Evolution Engine
Software Architecture Document (SAD)

Gil Duarte da Costa

{ gsaurus@gmail.com }

DOCUMENT VERSION:

- 0.1

RELEASE/REVISION DATE:

- 14 / 08 / 2010

Abstract

Evolution Engine is a game engine mainly oriented for 2D games, with special emphasis on the beat'em up and side scrolling genres. EvE is on it's early stages and this document intends to describe it's general organization and how it should work when implemented. EvE is an engine opened to expansion, structured in modules that can be override for different game genres and technologies. It also has a strong emphasis on the game designers, providing them with useful tools to build up their games, reducing the amount of scripting needed. Engine's core is cross platform and so will be the first modules implemented, leaving however the opportunity for the development of modules for specific platforms. Network is planned for the first implementation, however it is not part of the core structure, it is an independent module.

Table of Contents

List of Figures

List of Tables

1 Documentation Roadmap

1.1 Purpose and Scope of the SAD

This SAD specifies the architecture of the EvE system module, which is the core module specifying the communication interfaces between the connectable modules. The document explains how the modules are connected and how they interact to form a game. This way programmers can more easily know what exactly is provided to and expected from each module. The document also advances information of the first modules implementation planned, and how game designers can produce games for it.

The information of this document follows a template developed at the Software Engineering Institute [SEI 10], and is also based on [CA 10], [CMU 10], [Mitra 08] and [Goulão 10].

The software architecture for a system is the structure or structures of that system, which comprise software elements, the externally-visible properties of those elements, and the relationships among them [Bass 03]. Each structure is documented as an architectural view. The views of this SAD are oriented for analysis purposes and focused on the stakeholders needs.

This SAD should be updated and/or extended as the project evolves.

1.2 How the SAD Is Organized

This SAD is organized into the following sections:

- **Section 1 - Documentation Roadmap** provides information about this document and its intended audience. It provides the roadmap and document overview.
- **Section 2 - Architecture Background** provides information about the EvE architecture and explains the decisions that led to it. It describes the background and rationale for the proposed solution, as well as the constraints and requirements involved.
- **Section 3 - Views** constitutes the most relevant part of the documentation. Each view describes part or the whole EvE's system structure from a specific viewpoint (see Section 0).
- **Section 4 - Relations Among Views** specify how elements in one view maps to elements in another view.
- **Section 5 - Referenced Materials** provides look-up information for documents that are cited or pointed in this SAD.

- **Section 6 - Directory** has definitions of technical terms and acronyms used in this document that are specific to the scope of this project.

1.3 Stakeholder Representation

There is three kinds of relevant stakeholders on this project:

- **Project Developers** are the ones concerned with the project's development, it's costs and learning process.
  - Project Leader is the author of this SAD and is concerned that the final product is of good quality and easy to extend. More specifically he's interested on enabling the creation of a large line of products that are easily created with the engine. Those products may be created by the main team associated to the engine and by independent users. Those are expected to be non experts and having small knowledge about the engine, thus the final product must be of easy learning and simple to use.
  - Software Architect is also the project leader and author of this SAD, he's concerned that the architecture is modular so that each programmer can focus on very specific and independent tasks. Not only simplifies the programming effort, but also enables concurrent and independent development, increasing the development rate.
  - Programmers are the ones producing the software executable artefacts. The project leader is also programmer. Programmer is mainly concerned with reduced development time and effort. They're concerned on getting results quickly, but are aware that the code changeability, reuse and modularity are important because it shortens the time needed to modify or create new functionalities, witch are important qualities on a project intended to live long. They're also concerned on producing efficient code that can run with low spatial and temporal costs, because of realtime games requirements.
- **Game Developers** are the ones interested on producing games using EvE. This class of stakeholders includes:
  - Game Designers are the ones that plans the game structure, defines how the characters and entities interact, and how players interact with the game. They are interested on easily define characters behaviour and expect that the gameplay flows well and as they expect.
  - Graphic Artists produces the graphical content of games. They want to easily integrate their work in the game. They don't want to waste time learning that process and don't want to see their work being constrained by system limitations.
  - Audio Artists produces the auditive content of games. They want to easily integrate their work in the game. They don't want to waste time learning that process and don't want to see their work being constrained by system limitations.
  - Storyline writers defines the sequence of the main events on the game. They expect that what they imagine is reproducible with the engine, so the engine must be dynamic enough to enable very different event definitions.
- **Players** are the end users of the games developed using EvE. They are concerned with the flow of the games, response time, and that games are available on their favourite operating system and platform.

