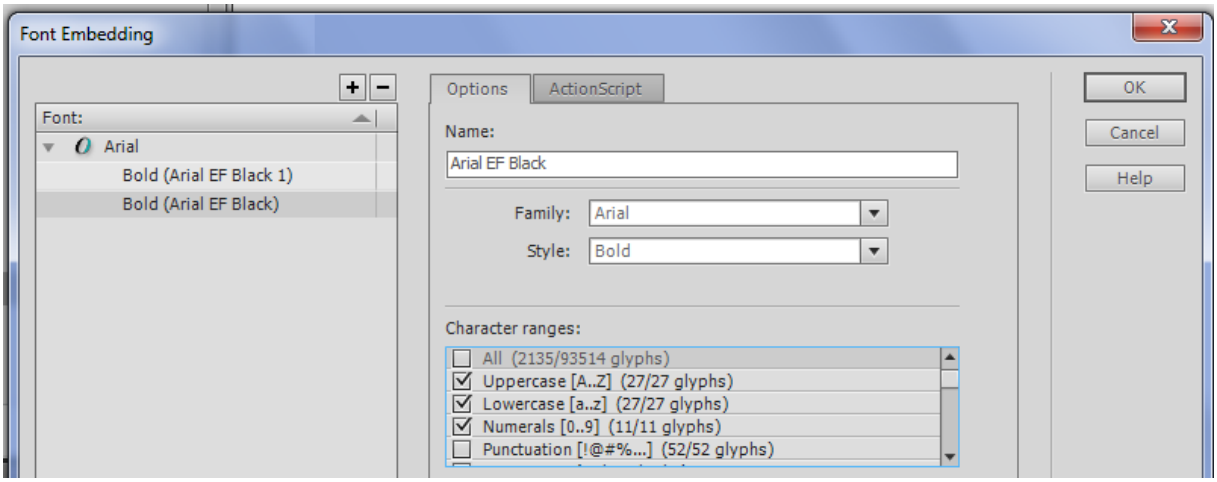
```
public function displayCards(clickable:Boolean = true){
        xx = 0;
        yy = 0;
        w = displayWidth * 1/Constants._CardWidth;
        removeCards();
        for(i = 0; i < cards.length; i++){
                addChild(cards[i]);
                cards[i].mouseEnabled = clickable;
                if(xx < 739-114-5) {
                        xx += 114+5;
                }
                else{
                        xx = 114+5;
                        yy += 80+5;
```

```
                    }
                    cards[i].x = xx;
                    cards[i].y = yy;
                    cards[i].scaleX = w;
                    cards[i].scaleY = w;
                }
            }
```

We've to account for the rotated card's width, which is 114 pixels, to calculate the card's position in the screen and when to change rows. In other words, if we placed a card in x=0 of the widget, it would be in the left outer part of it, so we've to reposition them. It's not the most elegant solution ever, but it's the only one that did work correctly.

Another problem is that, in AS3, when you rotate a MovieClip, the TextFields inside it don't rotate according to the parent's reference coordinate –the card's corner in our case. They do rotate according to the top left corner of the text. Every known Flash developer blasphemes at Adobe –me too after losing another couple days with it- for not fixing it to make the text fields rotate according to the parent's MovieClip. The solution is to embed the font bitmaps into our game:

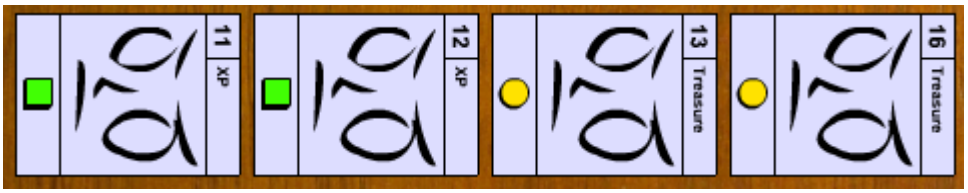




When we embed a font bitmaps in the game binaries, we have access to them via code. That means we tell our game via code to render the fonts from their glyph bitmaps instead of asking the operative system

to display them, which in the end treats them like DisplayObjects and makes our rotations take the parent's reference point.

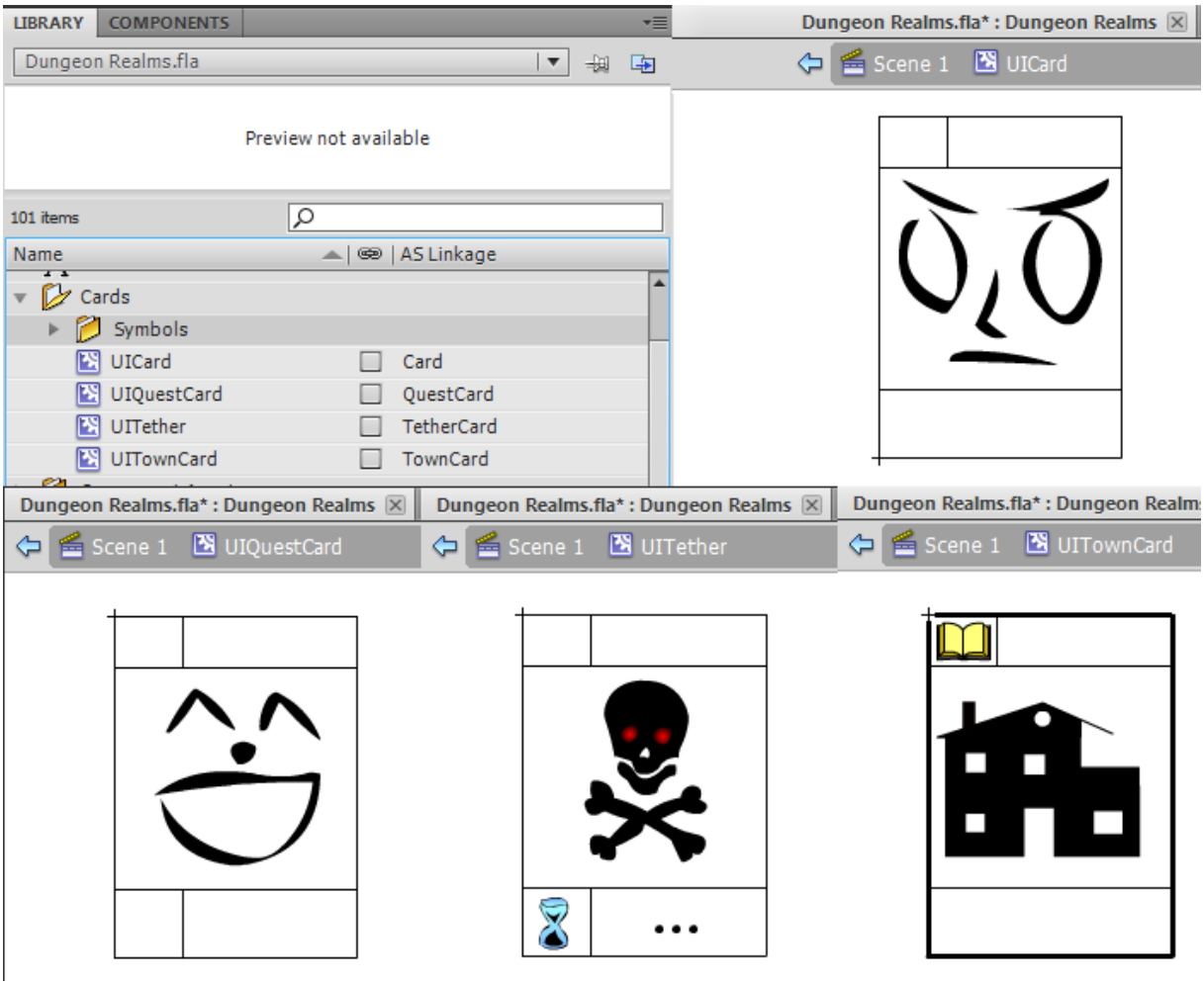
Finally, the result:



8.10.7 Cards

As we've said, in Dungeon Realms, cards can have multiple ways to work and that is reflected in their appearance. For example, town cards have a book graphic where others have the initiative, tethers have a sandglass where quest cards have the buying cost of quest cards and three points where the effect of these is and player and town cards have the bottom area without a vertical separation.

So, since an image is worth more than any explanation, here it is:



To achieve that we've drawn the graphics using our Flash IDE and to then assemble the drawings into various rectangles. We've drawn all of the cards 200 pixels high and 143 pixels wide, which maintains the aspect ratio of the paper cards. Note, however, that since all of the graphics are vector-based, the cards can be expanded or shrunken as much as wanted without any loss.

Note that in the AS column we've given a class name to link those MovieClip to every of our written ActionScript files. The linkage is automatic from here, as once we make a declaration:

```
public class Card extends MovieClip{
	/*
		implementation
	*/
}
```

The VM will look for the exported Card clip of the library and apply to the object. That means that adding a Card to the display list will show our Card graphic, a TownCard class will show the TownCard symbol, and so on.

To turn the cards, and after a lot of debugging of which we've talked about when exposing the interface's town widget, we finally settled with the GreenSock Animation Platform [22]. GSAP is a suite of tools for scripted animation that consists of a core animation API extended through some modules to provide functionalities such as controlling the flow of animations, timeline sequencing and so on.

The coolest thing about GSAP is that it' an API not only for AS3, but for JavsScript too. So yeah, you can use the same animation toolset for Flash and HTML5 projects. Same API and same robust feature set. So, since I'm to get into an animation API for the first time, it seems kind of nice to be able to use the same one through JavaScript.

So, to add and use a rotation animation, we have to:

```
import com.greensock.TweenMax;
public class Card extends MovieClip {
	protected var animTurn:TweenMax;
	public function Card(){
		/* code */
		animTurn = new TweenMax(this,1,{rotation:90, paused:true});
	}
	public function turn(){
		/* code */
		animTurn.play();
	}

	public function straighten(){
		/* code */
		animTurn.reverse();
	}
}
```

It's very straightforward and reduces a lot the work we've had to do instead with the integrated AS3 tweening functions, mainly to the lack of reversing and playing animations easily.

8.11 Shuffle cards

AS3 random number generator does not allow using a seed to be able to get the same sequence as many times as we want, hence being totally undeterministic and not useful for debugging reasons. That means we need to implement our own algorithm.

Since it's not the objective of this project to reinvent the wheel, we're using a known and proved algorithm: we're implementing in AS3 the subtractive random number generator, by Donald E. Knuth [23].

Why this one? Because it's easy and short to implement and because it's also used in Microsoft's .NET libraries, hence there's a warrantee it's good enough. The implementation is in our SRandom class and provides the public function nextMinMax(min,max), that returns a pseudo-random number between values min and max.

So, for example, to shuffle a table that contains the event deck cards previously loaded in order -all level 1 cards plus 2 tethers, all level 2 cards plus 2 tethers, etc-:

```
public function shuffleEventDeck(){
        shuffleSubArray(eventDeck, 0, 11);
        shuffleSubArray(eventDeck, 11, 22);
        shuffleSubArray(eventDeck, 22, 34);
        shuffleSubArray(eventDeck, 34, 45);
}
protected function shuffleSubArray(a:Array, stt:uint, end:uint){
        for(var i:int = stt; i < end; i++){
                _rand = rand.nextMinMax(stt, end);
                _card = a[i];
                a[i] = a[_rand];
                a[_rand] = _card;
        }
        return a;
}
```

With this method, we can also shuffle player decks and discard piles.

8.12 Card (ambient) images

To load card icons, we can use the XML inherent extensibility to create nodes that contain a path to find a card in the hard disk or Internet –it's done the same way. For example:

```
<card>
        <!—information nodes
        <pic>
                /assets/Cards/dollar.png
        </pic>
</card>
```

To process it, we add inside our Cards.xml (for) iterators:

```
t[i] = new GameCard();
t[i].setImage(xml.playercards.card[i].pic);
//rest of the for
```

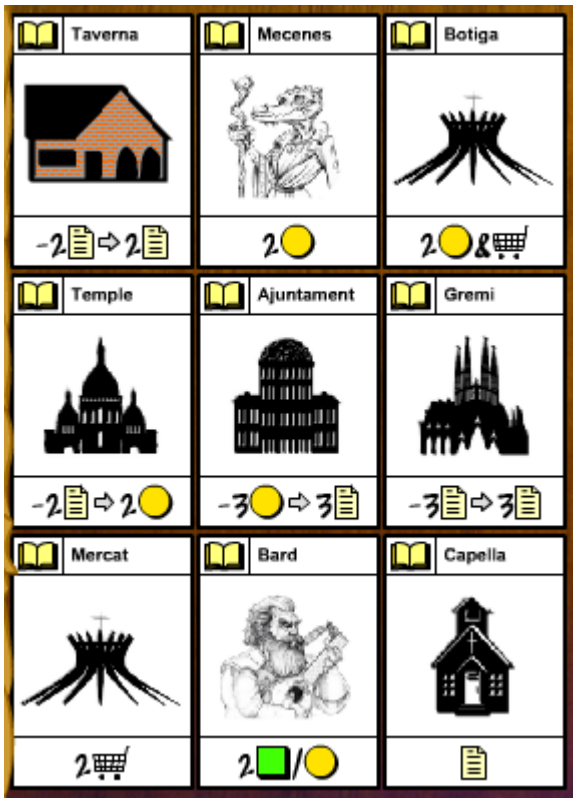
This then ends up in the card's code:

```
var ldr:Loader;
var image:Bitmap;
    public function setImage(s:String){
        ldr = new Loader();
        ldr.contentLoaderInfo.addEventListener(Event.COMPLETE, imageReadyHandler);
        var fileRequest:URLRequest = new URLRequest(s);
        ldr.load(fileRequest);
    }

    protected function imageReadyHandler(e:Event){
```

```
            ldr.contentLoaderInfo.removeEventListener(Event.COMPLETE, imageReadyHandler);
image = new Bitmap(e.target.content.bitmapData);
            addChild(image);
            image.width = 140;
            image.height = 100;
            image.x = 0;
            image.y = 45;
}
```

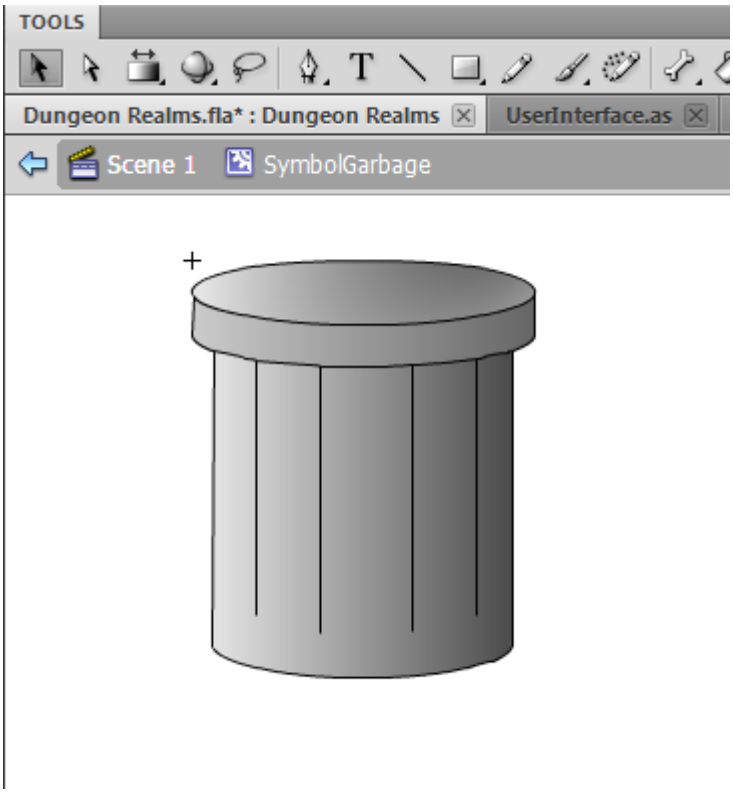
As always in the crazy world of AS3, we have to use events –did you think of it beforehand this time? We create a loader object to which we pass an URL to load. This resource locator can either be in our hard disk or in the (Internet) cloud, with exactly the same code. We have to add a listener that will fire when the loader is finished getting the image from whatever source.

The handler loads the data into a Bitmap object, and then we only need to add it to the display tree and set its coordinates and size. The result:



8.13 Card (effect) images

For the effect symbols we've chosen a different approach, because these images are very small and, using a regular image, the size scaling would make them low-quality as pixels would blur very easily. Hence, we've decided to take profit of the vector drawing capabilities of our IDE, Adobe Flash Professional.



We proceed drawing every effect symbol as in the upper garbage image. We do also configure them to be exported at runtime to use the mas objects, as we explained a few sections before, and give them runtime names following the pattern Sym*, where * is the name we give to the symbol to recognize it, for example SymGarbage.

To process them with less code, we take profit of the powerful typing of AS3, by declaring a vector of object names:

```
public static const V_CLASSES:Vector.<Class> = new <Class>
        [null,null,null,null,
        SymPX,SymGold,SymCard,SymCard, SymGarbage, SymCart, SymMan,
        SymBook, SymDoubleUse, SymDoubleGold, SymDoublePX, SymHand];
public static const V_EMPTY = 0;
public static const V_AND = 1;
public static const V_OR = 2;
public static const V_PX = 4;
public static const V_MONEY = 5;
//etcetera
```

So we then, in the Card object, have the following code that builds up the pre-effect and post-effect sprites, as well as introducing any operator symbols present. The possible patterns are:

- An effect-glyph can either be:
  - Symbol, without anything entra
  - Number followed by symbol
  - + or – sign, followed by number and then by a symbol
- Pre and post effects can have the same patterns:
  - Effect-glyph (just glyph from now on)
  - Glyph, plus OR /, plus glyph
  - Glyph, plus AND &, plus glyph

- When, there's a pre-effect, there's always the pattern:
    - Pre, plus ARROW, plus Post

Hence, we first check if the effect array has a pre-effect in the first conditional block, if there's we build the graphic up until the arrow. In any case, we then add the post-effect. We center all of them depending on the size of the lower part of the card: if there's no initiative, then the space is wider.

```
private function effectPaint(){
        var dx:uint = 0;
        var spriteEffect:Sprite = new Sprite();
        var spriteArrow:Sprite;

        if(pre[Constants.V_EMPTY] == 0){
                spritePre = toSprite(pre, true);
                spriteEffect.addChild(spritePre);
                spriteArrow = new SymArrow();
                spriteArrow.x = spriteEffect.width+5;
                spriteEffect.addChild(spriteArrow);
        }

        spritePost = toSprite(post);
        if(pre[Constants.V_EMPTY] == 0)
                spritePost.x = spriteEffect.width+5;
        spriteEffect.addChild(spritePost);

        addChild(spriteEffect);
        if(wide){
                spriteEffect.x = (Constants._CardWidth - spriteEffect.width)*0.5;
        }
        else{
                spriteEffect.x = _wideDx
                        + (Constants._CardWidth - _wideDx - spriteEffect.width)*0.5;
        }
        spriteEffect.y = 167;
}
```

The toSprite method we call from the previous function, basically iterates over the effects and adds symbols one after the next, checking if there's need to add / or & at any moment.

```
private function toSprite(v:Vector.<int>, neg:Boolean = false):Sprite{
        var g:Sprite = new Sprite();
        var m:MovieClip;
        var dx:uint = 0;
        var OR:Boolean = (v[Constants.V_OR] == 1);
        var AND:Boolean = (v[Constants.V_AND] == 1);
        for(var i:uint = Constants.V_START; i <= Constants.V_END; i++){
                if(v[i] != 0){
                        m = new Constants.V_CLASSES[i]();
                        if( v[Constants.V_OR] == 1) dup(Constants.V_CLASSES[i]);

                        if(v[i] != 1)
                                addNumber(g, v[i], neg);
                        if(g.numChildren > 0)
```

```
                    m.x = g.width+2;
                g.addChild(m);

                if(OR || AND){
                    if(OR){
                        m = new SymOr();
                        OR = false;
                    }
                    if(AND){
                        m = new SymAnd();
                        AND = false;
                    }
                    m.x = g.width+2;
                    g.addChild(m);
                }
            }
        }
        return g;
}
```

9 Iteration 3

9.1 Objective

Add interaction to the User Interface so that user clicks on the cards can be tracked and processed.

9.2 Backlog

B31 - Trace user clicks in different parts of the screen.

B32 - Get the clicks processed by the game engine.

9.3 Effort Chart

First time ever we've accomplished an objective in less than the estimated time! Note, however, the mouse click treating will give work later in the game, when depending on the game situation some elements must change their behavior.

Ref	Concept	Estimate	Real
B31.T1	Research how to track clicks	2	4
B31.T2	Implementation	5	5
B32.T1	Define communication between the screen and the game engine.	3	3
B32.T2	Implementation	15	9
	Total	25	23

9.4 Tracking user clicks

ActionScript 3 is so cool that it already provides some basics to allow for user interaction. We couldn't expect less of a language that has as a main goal to develop interactive rich applications. Could we?

9.4.1 MouseEvent

A MouseEvent object is dispatched into the event flow whenever mouse events occur. A mouse event is usually generated by a user input device, such as a mouse or a trackball that uses a pointer.

When nested nodes are involved, mouse events target the deepest possible nested node that is visible in the display list. This node is called the target node. To have a target node's ancestor receive notification of a mouse event, use EventDispatcher.addEventListener() on the ancestor node with the type parameter set to the specific mouse event you want to detect.

Note that to dispatch MouseEvents to an object that is listening, we need a MovieClip that provides an actual interface –an image or a button, for example- for the pointer to interact with it. Unless we artificially dispatch MouseEvents crafted via code instead of actual mouse actions, of course.

AS3's mouse events are actually complete and complex to use correctly when the developer is trying non-trivial tasks –and many newbies to the language do actually have a lot of problems with it. As for the basic MouseEvent class, we have available 18 different mouse actions that can be tracked: click different

mouse buttons, roll over/out the display element and even an event for when we click inside the element but release the mouse outside it.

Because we, the developers, can never have enough to play with, the MouseEvent class is then be subclassed into a dozen different event classes. From DragEvents, which are pretty general, to subclasses specific to certain element types –for example, ChartEvents are used to track clicks on chart matrices to get rows and columns without a hassle.

Going back to the basic MouseEvents, they not only tell the listener that an action was taken on it, but also gives access to a couple dozen variables so you don't have to mess with code: we can know the exact (x,y) screen coordinates of the click, whether the shift/control/alt keys where pressed, the (x,y) distances from the last click… Didn't you get tired to write big switch blocks just to know if a key was pressed when a mouse click occurred?

9.4.2 How do they work?

So after that commercial-looking babble, let's get to the basics. An ActionScript Event –which is the superclass of MouseEvent, among a zillion others- is any interaction happening in the Flash environment, whether it was a mouse click, the movement of the timeline to a new frame, the completion of the loading process of an external asset, or any other occurrence.

ActionScript can make any object react to any event by using an Event Listener. An event listener is basically a function created with the very specific purpose of reacting to an event. An object can react to an event using an event listener. This is done by using the .addEventListener method. This method simply registers an Event Listener and an Event to an object:

```
myObject.addEventListener(Event, eventListenerFunction);
```

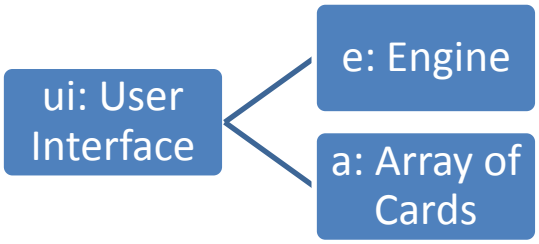
Our Event Listener will obviously have to be specified by declaring the function the same way any other function is declared in ActionScript, the only difference is that the listener function must have one argument to represent the event. This event argument can have any identifier as its name –as a non-written standard, over 90% AS3 developers use the letter 'e' for it:

```
function eventListenerFunction (e:Event):void{  /*Code*/  }
```

So for example, if we want our graphical objects –cards- placed on stage to act like a button by reacting to a mouse click over it, we can simply register an event and an event listener to it this way:

```
public class Card extends MovieClip {
    public function init(){
            addEventListener(MouseEvent.CLICK, clickHandler);
    }
    protected function clickHandler(e:MouseEvent){
            Registry.debuglog("Card.clickHandler " + nm );
            if ( check there's no restriction for the click )
                    /*  take some action  */
    }
}
```

Note, as you may have inferred by the first example, listeners do not need to be added to the object itself as in this Card example. For example, if we had these classes:



We could do something in the line of:

```
for each (Card c in a)
        c.addEventListener(MouseEvent.CLICK, e.clickHandler);
```

That is, from the ui instance we can add event listeners using known object references and tell them to dispatch to another object, the Engine e in the example. Note, however, that while the idea is cool we have two problems: it's usually a good idea to process the listener in the object that causes it and our game engine should not be processing loads of display-related data.

9.5 Getting the game engine to get the user input

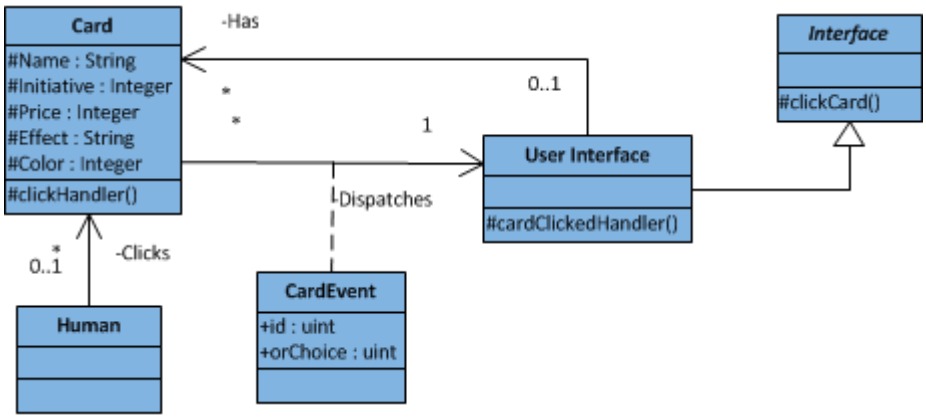
For once we're going to overlook the *do the simplest thing that could possibly work* we argued about before. The simplest thing would be to directly dispatch the MouseEvents to the engine. However, that implementation has shortcomings everywhere you look, the main being that the code isn't extensible or anything near it because the engine is processing exactly mouse events –which is mainly presentation layer data.

We would prefer the engine to process more general data. Why? Well, so far we've called User Interface to the User Interface. But I've been lying to you: it should be named Mouse Interface to be well-spoken.

Take, for example, a student after me wants to make a tablet version of the game: the engine won't be able to take the data from the TouchEvents that the touch screen creates, so the code will need to be redone or replicated into similar functions to treat data that, in the end, refers to the same objects, our cards.

That means we want to pass to the engine just the basic information of the card. We can get the the card ID easily from any kind of inputs: touch screens, mouse, digital pens. Or we could even give the engine ID after ID with a debugging algorithm to test outputs.

Behold, reader, as here's the solution:



Our Card objects are as we wrote before:

```
public function init(){
        addEventListener(MouseEvent.CLICK, clickHandler);
}
protected function clickHandler(e:MouseEvent){
        Registry.debuglog("Card.clickHandler " + nm );
        if ( check there's no restriction for the click ){
                var ce:CardEvent = new CardEvent(CardEvent.ON_CLICK);
                ce.id = this.id;
                dispatchEvent(ce);
        }
}
```

If we wanted to add capabilities for touch screen, we could just add in the init function:

```
addEventListener(TouchEvent.CLICK, clickHandler);
```

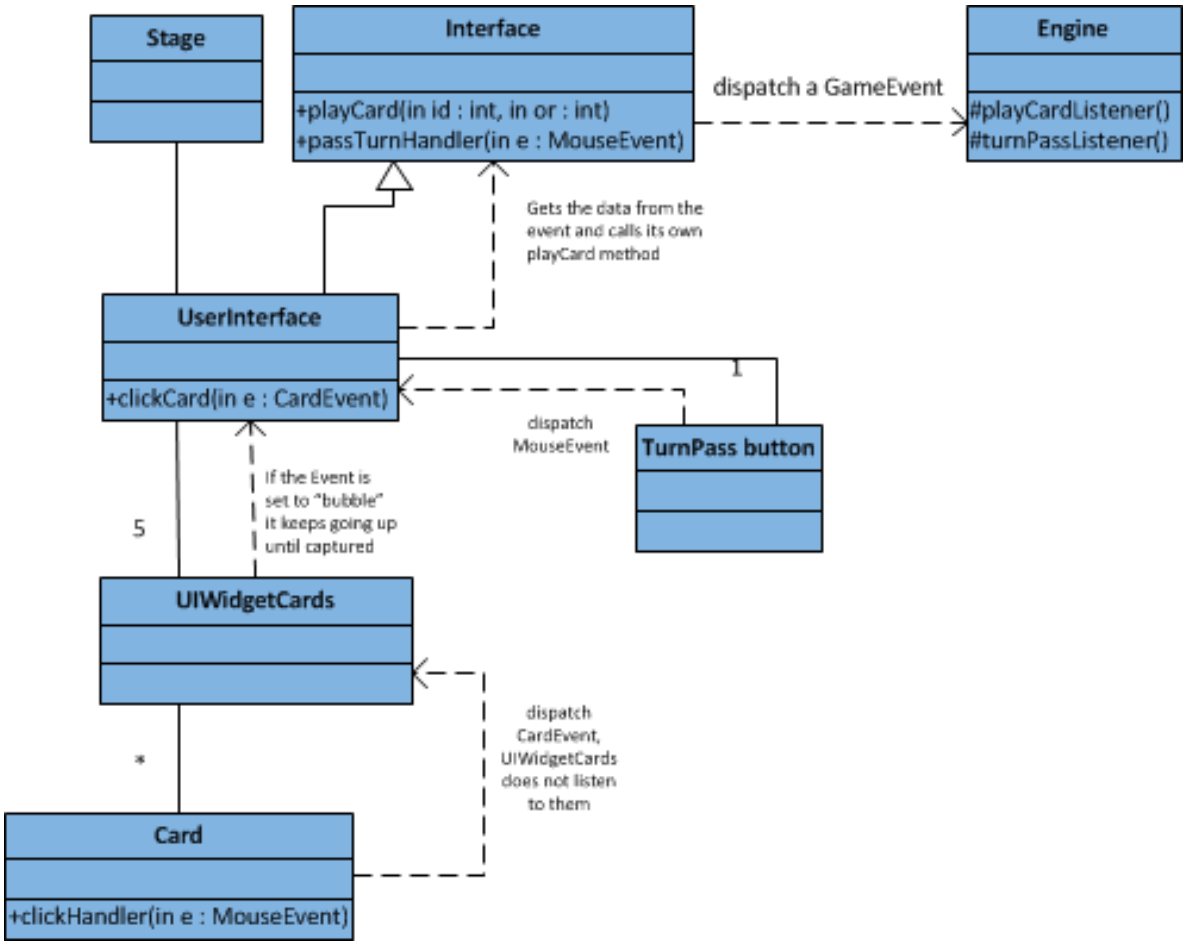
Although we would be better putting an if statement to add only whether one or the other listener because AS3 event listeners are as much resource-hungry as they are powerful. Note that what our clickHandler function does is to actually transform the input event –with the mouse in our program- into a CardEvent. What's that, you ask? An Event type we programmed ourselves for our presentation layer! We'll show them later.

Our UserInterface object, has the following code:

```
addEventListener(CardEvent.ON_CLICK, cardClickHandler);
protected function cardClickHandler(e:CardEvent){
        trace("UI.cardClickHandler");
        clickCard(e.cardID,e.cardOR);
}
```

Note how it listens to the same event type that the Cards dispatch. For the UI to catch the events, they must be launched from an object that is in its display tree. That is, an object listener won't magically react to an event it listens for.

By default, an event is only dispatched to it's direct parent in the display hierarchy, whether it is listening to it or not. AS3 events have a parametter known as bubble, that changes that behaviour. When an event is set to bubble it keeps scaling up the display hierarchy:



Putting it up together: every Card listens for MouseEvents caused by mouse clicks in them. The card then creates a CardEvent that contains the ID of the card that has been clicked and, if it's a card with multiple choice, the one that the player has chosen.

```
package as3.presentation
{
    import flash.events.Event;
    public class CardEvent extends Event
    {
            public static const ON_CLICK:String = "onClick";

            public var cardID:uint;
            public var cardOR:uint = 0;

            public function CardEvent(type:String,
                    bubbles:Boolean=true,
                    cancelable:Boolean=false):void
            {
              super(type, bubbles, cancelable);
            }
            override public function clone():Event {
                    // Return a new instance of this event with the same parameters.
                    return new CardEvent(type, bubbles, cancelable);
            }

    }
}
```

The CardEvent is the class we use inside the presentation layer to communicate card data. Note we define the bubbles parameter to true by default, since we'll be always using our Event in that setting. The cancelable parameter is not needed by us; setting to true allows to call the stopPropagation() and stopImmediatePropagation() methods later on. Both methods basically abort the bubbling process. We just add it –false by default- because the superclass creator requires it.

We do also need to override the Event's method clone(), just for the correct execution in the VM [24]. If we don't provide it then the superclass' function is called, which returns an Event instead of a CardEvent and strange things happen during execution. This clone() is used by the VM to do some of its magic during execution time.

The UserInterface class listens for CardEvents, as well as for when the button to pass turn is pressed:

```
public class UserInterface extends Interface{
	public function UserInterface() {
		super();
		Registry.screenLog.appendText("UI - being instantiated\n");
		//add childs
		Helpers.addMCat(enemyStats,0,0,this);
		Helpers.addMCat(town,0,293,this);
		/*
		…etcetera…
		*/
		addEventListener(Event.ADDED_TO_STAGE, addedToStageHandler);

	}
	private function addedToStageHandler(e:Event):void{
		Registry.screenLog.appendText("UI " + stage + "\n");
		Registry.stage = stage;
		removeEventListener(Event.ADDED_TO_STAGE, addedToStageHandler);
		addEventListener(CardEvent.ON_CLICK, cardClickHandler);
		pass.addEventListener(MouseEvent.CLICK, passTurnHandler);
	}
}
```

The UI creator initializes its variables and display objects and then listens for when it is added to the Stage, which is done listening to the Event.ADDED_TO_STAGE event –they are like a plague in AS3! This pattern follows the one we explained at some point: in AS3 we don't want the creators to really make the objects start rolling. In a DisplayObject in particular, there's no access to the Stage until it's added to it – unless, of course, we have a global reference to it and code the object to add itself to the display list.

Our click handler listeners is also in the UserInterface, as simple as follows:

```
public class UserInterface extends Interface{
	protected function cardClickHandler(e:CardEvent){
		Registry.dlog("UI.cardClickHandler");
		clickCard(e.cardID,e.cardOR);
	}
}
```

It just extracts the ID and possible option selected from the event and calls its method clickCard. Both this method and the handler for the pass turn button are implemented in the superclass:

```
public class Interface extends MovieClip{
```

```
        protected var table:Table;

        public function Interface() {}

        public function clickCard(_id:uint, _or:uint=1){
            Registry.dlog("Interface dispatching clickCard Event "+_id+"-"+_or);
            var _evt:GameEvent = new GameEvent(GameEvent.ON_CARD_CLICK);
            _evt.id = _id;
            _evt.choice = _or;
            dispatchEvent(_evt);
        }

        protected function passTurnHandler(me:MouseEvent){
            Registry.dlog("UI.passTurnHandler");
            passBlock();
            var _e:GameEvent = new GameEvent(GameEvent.ON_TURN_PASS);
            dispatchEvent(_e);
        }

        //virtuals
        public function mouseBlock(){}
        public function mouseUnblock(){}
        public function passBlock(){}
        public function passUnblock(){}
        public function allowOnlyHand(b:Boolean){}
        public function allowOnlyTown(b:Boolean){}
        public function allowOnlyEvents(b:Boolean){}
        public function reuseMode(b:Boolean){}
        public function addPreOKButton(){}
        public function update(){}
        public function setTable(t:Table){}
    }
```

As can be seen what both methods is to dispatch GameEvents, that are our domain layer events for which the game engine is listening. The Interface class does also have a dozen empty methods. That's because in the engine we don't want to be checking whether the Interface is implementing such things as actual graphics, so when it is an AI Interface things such asking to add a hover effect to better show, for example, the cards in hand won't have any effect.

It is also possible to circumvent the compiler crying if we don't ask it to go for an strict mode. If so, then it can oversee things such as calling a method that does not exist because we're kind of telling it that we know what are we doing –which in this case would be that, during execution time, that Interface method will exist because we'll be working with a subclass instance that has it. However, it oversees many things and can lead to a much harder debugging of simple typing errors, so it's better to keep a strict compiling mode and have those empty methods.

10 Iteration 4

10.1 Objective

Develop (multiplayer) communication between two game engines, at which point real interactivity will start.

Note the original proposed planning had the idea of implementing communications on iteration 5. The reason we advanced it is to try to implement a game engine serialization, so the game engine logic would get simpler to implement as, technically, there wouldn't "be" an opponent –a game state would be built from scratch, presented to the player, and then serialized again to send it. We'll explain in a following section we tried to do this with JSON and failed, thus we took another approach.

10.2 Backlog

B41 - Research how to communicate two flash applications.

B42 - Research and test different candidates

B43 - Implement the choice

10.3 Effort Chart

Perfect iteration schedule, if it wasn't for a crazy attempt at JSON, which is explained later in this section.

Ref	Concept	Estimate	Real
B41.T1	Research communication over the network	2	2
B42.T1	Research server candidates	5	5
B42.T2	Define communication protocol	3	3
B42.T3	Discarded JSON prototype	5	20
B43.T1	Implementation	15	15
	Total	30	45

10.4 Multiplayer over network

A multiplayer video game is a video game in which more than one person can play in the same game environment at the same time. Video games are often single-player activities that put the player against pre-programmed challenges and/or AI-controlled opponents, which often lack the flexibility and ingenuity of regular human thinking.

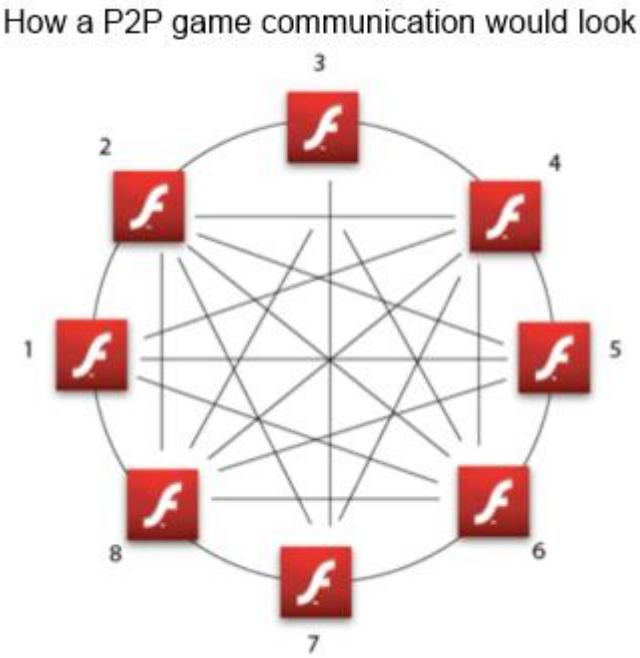
Multiplayers components allow players to enjoy interaction with other individuals, be it in the form of partnership or competition, which is our case. There are infinite options to communicate and synchronize multiple game clients, so let's get a bag of cookies before discussing about it.

10.4.1 Architecture: client-server vs P2P

Firstly, it's important to identify what we mean by multi-player and what implications that has. We're talking about non-local multi-player, over the internet rather than at the same computer. Of course this means there needs to be some kind of way for the players to communicate with each other over a network.

There are basically two architecture models to create a multiplayer game: client-server and peer to peer. The later, also called P2P, means there's no central overlord that dictates what's happening in the game nor relaying packets from one player's computer –or console- to another's.

Peer to peer architectures are often chosen for games because of a great benefit they have: there's no need to mount a server infrastructure and to later pay its electricity, bandwidth and maintenance bills; plus the multiplayer can live forever because it doesn't depend on the server service life –which is the time the game keeps being rentable. This model is the most used on non-AAA games just because it eliminates the big economic factor of complex server architectures.



Not everything are goodies, though, as P2P do also have detracting factors: it's a few magnitudes easier to hack a multiplayer game with a peer to peer architecture than a game with a central server that arbiters and oversees the game; and even when accomplished the server's owner just revokes the license of the players using hacks. This means most players just disregard playing P2P multiplayer with anyone that is not known by them that is trusted not to cheat.

All in all, a peer to peer architecture sounds like the choice for us, doesn't it? The answer is NO.

We've chosen the server-client model rather than peer-to-peer, because that's what our target platform client, Adobe Flash, supports. Bad luck, ours.

But wait, client-server architectures aren't that bad! A middleman server provides a security layer for the clients, as one doesn't even need to know the actual address of the adversary. Also, for a non real-time game like Dungeon Realms, a turn-based game, the server implementation is much easier than for a real-time one.

We don't have to worry about the concurrency problems that arise server-side with a multiplayer game with real-time clients. Goodbye to the worries about lag and his friends: a human player does already take quite some time thinking a move –it's not a problem even if a busy server adds a couple seconds after a player has taken 20 to play a card. It's impossible to notice.

Furthermore, even if it's not in the reach of this project, this architecture leaves the doors open to building a server for the game that automatically finds and matches opponents, or that has a chat lobby for players to manually challenge each other. Players could even get registered and maintain an ELO ranking and whatever the developer wanted.

10.4.2 TCP vs UPD

10.4.2.1 TCP

TCP stands for "transmission control protocol". Together with the lower-tiered Internet Protocol (IP) it forms the backbone for almost everything you do online, from email to managing your bank account online, it's all built on top of TCP/IP.

If you, the reader, have ever used a TCP socket, then you'll know that it is a reliable connection based protocol. This simply means that you make a connection between two machines and you send data between the two computers much like you are writing to a file on one side, and reading from a file on the other.

This connection is reliable and ordered, meaning that all data you send is guaranteed to arrive at the other side in the same order that you wrote it. It's also a stream of data, meaning that TCP takes care of splitting up your data into packets and sending those across the network for you.

10.4.2.2 UDP

Instead of treating communications between computers like writing to files, what if we want to send and receive packets directly? We can do this using UDP. UDP stands for "user datagram protocol" and it is another protocol built on top of IP, just like TCP, but this time instead of adding lots of features and complexity it is just a very thin layer over IP.

With UDP we can send a packet to a destination IP address and port, and it will get passed from computer to computer until it arrives at the destination computer or is lost along the way. UDP packets are like the bomber aircrafts of World War II: you launch them and just hope that they will get through to their destination.

On the receiver side, we just sit there listening on a specific port and when a packet arrives from any computer (remember there are no connections!), we get notified of the address and port of the computer that sent the packet, the size of the packet, and can read the packet data.

UDP is actually an unreliable protocol. In practice, most packets that are sent will get through, but you'll usually have around 1-5% packet loss, and occasionally you'll get periods where no packets get through at all -there are lots of computers between you and your destination where things can go wrong.

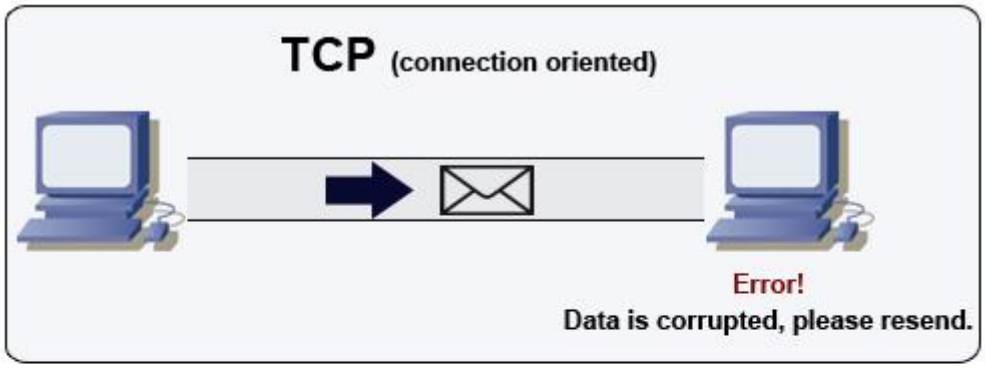
There is also no guarantee of ordering of packets. You could send 5 packets in order 1,2,3,4,5 and they could arrive completely out of order like 3,1,2,5,4. In practice, they will actually arrive in order almost all of the time, but again, you cannot rely on this!

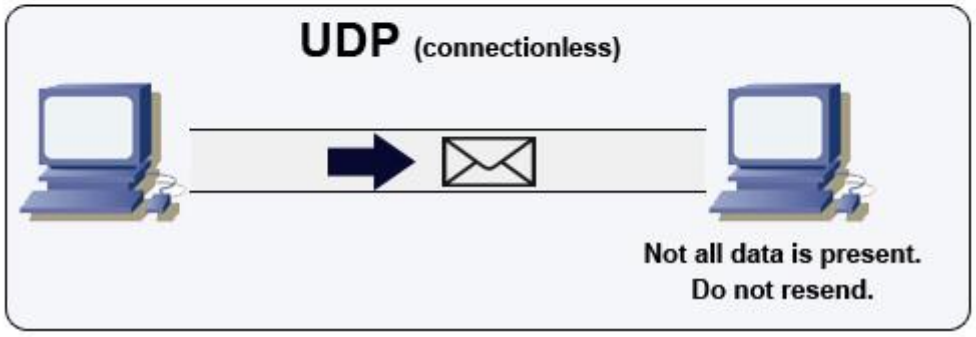
Finally, although UDP doesn't do much on top of IP, it does make one guarantee for you. If you send a packet, it will either arrive in whole at the destination, or not arrive at all. So if you send a 256 byte packet to another computer, that computer cannot receive just the first 100 bytes of the packet, it must get the full 256 bytes of data. This is pretty much the only guarantee you get with UDP, everything else is up to you!

10.4.2.3 Choices, choices, choices!

Just kidding: the choice of whether to use TCP or UDP communication protocols has also been made for us by the fact that Flash only supports TCP. So you had to read a page resuming each protocol; sorry for that, but knowledge is power.

Being limited to TCP is a time-bomb if you want to develop a fast paced multi-player game in Flash because, in it, the protocol uses Nagle's algorithm [25] which tries to group separate packets of data together into larger payloads before actually sending them, which can introduce lag. To that, you sum each packet requires ACKs, introducing extra ping, and goodbye to the real-time experience. Well, not only with Flash, but with any implementation language you'd want to use UDP.





Just for the record, with UDP we would just stream the data and if something gets lost or corrupted, well, the other side will have to manage that. Most decent real-time multiplayer engines do actually mix both protocols. For example, when you fire a bullet at a player -in a shooter-, your objective's coordinates have been received by UDP, and so probably are your bullet direction and speed. The server will do its calculations and you will probably rant when it fails: a packet telling you fired three bullets in this frame may just fail to arrive or do it a couple seconds late; it may also happen that your game client was showing your enemy at some coordinates that weren't actually the ones that corresponded to that frame but to a few sooner. Sometimes there's a bigger desynchronization and you just see a character randomly moving around an approximated trajectory –the packets are arriving very unordered for a while.

However, at the end of the day using TCP with Flash it is possible to disable Nagle and thereby remove this lag inducing restriction, so all wouldn't be lost if connection lag was a priority –and that's actually what developers do to implement that type of game. But remember we didn't care about lag? Good luck, ours!

In any case, at the end of the day anyone set on using UDP for their game will have to implement their own version of TCP's reliable, in-order delivery policy anyway, so once Nagle has been disabled there really isn't that much separating TCP and UDP in terms of raw speed.

10.4.3 Client-server action synchronization

Any game has calculations that occur to determine the result of actions. These can range from combat calculations to important economic transactions involving game items or deciding which unit and in which second to put to the production queue. If it's a priority to prevent players cheating, it's important that these calculations are not done on the gamer's computer, because they can easily modify the result of such calculations.

Due to the speed of light and other physical limitations, it's not instant to send or receive data from other computers. As an example, here in Spain we typically see response times between machines of around 50-250ms. Note we have ultra-crappy connections, though, considering we're a country that it's in the Champions League of the economy.

All online games have this same situation that affects any internet connection. In games where the server controls the game, it has to dictate whether things happen or not, but there's a 50-250ms delay –to put our numbers- before data gets to the server and back. There are three ways that games can solve this:

- Trust the client. This means people can cheat, but the results are instant. It's the easiest approach because there is no need to implement the game mechanics on both ends and there's no need of extra code to check if actions are permitted.
- Wait until data arrives back from the server before doing anything [26]. This is a very common strategy in RTS games. If you click to move, the unit will only start moving once the server says so, which is 50-250ms later. If you are on the lower end of the ping, you quickly get used to the small lag and everything feels pretty good. If you're on the bad end it feels like you're playing drunk.
- Start predicting the result of the action as though the server said yes, immediately. When the server later gets back to you with a result, factor it in. This is what good games do. It means that when you click to move, or click to attack, it occurs instantly and feels great. The problem is what happens when the server decides that the action can't have occurred - that's when the game gets badly out of sync and the games that use this solutions need a lot of server code to correct it –for example with rubber-banding.

Since Dungeon Realms is a turn-based game, the latter option is definitely out of the equation. It would mean implementing in the client just a dumb graphic interface that, when you clicked to play a card, it played and, maybe after 250ms, it returned to its place because the server considered it an impossible move.

That means it is between the first and second options. With the second option we'd need the server side to dictate actions, what we don't really want because it will later become a hassle for us: you still don't know it at this point of the reading, but the server won't be implemented in the same language than the client –that means duplicating code, and bugs, in two different languages.

What leaves us with the former option, which is fine because we basically want to use the server as a relay between both clients.

10.5 The server

10.5.1 Network sockets

A network socket is network interface- an endpoint of an inter-process communication flow across a computer network. A socket's address is the combination of an IP address and a port number, much like one end of a telephone connection is the combination of a phone number and a particular extension. Based on this address, network sockets deliver incoming data packets to the appropriate application process or thread that listens to the corresponding port.

Sockets are accessed with an application programming interface (API), usually provided by the operating system, that allows application programs to control and use them. This means that, using sockets, we don't need to program low-level networking functionalities and we aren't tied to an specific programming language.

Network sockets can be used through every mainstream language, and the communication is independent of the implementation of each side. An android tablet can use a socket with a Java program to communicate with a supercomputer server socket programmed in Fortran, just to mention some extremes. That's basically why almost all projects go for them.

Note, however, from an executing program point of view sockets are very similar to system files: you can write data to them and you can read data from them. This means there are no functionalities over them and is job for the developer to know how to separate, classify and process the data that arrives. That is, a socket is structureless.

10.5.2 Server implementation language

The socket approach means we have a wide –tendig to infinite- amount of choices for the server. So, our first decision is to leave out any option that is not thought to communicate flash applications. This leaves us without the typical Java and C++ choices for socket servers, since there are other, more focused on our task, options. Let's take a glance.

Platform	GPL	Cost	Complexity	Coolness
Flash Media Server	No	Starts at $995,	Hard	Flash community stops talking to you if you use it.
SmartFoxServer	No	Starts at 200€, which limits the number of connections to 100.	Medium	Uncool. A real game server needs a >1000€ license.
Node.js	Yes: MIT	Free	Easy	Very cool, and trendy.
Socket.io	Yes: MIT	Free	Medium	Kind of Node.js on steroids.

10.5.2.1 Flash Media Server

FMS is the most expensive solution and only really used few scenarios like Facebook games developed with Flash. If we could use it, there are some great functionalities that allows to share object instances between game clients.

That means a client would play a card, the engine would process it and, automatically –no socket messing anywhere- the other client would see the result in its screen. Basically, we could implement it in a way that all the engine and data was shared between both clients and that each client had a unique UserInterface that displayed the cards in the correct way –the hand of each player at each end, and the grounds swapped up-and-down.

However, the basic Standard FMS license costs $1000 and has a lot of restrictions. The Professional license costs $4500 and, finally, the unlimited license has an unknown price: you've got to contact Adobe and sign a NDA to negotiate.

10.5.2.2 SmartFoxServer

SmartFoxServer is a platform for rapidly developing multi-user applications and games with Adobe Flash/Flex/Air, HTML5, Android, Unity3D, Apple iOS and Java and comes with a rich set of features –which also makes it quite hard to start using due to the overwhelming numbers of possibilities. SFS is the typical choice for medium-sized multiplayer games that need a tested platform from which to build up the concurrency.

The prices start at 200€ and go up to €3500 for the full-fledged server.

10.5.2.3 Node.js

Node.js is a server-side software system created in 2009 and designed for writing scalable Internet applications, notably web servers. Programs are written on the server side in JavaScript, using event-driven, asynchronous I/O to minimize overhead and maximize scalability. It contains a built-in HTTP server library, making it possible to run a web server without the use of external software, such as Apache or Lighttpd, and allowing more control of how the web server works.

Concretely, Node.js is a packaged compilation of Google's V8 JavaScript engine, the libuv platform abstraction layer, and a core library, which is itself primarily written in JavaScript. Unlike most JavaScript programs, it is not executed in a web browser, but instead as a server-side JavaScript application.

It's totally free to use thanks to its MIT license and, currently, it's a very used solution by developers that aren't afraid of platforms without extensive documentations –the ones that actually help computing evolve. The reason it's that it is, in almost every scenario, the most lightweight server available so the resources required for high concurrencies are very low.

10.5.2.4 Socket.io

Socket.IO is a JavaScript library for realtime web applications. It has two parts: a client-side library that runs in the browser, and a server-side library for node.js. Both components have a nearly identical API. Like node.js, it is event-driven.

Socket.IO primarily uses the WebSocket protocol, but if needed can fallback on multiple other methods, such as Adobe Flash sockets, JSONP polling, and AJAX long polling, while continuing to provide the same

interface. Although it can be used as simply a wrapper for WebSocket, it provides many more features, including broadcasting to multiple sockets, storing data associated with each client, and asynchronous I/O.

Since we want the server to relay packets between the clients, all the extra client-side functionalities it adds on top of node.js are pretty much jitter to us.

10.5.3 What to send through the sockets

Out of the candidates, Flash Media Server and Smart Fox Server are out of the equation. Once and if we had a game to enter into a production and marketing stage, SFS would likely be an option to consider since it's got even chat, game lobbies and many other functionalities the average user expects on a multiplayer game.

However, we want a much simpler approach to the network functionalities since we don't need any extra functionalities to actually play a game between two humans. That leaves us with Node.js and Socket.io. The latter is, at the end of the day, a built up around Node.js to provide extra methods to games developed with Flash, Android, iOS and such.

We've elected Node.js, out of the two, due to:

- It's our first approach ever to JavaScript, the language behind both, and it's better so start from the basics.
- Node.js is simpler to start with: no API documentation to read for game-specific calls.
- In Node.js we can use the basics, so compatibility with future language version seems pretty much granted. Meanwhile, using Socket.io's young API may make the game unplayable to anyone that wants to try the game in the future.

All in all, it seems a bad idea to use Socket.io without having ever used Node.js. Let's see how we transmit data.

10.5.3.1 JSON – our first (failed) choice

JSON a RFC standard to code information. JavaScript Object Notation is a text-based open standard designed for human-readable data interchange. It is derived from the JavaScript scripting language for representing simple data structures and associative arrays, called objects. Despite its relationship to JavaScript, it is language-independent, with parsers available for many languages.