Stakeholders concerns are tabled next. Interest rate are represented as follows:

0 - don’t care
1 - not very concerned
2 - concerned
3 - very concerned

Project developers:

Concerns	Leader	Architect	Programmers
Performance	3	2	3
Modularisation	2	3	2
Extensibility	3	3	2
Reuse	3	3	2
System readability	1	3	2
Learnability	3	2	1
Easy to use	3	2	2
Security	3	2	2
Project costs	3	1	3

Table 1 - Development Stakeholders concerns

Game developers:

Concerns	Game Designers	Graphic Artists	Audio Artists	Storyline Writers
Performance	3	0	0	1
Extensibility	2	0	0	1
Reuse	2	2	2	1
Learnability	3	1	1	2
Easy to use	3	3	3	1
Security	1	3	3	0

Table 2 - Game developers and players concerns

Players are only concerned with the performance of the engine and if games have a natural gameplay. They like variety.

1.4 Viewpoint Definitions

The SAD employs a stakeholder-focused, multiple view approach to architecture documentation, as required by ANSI/IEEE 1471-2000, the recommended best practice for documenting the architecture of software-intensive systems [IEEE 1471].

The following table summarizes the stakeholders in this project and the viewpoints that have been included to address their concerns. Viewpoints are defined on this section subsections.

Stakeholder	**Viewpoint(s) that apply to that class of stakeholder's concerns**
Project Leader	Modules Decomposition, Components, Interfaces, Runtime
Software Architect	Modules Decomposition, Components, Packages, Classes, Runtime
Programmers	Modules Decomposition, Components, Game Organization, Interfaces, Packages, Classes, Runtime
Game Designers	Modules Decomposition, Components, Game Organization, Interfaces, Scripting API, Runtime
Graphic Artists	Interfaces
Audio Artists	Interfaces
Storyline Writers	Interfaces
Players	Game Organization

Table 3 - Relation of Stakeholders and Viewpoints

1.4.1 Modules Decomposition Viewpoint Definition

1.4.1.1 Abstract

Views conforming to the modules decomposition viewpoint partition the system into a unique non-overlapping set of hierarchically decomposable implementation units (*modules*), and shows the relations between them.

1.4.1.2 Stakeholders and Their Concerns Addressed

Stakeholders and their concerns addressed by this viewpoint include:

- Project leader, concerned on how the engine high level architecture meets the requirements;
- Software architect, who defines the modules presented on this view.
- Programmers that needs an overview of the engine's organization.
- Game designers that are interested on knowing how the system works without knowing the details of each module.

1.4.1.3 Elements, Relations, Properties, and Constraints.

Elements of the module decomposition viewpoint are modules, which are units of implementation that provide defined functionality. Modules are hierarchically decomposable; hence, the relation is "is-part-of." Properties of elements include their names, the functionality assigned to them (including a statement of the quality attributes associated with that functionality), and their software-to-software interfaces. The module properties may include requirements allocation, supporting requirements traceability.

1.4.1.4 Language(s) to Model/Represent Conforming Views.

Views conforming to the module decomposition viewpoint may be represented by (a) plain text using indentation or outline form [Clements 02]; (b) UML, using subsystems or classes to represent elements and "is part of" or nesting to represent the decomposition relation; (c) other notations like Acme models or *ad hoc* models which meets the elements and properties of the viewpoint.

1.4.1.5 1.5.1.6 Viewpoint Source.

[Clements 02, Section 2.1] describes the module decomposition style, which corresponds in large measure to this viewpoint.

1.4.2 Components Viewpoint Definition

1.4.2.1 Abstract

Views conforming to the components viewpoint represents independent programming modules and how they are connected by offered and required interfaces.

1.4.2.2 Stakeholders and Their Concerns Addressed

Stakeholders and their concerns addressed by this viewpoint include:

- Project leader, concerned on how the engine components are connected;
- Software architect, who defines the components presented on this view.
- Programmers that will implement those components needs to know how they are related with each other.
- Game designers that are interested on knowing how the system components are related without knowing the details of each one.

1.4.2.3 Elements, Relations, Properties, and Constraints.

Elements of the components viewpoint are individual coding components, usualy in DLL format. Components relations, properties and constraints are as defined at [OMG-UML].

1.4.2.4 Language(s) to Model/Represent Conforming Views.

Views conforming to the components viewpoint should be represented by UML component diagrams.

1.4.3 Game Organization Viewpoint Definition

1.4.3.1 Abstract

Views conforming to the game organization viewpoint represents how the resulting games produced with EvE are organized.

1.4.3.2 Stakeholders and Their Concerns Addressed

Stakeholders and their concerns addressed by this viewpoint include:

- Programmers, to have an overview of how components link together to form an executable game.

- Game designers, that should know how games runs over EvE;
- Players that may be interested on switching existing modules by themselves, they must know how they are connected constituting the executable game.

1.4.3.3 Elements, Relations, Properties, and Constraints.

Elements of the game organization viewpoint are modules (usualy DLLs) and other elements that are important on the executable game.

1.4.3.4 Language(s) to Model/Represent Conforming Views.

Views conforming to the game organization viewpoint may be represented by (a) plain text using indentation or outline form [Clements 02]; (b) *ad hoc* notation which meets the elements and properties of the viewpoint.