The JSON format is often used for serializing and transmitting structured data over a network connection. It is used primarily to transmit data between a server and web application, serving as an alternative to XML.

This was our original choice to sync the game. The idea was to code game instances into JSON using the as3corelib library [27] and transmit them to the other client to then rebuild the instances in the opponent's machine.

While the idea was cool –transmit the game's state coded as JSON-, we had to drop it after a week worth of tries to make it work correctly. The vast amount of pointers the game has made it quite a job to do things correctly, not to account all the pointers the Flash VM makes behind the scenes for the DisplayList and event listeners.

10.5.3.2 Yet again thinking simple things than can work

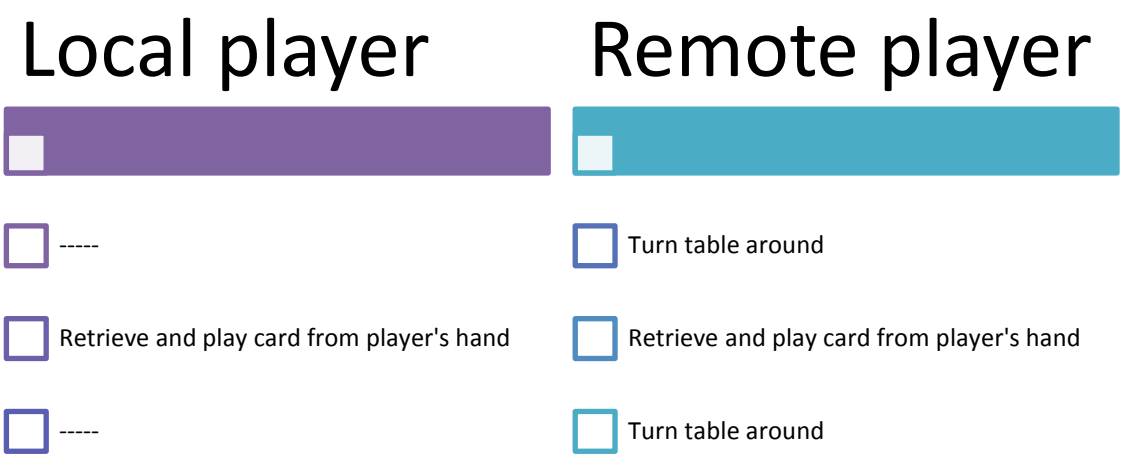
After the failed approach with JSON, we decided to take a whole new approach. Why try to rebuild whole game states, as if they were savegames, from plain text data? Yes, it's truth that this approach would have actually allowed to directly implement functionalities such as play by mail –save JSON to a file, send it by mail, and load it in the other client.

However, do we need it? No. We just need to get player actions synchronized at both ends. And we could do it simpler that with the JSON approach. It's already written in the last line: get player actions synchronized.

Remember our engine got just a card ID –and possible 'or' operator choice- from the presentation layer? What if it also got the same information from the data, socket, layer?

Going further still: if we could, in some way, virtually turn the game table so that we played the opponent's actions as if they were the local one's… Ah, yes, we can actually do that: we're the developers and we can mess around with our data structures!

So, for example, to play a card from hand:



That is, we could use the same functions and algorithms that, from their point of view, would access and mess with exactly the same data structures –they would never need to touch any data structure that contains opponent's cards.

10.6 Packets to synchronize the clients actions

As with any modern language, AS3 provides socket support through the API. To instantiate the class that will contain the actual socket connection, we do:

```
import flash.net.Socket;

package as3.data {

        protected var s:Socket = new Socket();

        public function SocketLord() {
                s.addEventListener(IOErrorEvent.IO_ERROR, handlerIO_Error);
                s.addEventListener(SecurityErrorEvent.SECURITY_ERROR,
                                                        handlerSecurity_Error);
```

```
            s.addEventListener(Event.CONNECT, handlerConnect);
            s.connect(serverIP, serverPort);
        }
    }
```

Yet again there are listeners around: we've already told that AS3 relies heavily on them and that it was the reason many non-expert developers fail at it. We have to add a couple listeners for possible errors, and another that will get an event when the connection is correctly done. We then call the method .connect(IP, port).

```
        protected function handlerConnect(e:Event){
            Registry.screenLog.appendText("Succesfully connected to server");
            s.removeEventListener(Event.CONNECT, handlerConnect);
            s.addEventListener(ProgressEvent.SOCKET_DATA, handlerTCP);

            timer = new Timer(1000);
            timer.addEventListener(TimerEvent.TIMER, handlerOut);
            timer.start();
        }
```

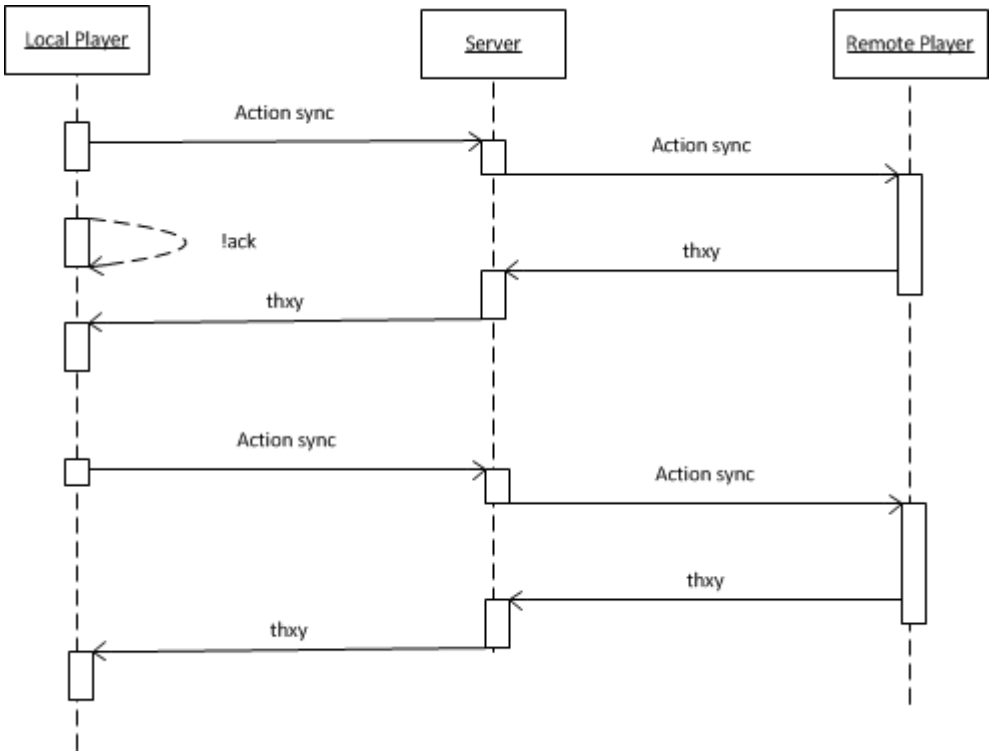
Once our connected event gets to it, we remove the listeners –resource awareness- and add a listener that will get a ProgressEvent.SOCKET_DATA event whenever new data gets to the socket . We also start a timer that, every 1000 milliseconds, will send out data to the server if there's anything to be sent. Remember that when we started the multiplayer discussion we told that a low ping was not a necessity: player's already take a while playing a card, so adding half a second is not a deal.

Furthermore, the decision was actually taken after noticing that, if a player quickly played cards, the opponent's had some confusion with the order of events when he saw, for example, three cards disappearing from the event line, a couple getting to the opponent's playing area and the px and gold numbers increasing or decreasing. All of that in, maybe, just a couple seconds. Adding a forced delay, of up to a second, between moves makes the game much more understandable for the player that is currently idling.

So, our socket output function looks like this:

```
        protected var outgoing:Array = new Array();
        protected var ack:Boolean = true;
        protected function handlerOut(e:TimerEvent){
            if(!ack) return;
            if(outgoing.length == 0) return;
            Registry.dlog("SocketLord.handlerOut " + outgoing[0]);
            s.writeUTFBytes(outgoing.shift());
            s.flush();
            ack = false;
        }
```

We have an array were the actions to be sent are queued and a Boolean that tells us if the remote player has already confirmed he has received last actions. If so, and if there is an action to be sent next, we send it. The following sequence diagram illustrates the simple sync algorithm



The local player sends an action to be synced, that is relayed through the server to the remote player. Then the output function may be fired again, but since the packet hasn't still been acknowledged, it waits. After an indeterministic amount of time, the server relays a THXY packet that the remote player has sent. Since developers are very polite persons, our ACK packets are actually coded as THXY, which stands for 'thank you'. After it, the local player can send the next action.

So, how does it work in the receiving end? Make some memory and recall we had wrote:

```
s.addEventListener(ProgressEvent.SOCKET_DATA, handlerTCP);
```

When we get an input packet, we get an event dispatched to our handlerTCP function.

```
protected var _beenVerified:Boolean = false;
protected function handlerTCP(e:ProgressEvent){
        Registry.dlog("SocketLord.handlerTCP");
        var _l:uint = s.bytesAvailable;
        if(_beenVerified)
        {
                /* code */
        }
        else
        {
                _beenVerified = true;
                trace(s.readUTFBytes(_l));
                s.writeUTFBytes("OK");
                s.flush();
        }
}
```

The first ever packet that gets in will go to the else at the bottom. That else is basically a way to circumvent a Flash behavior. You see, when a Flash/AIR program connects to a server it asks for a policy file, which is an XML that contains to which ports of the server the program can connects, if cross-domain is allowed, and optional diverse security information. It's basically a way to avoid DDoS attacks using Flash

programs embedded in web pages. So, when the client first connects to a server, it asks for the policy file –if it doesn't get one, a security error event is thrown.

After getting the policy, there's a random behavior –<irony>Adobe's known excellent implementations</irony>-where the socket either reconnects –with a new local port- to the server using the data from the policy file, or it stays connected using too, the data from it –unless it first connected to a port not allowed by the policy file, in which case the Socket would disconnect as per the API design.

So, if you ever try the game and see that two consecutive connections appear in the server from a same client, you now know it's not a bug by our part: it is an implicit behavior of AS3. By the way, this brought quite a bit of headaches to this poor guy that didn't understand where the problem was while debugging, until it was discovered it was not a bug of the game.

The solution to avoid the problems it caused? To implement a way in the server to differentiate whether a new connection needed a new policy file to be sent or not. We've done it with an if statement, although we could have also implemented two separated data input event handlers: one that would only be executed the first time, and that would then change the listener to the other method, that would contain the code inside the if.

```
var server = net.createServer( function(socket)
{
	var socketData = "";
	var player;
	/*
	Code
	*/
	socket.write(policy()+'\0');
	socket.on('data', on_policy_check);
	/*
	Code
	*/
}
function policy() {
	var xml = '<?xml version="1.0"?>\n<!DOCTYPE cross-domain-policy SYSTEM'
	xml += ' "http://www.macromedia.com/xml/dtds/cross-domain-policy.dtd">\n'
	xml+= <cross-domain-policy>\n';
	xml += '<allow-access-from domain="*" to-ports="*"/>\n';
	xml += '</cross-domain-policy>\n';
	return xml;
}
```

Whenever a new connection arrives, we call the policy() method, that returns a basic policy file for the client, and write it to the socket. Then we add a listener –they're here, too!- with the socket.on call, linking the 'data' event, that arrives whenever there is incoming data in the socket, to the on_policy_check() method.

What this function does is, first, change the listener for incoming packets to call another method, on_data(). We check that the same client isn't asking twice for the same policy file, scenario in which the file configuration is wrong or corrupt. We do also save the new client's IP and port, as well as a pointer to the socket itself. We do this to be able to pass the packets that arrive to one socket instance to the socket that will write the same in the other end.

```
function on_policy_check(data) {
	console.log("  client answer to policy: " + data);
	socket.removeListener('data', on_policy_check);
	socket.on('data', on_data);

	if(data == '<policy-file-request/>\0'){
		console.log("Error: client requested policy file twice?");
	}
	else{
		if(nPlayers < 2){
			playerData[nPlayers][_IP] = socket.remoteAddress;
			playerData[nPlayers][_PORT] = socket.remotePort;
			playerData[nPlayers][_SOCK] = socket;
			console.log("  sending seed");
			socket.write("0016Seed"+gameseed+"000"+nPlayers);
			player = nPlayers;
			nPlayers++;
			console.log("  assigned as PLAYER " + (player+1) +"\n");
		}
		if(nPlayers == 2){
			//resetAcks();
		}
	}
}
```

We also assign a number to the player and add 1 the total number of connected players. Lastly, we seed the game seed to the client together with his player number -1 or 2. The game seed is a natural number that is used by the client to generate a game. The seed is used in the random number generator for two reasons: to be able to play exactly the same game two times for testing, and to be able to create infinite game possibilities through adding randomness in the card shuffling with this seed.

So, back to when the policy file bureaucracy is done, we had the main if block:

```
		if(_beenVerified)
		{
			buffer.writeUTFBytes(s.readUTFBytes(_l));
			if(buffer.length >= 4 && buffLen == 0){
				//new packet
				buffer.position = 0;
				buffLen = uint(buffer.readUTFBytes(4));
				buffer.position = _l;
			}

			if(buffer.length == buffLen){
				//packet completed
				processPayload();
				buffer.clear();
				buffLen = 0;
			}
			else if(buffer.length > buffLen){
				this.write(buffer.toString());
				throw new Error("Handler TCP buffer overflown");
			}
		}
```

In the first line we read the data available from the socket and save it into a buffer, which is a ByteArray. This class provides methods and properties to optimize reading, writing, and working with binary data. According to Adobe, the ByteArray [28] class is only for advanced developers who need to access data on the byte level. So, yeepi! I'm an advanced AS3 developer!

In-memory data is a packed array (the most compact representation for the data type) of bytes, but an instance of the ByteArray class can be manipulated with the standard [] (array access) operators. It also can be read and written to as an in-memory file, using methods similar to those in the URLStream and Socket classes. It even provides methods to pack the data with zlib, deflate, and lzma compression. So it's quite a powerful class to manage byte data, that is what our sockets send and receive.

As when managing the access of a file in most languages APIs, here we've got a position variable in the buffer instance to put the I/O pointer where we want. When we've got at least 4 bytes -32 bits- in the socket we start looking at what's in the message. Why? Because we precede every message with the total length, in bytes, of the packets. For example, the previous THXY ack packets are actually sent as received as the string '0008THXY'. It's the way to know when a packet payload is finished.

So we look at the first 4 bytes and then we know how long the message will be, which is stored in the buffLen var. At the point where the total buffer length is the one we read in the first four bytes, we can proceed to process the packet in the processPayload() method to then clear the buffer for the next packet.

10.6.1 Packets: big or small?

We've got quite a bit of different packets in Dungeon Realms due to the complexity of some actions. For example, let's say we have an arrow operator where you can discard up to three cards and then draw up to three cards. How's the use case?

1. Click the card
2. Choose how many cards we want to discard.
3. Click every card we want to discard.
4. Draw the same number of cards.

To synchronize it we had to choose:

- Send all steps at once: the packet would contain the packet length, followed by the playcard action, followed by the number of cards to discard, and followed by every card id to discard.
- Atomize the transactions: send a packet to place the card in the playground, then send a packet for every discarded card, then send a packet to confirm and sync the number of cards drawn.

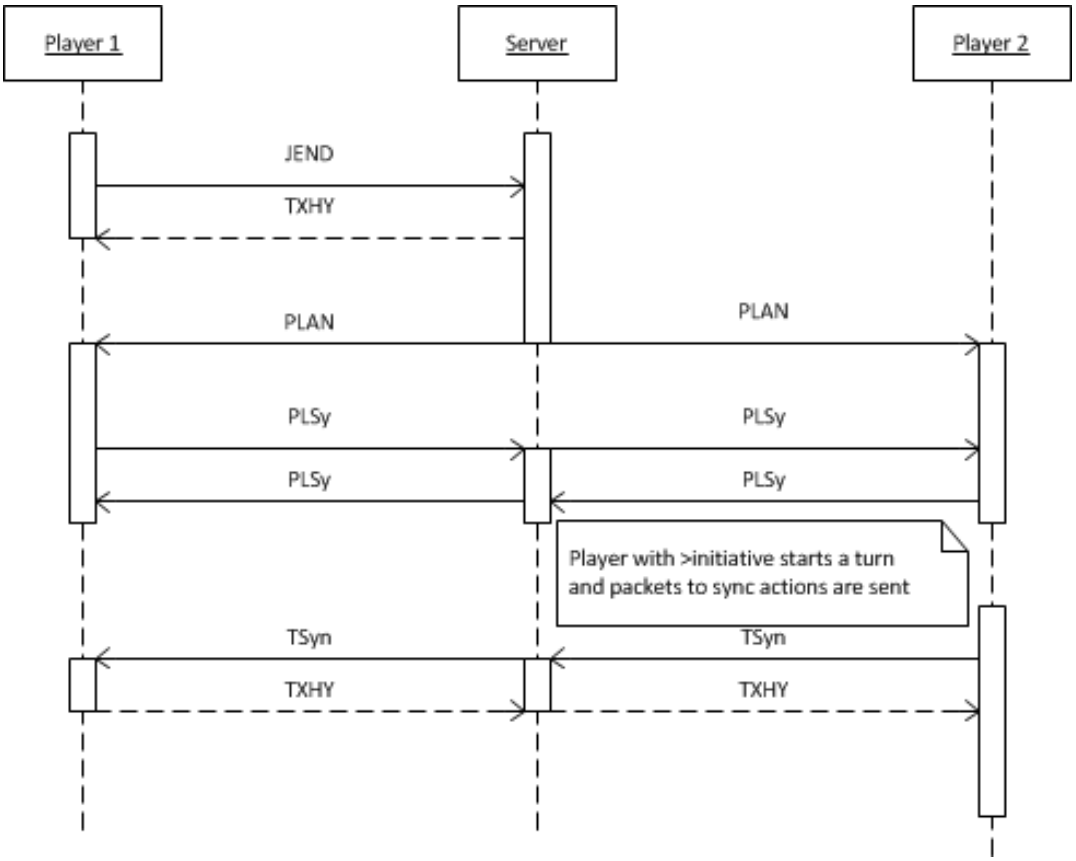
We chose the second option because it's easier to process small actions one by one that to implement a for algorithm converting card's id to the actions to take for each one. It's also more natural to the way the engine works: it responds to every user action, so we synchronize each step the engine takes one by one. It's also more natural for the player watching the opponent's actions to see them as they happen instead of getting a huge change in the game table at once.

10.6.2 Packet types

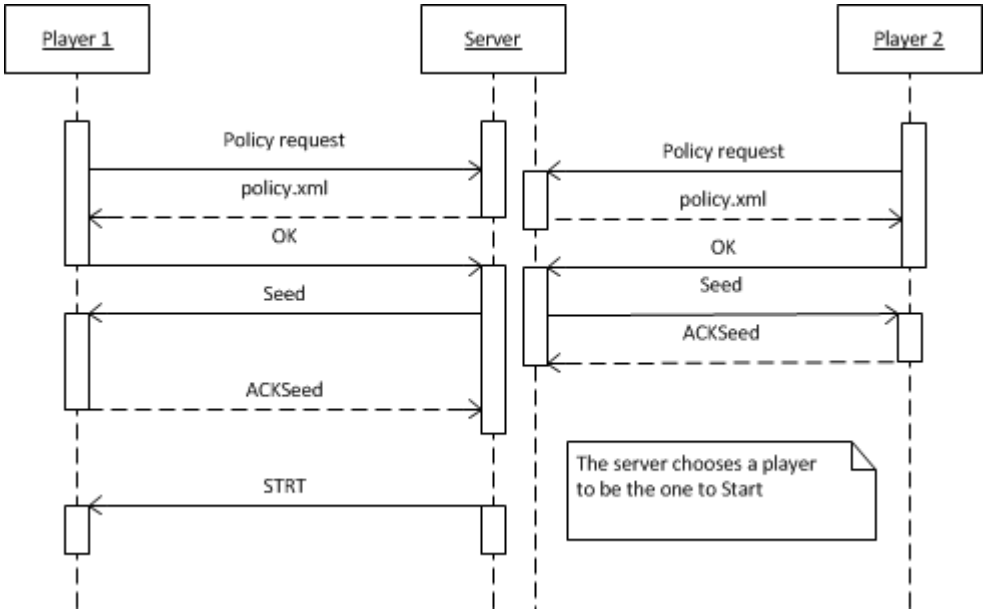
Here are the (atomized) packets we use to synchronize both clients

Packet length	Packet header	Purpose	Bytes 9-12	Bytes 13-16	Bytes 17-20
0016	Seed	Game creation	#seed	#player	
0008	STRT	Game can STaRT			
0008	Plan	Starting plan phase			
0012	PlSy	Synching plan phase	Card id		
0016	TSyn	Turn SYNc action	Card id	Or operator choice	
0012	TSBu	Turn Sync: BUy card	Card id		
0012	TSNu	TS: NUll town	Card id		
0012	TSRe	TS: REuse effect	FSA		FAS
0012	TSSH	TS: Save Hand effect	Card id		
0020	TSPF	TS: Pre Full, for arrow operator that don't to select cards.	Card id	#actions	Type of effect
0016	TSPC	TS: Pre Click, for arrow operator that need to select cards.	Card id	Or operator choice	
0020	TSPE	TS: Pre End, arrow operator finished	Card id	*unused	#actions
0012	TSPS	TS: Pre Start, pre is finished, effect can start.	Card id		
0008	TEND	Turn ENDed			
0008	JEND	Journey ENDed			
0008	THXY	Acknowledgement packet			

The JEND packet is special in the sense that it is never received by a client. When a journey ends the server gets two JEND packets and then a plan phase starts where both clients have to select a card and the one with a higher initiative starts proceeds to start the turn –illustrated by the TSyn packet that represents the player with higher initiative started a turn and used a card. The flow is as follows:

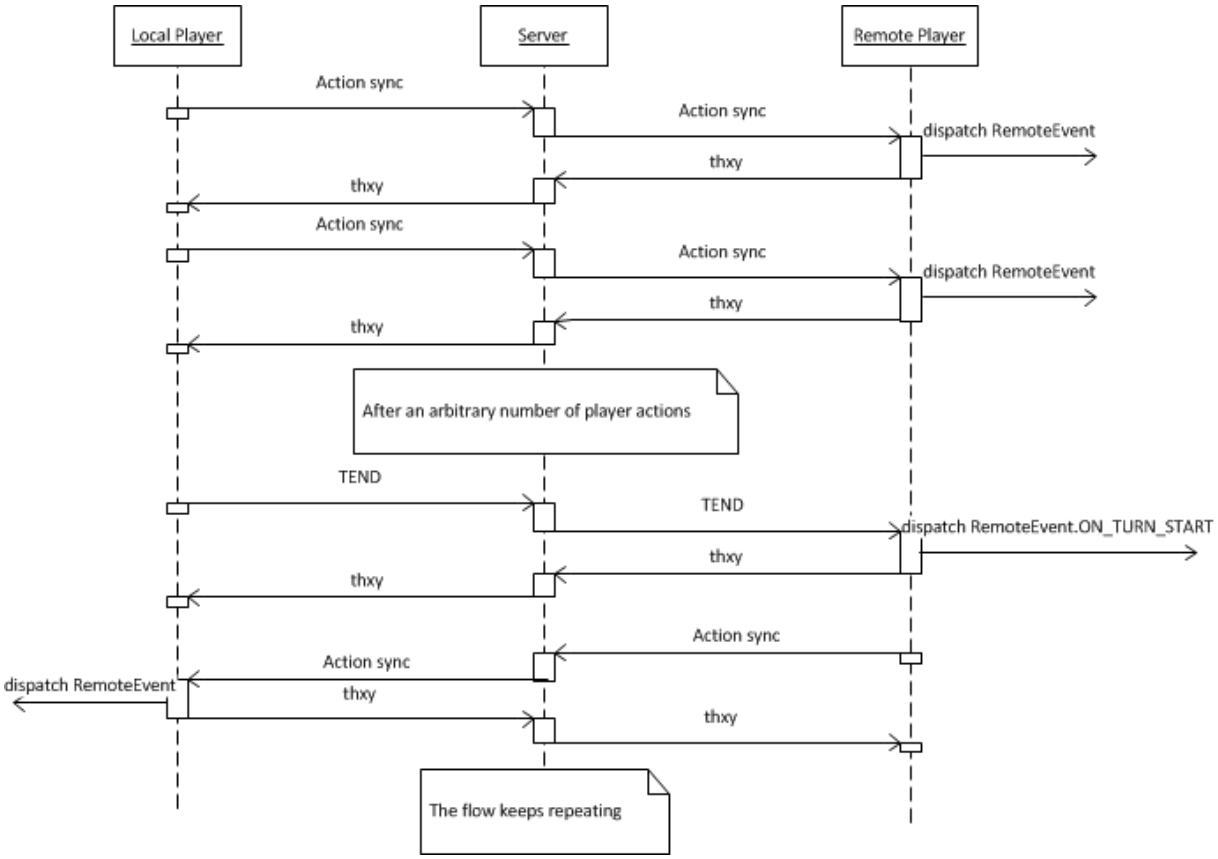


The other special scenario is at game start. The Seed packet is only sent by the server, and that's done after the policy file check.



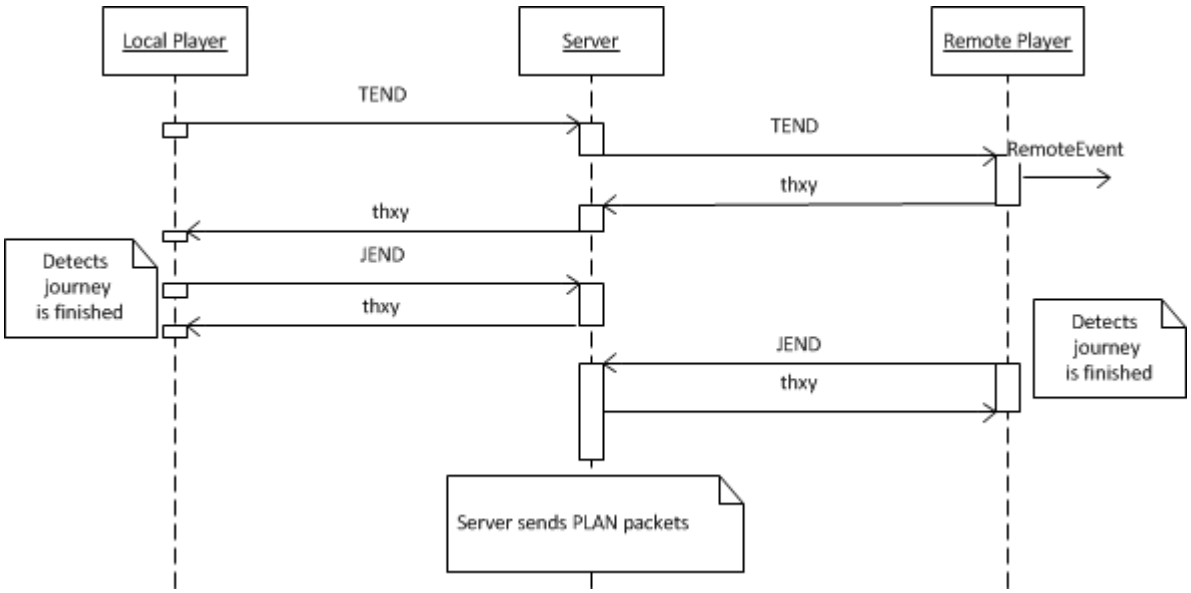
Note that the two lifelines under the Server mean to represent how the networking with each client is totally independent. Policy requests can arrive at any moment and Seed packets too. When the server has gotten an acknowledgement that both clients have finished starting up the engine with the given seed, then it sends to one a packet to start the game –STRT.