1.4.4 Editor Interfaces Viewpoint Definition

1.4.4.1 Abstract

Views conforming to the interfaces viewpoint represents the software presentation and way of interact with the final users.

1.4.4.2 Stakeholders and Their Concerns Addressed

Stakeholders and their concerns addressed by this viewpoint include:

- Project leader, who evaluates how the final product will looks like;
- Programmers that will code the interfaces
- Game designers, graphic artists, audio artists and story writers because they are going to use the interfaces to produce their games.

1.4.4.3 Elements, Relations, Properties, and Constraints.

Elements of the interfaces viewpoint are screens. Relations are the links from each screen to another.

1.4.4.4 Language(s) to Model/Represent Conforming Views.

Views conforming to the interfaces viewpoint should be represented by screen drawings.

1.4.5 Scripting API Viewpoint Definition

1.4.5.1 Abstract

Views conforming to the scripting API viewpoint represents the interface of methods available for scripts. That interface provides access to EvE internal functionalities in order to allow external scripts to decide the behaviour and sequence of events of the application.

1.4.5.2 Stakeholders and Their Concerns Addressed

Stakeholders and their concerns addressed by this viewpoint include:

- Game designers that will have to define the behaviour of the entities and sequence of events of their games;

1.4.5.3 Elements, Relations, Properties, and Constraints.

Elements of the scripting API viewpoint are classes and functions. Their properties and constraints are as defined at [OMG-UML].

1.4.5.4 Language(s) to Model/Represent Conforming Views.

Views conforming to the scripting API decomposition viewpoint may be represented by (a) plain text using indentation or outline form [Clements 02]; (b) UML, using subsystems or classes to represent elements, functions and relations; (c) *ad hoc* notation that meets the elements and properties of the viewpoint.

1.4.6 Packages Viewpoint Definition

1.4.6.1 Abstract

Views conforming to the package viewpoint represents how the code is divided into separated and hierarchical packages.

1.4.6.2 Stakeholders and Their Concerns Addressed

Stakeholders and their concerns addressed by this viewpoint include:

- Software architect, because he defines the packages structure;

- Programmers because they code the content of those packages.

1.4.6.3 Elements, Relations, Properties, and Constraints.

Elements of the packages viewpoint are packages. Their relations, properties and constraints are as defined at [OMG-UML].

1.4.6.4 Language(s) to Model/Represent Conforming Views.

Views conforming to the packages viewpoint should be represented by UML package diagrams.

1.4.7 System Classes Viewpoint Definition

1.4.7.1 Abstract

Views conforming to the classes viewpoint represents the programming classes of the System module, their methods, fields and relations.

1.4.7.2 Stakeholders and Their Concerns Addressed

Stakeholders and their concerns addressed by this viewpoint include:

- Software architect, who defines the general organization of the classes, and keep in contact with programmers to update the classes documentation as needed;
- Programmers, they implement the classes.

1.4.7.3 Elements, Relations, Properties, and Constraints.

Elements of the classes viewpoint are programming class units. Classes relations, properties and constraints are as defined at [OMG-UML].

1.4.7.4 Language(s) to Model/Represent Conforming Views.

Views conforming to the classes viewpoint should be represented by UML class diagrams.

1.4.8 Runtime Viewpoint Definition

1.4.8.1 Abstract

Views conforming to the runtime viewpoint represents communication between objects and entities during an execution scenario.

1.4.8.2 Stakeholders and Their Concerns Addressed

Stakeholders and their concerns addressed by this viewpoint include:

- Project leader, who monitors the project development and must know the expected behaviour of the software.
- Software architect, who defines how the structural elements behave with each other;
- Programmers that will implement the classes that will make possible these execution scenarios. Runtime viewpoint serves as reference to them to know how system elements communicate and work together.
- Game designers to know how elements interact, in order to produce more consistent scripts.

1.4.8.3 Elements, Relations, Properties, and Constraints.

Elements of the runtime viewpoint are objects, entities and operations. Their properties and constraints are as defined at [OMG-UML].

1.4.8.4 Language(s) to Model/Represent Conforming Views.

Views conforming to the runtime viewpoint should be represented by UML interaction diagrams.

1.5 How a View is Documented

A view is a representation of a whole system from the perspective of a related set of concerns [IEEE 1471]. Concretely, a view shows a particular type of software architectural elements that occur in a system, their properties, and the relations among them. Section 3 of this SAD contains one view (possibly containing subviews) for each viewpoint listed in Section 1.4. Views are documented as follows, where the letter *i* stands for the number of the view (1, 2, etc.) and j stands for the number of the sub-view (1, 2, etc.), when applycable.