Lastly, the normal synchronization flow, is as follows:



As you can see, the active player sends the packets we saw before that are used to synchronize actions one after one. It waits to receive an acknowledgement before sending the next. When the player chooses to end its turn, a TEND packet is sent, that tells the other player he can start his turn. But, what happens if that player can't start a turn because he has no followers left? Then both sides detect there are no followers left on either side –or there are no free towns- and send a JEND packet to the server.

The server then proceeds to send PLAN packets to both players, as we saw at the start of the section.



10.7 The game engine POV

Lastly, how does the engine make sense of all of that? You fear the answer and you're correct: events and listeners. Hooray! But first, we'll take a glance at how to send packets, which is easier.

We have a write method that queues outgoing packets in our output buffer that we explained before – the one that tries to write on the socket once a second.

```
protected function write(_data:String){
        Registry.dlog("enqueuing in socket " + _data);
        outgoing.push(_data);
}
```

The method is protected because we have a public function for each type of packet. For example:

```
public function journeyEnd(){
        Registry.dlog("SocketLord.journeyEnd");
        write("JEND");
}

public function endTurn(){
        Registry.dlog("SocketLord.endTurn");
        write("0008TEND");
}
public function playCard(id:uint, o:uint){
        Registry.dlog("SocketLord.playCard");
        var _id:String = uToString(id);
        var _or:String = uToString(o);
        write("0016TSyn"+_id+_or);
}
protected function uToString(n:uint):String{
        if(n < 10) return "000"+n;
        if(n < 100) return "00"+n;
        if(n < 1000) return "0"+n;
        throw new Error("SocketLord.uToString overflow " +n);
}
```

We've copied three of the packet's functions here. The JEND that is meant for the server and hence is not preceded by its length. The TEND packet that, this time, is preceded by its length but has no extra information after the packet type. Lastly, there's the TSyn packet that consists of the length, 16, the packet type, the id of the card played and the effect selected by the player. The uToString function forms correctly the number to occupy 32 bits by appending the correct number of 0 before each one.

At the receiving end, the processPayload method where we left things a while ago, has a switch that differentiates incoming packets:

```
protected function processPayload(){
        Registry.dlog("SocketLord.processPayload\n   " + buffer.toString());
        var _evt:RemoteEvent;
        var msg:String;

        buffer.position = 4;
        var sss:String = buffer.readUTFBytes(4)
        switch(sss){
                case "TEND":
                        msg = "SL dispatching ON_TURN_START";
                        _evt = new RemoteEvent(RemoteEvent.ON_TURN_START);
                        break;
                case "TSyn":
```

```
                                msg = "SL dispatching ON_TURN_SYNC";
                                _evt = new RemoteEvent(RemoteEvent.ON_TURN_SYNC);
                                _evt.id = uint(buffer.readUTFBytes(4));
                                _evt.choice = uint(buffer.readUTFBytes(4));
                                break;
                        /* other cases */
                }
                if(sss != "Seed" && sss != "Plan" && sss!="PlSy"){
                        s.writeUTFBytes("0008thxy");
                        s.flush();
                }
                Registry.dlog(msg);
                dispatchEvent(_evt);
        }
```

We position the buffer I/O pointer at byte 4 and read 4 bytes from there, so we get the packet type. We then switch that –yes, AS3 allows to switch statements with the content of Strings!- and proceed accordingly; we've copied two examples. In the TEND case, we just create an ON_TURN_START event and dispatch it. In the TSyn case we create, likewise, an ON_TURN_SYNC event but also extract the card id and the or operator selection. Every time we read an amount of bytes, the buffer pointer moves by the same amount, hence we don't mess with it.

10.8 RemoteEvent

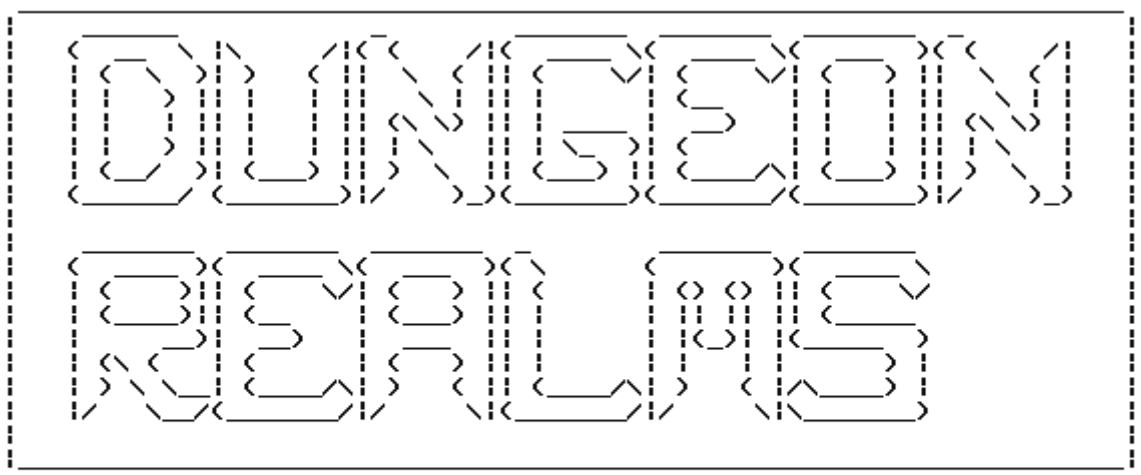
Finally, let's take a look at the last piece left to explain. The RemoteEvent class is very similar to our presentation layer's CardEvent, but the main difference is that we use them to throw events from the data layer. We could have just one type of custom event for the whole game, but it really becomes a mess when defining together a lot of string constants that don't have anything in common.

```
public class RemoteEvent extends Event {

        public static const ON_TURN_SYNC:String = "onTurnSync";
        public static const ON_TURN_SYNC_BUY:String = "onTurnSyncBuy";
        public static const ON_TURN_SYNC_NULLTOWN:String = "onTurnSyncNullTown";
        /*
        And so on for every packet type
        */
        public var id:int;
        public var choice:uint;
        public var num:uint;

        public function RemoteEvent(type:String,
                                    bubbles:Boolean=false,
                                    cancelable:Boolean=false):void{
          super(type, bubbles, cancelable);
        }
        override public function clone():Event {
                // Return a new instance of this event with the same parameters.
                return new RemoteEvent(type, bubbles, cancelable);
        }

}
```

Just note how every event type is, actually, a unique string identifier. The reason to that is that the Event superclass is implemented using Strings to identificate event types.

10.9 The server in all its glory

```
PS Z:\DRserver> node .\server.js
```



```
Server started on port 8000
Game seed is 2100

NEW CONNECTION
  address 127.0.0.1:50609
  sending policy to client
  client answer to policy: OK
  sending seed
  assigned as PLAYER 1

DATA: message received from player 1
->ACKSeed

NEW CONNECTION
  address 127.0.0.1:51164
  sending policy to client
  client answer to policy: <policy-file-request/>
Error: client requested policy file twice?
CLOSE: player undefined disconnected.

NEW CONNECTION
  address 127.0.0.1:51165
  sending policy to client
  client answer to policy: OK
  sending seed
  assigned as PLAYER 2

DATA: message received from player 2
->ACKSeed

ACK reset

Sent to player 1 <- 0008TEND

DATA: message received from player 1
->0008thxy

DATA: message received from player 1
->0016TSyn00120000
```

Note we've even made a big flashy ASCII title, as can't be otherwise with any text-based videogame-related program. Note how there is a dropped connection due to a double policy petition but the server keeps working flawlessly.

11 Iteration 5

11.1 Objective

Implement the game rules, so a game can actually be played between two human players.

11.2 Backlog

B51 - Create the game engine barebone

B52 - Implement player stats: px, gold, followers, number of cards in each place.

B53 - Implement data structures for the game engine.

B54 - Implement each of the card effects.

B55 - Implement each of the card operators –and, or, arrow.

B56 - Implement card playing rules.

B57 - Integrate calls with the socket class.

B58 - Implement game flow: turn, journeys, phase...

B59 - Implement victory conditions

11.3 Effort chart

We can only recognize there's been a lot of under-estimating! The extra time has got mainly (80%) to debugging. As the game has got more and more functionalities every bug has been harder to track; the #1 bug in the ranking was to write a return instead of a break inside a switch, what meant a dozen hours, and the patience, lost to debugging!

Ref	Concept	Estimate	Real
B51.T1	Design the game engine	5	10
B51.T2	Implement the skeleton	2	2
B52.T1	Define stats to track	3	3
B52.T1	Implementation	3	5
B53.T1	Define game engine data and variables	5	10
B53.T2	Implementation	25	30
B54.T1	14 different effects (we account them together instead of in 14 tasks)	$14 \times \sim 5 \approx 75$	100
B55.T1	And operator	2	2
B55.T2	Or operator	10	15
B55.T3	Arrow operator	10	30
B56.T1	Implement and test rules to allow only legal movements	30	50
B57.T1	Implement socket calls	10	15

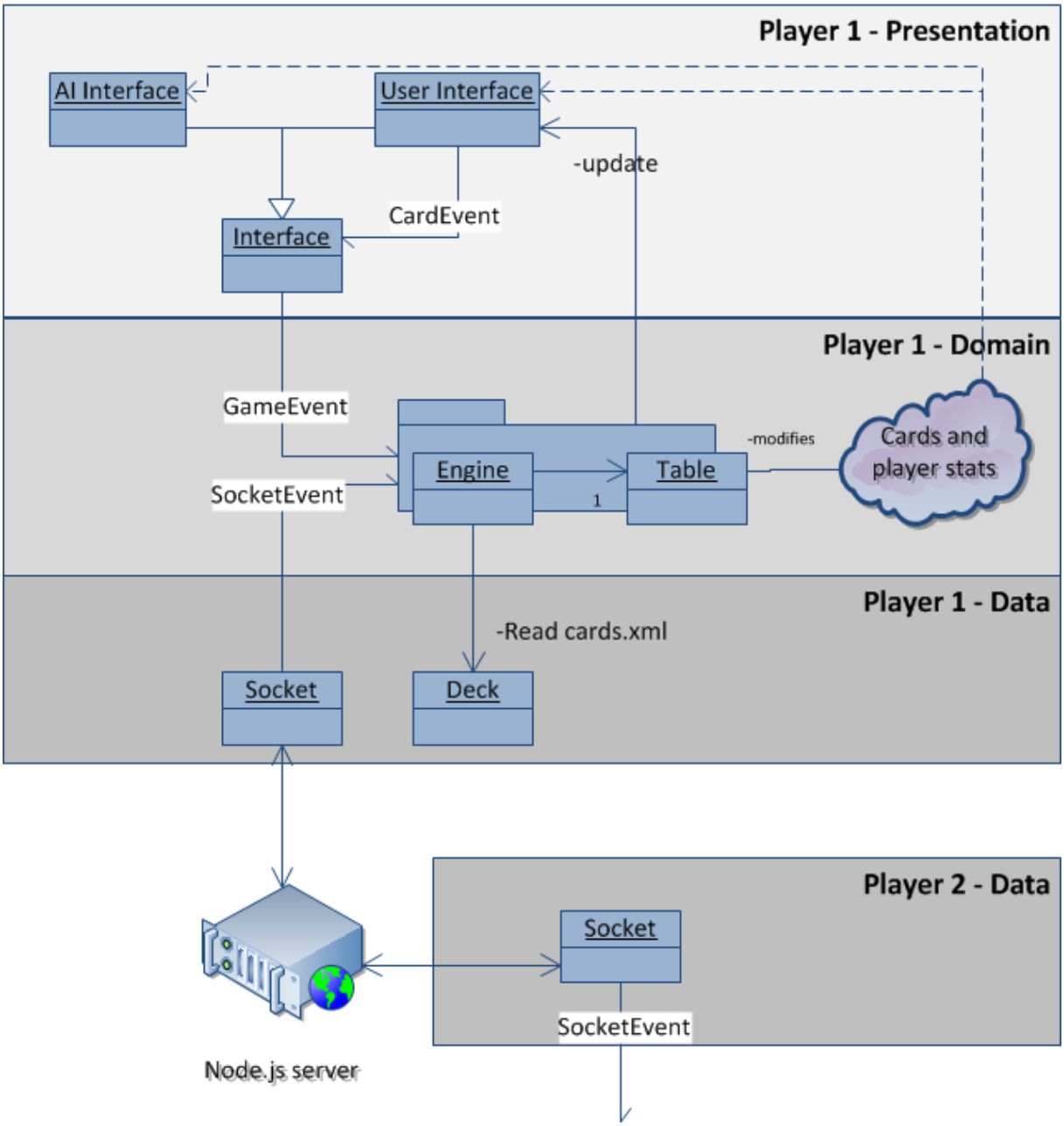
B57.T2	Test and polish	5	10
B58.T1	Implement turns	15	15
B58.T2	Implement planification phase	15	20
B58.T3	Implement journey end	15	15
B59.T1	Design how victory conditions work	10	10
B59.T2	Implementation	5	5
	Total	245	347

11.4 The game engine

Our Dungeon Realms engine, as in any game, is expected to become quite big with all the functions it will need to work. Because of that we've decided to use the component design pattern. This is commonly used in game development when an object traverses layers in its implementation, to avoid it. In our case, we'll separate our game engine into two separate classes: the Engine and the Table.

They both work together for the game rules to work, but the Engine is the domain module that talks with the data layer, while the Table is the responsible to make changes reflect the screen. These two classes are highly coupled between them by their nature; however, to avoid bad implementation choices and to keep a more streamlined approach, the Table won't keep any reference to the Engine, and will be this last one that will always push data to the former.

To understand it better, you can understand it thinking the Engine class processes game logic while the Table class processes game data. The game data plus the game logic make the actual game.



The image shows, at a very big glance, how the game components interact between them. The interfaces know information about the game state –cards and stats-, so the player can interact with it, and communicate the actions, which are asynchronous by nature, via events. The Engine then has processes the data to attain the desired action, delegating on its Table component all the actions that depend on the actual game data.

11.5 Structuring game data

11.5.1 Player stats

To keep track of each player stats, we have a dedicated class. We store the six basic stats that can be seen on the screen plus some other data:

- extraMen: game rules state the maximum extra followers that can be gained in a journey are two. We store here this number of followers gained during a journey.
- menAva: only one follower can be used per turn. This boolean marks whether one has been used this turn.
- dragonsKilled: keeps track of the number of dragons killed. Two dragons mean the player wins the game.

```
public class PlayerStats {
        protected var px:uint = 0;
        protected var hand:uint = 0;
        protected var deck:uint = 9;
        protected var discard:uint = 0;
        protected var money:uint = 0;
        protected var men:uint = 3;
        protected var extraMen:uint = 0
        protected var maxMen:uint = 3;
        protected var menAva:Boolean = true;
        protected var dragonsKilled:uint = 0;

        public function availableMen():Boolean{return menAva && men > 0;}
        public function getPX():uint{return px;}
        public function getMoney():uint{return money;}
        public function getHand():uint{return hand;}
        public function getDeck():uint{return deck;}
        public function getDiscard():uint{return discard;}
        public function getMen():uint{return men;}
        public function getExtraMen():uint{return extraMen;}
        public function getDragonsKilled():uint{return dragonsKilled;}
        /*
        Continues
        */
}
```

There's a getter function for each parameter that directly return the values, except for the query for available men, that not only needs that no followers have been used this turn but also that there are still at least one remaining.

We then need a lot of functions to precisely attend the engine's needs. To avoid you reading through a couple dozen functions, we'll just copy an extract.

```
public class PlayerStats {
        /* continues */
        protected function incMen(){
                if(extraMen == 2) return;
                men++;
                extraMen++;
```

```
        }
        public function setMen(n:uint){men = n;}
        public function decMen(){men--; menAva = false;}
        public function subMen(n:uint){men -= n;}
    }
```

In this example, we see the need a regular setter –for example to reset the number of followers on new turns-, a standard subtractor –some cards require paying followers- plus a function to pay when placing a follower on turn –that subtract only one, but also mark there is no longer a man available this turn-, and lastly a function to add extra followers to the pool –that does nothing when the maximum has already been reached. Lastly, we instance and reference it through our Table class so the stats are accessible through our domain layer:

```
public class Table extends EventDispatcher{
        protected var p1Stats:PlayerStats = new PlayerStats();
        protected var p2Stats:PlayerStats = new PlayerStats();
        /* etc */
}
```

The number of cards of the players is refreshed when an update is called throughout the game engine. Other stats such as the money available are changed when an effect modifies them, so there's no need to update them in this call:

```
public class Table extends EventDispatcher{
        protected function updateStats(){
                p1Stats.setDeck(playerDeck.length);
                p1Stats.setDiscard(playerDiscard.length);
                p1Stats.setHand(playerCards.length);

                p2Stats.setDeck(opponentDeck.length);
                p2Stats.setDiscard(opponentDiscard.length);
                p2Stats.setHand(opponentCards.length);
        }
        /* etc */
}
```

We'll return to the PlayerStats a bit later, when discussing on the game logic implementation.

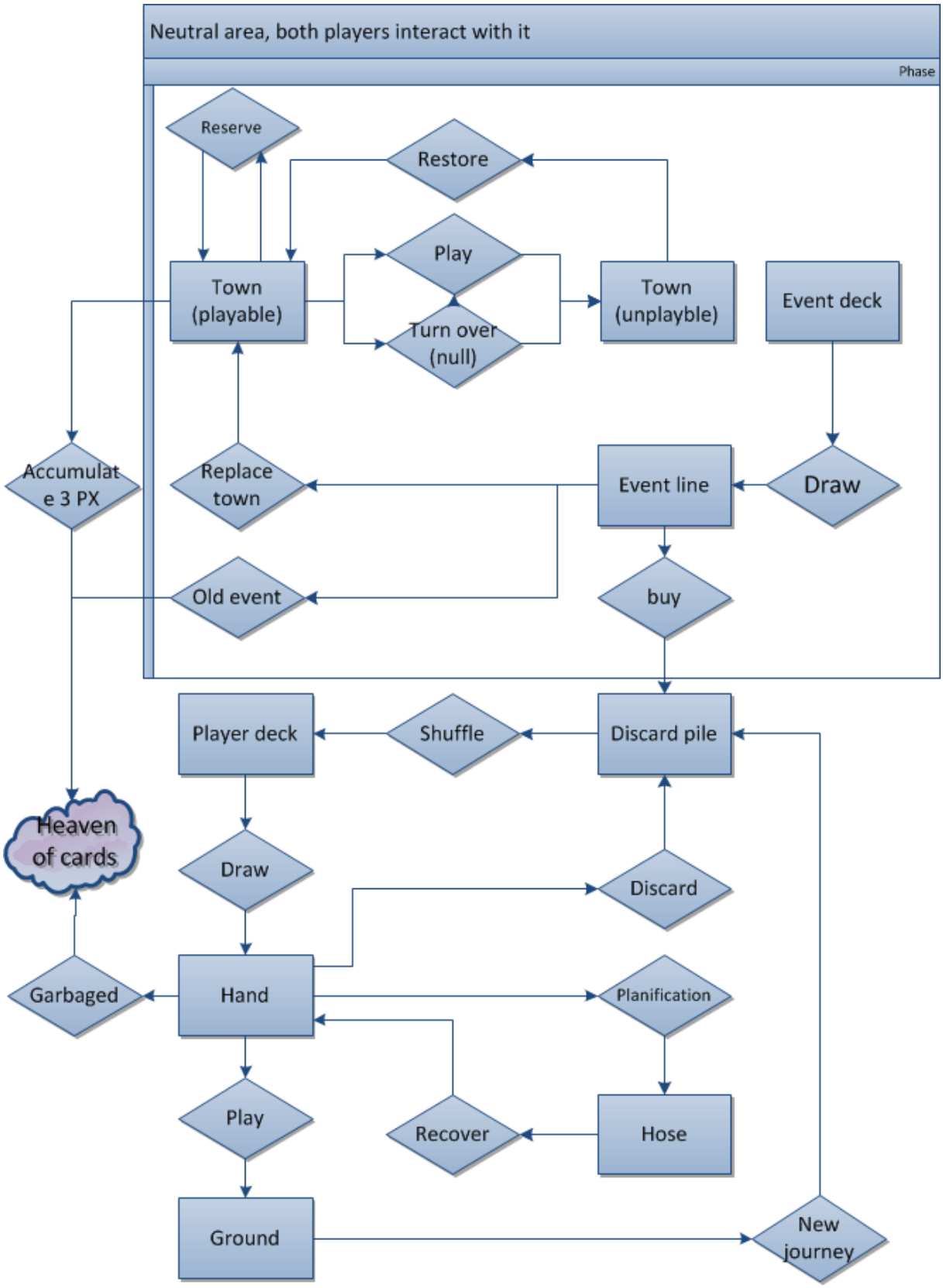
11.5.2 Cards

(The actual) Dungeon Realms is a game played over a hard surface, be it a table or even the floor. Cards are organized in groups, in whatever surface is chosen, that represent things or places within the game. This is a big overview of each group, omitting game rules to avoid a four-page block:

- **Event deck**: this is the source of events.
- **Event line**: represent the quests and events the adventurers face during the game. At the start of a journey 6 quest cards are placed in the line and players can acquire them during their turns through the corresponding card effect or by paying the corresponding amount of money.
- **Town**: consists of nine cards with special playing rules that represent the town of the adventurers. One can be reserved per turn with a follower and they can be used anytime until the next journey.
- **Deck**: this is the source of resources, cards, for the player.

- **Hand**: the cards in the hand are the assets and means available for the adventurer, the player, to solve the quests that happen and are available to interact with through the event line. These cards are drawn from the deck.
- **Hose**: the planification effect let's save a card apart to recover it in the next plan phase.
- **Ground**: when a player plays a card from hand, it is placed in the ground, rotated ninety degrees, and its effects are played.
- **Discard pile**: after acquiring or discarding a card, it is placed here. When there are no cards remaining in the deck, all of the cards in the discard pile are taken, shuffled and transformed into the new deck.

The life cycle of cards in Dungeon Realms, is illustrated in the following graph. Note how the hand is the place with more options to output cards and the discard pile is the main sink. Also note that, except the marked neutral zone, the rest is actually duplicated –although only drawn once- since there are two players.



After a though, one quickly gets to the conclusion there's the need for en event deck, an event line and a town that both players will interact with. For every player, there's de discard pile, deck, hand and playing area –ground-, plus the hose to save cards for a future journey with the planification effect.

Hence we have a total of five groups of cards for each player, plus three common groups that don't pertain to any player:

```
public class Table extends EventDispatcher{
	protected var playerDeck:Array = new Array();
	protected var playerCards:Array = new Array();
	protected var playerGround:Array = new Array();
	protected var playerDiscard:Array = new Array();
	protected var playerHose:Array = new Array();

	protected var opponentDeck:Array = new Array();
	protected var opponentCards:Array = new Array();
	protected var opponentGround:Array = new Array();
	protected var opponentDiscard:Array = new Array();
	protected var opponentHose:Array = new Array();

	protected var townCards:Array = new Array();
	protected var eventDeck:Array = new Array();
	protected var eventLine:Array = new Array();
	/* etc */
}
```

We use again the almighty Array class, that let us work with it as a table –indexes-, a queue –shift, unshift-and a stack –push, pop-. It's inefficient speed-wise due to all the functionality it provides, but since we want to make few operations on them it won't be noticed.

And now comes the magic. We already explained how interesting it would be to be able to swap player data so the same functions would on either player:

```
	public function swapPlayers(){
		var ac:Array = opponentCards;
		opponentCards = playerCards;
		playerCards = ac;
		/*
		Does the same for every player's array
		*/
		var ps:PlayerStats = p1Stats;
		p1Stats = p2Stats;
		p2Stats = ps;

		for(var i:int = 0; i < townCards.length; i++){
			(townCards[i] as Card).swapPlayers();
		}
		Registry.rules.swapPlayers();

	}
```

This call swaps every one of the player arrays in our Table class. It also swaps the playerStats instances, so the effects apply on the correct numbers. We do also call a swap on every town cards, since, although being a neutral area, players can reserve them; so what this town swap actually do is to check if the card is owned by a player and, if so, swap the player –local for the opponent, and the other way around. Lastly, we call a swap on a Registry.rules that we'll see later, but that's basically a class that provides tests to know if cards can be played.

11.6 Card effects

As we said, or as you should know if you read the rules, there are quite some different effects in Dungeon Realms. Now that you know how the data is structured, let's see how do we apply the effects.

Unless when stated otherwise, the code shown here pertains to the Table class –so we can avoid writing every time.

11.6.1 Add and remove money

We have the current player's gold stored in the PlayerStats instance called p1Stats. So:

```
p1Stats.addMoney(amount);
p1stats.subMoney(amount);
```

Accessible in our Table, add and subtract amounts of money.

11.6.2 Add PX

In the same fashion we can add experience. Note that we never need to subtract PX points:

```
p1Stats. addPX (amount);
```

11.6.3 Draw card

The instance playerCards has the destination group of cards, and a couple pages before you saw player cards are always drawn from the player deck. We have the call:

```
playerDraws(amount);
```

That is then chained to these two functions:

```
public function playerDraws(n:uint){
        Registry.dlog("Player drawing " + n);
        for(var nn:uint = 0; nn < n; nn++)
                drawOne();
}
protected function drawOne(){
        Registry.dlog("   drawing a card);
        playerCards.push(playerDeck.pop());
        if(playerDeck.length == 0)
        {
                shuffle(playerDiscard);
                while (playerDiscard.length > 0) {
                        playerDeck.push(playerDiscard.pop());
                }
        }
}
```

That is, we divide the task of drawing an arbitrary amount of cards into N times the task of drawing one card. This way it's easier to detect when there's the need to reshuffle the player's discard pile into a new deck. So the drawOne methods pushes the top card of the deck to the player hand and, if the deck gets empty, the discardPile is converted into a new deck. Note that we don't do a playerDeck = playerDiscard, as may seem natural, because these arrays could be pointed in any other instance and hence would break the integrity of the data. Pushing every card from one array to the other makes the code foolproof to it.

11.6.4 Discard card

Discarded cards go from hand to the discard pile:

```
public function disCard(_id:uint){
        Registry.dlog("table.disCard");
        var i:int = Helpers.indexOf(playerCards, _id);
        if(i<0) throw new Error("Table.disCard - card not found");
        playerDiscard.push(playerCards[i]);
        playerCards.splice(i,1);
        updateStats();
}
```

We first locate the card instance in the player hand through its ID, use the push as before, and lastly we splice the hand array –removes from the array the number of cards passed in the second parameter, starting from index 'i'.

11.6.5 Garbage card

It proceeds almost identically to the discarding we just saw:

```
public function garbageCard(_id:uint){
        Registry.dlog("table.garbageCard IN HAND");
        var i:int = Helpers.indexOf(playerCards, _id);
        playerCards[i].destroy();
        playerCards.splice(i,1);
        updateStats();
}
```

Note, however, that instead of pushing the card into the discard pile, garbaged cards are definitely removed from the game. Hence, we splice the given card from the hand array after calling its destroy method:

```
public class Card extends MovieClip {
        /* etc */
        public function destroy(){
                graphics.clear();
                removeEventListener(MouseEvent.CLICK, clickHandler);
                tfName = null;
                tfEffect = null;
                pre = null;
                post = null;
                animTurn = null;
        }
}
```

The VM garbage collector sucks a lot, so it's a very good practice, when programming AS3, to always clean everything on objects so they get actually garbaged from the application's memory. In our case, we set to null every reference to other objects in our instance, clear any graphics data in it –for example, the background painted in some colors, and remove the listener –for example, not doing this, prevents the VM to ever garbage this instance because a listener/handler points to it.