- Section 3.i: View name
- Section 3.i.j: Sub-view name

- Section 3.i.j.1: View Description.
- Section 3.i.j.2: Primary presentation. This section presents the elements and the relations among them that populate this view, using an appropriate language, languages, notation, or tool-based representation.
- Section 3.i.j.3: Element catalog. Whereas the primary presentation shows the important elements and relations of the view, this section provides additional information needed to complete the architectural picture.
- Section 3.i.j.4: Variability mechanisms. This section describes any variabilities that are available in the portion of the system shown in the view, along with how and when those mechanisms may be exercised.
- Section 3.i.j.5: Architecture background. This section provides rationale for any significant design decisions whose scope is limited to this sub-view.
- Section 3.i.j.6: Related views. Refers to views of the system related to this specific view. May provide a context diagram showing how the part of the system represented by this view contextualises with parts represented by other views..

1.6 Relationship to Other SADs

This SAD refers to a project in development. At the moment, any other document was produced, however it's possible that SADs for specific software components are made in a later stage (like for EvE editor or even for each module implementation of the MVC).

1.7 Process for Updating this SAD

Discrepancies, errors, inconsistencies, or omissions on this SAD should be reported to gsaurus@gmail.com. Reports will be evaluated within a week, and an answer will be sent back to the submitter about the result of the report evaluation.

Extensions proposals should also be sent to gsaurus@gmail.com for evaluation. The author of this SAD is responsible for keeping the document up to date and synchronized with the development whenever possible.

2 Architecture Background

2.1 Problem Background

EvE was part of the Streets of Rage Evolution project (SoRE). The development started to be based on SoR style, but we figured out that this restriction brings implications to possible unexpected new features. So, it was decided to generalize EvE for any kind of beat'em up games (as much as possible). That implies to remake all the code, so the first working version was discontinued and turned into a prototype, where several features were tested.

Any documentation was produced for the abandoned prototype version.

A generic and stable version of EvE is on the works since January 2010, and it is not only targeted for the beat'em up style, but for many other styles, depending on the imagination and will of the game designers and the EvE team itself, who could make specific EvE modules for certain styles.

2.1.1 System Overview

To build games, EvE gets as input a variety of sources from the game designers (images, animations, sounds and musics), plus some configuration files and scripts for behaviour. This input is made though an editor program. It runs over the engine to show how the game should look like and to save/load the data into/from the game database.

Finally, end users plays an executable game that runs over the engine using the database built with the editor.

The EvE context diagram is as follows:

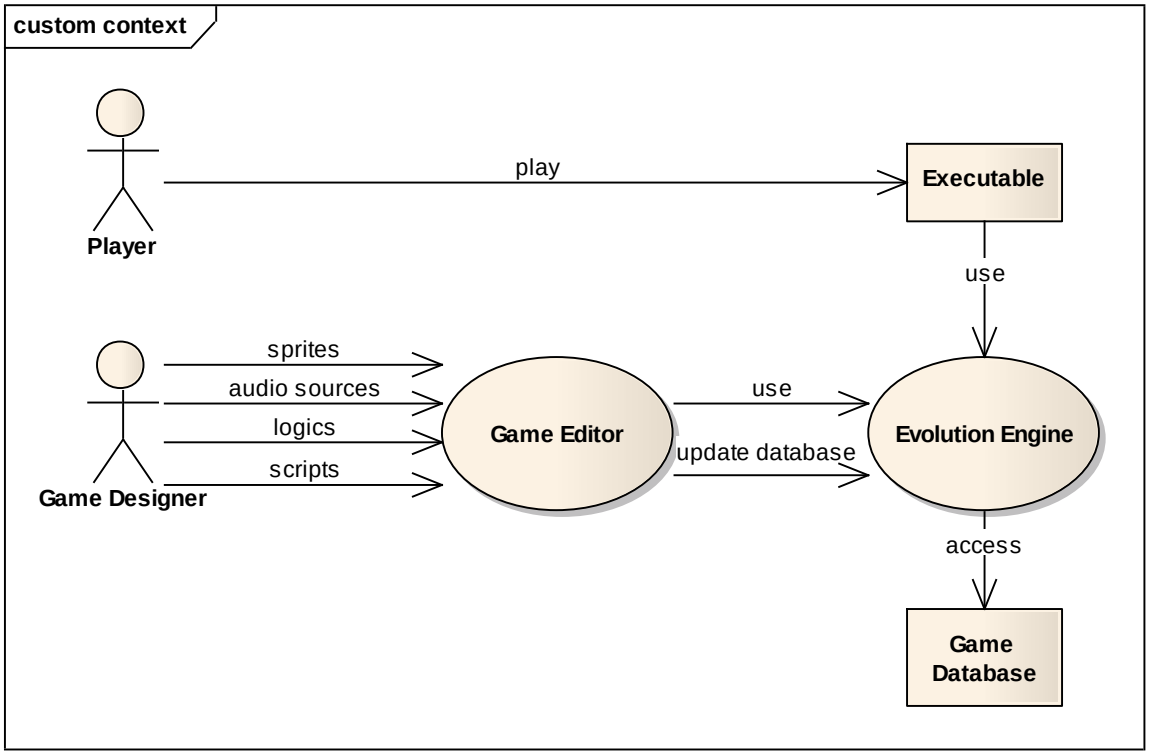

Figure 1 - EvE Context Diagram

Some key concepts are identified here:

- Processing and transformation of resources input into game data.
- Database management;
- Engine logics;
- Processing of player input;
- Production of output (audiovisual) to player.