11.6.6 Buy/acquire card

Cards can be acquired or bought for free with the cart effect and they can also be bought for the cost written into the card by paying money.

```
public function payMoney(n:uint){
	Registry.dlog("table.payMoney");
	p1Stats.subMoney(n);
}
public function cartCard(_id:uint){
	Registry.dlog("table.cartCard");
	var i:int = Helpers.indexOf(eventLine, _id);
	if(i<0) throw new Error("Table.cartCard - card not found");
	eventLine[i].removeFromEvents();
	playerDiscard.push(eventLine[i]);
	eventLine.splice(i,1);

	for(var j:int = i-1; j >= 0; j--){
		if((eventLine[j] as Card).isTether()){
			eventLine[j].removeFromEvents();
			playerDiscard.push(eventLine[j]);
			eventLine.splice(j,1);
		}
	}
	updateStats();
}
```

To achieve the effect, we have a call to pay money that routes to the player stats, plus a method to get the card. We have the payMoney function for compatibility: if any change in the player stats code arises, we just make a change here, versus having to change a call to the player stats every time we need to pay money.

The cartCard receives the ID of the card to acquire and finds it through the indexOf call. It then tells the card it's removed from the event line:

```
public function removeFromEvents(){
	inEventLine = false;
}
```

This Boolean is used to know when the card will be clickable depending on the situation. For example, a card can't be acquired if the inEventLine is set to false.

Back to the previous function, we then push the card's reference to the player discard pile –where acquired events go- and remove it from the event line array with the splice method.

Lastly, we check if there's any Tether at the left of the acquired card, because the rules stated that, in that case, it's also acquired by the player. We mimick in the Tether the remove-push-splice we did to the actual event acquired. A final call to updateStats() keeps the coherence of the data available to the interface.

11.6.7 Add and remove followers

Similarly again to money, we can do:

```
p1Stats.addMen(amount);
```

```
p1Stats.subMen(amount);
```

And, when we're using a follower to reserve a town:

```
p1Stats.decMen(amount);
```

Just note that the admen call does not plainly add an amount to the value, as there was the maximum extra followers we mentioned earlier:

```
public class PlayerStats {
	public function addMen(n:int){
		for(var i:uint = 0; i < n; i++){
			incMen();
		}
	}
	protected function incMen(){
		if(extraMen == 2) return;
		men++;
		extraMen++;
	}
}
```

11.6.8 Null town

To achieve this effect, we have the call:

```
public function nulltown(_id:uint){
	var i:int = Helpers.indexOf(townCards, _id);
	townCards[i].nullTown();
}
```

That first finds the index of the card ID parameter and then calls it's nullTown method.

```
public class Card extends MovieClip {
	public function nullTown(){
		if(!inTown) throw new Error("Trying to destroy a card that is not in town");
		nulled = true;
		alpha = 0;
	}
	public function denullTown(){
		if(!inTown) throw new Error("Trying to rebuild a card that is not in town");
		nulled = false;
		alpha = 1;
	}
}
```

The call makes it disappear from the screen –alpha = 0 means totally transparent- and sets a Boolean to mark it has been nulled. The denull, called when a journey ends, restores the town status.

11.6.9 Save card or planification

The save effect, or planification, puts a card into your hose until the next plan phase, thus giving you extra cards for the next journey:

```
public function saveHand(_id:uint){
	Registry.dlog("table.saveHand");
	var i:int = Helpers.indexOf(playerCards, _id);
```

```
            playerHose.push(playerCards[i]);
            playerCards.splice(i,1);
            updateStats();
        }
```

Basically, we do our trustworthy find index-push-splice-update. And when a new journey comes:

```
        while(playerHose.length > 0)
            playerCards.push(playerHose.pop());
        /*
            etc
        */
        swapPlayers();
        while(playerHose.length > 0)
            playerCards.push(playerHose.pop());
        /*
            etc
        */
        swapPlayers();
```

We add all the saved cards to the hand. Note that we can use exactly the same code by using the swapPlayers() we showed earlier.

11.6.10 Double gold and double PX

We show these two together since both are coded equally. From the Table we can call:

```
p1Stats.doubleMoney();
p1Stats.doublePX();
```

In the PlayerStats class is implemented:

```
public class PlayerStats {
    protected var moneyMult:uint = 1;
    protected var pxMult:uint = 1;

    public function addPX(n:uint){px += n*pxMult;}
    public function doublePX(){pxMult = 2;}

    public function addMoney(n:uint){money += n*moneyMult;}
    public function doubleMoney(){moneyMult = 2;}
}
```

We have a multiplying factor that by defaults it's one. When we use a doubling effect, we set it to 2. Whenever we add gold or experience, we always multiply the value by this factor –that is always 1 or 2.

11.6.11 Reuse

This effect allows replaying an already used card, which means it must be either in the player's ground or, either, reserved and used –by this player- in town. That means we introduce some alternate rules to allow when a card is being played:

```
        protected function clickHandler_reuse(e:MouseEvent = null){
            if(inEventLine) return;
            if(isGesta()) return;
            if(inTown && (!mine || !rotated)) return;
```

```
                dispatchClick(id);
        }
```

That is: if the card is not in the player's ground –what's means it's been used before-, or either in town and owned and used by the player, we don't take action. If any of these two conditions is met, then the click is treated.

11.7 Card operators

11.7.1 And & Or

We better explain these two at the same time, since they are inherently treated by our effect iterator. Our applyPost(…) function takes a card instance, an 'or' value stating the effect selected to apply and the times –multiplicator, number preceding the effect- to apply (some of) the effects.

```
protected function applyPost(_c:Card, _or:uint, _n:uint = uint.MAX_VALUE){
        …
}
```

The _or parameter always has the values 0 or 1:

- If card does not have Or operator, then always, **_or = 1**
- If card does have Or operator:
  - If left effect selected, then **_or = 1**
  - If right effect selected, then **_or = 2**

Note that when there is an Or, there is never an And or an Arrow operator. When there is an Or operator, _or is 1 independently if there is an And or an Arrow.

The _n parameter is the maximum times allowed by the rules to apply an effect. For example, a card may have an arrow operator preceded by the effect 'discard up to 3 cards' but, if the player does only have 2 cards in the hand then _n is passed with value 2.

Since the times an effect can take place is restricted by both what's written in the card **and** the current game status –player stats, cards available…-, then the actual maximum times that effect can take place is the minimum of those two values: var _num = Math.min(p[i], _n); where p[i] is the value of the XML effect node we parsed to the card effect table.

```
protected function applyPost(_c:Card, _or:uint, _n:uint = uint.MAX_VALUE){
        Registry.dlog("Table.applyPost OR: " + _or + " " + _n);
        var _first:Boolean = (_or < 2);
        var _ret:Boolean = true;
        var p:Vector.<int> = _c.getPost();
        var _evt:GameEvent;

        for(var i:int = Constants.V_START; i <= Constants.V_END; i++){
                if(p[i] != 0 && _first){
                        var _num = Math.min(p[i], _n);
                        switch(i){
                                case Constants.V_PX:
                                        Registry.dlog("Table - gain px " + _num);
                                        p1Stats.addPX(_num);
                                        break;
                                case Constants.V_MONEY:
                                        Registry.dlog("Table - gain mo " + _num);
```

```
                        p1Stats.addMoney(_num);
                        break;
                    /*
                    Rest of the effects
                    */
                    }
                    if(p[Constants.V_OR] == 1) return;
                }
                else if(p[i] != 0) _first = true;
            }
        }
```

The first thing we do is to check if _or has the value 2 or not: if it's 1 we'll apply the first effect we find. If it's 2 we'll skip it because it means two things:

- This is a card with an Or operator
- The right effect has been selected

So we have the boolean _first that's true whenever the first effect has to be applied and false when it has to be skipped. When _or=2 and we skip the first effect, then we mark _first = true –that is, to apply the first effect we find from that point:

```
if(p[i] != 0 && _first){
```

When we apply an effect we always check:

```
if(p[Constants.V_OR] == 1) return
```

Hence when we are processing an Or card we stop applying effects after the first. If it's a card with an And operator, it will keep applying every effect found.

11.7.2 Arrow

The arrow operator is more special than the two previous ones. To apply the effect in the right of the arrow, we first need to have applied the effect in the left. For example, if the right says to draw 3 cards and the left to discard 3, it's not the same to discard these 3 after drawing than doing it before.

```
protected function playCardHand (_id:uint, _or:uint = 0){
        var i:int = Helpers.indexOf(playerCards, _id);
        if(i == -1) throw new Error("Table.playCardHand - invalid card ID");

        var _card:Card = playerCards[i];
        if(_card.isTether()) return;
        if(_card.hasPre()){
            Registry.dlog(" has pre");
            if(!Registry.rules.canPlayCardPre(_card)){
                Registry.log(" no can play");
                dispatchEvent(new GameEvent(
                                GameEvent.ON_CARD_CLICK_CANT));
                return;
            }
            Registry.selectNum.init(_card, Registry.rules.preMaxValue(_card));
            _pdCard = _card;
```

```
                        playerGround.push(playerCards[i]);
                        playerCards.splice(i,1);
                        _card.turn();

                        var _e:GameEvent = new GameEvent(GameEvent.ON_PRE);
                        _e.id = _card.getPreType();
                        _e.choice = Registry.rules.preMaxValue(_card);
                        dispatchEvent(_e);
                        return;
                    }
                    /* else proceed normally */
                }
```

Hence, when we check a card has a pre-effect, or what's the same, an arrow operator, we enter the big if block. If our Rules class says we can't play it due to restrictions –e.g. garbage hand cards when there aren't any left- we return throwing an Event that notifies it.

When the test succeeds, we check the maximum times the effect can be applied, remove the card from the hand and put it on the playing area. We first initialize a control, used to manually select the times we want to use the effect, with those values. We also save the card reference in _pdCard for later use. Lastly, we throw an ON_PRE event that carries the type of effect and the maximum times it can be applied. This event is captured in the Engine class, that manages the game logic.

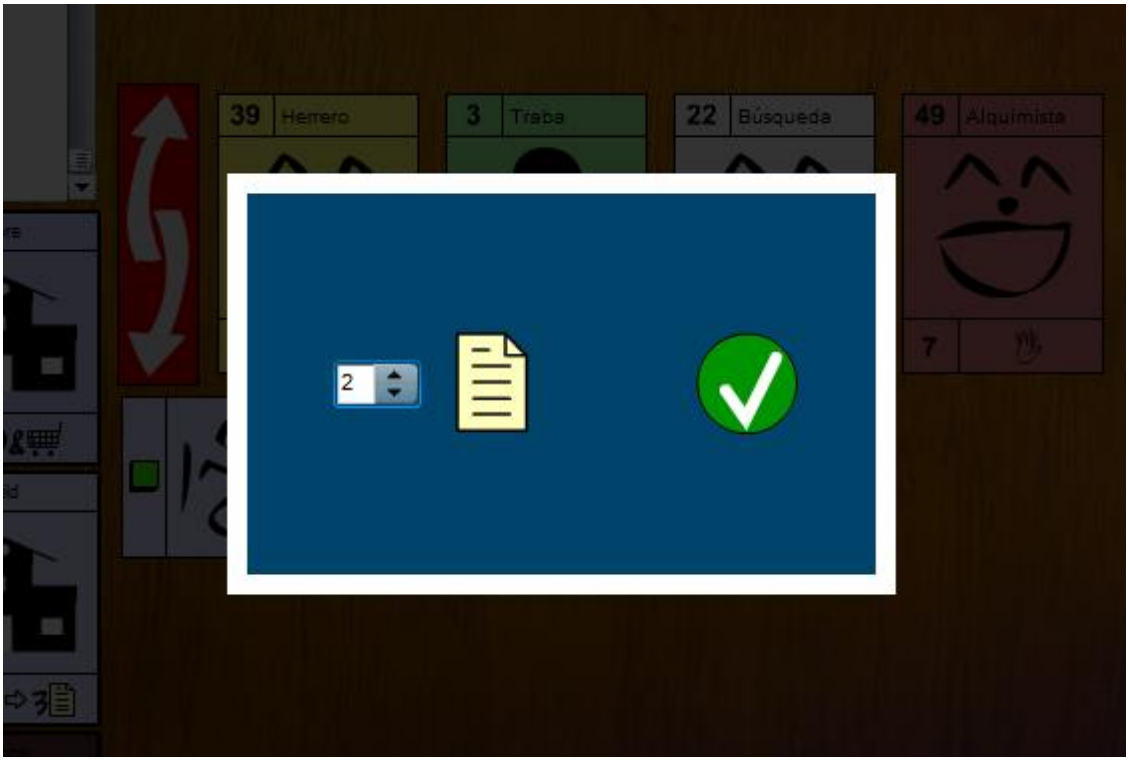
```
public class Engine{
        protected function cardClickHandler(e:GameEvent){
                Registry.dlog("Engine.cardClickHandler " +e.id);
                cardClickListeners(true);
                table.playCard(e.id,e.choice);
        }
        protected function cardClickListeners(_on:Boolean){
                Registry.dlog("Engine.cardClickListeners "+ _on + " " + Registry.currentPhase);
                if(_on && Registry.currentPhase != Constants.PH_PLAN){
                        table.addEventListener(GameEvent.ON_PRE, cardPreHandler);
                        table.addEventListener(GameEvent.ON_CARD_CLICK_CONFIRM,
                                                        cardClickConfirmHandler);
                        /* other events */
                }
                else{
                        table.removeEventListener(GameEvent.ON_PRE, cardPreHandler);
                        table.addEventListener(GameEvent.ON_CARD_CLICK_CONFIRM,
                                                        cardClickConfirmHandler);
                         /* other events */
                }

        }
        protected function cardClickConfirmHandler(e:GameEvent = null){
                Registry.dlog("Engine.cardClickConfirmHandler");
                cardClickListeners(false);
                /* code */
        }
}
```

As you can see, the Engine activates the capture of some events before delegating into the Table to apply the effects of a card. Remember the ON_CARD_CLICK GameEvent was received when a card was clicked –and listened by cardClickHandler? The ON_CARD_CLICK_CONFIRM is received when all of the effects of the card have been applied.

But, when applying the left part of an arrow operator, we throw an ON_PRE event that ends up here:

```
        protected function cardPreHandler(e:GameEvent){
                Registry.dlog("Engine.cardPreHandler");
                cardClickListeners(false);
                preType = e.id;
                numPreMax = e.choice;
                Registry.selectNum.addEventListener(
                                GameEvent.ON_PRE_CHOICE, cardPreChoiceHandler);
                }
}
```

We deactivate the normal card event listeners, extract the type effect and the maximum times –by the rules- it can be applied and add a listener to the control initialized in the Table. The number-select control will notify the user selection through the ON_PRE_CHOICE event. There's a screen of the control:



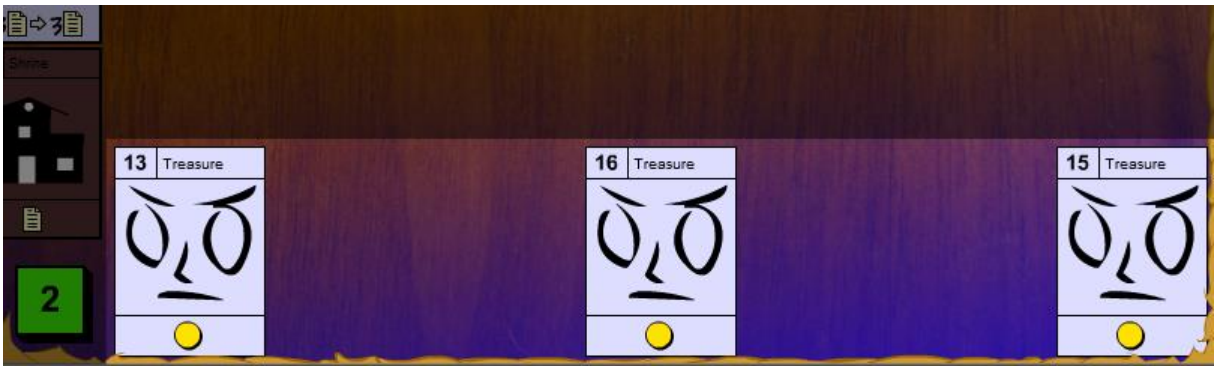
When the user presses the big round green V, the event is thrown and the overlay disappears. The Engine then recovers the execution flow again:

```
        protected function cardPreChoiceHandler(e:GameEvent){
                Registry.dlog("Engine.cardPreChoiceHandler");
                if(preType == Constants.V_FOLLOWER){
                        Registry.dlog(" pre is follower --> resume");
                        table.payFollower(e.choice);
                        table.resumePost(e.choice);
                        table.after();
                        sl.playCardPre(e.id, e.choice, Constants.V_FOLLOWER);
```

```
            cardClickListeners(true);
        }
        else if(preType == Constants.V_MONEY){
            Registry.dlog(" pre is money --> resume");
            table.payMoney(e.choice);
            table.resumePost(e.choice);
            table.after();
            sl.playCardPre(e.id, e.choice, Constants.V_MONEY);
            cardClickListeners(true);
        }
        else{
            sl.playCardPreStart(e.id);
            numPreMax = e.choice;
            numPreActions = 0;
            ui.allowOnlyHand(true);
            ui.removeEventListener(GameEvent.ON_CARD_CLICK, cardClickHandler);

            switch(preType){
                case Constants.V_DISCARD:
                    ui.addEventListener(GameEvent.ON_CARD_CLICK,
                                        cardClickHandler_Discard);
                    break;
                case Constants.V_GARBAGE:
                    ui.addEventListener(GameEvent.ON_CARD_CLICK,
                                        cardClickHandler_Garbage);
                    break;
                default:
                    throw new Error("cardPreChoiceHandler: " + preType);
                }
            }
            cardClickConfirmHandler();
        }
```

There are various possibilities depending on the pre-effect:

- Pay followers: we give the Table the instruction to reduce the follower number and continue applying the post-effect.
- Pay money: this one can also continue with no more interaction. Proceeds as the follower-fee but reducing the amount of money.
- Discard or garbage cards: this one requires more user interaction, so it can't continue executing like the two latest. We redirect the card click listener to either a listener to discard cards or to garbage them and set the total amount of cards to treat in numPreMax. We also instruct the User Interface to put an overlay on the screen so the user can focus only on his hand cards.



Now, every time the user clicks one of the cards, it will route to the new (temporal) card click handler. We show the case of discard –garbage is similar:

```
protected function cardClickHandler_Discard(e:GameEvent){//PRE
        Registry.dlog("Engine.cardClickHandler_Discard");
        table.disCard(e.id);
        if(!updatePreParams())
                sl.playCardPreClick(e.id, Constants.V_DISCARD);
        else
                sl.playCardPreEnd(e.id, Constants.V_DISCARD, numPreActions);                }

protected function updatePreParams():Boolean{
        var preFinished:Boolean = false;
        numPreActions++;
        if(numPreActions > numPreMax)
                throw new Error("Engine.updatePreParams - overflow");
        if(numPreActions == numPreMax){
                ui.removeEventListener(GameEvent.ON_CARD_CLICK,
                                                        cardClickHandler_Discard);
                ui.removeEventListener(GameEvent.ON_CARD_CLICK,
                                                        cardClickHandler_Garbage);
                ui.addEventListener(GameEvent.ON_CARD_CLICK, cardClickHandler);
                cardClickListeners(false);
                ui.allowOnlyHand(false);

                table.resumePost(numPreMax);
                table.after();
                preFinished = true;
        }
        ui.update();
        return preFinished;
}
```

For every card clicked, we discard it. We then update the number of actions –cards discarded- taken and, if the effect is finished, we restore the card clicks to the original listeners, remove the UI overlay and resume the post-effect. We also instruct the opponent with one of our defined packets how to proceed depending if the pre has finished or it's an intermediate step.

The resumePost() method of the Table effect, retakes the processing of the card instance, stored in _pdCard, effects;

```
        var _pdCard:Card
        public function resumePost(num:uint, postDone:Boolean = false){
```

```
            if(whereIs(_pdCard.getID()) == Constants.IN_HAND){
                var _io:int = Helpers.indexOf(playerCards, _pdCard.getID());
                playerGround.push(playerCards[_io]);
                playerCards.splice(_io,1);
            }
            resumeTownPost(num,postDone);
        }
```

In case the card is in hand, we need to push it to the playing area. In any case, we call the resumeTownPost(…) method with the number of times the pre-effect was applied and whether the post-effect was already processed by any other means, just to be sure. Note this method processes effects from both hand and town cards because it's the same code:

```
    protected function resumeTownPost(num:uint, postDone:Boolean = false){
        Registry.dlog("Table.resumeTownPost "
            + num +" " + postDone + " " + _pdCard.getID());
        if(!postDone){
            applyPost(_pdCard, 0, num);
            var _evt:GameEvent =
                        new GameEvent(GameEvent.ON_CARD_CLICK_CONFIRM);
            _evt.id = _pdCard.getID();
            _evt.choice = num;
            dispatchEvent(_evt);
        }
        _pdCard.turn();
        _pdCard = null;
    }
```

So, if the post-effect was still done, we call the applyPost(…) function and throw, finally, an ON_CARD_CLICK_CONFIRM event.

11.8 Playing rules

For one side, we have a Rules object with a couple references to each of the player stats instances. This class ducktapes together many functions to control when and how cards can be played:

```
    public class Rules {

        protected var p1Stats:PlayerStats = new PlayerStats();
        protected var p2Stats:PlayerStats = new PlayerStats();

        public function Rules() {}
        public function setStats(p1:PlayerStats, p2:PlayerStats){
            p1Stats = p1;
            p2Stats = p2;
        }
    }
```

11.8.1 Turn tests

There's a recurrent need to know when a player can still start a turn and when a journey is finished:

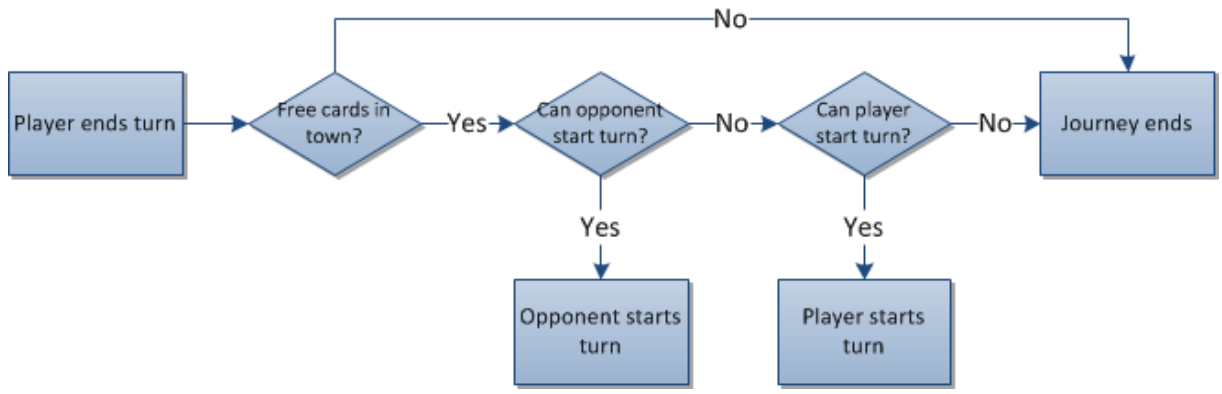
```
        public function canIStartTurn():Boolean{
            trace("canIStartTurn", p1Stats.getMen() > 0);
            return p1Stats.getMen() > 0;
```

```
}

public function canOpStartTurn():Boolean{
        trace("canOpStartTurn", p2Stats.getMen() > 0);
        return p2Stats.getMen() > 0;
}

public function isJourneyFinished(a:Array):Boolean{
        Registry.dlog("Rules.isJourneyFinished);
        if(!remainingTowns(a)) return true;
        Registry.dlog("   there are towns remaining");
        return !(canIStartTurn() || canOpStartTurn());
}
```

A player can start a turn when it still has followers left. To test when a journey has finished, we first need to know if there are still towns that haven't been reserved:

```
protected function remainingTowns(a:Array):Boolean{
        var _c:Card;
        for(var i:uint = 0; i < a.length; i++)
                if( (a[i] as Card).isAvailable() )
                        return true;
        return false;
}
```

If there are no remaining towns, we return true, which means the journey is finished –because in every turn a town must be reserved. In case there is, indeed, some remaining, we then decide based on the players status: if any of them can start a turn, then it's not finished.



11.8.2 Town playability tests

A town cards can be reserved if there are men available. It can be used if it's in possession of the current player and is not rotated. It can also be played when rotated if the game has de reuse mode active, what happens when the reuse effect is being applied –waiting for the user to select a card to replay.

```
public function canReserveTown(c:Card):Boolean{
        Registry.dlog("Rules.canReserveTown ");
        if(p1Stats.availableMen()) return !c.isReserved();
        return false;
}

public function canPlayTown(c:Card):Boolean{
        Registry.dlog("Rules.canPlayTown");
```

```
            if(c.isMine() && !c.isRotated()) return true;
            if(c.isMine() && c.isRotated() && Registry.modeReuse) return true;
            return false;
        }
```

Note that because our player swapping the tests are easy to conduct in the rules: we mentioned we had a swapping method in town cards, that switches the Boolean that marks if the current player is in its possession.

11.8.3 Event playability test

A card in the event line can be bought if it's not a tether and the player has enough money. Note there's no need to test if we're acquiring through a card cart effect, because then the only we can't do is get a tether:

```
        public function canBuyEvent(c:GameCard):Boolean{
            trace("Rules.canBuyEvent");
            if(c.isTether()) return false;
            var _qc:QuestCard = c as QuestCard;
            return _qc.getPrice() <= p1Stats.getMoney();

        }
```

11.8.4 Arrow operator playability test

The left-operand of cards with an arrow operator is not only a cost to pay, but also a restriction on whether a card can be played. To know if a card with the arrow can be played, we have a switch on the possible effects that can appear on the left side:

```
    public function canPlayCardPre(c:Card):Boolean{
        var _pre:Vector.<int> = c.getPre();
        for(var i:uint = Constants.V_START; i <= Constants.V_END; i++){
            if(_pre[i] != 0){
                switch(i){
                    case Constants.V_MONEY:
                        return 0 < p1Stats.getMoney();
                    case Constants.V_GARBAGE:
                    case Constants.V_DISCARD:
                        return 0 < p1Stats.getHand();
                    case Constants.V_FOLLOWER:
                        return 0 < p1Stats.getExtraMen();
                }
            }
        }
        return false;
    }
```

Since the Rules class has access to the player stats, the checks on each resource are easy.

11.8.5 Quantifying the arrow operator

As we said previously when explaining how to apply the arrow operator, we need to know the maximum number of times that effect can actually be played:

```
    public function preMaxValue(c:Card):uint{
        var _pre:Vector.<int> = c.getPre();
```

```
            for(var i:uint = Constants.V_START; i <= Constants.V_END; i++){
                if(_pre[i] != 0){
                    switch(i){
                        case Constants.V_MONEY:
                            return Math.min(_pre[i], p1Stats.getMoney());
                        case Constants.V_GARBAGE:
                        case Constants.V_DISCARD:
                            return Math.min(_pre[i], p1Stats.getHand());
                        case Constants.V_FOLLOWER:
                            return Math.min(_pre[i], p1Stats.getMen());
                    }
                }
            }
            throw new Error("Rules.preMaxValue – effect id not found");
        return 0;
        }
```

So, for example, when discarding o garbaging cards pre[i] contains the number in the card, so we also get the number of cards in the hand and return the minimum of both. That is, if the card says 3 but there are 2 cards in hand –apart from the one being played-, a 2=min(3,2) will be returned so the player will only be able to select a 1 or a 2 by the method we saw when showing how to apply the arrow effect.

11.8.6 Hand card playability test

Hand cards can always be played, except when the arrow operator is present. That means we don't need any extra function because when the arrow is present, the playCard methods in Table will detect it and use the corresponding prior call.

11.9 Integrating socket calls in the game engine

As we explained in the corresponding iteration, our socket transmits unitary data packets, in the sense they transmit individual moves.

Now that after more than a hundred pages you have seen many actions in the game are complex and need several user interactions to completely apply some cards effects, you probably understand and share that approach: the opponent gets more smaller and understandable updates about the game more frequently.

What does that mean? That we need different calls for different types of moves. If we transmitted the turn data all at once we could just have one packet type and parameterize it with a lot of options –much how a TCP packet works.

Is that a problem? Not at all. The nature of the game engine, that makes small amounts of work after every event, means we already have natural places where we can put our socket calls. For example:

```
            var sl:SocketLord = new SocketLord();
            protected function cardClickConfirmHandler(e:GameEvent = null){
                Registry.dlog("Engine.cardClickConfirmHandler");
                cardClickListeners(false);
                if(e != null) sl.playCard(e.id,e.choice);
                table.after();
                ui.update();
            }
```

After a card has been played we receive that ON_CARD_CLICK_CONFIRM we explained. That means at this point the card has been a) tested for playability and b) actually played without problems. So we tell it to the other end of the line –we already saw how the socket class is implemented. Another example:

```
protected function cardClickHandler_SaveHand(e:GameEvent){
        Registry.dlog("Engine.cardClickHandler_SaveHand");
        table.saveHand(e.id);

        sl.saveHand(e.id);

        ui.removeEventListener(GameEvent.ON_CARD_CLICK, cardClickHandler_SaveHand);
        ui.addEventListener(GameEvent.ON_CARD_CLICK, cardClickHandler);
        Registry.modeSaveHand = false;
        ui.allowOnlyHand(false);
        ui.update();
}
```

That's the function called –by an event- when the user clicks the card he wants to save by using the save-hand effect. So, just after telling our Table to save it in the hose, we also notify the opponent.

All other socket calls are integrated in the same way: whenever the state of the Table changes, it notifies the opponent's game to modify it in the same way.

Lastly, only the player that is not active has to wait for socket input. That means we need the functionality to switch when we want to listen and when to not for socket events:

```
protected function waitOpponentEvents(b:Boolean){
    Registry.dlog("waitOpponentEvents " + b);
    if(b){
        sl.addEventListener(RemoteEvent.ON_TURN_START, turnStart);
        //direct effects
        sl.addEventListener(RemoteEvent.ON_TURN_SYNC, turnSync);
        //effects that require actions
        sl.addEventListener(RemoteEvent.ON_TURN_SYNC_BUY, turnSyncBuy);
        sl.addEventListener(RemoteEvent.ON_TURN_SYNC_REUSE, turnSyncReuse);
        sl.addEventListener(RemoteEvent.ON_TURN_SYNC_NULLTOWN, turnSyncNulltown);
        sl.addEventListener(RemoteEvent.ON_TURN_SYNC_SAVE_HAND, turnSyncSaveHand);
        //effects that require previous selections
        sl.addEventListener(RemoteEvent.ON_TURN_SYNC_PRE, turnSyncPre);
        sl.addEventListener(RemoteEvent.ON_TURN_SYNC_PRE_START, turnSyncPreStart);
        sl.addEventListener(RemoteEvent.ON_TURN_SYNC_PRE_CLICK, turnSyncPreClick);
        sl.addEventListener(RemoteEvent.ON_TURN_SYNC_PRE_END, turnSyncPreEnd);
    }else{
        sl.removeEventListener(RemoteEvent.ON_TURN_START, turnStart);
        sl.removeEventListener(RemoteEvent.ON_TURN_SYNC, turnSync);
        sl.removeEventListener(RemoteEvent.ON_TURN_SYNC_BUY, turnSyncBuy);
        /*
        Etc
        */
    }
```

Basically, there's a listener for every of the packet types, that are linked through events to the functions that take the required actions. All of these actions follow the pattern:

```
protected function turn-sync-listener(e:SocketEvent){
```

```
            table.swapPlayers();
            /*
            Take action
            e.g. call the normal functions that take action on GameEvents
            */
            table.swapPlayers();
    }
```

This way, we can reuse lots of code instead of needing to have different functions to act on the player or its opponent, or putting if/else blocks everywhere to differentiate. To save a lot of trees, we won't copy here all those listeners and which functions do every one call: it's already in the game code and it's intuitive enough to understand and to follow the code.

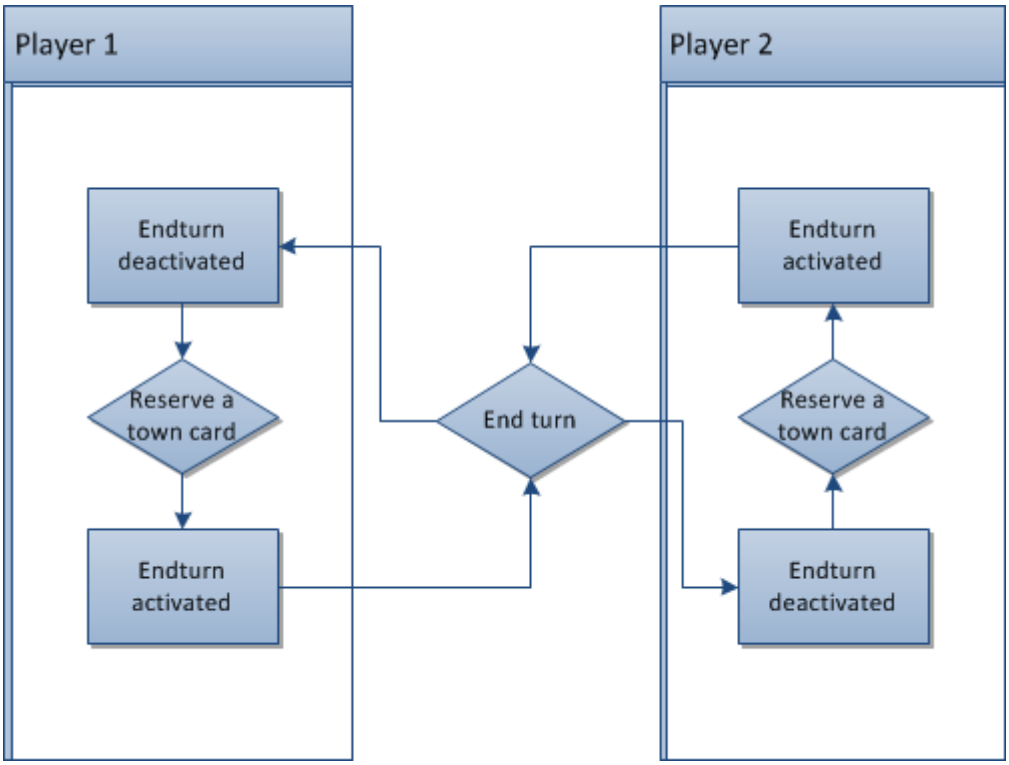
11.10 Game flow

We already have pretty much all we need to advance through phases in the game, as what it mainly involves is changing which players controls the flow of the game from that point, what we already implemented in our socket class. That means we have the implementation, and the server, to actually make possible this action's ownership.

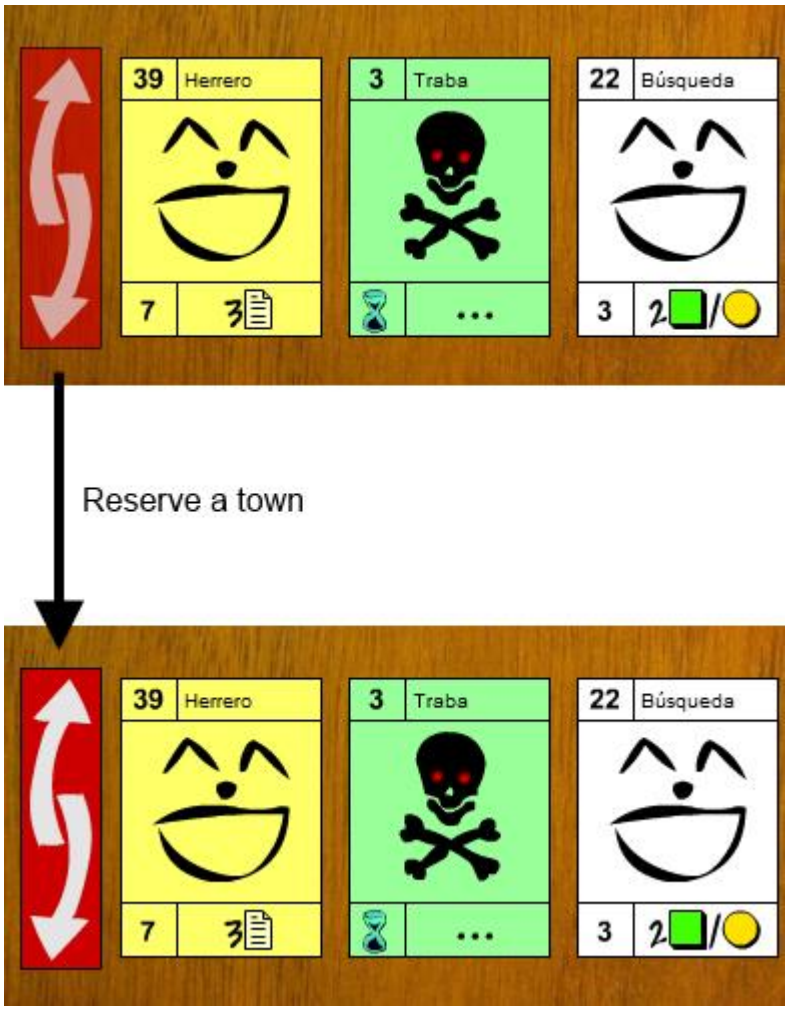
That means what's left is to give the user –human or not- the possibility to end the turn, and the game engine, together with the socket class and the engine, will take the logic steps needed to decide if the same player will repeat a turn, if it's the opponent who will take control, or if it's a journey-end and a new maintenance phase.

11.10.1 Turn ending

A few pages back, you've got a flowchart of the end-turn scenario. Just take look if it's not fresh in your mind. Now, we can see the only thing remaining is some way to enter in the first square of the graph: a method to tell the turn has to end. To complement that flowchart, take the next one into account and mix them both into your head –an actual flowchart representing the options mix is harder to understand than taking a moment to think.



When a player ends its turn, any of the players can take the next turn. In any case, any of them is able to end a turn: only the one than is playing can end it after reserving a town card before. We can represent this in the UI by graying out the endturn button until the player has reserved a town card.



In our UserInterface code, we add:

```
protected var pass:PassTurn = new PassTurn();
protected function addedToStageHandler(e:Event):void{
        Registry.dlog("UI " + stage + "\n");
        /* code */
        pass.addEventListener(MouseEvent.CLICK, passTurnHandler);
}
override public function passBlock(){
        pass.alpha = 0.5;
        pass.enabled = false;
        pass.mouseEnabled = false;
}
override public function passUnblock(){
        pass.alpha = 1;
        pass.enabled = true;
        pass.mouseEnabled = true;
}
```

The Block function fades the button –top image- and disables mouse interaction with, while the Unblock one –bottom image- unfades and activates it. You can see the clicks over the button are received by the passTurnHandler, that is implemented in the superclass Interface:

```
protected function passTurnHandler(me:MouseEvent = null){
        Registry.dlog("UI.passTurnHandler");
        passBlock();
        var _e:GameEvent = new GameEvent(GameEvent.ON_TURN_PASS);
        dispatchEvent(_e);
}
```

Note that it accepts a null as a parameter: it can be called either by the listener or by a function call in the code, which is useful if you consider that's what the AI will do using another Interface subclass.

11.10.2 Journey end, maintenance phase

Whenever an end turn happens, it can happen it ends up being a journey end, in which case the game enters into maintenance phase and then players have to choose cards and the one with highest initiative starts a turn –we already saw how it worked.

When a journey ends, the endJourney function in our Engine is called through an if-else block:

```
if(Registry.rules.canOpStartTurn()){
        Registry.dlog("   canOpStartTurn");
        waitOpponentEvents(true);
        sl.endTurn();
}
else{
        if(Registry.rules.canIStartTurn()){
                Registry.dlog("   canIStartTurn");
                turnStart();
                sl.endTurn();
        }
        else{
                //end Journey
                Registry.dlog("   end Journey");
```

```
                    endJourney();
                }
            }

        protected function endJourney(){
            Registry.dlog("Engine.endJourney");
            table.endJourney();
            ui.update();
            turnPlanListeners(true);
            sl.journeyEnd();
        }
```

Which makes call on the Table to do the data-wise operations of a jorney-end, activates the listeners for the plan phase and tells the other player the journey has finished.

This function in Table is a big block of lines that has do these operations in this order:

1. Every empty town card receives an extra PX as a reward for using it in the next journey. If it gets 3 PX, the card is destroyed and substituted by the first event in the line, and by the first event in deck if the line is empty.
2. All cards in playing areas and hands go to respective discard piles.
3. All players get their followers back -3-. Extra followers –up to 2 possible- are removed.
4. Remove the first event in the line –another external adventurer solved it-. Move the remaining cards, if any, to the left and complete the event line –has to have 6 cards- with cards from the event deck.
5. If a town card was nulled, denull it.
6. New journey starts with a planification phase.

These steps are followed in this endJourney function of the Table class, note the numbered comments refer to the numbered list up here:

```
public function endJourney(){
        Registry.dlog("Table.endJourney");
        var _c:Card;
        //1. Recompensa de PX
        for(var i:int = townCards.length-1; i >= 0; i--){
            _c = townCards[i] as Card;
            _c.denullTown();
            if(!_c.isReserved()){
                //add extra PX
                if(_c.addExtraPX() >= 3){
                    garbageTownCard(i);
            }
            }
            else{
                _c.unreserve();
            }
        }

        //2. Descartes
        while(playerCards.length > 0) playerDeck.push(playerCards.pop());
        while(playerGround.length > 0) playerDeck.push(playerGround.pop());
        while(playerDiscard.length > 0) playerDeck.push(playerDiscard.pop());
```

```
        while(opponentCards.length > 0) opponentDeck.push(opponentCards.pop());
        while(opponentGround.length > 0) opponentDeck.push(opponentGround.pop());
        while(opponentDiscard.length > 0) opponentDeck.push(opponentDiscard.pop());
        for(i=0;i<playerDeck.length;i++) playerDeck[i].straighten();
        for(i=0;i<opponentDeck.length;i++) opponentDeck[i].straighten();

        if(Registry.playerNum == 1){
                shuffle(playerDeck);
                shuffle(opponentDeck);
        }
        else if(Registry.playerNum ==2){
                shuffle(opponentDeck);
                shuffle(playerDeck);
        }

        //3. And followers
        playerDraws(5);
        while(playerHose.length > 0) playerCards.push(playerHose.pop());
        p1Stats.newDay();
        p1Stats.setMen(3);

        swapPlayers(); //copypaste of the previous lines :D
        playerDraws(5);
        while(playerHose.length > 0) playerCards.push(playerHose.pop());
        p1Stats.newDay();
        p1Stats.setMen(3);
        swapPlayers(); //end copy paste

        for(var j:int=0;j<5;j++){
                Registry.dlog(playerCards[j].getID() + " --- " + opponentCards[j].getID());
        }

        //4. Event line
        if(eventLine.length > 0) eventLine.splice(0,1);
        buildEventLine();

        //5. Reconstruction
        //undestroy cards

        //6. New Journey --> back to engine.class processing
}
```

All the function calls have pretty much been already been explained or are either self-explantory by the name. We'll just show the buildEventLine() method. When reconstructing an event line, some cards get an extra cost:

- First, second and third card have the buying cost written in the card.
- Fourth and fifth, cost 1 extra coin than the card's value.
- Sixth card costs 2 extra coins than the card's value.

```
public function buildEventLine(){
                Registry.dlog("Table.buildEventLine - " + eventLine.length);
```

```
                drawEvent(6 - eventLine.length);
                (eventLine[0] as GameCard).setExtraPrice(0);
                (eventLine[1] as GameCard).setExtraPrice(0);
                (eventLine[2] as GameCard).setExtraPrice(0);
                (eventLine[3] as GameCard).setExtraPrice(1);
                (eventLine[4] as GameCard).setExtraPrice(1);
                (eventLine[5] as GameCard).setExtraPrice(2);
        }
        protected function drawEvent(n:uint){
                for(var nn:uint = 0; nn < n; nn++){
                        drawOneEvent();
                }
        }
        protected function drawOneEvent(){
                Registry.dlog(eventDeck.length + "cards remain in the table deck");
                eventLine.push(eventDeck.shift());
                if(eventDeck.length == 0)
                {
                        //game finished
                }
        }
```

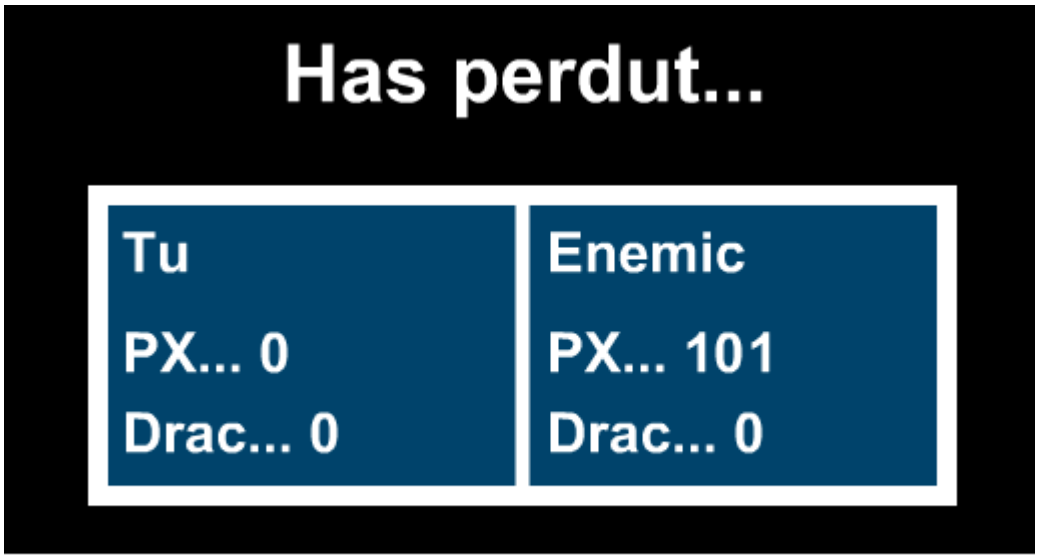
So we first draw the number of events missing to complete 6 intotal, and then we adjust the extra price. The setExtraPrice() call does nothing on Tethers, but on normal Quest Cards:

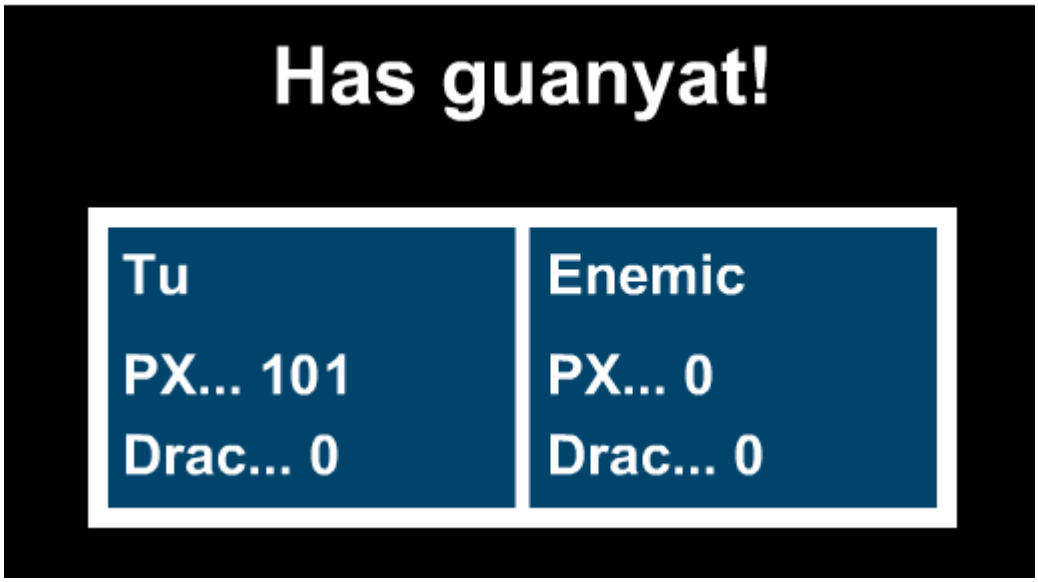
```
public class QuestCard extends GameCard{
    override public function setExtraPrice(p:uint){
            extraPrice = p;
            Helpers.initTextField(  tfPrice,
                                    (price + extraPrice).toString(),
                                    0, 168, 40, 24,
                                    tFormatBig,this);
     }

    public function getPrice():uint{return price + extraPrice;}
}
```

It instructs the card to show it's normal 'price' plus the 'extraPrice' in its bottom left box. It also sets extraPrice, and the getPrice methods returns the sum of both –this way we never alter the original price value.

11.11 Victory conditions

To capture when a victory condition is met, we have to add a listener to the Table class in the usual fashion that will catch an event thrown from PlayerStats when PX is over 100. From the Table we add to the event the values corresponding to the px and dragons killed, for each player. The Engine does sync the condition via Socket and, in both clients, a game over screen appears:





This screen is created by the class EndGameScreen and basically consists of 7 TextFields which content is initialized by consulting the Lang.xml file.

11.11.1 100 PX

When a player reaches 100 PX, he wins. The quickest and easiest way to get aware of it, is dispatching an event. From where? From the places where experience points are stored, of course:

```
public class PlayerStats extends EventDispatcher {
	/*
	Class code
	*/
	public function addPX(n:uint){
		px += n*pxMult;
		if(px >= 100){
			dispatchEvent(new GameEvent(GameEvent.ON_WIN));
		}
	}
}
```

Note we do need to subclass EventDispatcher to have access to the dispatchEvent method.

11.11.2 Two dragons killed

A player can also win by killing two dragons, which are quest cards of level 4 that can appear in the late game. In the same fashion, we notify if such condition is met:

```
public class PlayerStats extends EventDispatcher{
	/*
	Class code
	*/

	public function incDragons(){
		dragonsKilled++;
		if(dragonsKilled == 2){
			dispatchEvent(new GameEvent(GameEvent.ON_WIN));
		}
	}
}
```

11.11.3 Empty event deck

When the event line can't be completed with 6 cards or, what's the same, no more cards can be drawn from the event deck because it's empty, the game finishes –and the player with most PX wins. This time the correct place to get aware if this condition is met is in our Table class, that's where our card Arrays are controlled. So we just call our winHandler, that catches events too, with a normal function call.

```
public class Table extends EventDispatcher{
	/*
	Class code
	*/

	protected function drawOneEvent(){
		Registry.dlog(eventDeck.length + "cards remain in the table deck");
		eventLine.push(eventDeck.shift());
		if(eventDeck.length == 0)
		{
			winHandler(new GameEvent(GameEvent.ON_WIN));
	}
	}
}
```

12 Iteration 6

12.1 Objective

Build an Artificial Intelligence that can play Dungeon Realms versus an human player

12.2 Backlog

B61 - Design the AI.

B62 - Implement the designed AI

12.3 Effort Chart

We decided to stop thinking on an evaluation function before touching code, and instead play –literally- with the implemented AI to see how changes affected it. The implementation took longer than estimated mainly to bugs introduced and the difficulty to trace them in huge minimax trees.

Ref	Concept	Estimate	Real
B61.T1	Design how to generate game states	3	5
B61.T2	Design an evaluation function	3	1
B62.T1	Implementation	80	100
B62.T2	Play with modifications to the evaluation function	10	20
	Total	96	126

12.4 AI design

(Almost) Everything this humble developer knows about artificial intelligence has been learnt from Stuart Russell [29] and Peter Norvig [30], through some of their great papers but, mainly, from their book Artificial Intelligence: A Modern Approach [31]. I can't stop to recommend this reading, which teaches the basis for the AI of Dungeon Realms, to anyone interested in the field; Norvig's web site is also full of great articles –Norvig.com.

After this small preamble, how is video game AI created? The answer is that it needs two basic ingredients:

- The ability to, from any given game state, create (all the) possible game states that it can spawn. This is the same than simulating every possible move.
- A function to rate how good is any given game state, understanding it as how near is it from winning the game. An AI that wants to win usually choses the best rated state.

12.4.1 Evaluation function

As opposed to very complex games to evaluate, like Chess, Dungeon Realms is quite straight. Any game that is score-based to decide who wins has this common characteristic: the player with the best score is winning.

Hence, our AI must pursue the goal to increase its PX, the more it has the better. But, it also has to consider its enemy's score. Starting with 0 points each player, consider these scenarios:

1. Player A ends with 10 points, player B ends with 15 points.
2. Player A ends with 10 points, player B ends with 10 points.
3. Player A ends with 5 points, player B ends with 0 points.

What's better for A? Clearly the scenario 1 is not: he's losing. In scenario 2, A has more points that it has in scenario 3, is it better to choose option 2? No; consider calculating points(A)-points(B):

1. 10 – 15 = -5
2. 10 – 10 = 0
3. 5 – 0 = +5

Clearly, the last scenario is the best one because A has the most advantage. Hence:

- Being f(n) = rating of the game state, then
- $f(n) = points(A) - points(B),\ A, B > 0$

Also, since in Dungeon Realms a player's experience can never decrease:

- Being $S_{i \in 1..n}$ a game state achieved from S, then
  - $points(A)_{S_i} \geq points(A)_S$
  - $points(B)_{S_i} \geq points(B)_S$

So far, this formula looks good over paper and we'll try it on the implemented AI instead of squashing our brains. We'll show later how this evaluation functions is, while good on concept, bad for playing purposes.

12.4.2 Generating game states

Dungeon Realms has quite a variety of possibilities considering cards can be interacted with from various places and there is a lot of different effects. From any given game state, we can:

- Play any of the cards in hand.
- Reserve any of the unreserved town cards.
- Play any of the reserved town cards.
- End the turn, if already reserved a town this turn.
- If in planification phase: play any of the cards in hand without applying its effects.

And every card played, can also spawn multiple states depending on the effect:

- If it has an or operator, try each possibility.
- If it has an arrow operator, choose # times to use it, from 1 to the effect factor.
- For each discard or garbage effect, up to the factor choosed:
  - Discard/garbage any combination of cards for the required amount
- For each buy/cart effect:
  - Buy an affordable quest event
  - Cancel buying and lose the "remaining carts".
- If playing a planification effect:
  - Save for the next journey any of the cards in hand.
- If playing a lose-money, lose-follower effect:
  - Choose how much from 1 to the factor of effect
- If playing a double-use effect:

- Play any of the cards in the playing area.
- If playing a null-town effect:
  - Null any unreserved town.