Since here sprites are also called graphical models (GMs) because they can be anything (sets of 2D images, 3D models, etc).

2.1.2 Goals and Context

The main goal is to build a game engine that can be used by many to easily build sligtly different games. Several game genres can be supported, however some better than others. The genres most likely to be supported are side scrollers, fighting and beat'em up games, platform games, 2D shooters (like space shooters) and RPGs. Strategy games and MMORPGs are most likely to be poorly supported, they are very unique styles and can only be supported by programming specific modules for them, which isn't planned yet.

The engine and editor must be build as extensible units, so that the system can healthy evolve in the future. The results of it's evolution should be easy to configure with older modules and versions. It should be a very dynamic system.

2.1.3 Significant Driving Requirements

Requirements are of major importance when setting the architecture, identifying requirements is finding out how the system must be structured and how it must behave.

2.1.3.1 Design Constraints

When building the system the following design constraints (DCs) must be taken in consideration:

- **DC1:** Dynamic nature and evolutionary software.
- **DC2:** Realtime application.
- **DC3:** Netplay support.

2.1.3.2 Functional Requirements

The functional requirements (FRs) are separated in two: one targeting the game designers, corresponding to the editor FRs (EFR); other targeting the end users of the produced games, corresponding to the engine's core FRs (CFR).

2.1.3.2.1 Editor Functional Requirements

The following use cases diagram illustrate the required features:

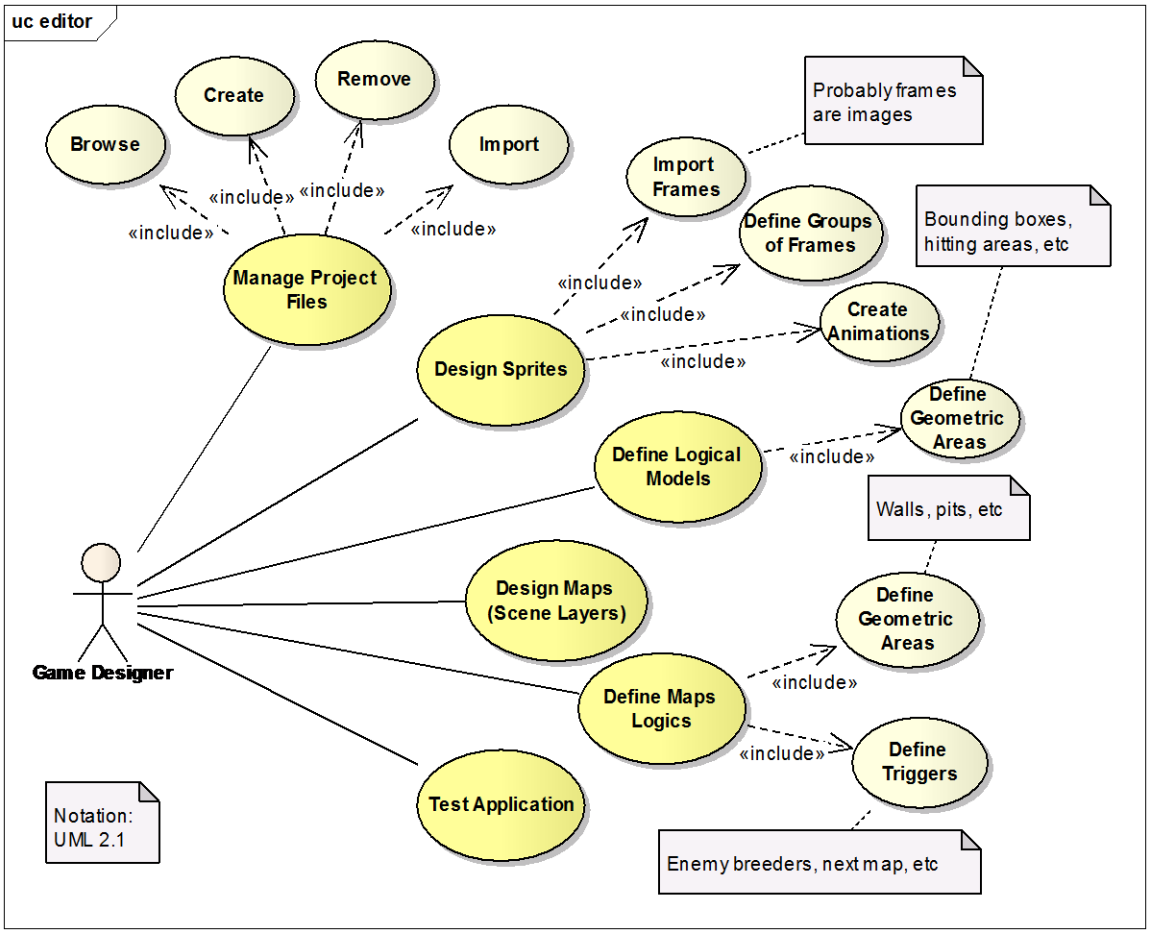

Figure 2 - Editor use cases.