12.5 AI implementation

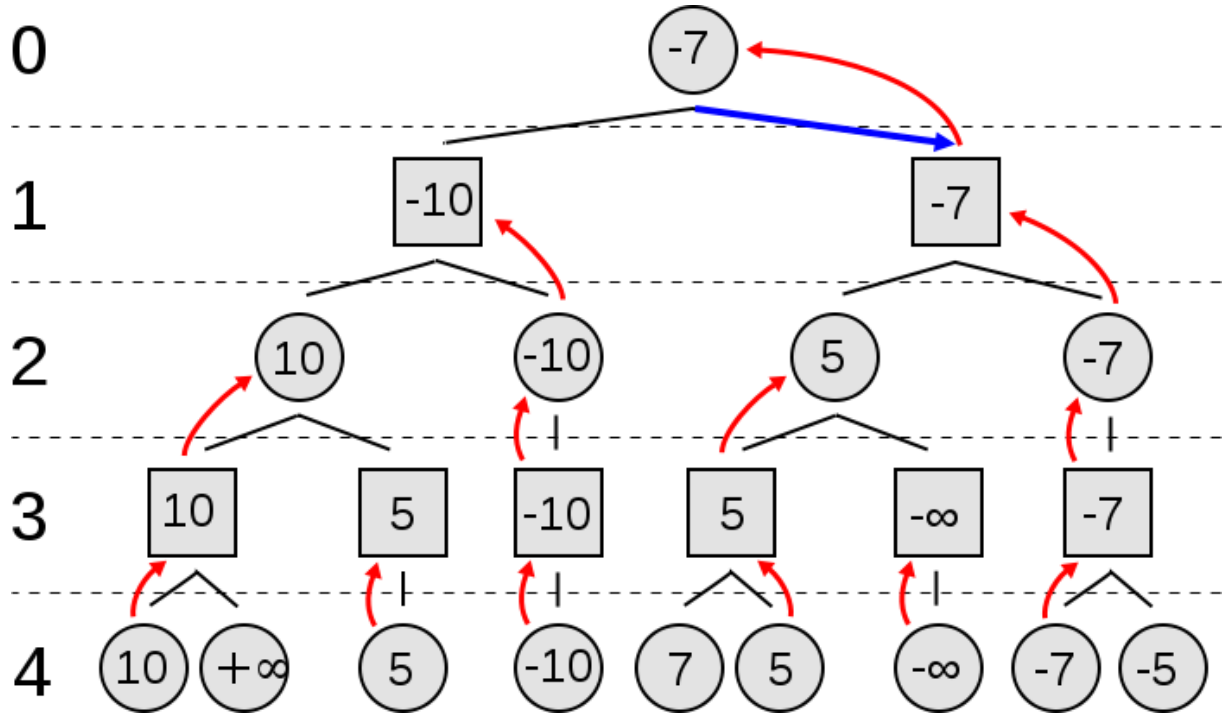
Programming an AI is a very serious business: an algorithm has to produce the illusion of human intelligence –which some humans actually lack. Since video game AI is centered on appearance of intelligence and good gameplay, its approach is very different from that of traditional AI; workarounds and cheats are acceptable –to achieve better processing performance and as long as they're not obvious to the player- and, in many cases, the computer abilities must be toned down to give human players a sense of fairness.

Note that since around 2005 some AI middleware started to appear for game development, and some of them have even become somewhat popular in the last handful years, being used in popular games [32]. The fields they cover, however, are almost-exclusively centered in pathfinding, navigation, collision-avoidance and machine-learning, which are the fields also related to the all-mighty military I+D businesses that use these same middleware for simulations.

12.5.1 Minimax

For a specific task like programming an artificial player for a card game, there's really not much more to do than manually programming it. For adversarial, turn-based, games, the choice is, almost-always, the powerful minimax and its derivatives.

In the following image, circles represent the moves of the player running the algorithm (maximizing player), and squares represent the moves of the opponent (minimizing player). The values inside the circles and squares represent the value α of the minimax algorithm. The red arrows represent the chosen move, the numbers on the left represent the tree depth and the blue arrow the chosen move.



This, however, does not directly adapt to our game nature, as minimax premise is that of alternate moves for each player. But, in Dungeon Realms, players can do lots of actions during their turn! The easiest

approach in this scenario is to introduce some kind of marker such that, for the player that is not active, the only possible action is one such that the game state remains equal –that is, takes no action on the state.

However, that would cause, with the basic minimax algorithm, than a tree is recursively built and processed to take one action, and then a new call is done that would rebuild a tree identical to a branch of the original one –since the ghost opponent would have left the state unmodified.

12.5.2 AI

And here's where an inspiration of pushdown automatons plus Markov's related theories on past-independent states can save the day. We will build a tree maximizing the AI's gains as minimax postulates, but returning with the "winner" –the best solution- state a stack with the succession of moves the AI needs to take.

```
public class AIInterface extends Interface{

	protected var ais:AIState = new AIState();
	protected var table:Table;
	protected var timer:Timer;
	private var nodesEvaluated:uint;
	private var moves:Vector.<AICard>;

	private function minimax(){
		movesVal = int.MIN_VALUE;
		nodesEvaluated = 0;
		var dsa:int = max(ais);
		Registry.log("AI evaluated " + nodesEvaluated +" possibilities");
		timer.start();
	}
	private function max(a:AIState):int{
		//terminal or sterile node -> return utility
		if(a.terminalTest()) return a.utility();
		var scs:Vector.<AIState> = getAllSuccessors(a);
		if(scs.length == 0) return a.utility();

		var tmp:AIState;
		var best:AIState;
		var val:int;
		var upper:int = int.MIN_VALUE;
		for(var i:int = 0; i < scs.length; i++){
			tmp = scs[i];
			val = max(tmp);
			if(upper < val){
				upper = val;
				setMoves(tmp, upper);
			}
		}
		return upper;
	}
	private var movesVal:int;
	private function setMoves(a:AIState, v:int){
		if(v > movesVal){
			moves = a.moves;
			movesVal = v;
```

```
		}
	}

	private function getAllSuccessors(a:AIState):Vector.<AIState>{
		nodesEvaluated++;
		var i:int;
		var r:uint;
		var b:AIState;
		var so:Vector.<AIState> = new Vector.<AIState>();

		switch(a.mod){
			case AIState._RES:
			//reserve a town card
				for(i = 0; i < a.townCards.length; i++){
					//trace("reserve",i);
					//gv.dlogAI("  _res " + i);
					if(a.canReserveTown(i)){
						//gv.dlogAI("    can " + i);
						b = new AIState();
						b.parseAIS(a);
						b.reserveTown(i);
						so.push( b );
					}
				}
			break;

			case AIState._PLAY_POST:
				b = new AIState(); b.parseAIS(a);
				b.resumePost();
				so.push( b );
			break;

			case AIState._DISCARD:
				for(i = 0; i < a.playerCards.length; i++){
					b = new AIState(); b.parseAIS(a);
					b.disCard(i);
					so.push( b );
				}
			break;

			case AIState._GARBAGE:
				for(i = 0; i < a.playerCards.length; i++){
					b = new AIState(); b.parseAIS(a);
					b.garbageCard(i);
					so.push( b );
				}
			break;

			case AIState._CART:
				//player can stop carting at any moment
				b = new AIState(); b.parseAIS(a);
				b.endCart();
				so.push(b);
				//or continue buying
```

```
		for(i = 0; i < a.eventLine.length; i++){
			if(a.canBuyEvent(i)){
				//gv.dlogAI("    " + i);
				b = new AIState(); b.parseAIS(a);
				b.buyEvent(i);
				so.push( b );
			}
		}
break;

case AIState._SAVE_HAND:
		for(i = 0; i < a.playerCards.length; i++){
			b = new AIState(); b.parseAIS(a);
			b.saveHand(i);
			so.push(b);

		}
break;

case AIState._PLAY:
		//ending turn is always an option
		b = new AIState();
		b.parseAIS(a);
		b.endTurn();
		so.push(b);

		for(i = 0; i < a.townCards.length; i++){
			switch(a.playMode_Town(i)){
				case AICard._PLAY_OR:
					//right effect
					b = new AIState(); b.parseAIS(a);
					b.playTown(i,2);
					so.push(b);
					//no break here
				case AICard._PLAY_YES:
					//left effect
					b = new AIState(); b.parseAIS(a);
					b.playTown(i,1);
					so.push(b);
					break;
				case AICard._PLAY_PRE:
					r = Math.min(a.getPreMax_Town(i),
						a.getPostMax_Town(i));
					for(j = 1; j <= r; j++){
						b = new AIState();
						 b.parseAIS(a);
						b.playPre_Town(i,j);
						so.push(b);
					}
					break;

				case AICard._PLAY_NO:
				default:
					break;
```

```
                        }
                    }
                    for(i = 0; i < a.playerCards.length; i++){
                        switch(a.playMode_Hand(i)){
                            case AICard._PLAY_OR:
                                b = new AIState(); b.parseAIS(a);
                                b.playHand(i,2);
                                so.push(b);
                            case AICard._PLAY_YES:
                                b = new AIState(); b.parseAIS(a);
                                b.playHand(i,1);
                                so.push(b);
                                break;

                            case AICard._PLAY_PRE:
                                r = Math.min(a.getPreMax_Hand(i),
                                    a.getPostMax_Hand(i));
                                for(j = 1; j <= r; j++){
                                    b = new AIState();
                                    b.parseAIS(a);
                                    b.playPre_Hand(i,j);
                                    so.push(b);
                                }
                            case AICard._PLAY_NO:
                            default:
                                break;
                        }
                    }
                break;
                case AIState._END_TURN:
                break;
            }
            return so;
        }
```

After you looked at the code, you must have realized AIInterface makes use of the class AIState. AIState is an abstraction of the game state: take everything in the Table, Cards and PlayerStats and stripe it of everything not useful –everything related to graphics, card rotations or variables that mark where they are, etc. We duplicate the date we need because, when programming a game AI, it's a best-practice to separate all data from the actual game state. Taking only what we need, objects are also much smaller and, hence, we get an speed gain.

An AIState instance, has a variable that defines the current game state, in the sense of which cards and actions can the player take:

```
public class AIState {
        public static const _RES:int = 1;
        public static const _PLAY:int = 2;
        public static const _CART:int = 3;
        public static const _SAVE_HAND:int = 4;
        public static const _DISCARD:int = 5;
        public static const _GARBAGE:int = 6;
        public static const _PLAY_POST:int = 7;
        public static const _END_TURN:int = -1;
```

```
        public var mod:uint;
        //implementation
}
```

This status is stored in the mod variable depending on the actions taken over the instance:

- _RES: only action possible is to reserve a town card.
- _PLAY: normal game state where town and hand cards can be played, or the turn can be finished.
- _CART: a card was played with a card effect, only action possible are to buy a card from the event line or to stop buying cards.
- _SAVE_HAND: similar to the previous, only action possible is to save a card from the hand
- _DISCARD: only action possible is to discard a hand card to the discard pile
- _GARBAGE: only action possible is to remove a hand card from the game
- _PLAY_POST: a card with a pre was played and the pre-effect done, only action possible is to apply that card's post-effect.
- _END_TURN: no more actions are possible.

So, for example –we won't copy here all the code because it's already in the game-, when the AI chooses to, given an AIState such that mod=_CART, buy an event –already has checked to have enough money for it-, this function is called:

```
public function buyEvent(i:int){
        moves.push(eventLine[i]);
        p1Stats.subMoney(eventLine[i].price);
        if(eventLine[i].isGesta())
                p1Stats.incDragons();
        playerDiscard.push(eventLine[i]);
        eventLine.splice(i,1);
        for(var j:int = i-1; j >= 0; j--){
                if(eventLine[j].tether){
                        playerDiscard.push(eventLine[j]);
                        eventLine.splice(j,1);
                }
        }
        nEffect--;
        if(nEffect < 0) throw new Error("Inconsistent #buyable-cards")
        else if(nEffect ==0){
                endCart();
        }

}
public function endCart(){
        var c:AICard = new AICard();
        c.cID = AICard._END_CART;
        moves.push(c);
        mod = AIState._PLAY;
}
```

The state is modified so that the player loses money and the event is added to the discard pile together with any tethers at its left. If there are no more credits –nEffect- available to continue buying, the endCart

function is called –which can also be called even if there are remaining credits-, which modifies the state to mark mod=_PLAY: hand and town cards can be played again, or the turn can be finished.

Note how in both functions, we have a moves.push(…) call. That's the moves stack we mentioned when discussing the AIInterface's algorithm. In this stack we put one after the other the cards the AI will play –if it's the selection-, plus some special markers. The AICard objects we stack are the AI representation of the cards –id, effects, price, extra PX and a few more fields-.

We've decided to represent special moves as cards, too. For example, a card with the id = -1, means to end the turn.

```
public class AICard {
	public static var _END_TURN:int = -1;
	public static var _END_CART:int = -2;
	public static var _PRE_CHOICE:int = -3

	//implementation
}
```

12.5.3 Making the moves as a human

And, back to the AIInterface, which subclassed our Interface that threw events to the game engine; how does it make the moves?

First thing we realized if we just iterated the moves is that, since the AI already knew all of them when it started, they were all made at the speed of light. No human could keep trace of the order of movements. The solution? Yet more events: this time, timed –what a pun.

Whenever it's a player's moment to take action, the mouseUnblock function is called –ok, the name should be changed, but it has been named like that for so many months…-:

```
override public function mouseUnblock(s:int = -1){
	ais = new AIState();
	ais.parseTable(table);

	if(s == Constants.PH_PLAN)
		randomPlan();
	else
		minimax();
}
```

Note than in planning phases, we just use a random card. This way, the AI does not always play perfectly.

```
private function timerMoves(e:TimerEvent){
	var c:AICard = moves.shift();
	switch(c.cID){
		case AICard._END_TURN:
			timer.stop();
			passTurnHandler();
			break;
		case AICard._END_CART:
			endCarting();
```

```
                    break;
                case AICard._PRE_CHOICE:
                    var _e:GameEvent =
                        new GameEvent(GameEvent.ON_PRE_CHOICE);
                    _e.id = c.reserved;
                    _e.choice = c.cOR;
                    dispatchEvent(_e);
                    break;
                default:
                    clickCard(c.cID, c.cOR);
                    break;
            }
        }
```

The timer event is instantiated in the AIInterface creator, and can be set to fire to whatever time we want –e.g. 1 second. Note the timer is started when there's a complete move stack to play, and it's stopped when there are no more actions left for the AI to take. The overall result is that the AI seems much more natural than letting it play at CPU speed.

12.6 Back to the state evaluation

We introduced earlier what seemed like a good way to evaluate our states, while advancing it wasn't actually as it seemed. Basically, we logically induced from some examples how the difference in PX between both players was an objective indicator to evaluate game states.

12.6.1 The problem

But, there's always a but when using induction to create general rules: our logic was flawless, but limited. With a god-style AI that can process every possible future state up to every possible game over scenario, then it would always prevail and smash opponents with our flawless logic. However, our AI has to evaluate a non-computable number of states and, as such, many secondary effects take place in its playing style if it only takes care about experience points:

- If the AI can't gain more PX in a branch than its opponent, it will never be taken.
    - Same happens if it gains PX, but less than its opponent
- It will prefer even 1 experience point to any other playing option, no matter how good the alternative might be in the mid or long run.

If you consider the starting turn, the AI has to calculate over 200.000 possible transitions for only its opening turn. If you ask to it to compute the opponent's, too, we're squaring that number:

- $(2 \times 10^5) \times (2 \times 10^5) = 4 \times 10^{10}$.

You can consider the opponent will have 8 possible towns to reserve instead of 9, so you get eight nights of 4 and it becomes a 3.55, which still doesn't change anything when you consider the faster CPUs available we have are in the factor of 10^9 Hz. The opening example we told does already need near 10 seconds to compute in a high-end computer with an overclocked 3.8 Ghz AMD Phenom II CPU. Trying to go for the next turn's computation we'd need 3.55×10^6 seconds, which is half an hour short of the 100 hours mark.

12.6.2 The heuristic solution

The previous explanations mean we can't rely on harsh computation power to have a strong artificial intelligence, and the solution is to use a heuristic function. How do we create them? There are two options:

1. Think and specificate a great evaluation algorithm and then implement it.
2. Implement a simple, yet generally-good, evaluator and playtest the AI to make adjustments and extensions.

As Norvig and Russell point in their world-acclaimed book, option number 1 is the best possible way to lose your time and fill the paper recycle bin. Option number 2 is the best approach, and is where experienced game AI developers shine over the rest: intuition and smelling are a factor –pity we don't have the experience.

Hence, we'll have to add entropy, beyond the PX reference, to our evaluator. To start with… is acquiring cards good? Yes. The more cards the player buys the more resources it will have available for success. If *C* stands for *Cards*:

$$C \equiv C_{hand} \cup C_{deck} \cup C_{ground} \cup C_{discard} \cup C_{hose}$$

Then, if we consider:

$$f_1(x) = |C| \times PX$$

We're pretty much giving a big twist to our AI, as it will now try to get new cards as well as increasing PX. We could also give different weights to each of the factors, but it is fine as of now.

However, if we look again at the definition of C_1, we have collateral effects, by its definition, once we put it into f_1:

- Drawing cards is not encouraged: a card in the deck has the same value than a card in the hand. A card that hasn't still been drawn is definitely useful for nothing in Dungeon Realms, so it should have less value.
- Using the planification –save card- effect won't happen: for one side, a card in the hose has the same value than in the hand, while in the other side of things, using this is a long-term decision the AI can't process for possible benefits; the result is it won't use it.
- Should the hand, discard pile and ground area number of cards should be valued equally? We think so:
    - Valuing a played card higher than keeping it in hand would encourage using some cards which do not give an actual benefit.
    - The discard pile is the main number affected when buying, as a just-acquired quest card goes here. Its value could be increased, but shouldn't: at some point when drawing cards, the deck gets empty and the discard pile is converted into the new deck, hence getting empty. This would mean that the drawing action would actually be penalized every time the AI draws more cards than the current amount in the deck.

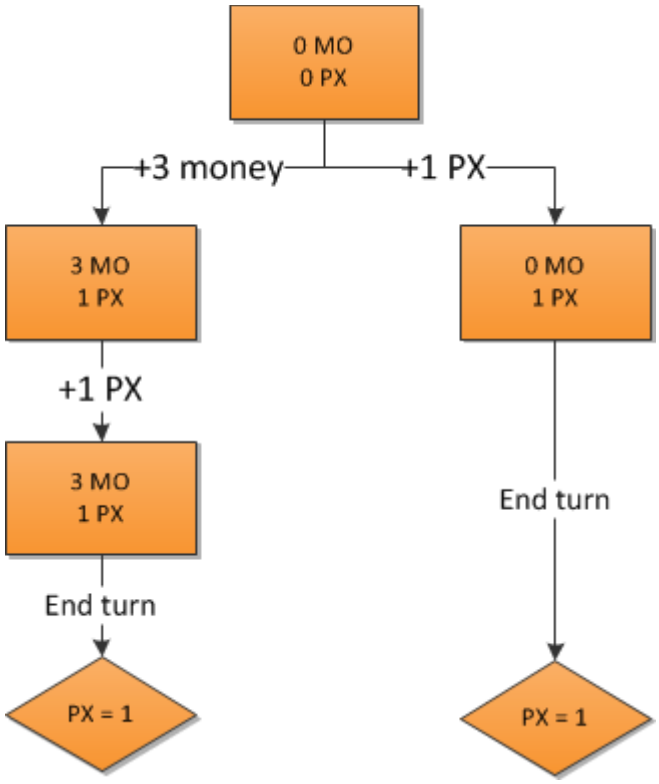
Hence, we add some sauce and are left with:

$$C_H = C_{hand} + C_{ground} + C_{discard} + 2 \times C_{hose} + \frac{C_{deck}}{2}$$

Considering the previous facts, we double the value of hosed cards and diminished the value of cards in the deck. And so far:

$$f_2(x) = C_H \times PX$$

Which is fine enough when tested, but still makes the AI seem dumb: what happens with money? When we instruct the AI to only care about experience points, we're omitting a secondary game value: money. Then again, if the AI was able to consider every possible future game state, it wouldn't be a problem. However, since it can only compute up to a certain point, coins can be left unused. Consider this (striped down) game state evaluation tree that illustrates the situation:



Assume the AI builds first the left branch –remember that minimax is a depth-first algorithm. It first gains three coins, whether through 1 or more cards in its hand or town, then 1 experience point and then ends the turn and, in the evaluation function, PX takes the value 1. The algorithm then builds the right branch in which it gains 1 experience point and then ends the turn; since the evaluation is not better, the branch is discarded and the AI keeps the left branch actions.

Also, having unused coins at the end of a turn, when they are lost, makes the AI seem everything but human. Hence, we will highly disregard that attitude and get our final heuristic equation:

$$h(x) = C_H \times (PX - MO)$$

This simple, yet highly tested, heuristic function gives great results and will challenge even the experienced Dungeon Realms players.

13 Extra: rule change, a.k.a. bug, for buying cards

After iteration 5, we got aware of a misunderstanding of the rules concerning how cards are bought:

- What was understood: you could buy quest cards by either paying its money cost or using a card with the cart effect/symbol, getting it for free.
- How the rule actually is: to buy a quest card, you must use a cart effect and also pay the money cost. That is, you can only opt to buy a card, always paying money for it, when using a cart effect. Unused cart effects are not stored in any way.

Since it's always good the learn from our own errors, we're documenting this rule change, or bug solving depending on how you want to view it, in this section for two reasons:

- This fact was known very late in the development when all of the documentation was done. Please be aware some of the texts and images referring to how this effect works may explain the actual rule incorrectly –nothing too serious, though.
- We'll fool-proof our code adaptability: Will it be easy?

Let's go for the task making some memory: our game engine did use pretty much the same calls and code to either buy or get a card for free. The difference is we did subtract the money from the player when not using the cart effect. So, we've had to:

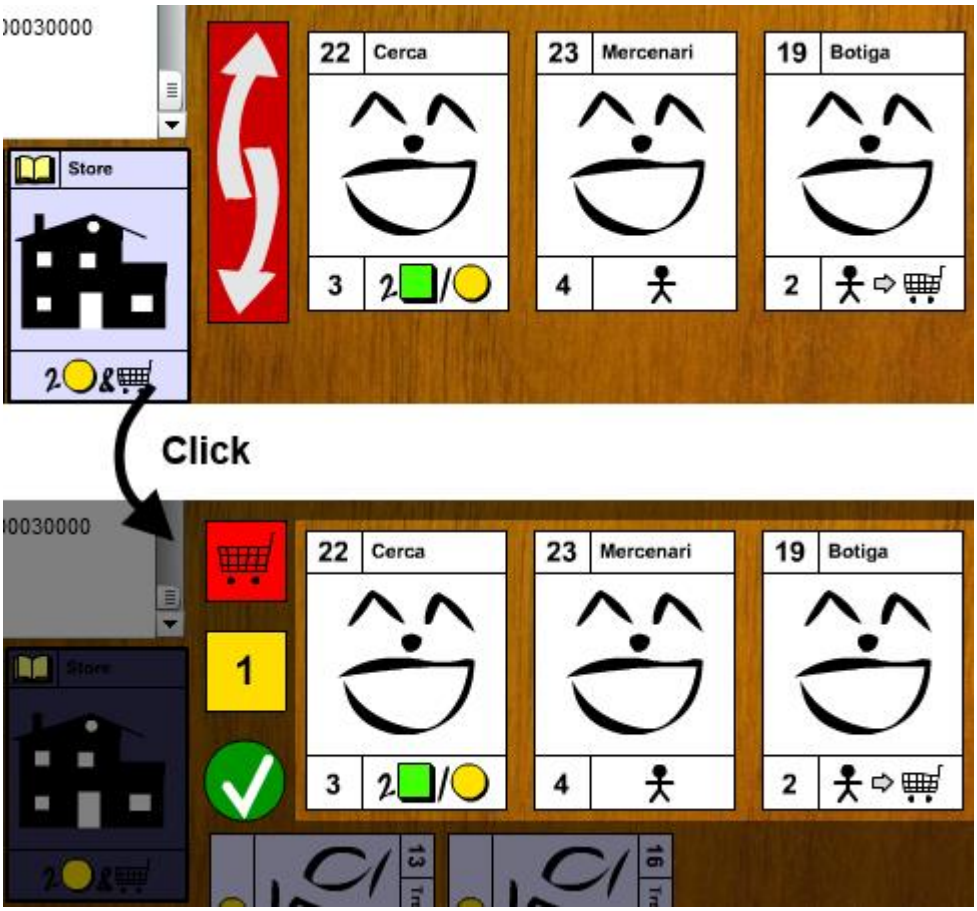
- Deactivate the possibility to buy a cart when not using the effect meant for it. This is done deactivating mouse interaction with the event line.
- The effectCartHandler(…) method prepared the game engine to process clicks on event cards as buy-for-free by redirecting the click events to a new handler. This, cardClickHandler_Cart(…), needs to check, someway, when there is enough money to buy a card and, if so, add a call added to subtract money from the player. We decide to do it in the cartCard(id) function in Table class, and make it return a Boolean false when the card can't be bought.

```
if((eventLine[i] as QuestCard).getPrice() < p1Stats.getMoney() return false;
```

- When it can be bought, we continue with the original code, but add a line to subtract the card cost.

```
payMoney((eventLine[i] as QuestCard).getPrice());
```

- (Only for the graphic interface) Add a button for the human player –UserInterface- to tell when he has finished buying, that will be used when a card allows to buy N cards and the player wants to buy less than N. When buying the Nth card, the button has to disappear.



The yellow counter states the number of carts that can still be bought, when it arrives to 0 the screen returns to the upper state. If the player decides to press the green button when he can still buy cards, the screen also returns to the upper state.

Total time taken: 3 hours, plus 1 hour to document it

14 Cost study

14.1 Human

This thesis spans an approximated effort of 740 human hours as we can see in the following table; more granular estimations can be viewed at each of the iteration's effort charts.

	Estimated	Real world	Deviation
Iteration 0	23	36	56%
Iteration 1	11	19	72%
Iteration 2	90	95	5%
Iteration 3	25	23	-8%
Iteration 4	30	45	50%
Iteration 5A	130	177	36%
Iteration 5B	115	170	47%
Iteration 6	96	126	31%
Rule change contingency	0	4	N/A
Project presentation	25	25	0
Cost study	25	20	-20%
…Total	570	740	29%

Globally, the cost of the project has been a 29% higher than previously estimated. Is that bad? Definitely not: it's an awesome number; in the real world, any software that does not get beyond the +50% mark is considered to be a great economic success by both the client and the provider, and we've been able to keep in this range.

14.2 Economic

The previous 740 hours to develop Dungeon Realms can be segregated into, approximately, 10% of pure software-design tasks, 15% testing and 75% of actual developing tasks –implement, redesign, learn.

Aggregated tasks per role	Cost (€/hour)	Time (hours)	Total cost (€)
Design	30	75	2.250
Developer	25	555	13.875
Tester	15	110	1.650
…Total			17.775

But the actual costs aren't limited to that, as there are another couple points to consider:

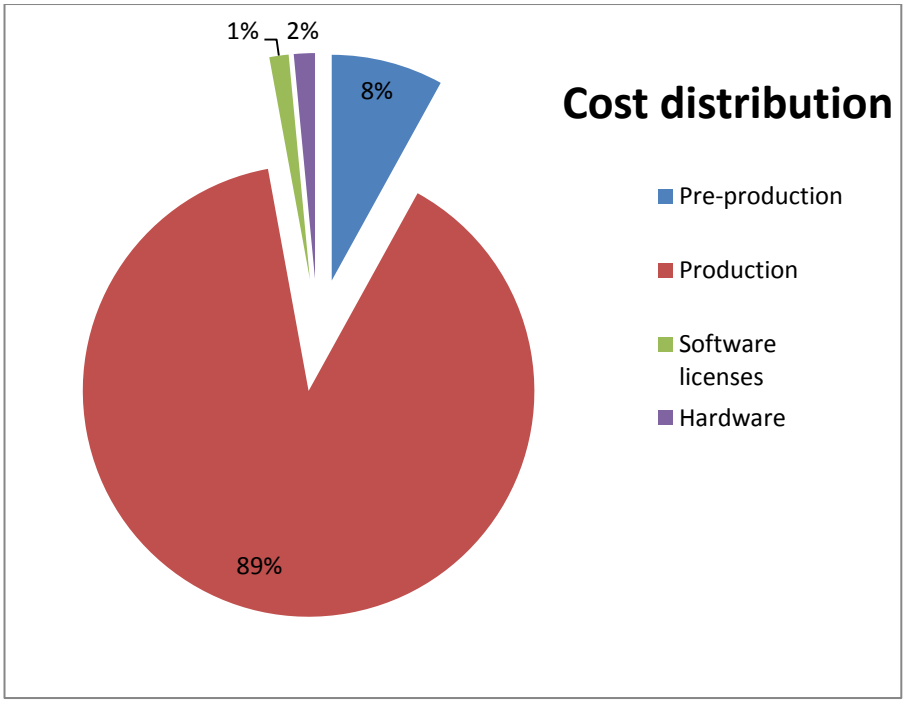
- Hardware: we required a PC to develop the game. Since to special or dedicated components are needed, a 300€ mainstream computer is enough.