Project files are all those concerning to sprite models, logical models, maps, maps logics and scripts. EFRs are listed bellow:

- **EFR1:** Browse, open, create and remove project files.
- **EFR2:** Import external files into the project.
- **EFR3:** Import and remove frames of a GM.
- **EFR4:** Group frames of GMs.
- **EFR5:** Create animations by defining and timing sequences of frames and grouped frames of a GM.
- **EFR6:** Define logical models (LMs), including their geometric properties.
- **EFR7:** Design maps (level backgrounds, usualy composed of layers).
- **EFR8:** Define maps logics, including geometric properties and action triggers.
- **EFR9:** Test the building data with the engine.
- **EFR10:** Support for special displaying effects (audiovisual).

2.1.3.2.2 Engine's Core Functional Requirements

CFRs are listed as follows:

- **CFR1:** Process input from one or various input devices, in order to control the game.
- **CFR2:** Process correctly the database input and display errors on invalid data or operations.
- **CFR3:** Display appropriately the database input, in concordance with the game logics.
- **CFR4:** Is able to update, save and load deterministic game states (specially because of DC3).

2.1.3.3 Quality Attributes Requirements

The quality attributes, also called non functional requirements (NFRs), are listed below:

- **NFR1: Costs and Schedule.** The project team has limited resources to work on the project because it is not being funded. Schedule is extended due to that, however shouldn't be too much extended or it is reported as a death project. Team should work as they can to accomplish the goals in one or two years.
- **NFR2: Extensibility**. System should allow the docking and or replacement of parts of it by completely new parts that works correctly with all other parts. This way games of new styles can be made and even existing games made with previous versions of EvE can be updated with few or any modifications from the game designers side.
- **NFR3: Modularity**. This is closely related with extensibility. Modularity allows concurrent development, easy replacement of modules and easy extension of existent modules, without affecting other parts of the system.
- **NFR4: Performance.** It is very important on realitme games (DC2). Games built with EvE must flow smoothly.
- **NFR5: Correctness.** The system should be deterministic and to always produce correct results free of bugs from it's side. Script bugs aren't direct responsibility of EvE team, except for scripts developed by them.
- **NFR6: Learnability.** Game designers shouldn't have much effort to learn how to transform their artistic resources into a playable game.
- **NFR7: Reusability**. Also close related with modularity, modules should work well with any other existent or future modules of the engine.
- **NFR8: Security.** The sources produced by the game designers shouldn't be accessed nor modified by others. Their copyrights must be granted. The system should be resistant to hacking attacks.
- **NFR9: Evolution.** Artefacts produced with previous versions of EvE should evolve with EvE itself with few or any modifications.

2.2 Solution Background

2.2.1 Architectural Approaches

By structuring the engine in a MVC pattern the presentation of the elements of a game is totally independent of their logical shape and behaviour. For example, two games with the same model can have different controllers (ex two different beat'em up games) and different views (ex. 2D or 3D); two games with the same view can have different models and controllers, etc.

Besides the variety of combinations of different modules, the MVC also helps on:

- The development since each module can be programmed independently as a connectable component;
- On the maintenance because if changes are required they probably focus on one module only. Each module can be improved and tested independently;
- On extending the engine with new implementations of one or more modules, instead of having to modify embedded parts of the project.

In the other hand, the effort required to plan, structure and implement is higger than a straight forward approach. But in the end it worth because it grants EvE a longer lifetime (extensibility and reusability quality attributes).

System is the module that connects the MVC together. It defines the interfaces of communication between the MVC and a set of utilities useful on all modules. This strategy makes the MVC modules completely independent between each other, forcing however that they are dependent of the System module. The option of letting the executable file to decide which implementations (dlls) are imported strengthens the modules independence. The executable file is an independent entity, the engine is unaware of who imports their modules.

Events are mainly exchanged with a publish/subscribe mechanism. This allows the events producers to be unaware of who is receiving the messages, enabling the diversification of independent subscribers.

The Model keeps two states of the game, the present and the future state. This duplication keeps data correct. If only one state existed, when one entity queries some properties of the state, those would have been already modified by a previously updated entity. Thus, all entities queries the same present state, and orders modification for the future state. Properties of the future state can however be overridden by other entities, but entities don't really know what will happen, they just order what they want or expect to happen. For example if a character hit another, it will order that the other is hit, and the first keeps unchanged. However when the other character is updated, if it detects that hits the first character, that one will be hit and the first keeps unchanged. This results on a mutual hit, where both characters gets hit. If only one state existed, only one of the characters would have been hit.

The architecture of EvE is still on an early stage and will certainly change soon. Everything must be taken in consideration. Some driving points are to enable:

- Diversification on game behaviours (scripting, specialized controllers, etc);
- Diversification on special effects (scripting, specialized view internal controllers);
- Game entities to store specific information, like entity name, energy, etc (extra data containers for each entity);
- To store global information, like game options, current level, player score, etc (extra data containers on the game state);
- The creation of complex HUDs, menus, loading and other screens (scripting, access to Model and Controller from the View module).

It is possible that many architectural aspects will only be cleared when EvE implementation starts.

2.2.2 Analysis Results

No significant executable artefacts were produced yet, EvE is on design process and early implementation just started. Only a few classes were coded and tested for data codification. Information is compressed/decompressed and encoded/decoded into/from files in one step or by streaming. Compression is made through the zlib library. Simple tests shown that this early code works for normal datafiles. Tests where made to stream a codified music, but the results aren't very promising. the decodification of the stream chunks and play on a sf::SoundStream doesn't work properly. More work has to be made with codecs. The development of EvE System module and the MVC modules should start in the near future.

2.2.3 Requirements Coverage

2.2.3.1 Design Constraints

- **DC1:** Dynamic nature and evolutionary software - Covered by the MVC design pattern and scripting support.
- **DC2:** Realtime application - Covered by deterministic state mechanism and frame skip techniques.
- **DC3:** Netplay support - Covered by deterministic state mechanism and a dedicated network module.

2.2.3.2 Functional Requirements

The EFRs are covered on the Editor Interfaces view.

The CFRs are covered on the System Classes and Runtime views.

2.2.3.3 Quality Attributes Requirements

The NFRs can be covered as shown on the following graph:

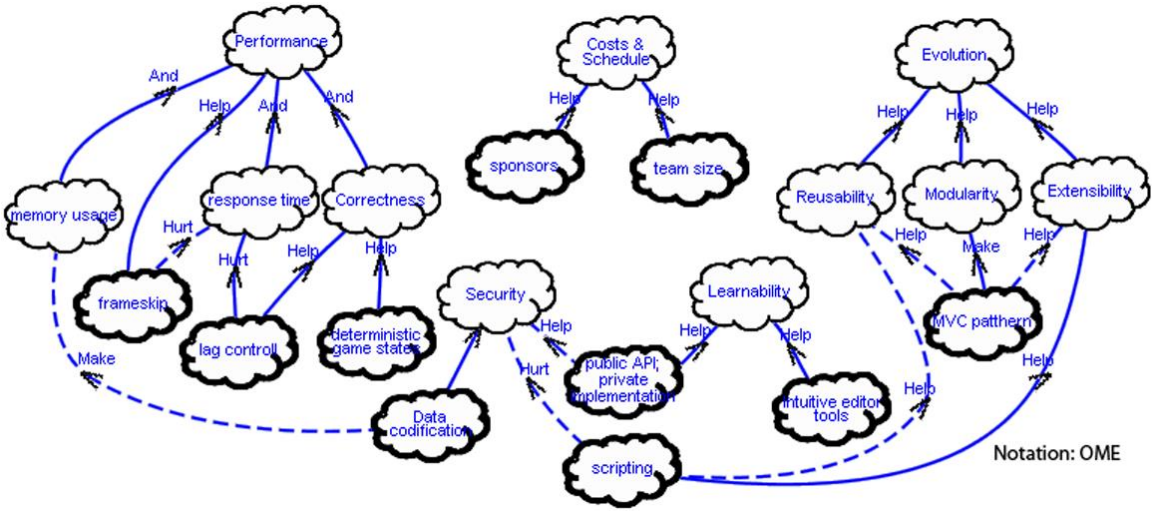

Figure 3 - OME NFR Framework graph of EvE NFR requirements

2.2.4 Summary of Background Changes Reflected in Current Version

N/A, this is the first version.

2.3 Product Line Reuse Considerations

The doors for a product line are evident on the modular nature of EvE. The editor and the scripting support are the biggest points for it. The first projects planned, that are also the main drivers of the engine, are the Streets of Rage Evolution and the Bare Knucle IV - two very different beat'em up games that can greatly benefit from EvE dynamics. But many small example demos are planned, with very different characteristics. At least one demo for each of the most known game styles (platforms, beat'em up, fighting, shooting, RPG...). Those demos will be opened to the community, perhaps to use as a base for more complete games. It is expected that the community can quickly produce many different games with EvE.