- Pre-production: the (cardboard) game design, done by José Carlos: we have to remember we've just converted a real world game to a PC game. This means Dungeon Realms did actually take some work to exist; in concrete, it took J.C. 40 hours to design the game and another 40 hours of play-testing to polish rules. If we assume an average cost of 20€ per hour, we've got a total of 1.600 euros to create Dungeon Realms.
- Software used and licensing cost:

Software	License cost (euros)	Notes
Microsoft Windows 7 Pro	139	Free for education*
Microsoft Office	125	
Microsoft Visio	399	Free for education*
Adobe Flash Professional	599	75% discount for students
Paint.net	0	
…Total	1.262	275€ with allowances

*As per the MSDNAA terms between the college –FIB- and Microsoft.

Hence, we've got a total of 1.600€ to create Dungeon Realms, 17.775€ for the actual production of the PC version, and 1.262€ in licenses of the needed software.



	Cost (euros)
Pre-production	1.600
Production	17.775
Licenses	275
Hardware	300
Total	19.950

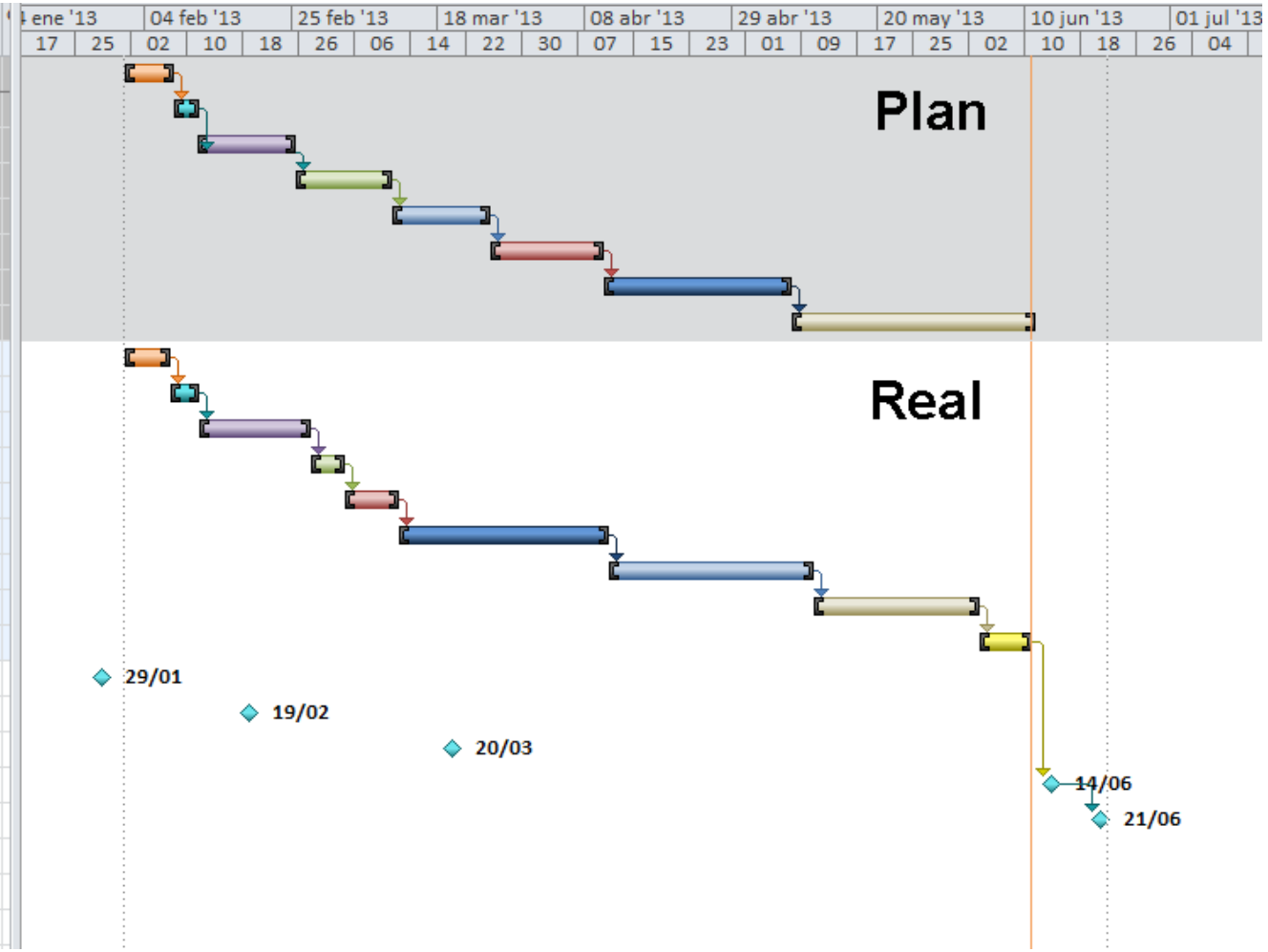
14.3 Time planification

In the next page, we provide a unified gantt graph to contrast the original planning –grayed out- compared with the actual time taken to complete the steps. Note we've colored differently each iteration of the gantt to facilitate a more visual way to note the distinctions.

As we mentioned in iteration 4, our original idea was to create a way to serialize the game engine to send it through the network –which could also be easily modified to be sent through mail. Since it seemed quite a challenge –we actually couldn't accomplish it-, we chose to do it before writing engine code that could depend on it.

Due to that fact, we got our major agenda disruption as we dropped the idea and chose a much easier and faster –red gantt bar- way to implement communications, but that had the cost of a slower game logic implementation –light blue.

Iteration 0 - Project study	40 horas
Iteration 1 - Setting up	15 horas
Iteration 2 - UI and card parsing	80 horas
Iteration 3 - Interactivity	80 horas
Iteration 4 - Engine event logic	80 horas
Iteration 5 - Communication	100 horas
Iteration 6 - Game rules	150 horas
Iteration 7 - Artificial Int.	200 horas
Iteration 0 - Project study	36 horas
Iteration 1 - Setting up	19 horas
Iteration 2 - UI and card parsing	95 horas
Iteration 3 - Interactivity	23 horas
Iteration 4 - Communication	45 horas
Iteration 5A - Game rules	177 horas
Iteration 5B - Engine event logic	170 horas
Iteration 6 - Artificial Int.	126 horas
Non-iteration documentation	45 horas
Project Registration	0 días
Project Registration (payment)	0 días
Project pre-Report	0 días
Project deadline (max)	0 días
Project presentation (max)	0 días



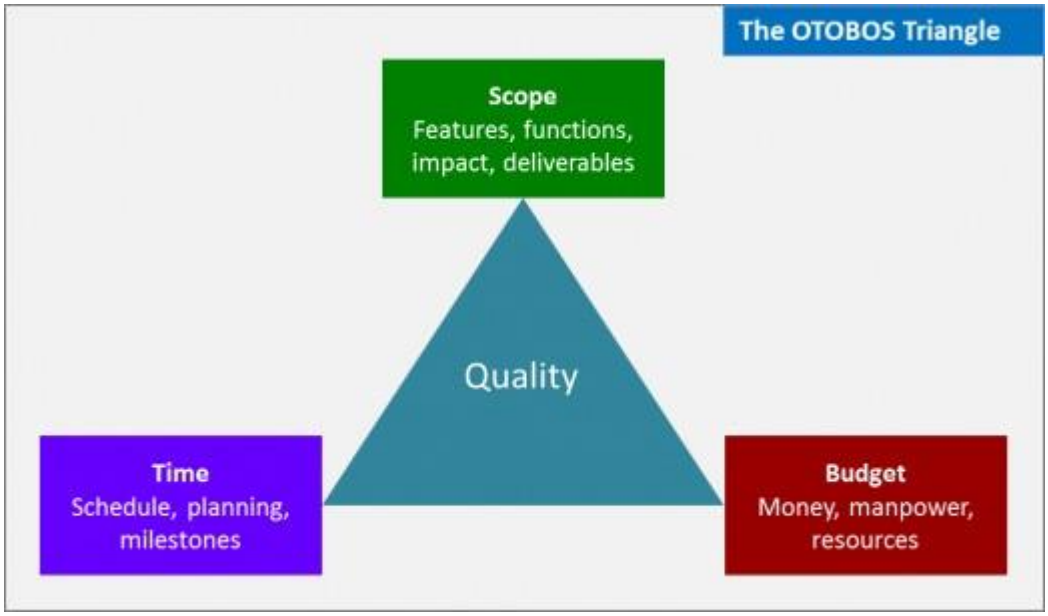
15 Conclusions

15.1 OTOBOS triangle

The OTOBOS is a project success measurement that stands for On Time – On Budged – On Scope. This is so because the three basic vectors of any project, of any discipline, can be measured through:

- Scope: exact definition of the project's goal. What does include and what doesn't include the project?
- Time: what's the must-be-done date for the project and the milestone and derivelable due dates?
- Budget (or Resources): how much workforce and resources can be invested in the project?

Experienced project managers teach to pick one of the vectors and never, ever, change it during the life of the project and, that this election, is usually behind the project's mean of existence. For example, a prototype to be shown at a showroom has a closed final date –either it's ready for the showroom or it doesn't need to be finished. Then, of the other two remaining, you're supposed to pick one of the vectors and try to contain it, while the risks and contingencies will make a pressure that will escape –make it grown- through the remaining vector.



As we wrote in the initial abstract, the idea behind the Dungeon Realms project was to port a cardboard game to a computer implementation, within the planification frame of a thesis:

- Scope:
  - ✓ Port the game Dungeon Realms to PC.
  - ✓ Support for multiple languages.
  - ✓ Play over Internet.
  - ✓ Play versus an Artificial Intelligence.
  - ✓ Deliver a mid-project report.
  - ✓ Demonstrate the use and strengths of agile methodologies within the frame of video game development.

- Time
  - ✓ Project deadline met: June 14, 2013
  - ✓ Presentation deadline met: June 21, 2013
  - ✓ Project planification calendar met: completed June 3 2013, eight days earlier than estimated.
  - ✓ Mid-project report deadline met: March 20, 2013.
- Budget/Resources:
  - ✓ Workforce: 1 developer.
  - ✓ Resources: 1 computer with software licenses; less 3% of the project budget.
  - ✓ Budget: 20.000€, only a fraction of the cost of a commercial game.

Hence, we consider this thesis, *Dungeon Realms: Design and Implementation*, an absolute success due to meeting all of the goals and subgoals.

15.2 Last words

Game development is in crisis—facing bloated budgets, impossible schedules, unmanageable complexity, and death march overtime. It's no wonder so many development studios are struggling to survive, while even more have already closed their doors. Fortunately, there is a solution: agile methods are already revolutionizing development outside the video game industry. In this project we've shown an approach to successfully apply these methods to the unique challenges of game development.

Having continuous integration as a main goal we've been able to keep the game always in such a state that, at every developer bedtime, it could always be compiled –short of two nights during all of the development. This means that, during the second half the project, the game has always been in a playable state, with the functionalities available at every moment. Thanks to this, small goals can always be achieved, increasing the will to work to see immeadiate changes and to recognize bad approaches immeadiately –just think we had spent the 100 hours planified for communications instead of being able to test and drop our idea at just 15 hours lost!

Increasingly, game developers and managers are recognizing that things can't go on the way they have in the past. Game development organizations need a far better way to work. Agile game development gives them that –and brings the profitability, creativity, and fun back to game development.

And, to finish, we'll relaunch and expand a question we made at the start of the project: will small, independent developers that use modern methodologies and techniques be able to take on the current, crashing, market? At this point you, my fellow reader, should've built an answer yourself.

16 Annex: Game rules

In Dungeon Realms you embody an amateur adventurer that leaves its hometown, avid of challenges and adventures. Slowly, as a hero of the story, you'll have to explore the región and loot with bravery and courage, facing many dangers and unimaginable ventures to, this way, gain experience, fame and increase your social status and level.

Relive with firsthand vigor and courage all the excitement of the adventures of an adventurer, palace intrigue and danger, and become the protagonist a bizarre, drunken and quarrelsome adventurer, in an exciting fantasy medieval district where they meet and are home to a multitude of stories, legends and fantastic stories.

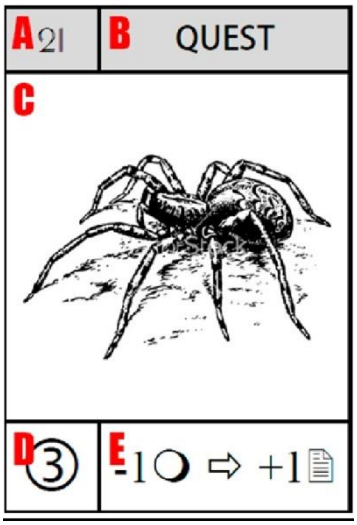
Defy destiny by resorting to the sword, searching great treasures, looting thief and malign creature lairs, get the most powerful magic objects, descend to the hell, visit graves in millenary subterranean cities, discover the far dwarf city of King Aquilon, plot at the palace, have fun with your mates in the Stag Tavern between quarrels, hookahs and spiced wine bottles, perform missions for the King or the thieves guild, save princesses and kill dragons. Only fame or death awaits.

COMPONENTS

- 18 initial cards (9 each player).
- 9 tether cards.
- 9 town cards. 🕮
- 36 quest:
    - 9 level 1 cards (white).
    - 9 level 2 cards (green).
    - 9 level 3 cards (yellow).
    - 9 level 4 cards (red).
- 10 adventurer markers (5 each player)
- PX markers with values 1, 5 and 25.
- Two +1 and one +2, markers for the event line

CARD STRUCTURE:

A: In the top left corner there's, normally, a number, that determines initiative. Initial player cards have values from 10 to 18, while tethers have 1 to 9. Level 1 cards to from 19 to 27, level 2 from 28 to 36, level 3 from 37 to 45 and level 4 from 46 to 53.



B: Name (and type) of card.

C: Ambient image of the card.

D: in the bottom left there's the card price in coins (MO) the player has to spend to acquire (successfully complete) that card.

E: Tis bottom box contains the effect of the card.

Deeds are special.

CARD TYPES

- **Tether:** Represent the obstacles you'll find during your adventures. Magic, curses, poison and illness.
- **Player cards:** 6mo/3px. The cards you start your adventure with.
- **Town cards:** 🕮 Nine of them form the Town. Represent the places and characters in which you'll be able to support yourself.
- **Quests:** Represent great adventures, places and characters that you'll find during your adventures. They go from level 1 to 4, the higher the more powerful they are.

INITIAL CONFIGURATION

1. Each player receives ha set of 9 initial cards (6mo/3px), shuffles them and places them upside down, creating a drawing Deck.
2. Each player receives 3 follower tokens. He/she will also have 2 additional in reserve to dispose of them.
3. Organize the event deck: shuffle every level separately together with tethers. For level 1, include tethers 1 and 2; for level 2, tethers 3 and 4; for level 3, tethers 5, 6 and 7; for level 4, tethers 8 and 9. Then create a unique upside down deck with level 4 at the bottom, then 3, 2 and 1, so quests appear in order.
4. Place the 9 town cards 🕮 at the center of the table. They form the Town and ara available

every Journey.

5. Place the 6 first cards (level 1) of the event deck forming an event line.
6. Determine the initial player randomly.

DEVELOPMENT:

A game is played in multiple Journeys (rounds) Every Journey consists of three phases:

- **Planification:** draw 5 cards from the Deck and determine who plays first.
- **Adventure:** place followers in town cards and play your hand cards. Apply effects.
- **Upkeep:** prepare next Journey.

PLANIFICATION

Hand: each player **draws 5 cards**. The discard pile is shuffled in the very moment the drawing deck becomes empty, even before the player needs to draw cards again.

Initial player: each player chooses a card from hand and shows it simultaneously. The one with higher initiative starts playing this Journey. In case of draw, order is reversed in respect to last Journey. Card is places in the table, turned 90º and effects are not applied.

ADVENTURE

Adventuring: Starting with the initial player, and alternatively, each player places, in a town, **one** of its followers in the Town and/or plays cards.

Followers: only one can be placed in each town card, and only one per turn.

Using cards: a player can, before and/or after placing a follower, use one or more of the cards in its hand or reserved.

- **From hand:** place it faceup in the table, turned 90º, and apply its effects.
- **From town:** if the player wants to reserve a town card, he places a follower over it. It's not obliged to use the card in the same turn. When used, turn the follower, apply the effects, and receive and accumulated PX in the card.

Turn end: if the player does not want top lay more cards and has finished reserving/using town cards, he passes the turn to the opponent.

No adventurers: if a player has no followers left at the starts of its turn, he must pass and can't use any card.

Journey end: when no player has followers left, the Journey is finished.

Acquire cards: a success in the adventure. To buy a card, the player must spend money obtained during the same turn through playing cards. Coins not used during a turn is lost.

- ***Price:*** three frist cards in the event line have their written cost; next two have their cost increased by 1; the last by 2.
- ***Tethers:*** if, at the left of the acquier card, there's one or more tethers, the player gets them too. All acquired cards to the discard pile.

Experience: the player gains PX through using cards that grant experience points. The current number is visible at every moment.

UPKEEP

In this order:

1. **PX grant:** every empty town card gets 1 extra PX point. This way, they become more attractive to players. If a card stocks three points, it's destroyed and substitued by the first card in the event line –or the first in the event deck if that's empty.
2. **Discarding:** all cards played and those still in hand, go to the discard pile.
3. **Followers:** players recover their followers from town. They get only 3 and, if there are remaining, they go to the general reserve.
4. **Event line:** remove, if there is, the first quest in the event line (another group solved it). Move left the remaining cards. Fill the empty spaces, up to 6 cards, with cards from the event deck.
5. **Reconstruction:** if a town card was destroyed by the black mage, turn it faceup so it's available again.
6. **New Journey:** a new Journey starts with Planification.

GAME OVER:

Game ends if any of these conditions are met:

- A player gets 100 or ore PX
- A player has won 2 dragons.
- The event line can't be completed.

WINNER:

The player with more PX wins. If draw: the one that got more PX in the last Journey. If still drawn: who won 2 dragons. If still: who has the card with highest initiative.

CARD EFFECTS

And/or: *whenever in an effect appers an & it means "and" (both effects take place); an / means "or" (you must choose one of the effects).*

Money: basic monetary unit. Everything's acquired with it. ❍

Experience: basic scoring method. ❑

Quest: adventuring means taking risks and defying monsters. Everything you resolve with exit becomes part of your curriculum.

Deeds: some cards are specially powerful and hard to resolve, but have higher rewards. A deed can't be applied over another deed. =

Hire followers: on ocasions, players will hire followers (mercs, militia…) to explore and fight.

Aids: things like contacts, information, rumours and prays, may give you additional information. To solve your quests.

Tethers: sometimes you have to carry with bad consequences of your actions.

Remove problems: to be succesful you'll have to remove the problems that arise and things that are no longer useful.

Influence: influencing the town might have its benefits.

Planification: to be succesful you need to plan your future actions.

EFFECT EXAMPLES

❑: gain 1 px.

+3❍: gain 3 mo.

+2: draw 2 cards.

: acquire 1 quest.

: gain 1 extra follower.

-3❍ ⇨ +3: spend up to 3 MO and draw up to 3 cards (1:1).

-2 ⇨ +2❍: garbage up to 2 cards from hand ajd gain up to 2 MO (1:1).

-1: null (turn bottom up), for this Journey, a town card that has no follower on it.

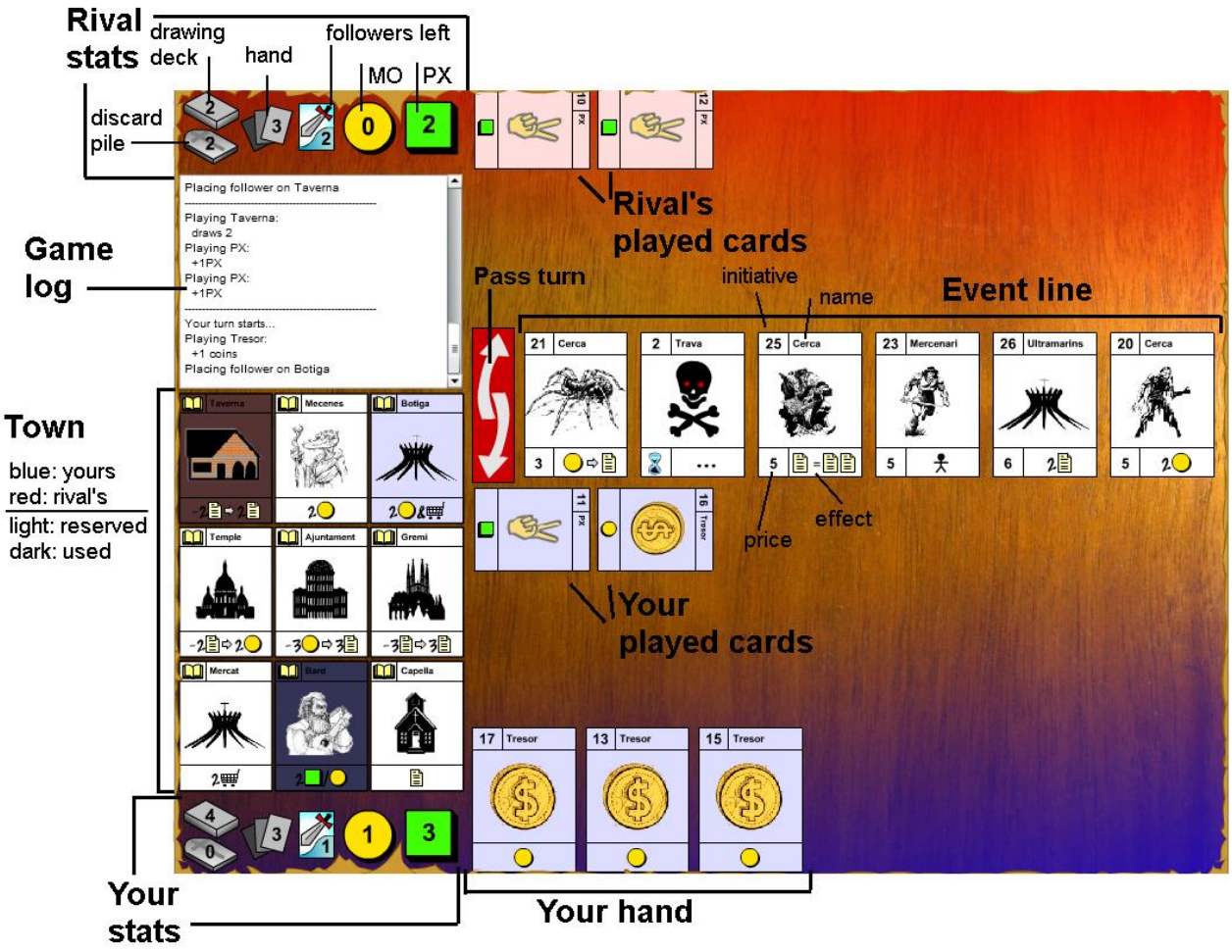
= : reuse a non-deed card.

❍ = 2x❍: double all MO obtained this turn .

❑ = 2x❑: double all PX obtained this turn.

: put in your hose a card from your hand; in the next Journey, add it to the 5 cards drawn during Planification

17 Annex: Reference PC manual



In the bottom you have your cards and stats; in the top, your opponent's. Dividing both player's grounds there's the event line. The red button with arrows at the left of the event line is used whenever the player wants to ends his turn.

At the left, there's the Town: a light blue card means you have reserved it and dark blue that you've used it –after a previous reserve-; in the same fashion, light red means the opponent reserved it and dark red that the opponent used it.

Lastly, the game log indicates the actions the players take, turn changes, journey ends, etc, as well as the number of possible movements the AI calculated –if playing versus it.

18 Bibliography

[1] J. C. d. D. Guerrero, "La BSK," [Online]. Available: http://www.labsk.net/wkr/archives/13834/. [Accessed January 2012].

[2] "Board Game Geek," [Online]. Available: http://www.boardgamegeek.com/boardgame/134282/dungeon-realms. [Accessed Juanuary 2013].

[3] V. G. development, "Wikipedia," [Online]. Available: http://en.wikipedia.org/wiki/Video_game_development.

[4] "About.com," [Online]. Available: http://classicgames.about.com/od/classicvideogames101/p/CathodeDevice.htm. [Accessed February 2013].

[5] "Wikipedia," [Online]. Available: http://en.wikipedia.org/wiki/Cathode_ray_tube_amusement_device. [Accessed February 2013].

[6] M. E. Moore and J. (. Novak, Game Industry Career Guide., Delmar: Cengage Learning..

[7] E. M. R. &. J. K. Larsen, "Silicon Valley fever: growth of high-technology culture. Basic Books.," 1984, p. 263.

[8] "Wikipedia," [Online]. Available: http://en.wikipedia.org/wiki/List_of_best-selling_game_consoles. [Accessed 2013].

[9] BASIC Computer Games, 1978.

[10] "Minecraft," [Online]. Available: https://minecraft.net/stats. [Accessed February 2013].

[11] Blizzard, "Blizz Planet," [Online]. Available: http://www.blizzplanet.com/. [Accessed 7 February 2013].

[12] S. Nunneley, "VG247," 30 January 2013. [Online]. Available: http://www.vg247.com/2013/01/22/minecraft-sales-hit-20-million-mark-for-all-platforms/.

[13] "Statistic Brain," [Online]. Available: http://www.statisticbrain.com/skyrim-the-elder-scrolls-v-statistics/.

[14] "Statistic Brain," [Online]. Available: http://www.statisticbrain.com/star-wars-the-old-republic-statistics/.

[15] K. Beck and e. al., "Manifesto for Agile Software Development," in *Agile Alliance*, 2001.

[16] K. Beck, Extreme Programming Explained: Embrace Change, Boston: MA: Addison-Wesley., 1999.

[17] "C2," 2013. [Online]. Available: http://c2.com/xp/DoTheSimplestThingThatCouldPossiblyWork.html.

[18] W3Schools. [Online]. Available: http://www.w3schools.com/xml/.

[19] Adobe, "AS3 Reference (ByteArray)," 07 03 2013. [Online]. Available: http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/utils/ByteArray.html#readUTFBytes().

[20] Adobe, "AS3 Reference (XML)," [Online]. Available: http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/XML.html#XML().

[21] "Wikipedia," [Online]. Available: http://en.wikipedia.org/wiki/Minimalism_(computing).

[22] GreenSock, "GreenSock," [Online]. Available: http://www.greensock.com/v12/.

[23] S. University, "Stanford.edu," [Online]. Available: http://www-cs-faculty.stanford.edu/~uno/taocp.html.

[24] Adobe, "Adobe AS3 dcumentation," [Online]. Available: http://livedocs.adobe.com/flex/3/html/help.html?content=createevents_3.html.

[25] "Wikipedia," [Online]. Available: http://en.wikipedia.org/wiki/Nagle's_algorithm. [Accessed 7 4 2013].

[26] B. Hook, "Book of Hook," [Online]. Available:

http://trac.bookofhook.com/bookofhook/trac.cgi/wiki/Quake3Networking. [Accessed 12 4 2013].

[27] D. Schall, "Git Hub," [Online]. Available: https://github.com/mikechambers/as3corelib.

[28] Adobe, "AS3 Reference," [Online]. Available: http://help.adobe.com/en_US/FlashPlatform/reference/actionscript/3/flash/utils/ByteArray.html.

[29] S. Rusell. [Online]. Available: http://www.eecs.berkeley.edu/~russell/.

[30] P. Norvig. [Online]. Available: http://norvig.com/.

[31] S. Russell and P. Norvig, Artificial Intelligence: A Modern Approach, Pearson, 2003.

[32] Autodesk, "Autodesk," [Online]. Available: http://gameware.autodesk.com/navigation.