3 Views

3.1 Modules Decomposition View

3.1.1 View Description

EvE follows the MVC pattern, thus it has tree main modules: Model, View and Controller, which communicate between each other.

3.1.2 Primary presentation

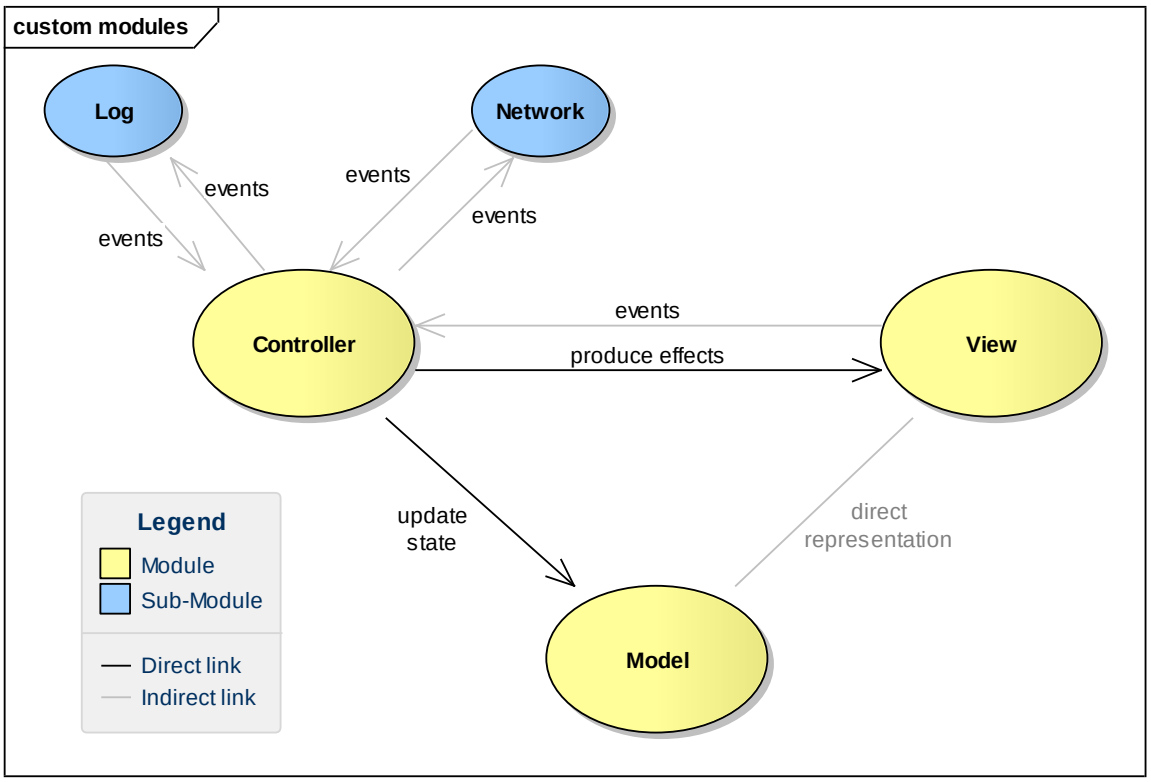

Figure 4 - EvE modules diagram

3.1.3 Element catalog

Model	Module responsible for data representation and game state management
View	Module responsible for the presentation of the game elements.
Controller	Module that decides what happens, when and how, defines the behaviour of the game.
Network	This module send by the network the events accepted by the Controller, and get the incoming events and communicate them to the Controller.
Log	Module that records all events accepted by the Controller, and can also be used in read mode, replicating all registered events.

3.1.4 Variability mechanisms

Many implementations of the modules Model, View and Controller can exist, any conjunction of them will result in different game styles.

Other sub-modules can be attached to specific implementations.

3.1.5 Architecture background

EvE doesn't use a pure MVC pattern. As is usual, the MVC is adapted to better fit the needs of the software. The communication between the modules are briefly explained bellow:

- The Model has indirect access to the View in order to automatically request the creation and destruction of representations for the objects being created and removed on the Model.
- The View has indirect access to the Model in order to keep track of the state of the Model objects being directly represented by the View.
- The View has indirect access to the Controller by a publish/subscribe pattern, in order to alert the Controller of the input events that are detected during the game.
- The View will probably have a more direct query access to the Controller and Model, in order to gather specific information to show on the screen (loading threads progress for example).
- The Controller has access to the View in order to allow the definition and display of special effects that occurs in response to Controller behaviours.
- The Controller has access to the Model to query and modify the game state.

Other modules like the network can be freely attached to the MVC modules as needed, but the communication between them are not well specified. However the Controller can easily communicate with other (perhaps unknown) modules via a publish/subscribe mechanism.

3.1.6 Related Views

- Components
- Game Organization
- Packages
- Runtime

3.2 Components View

3.2.1 View Description

This view presents how the components of EvE are connected with common interfaces.

3.2.2 Primary presentation

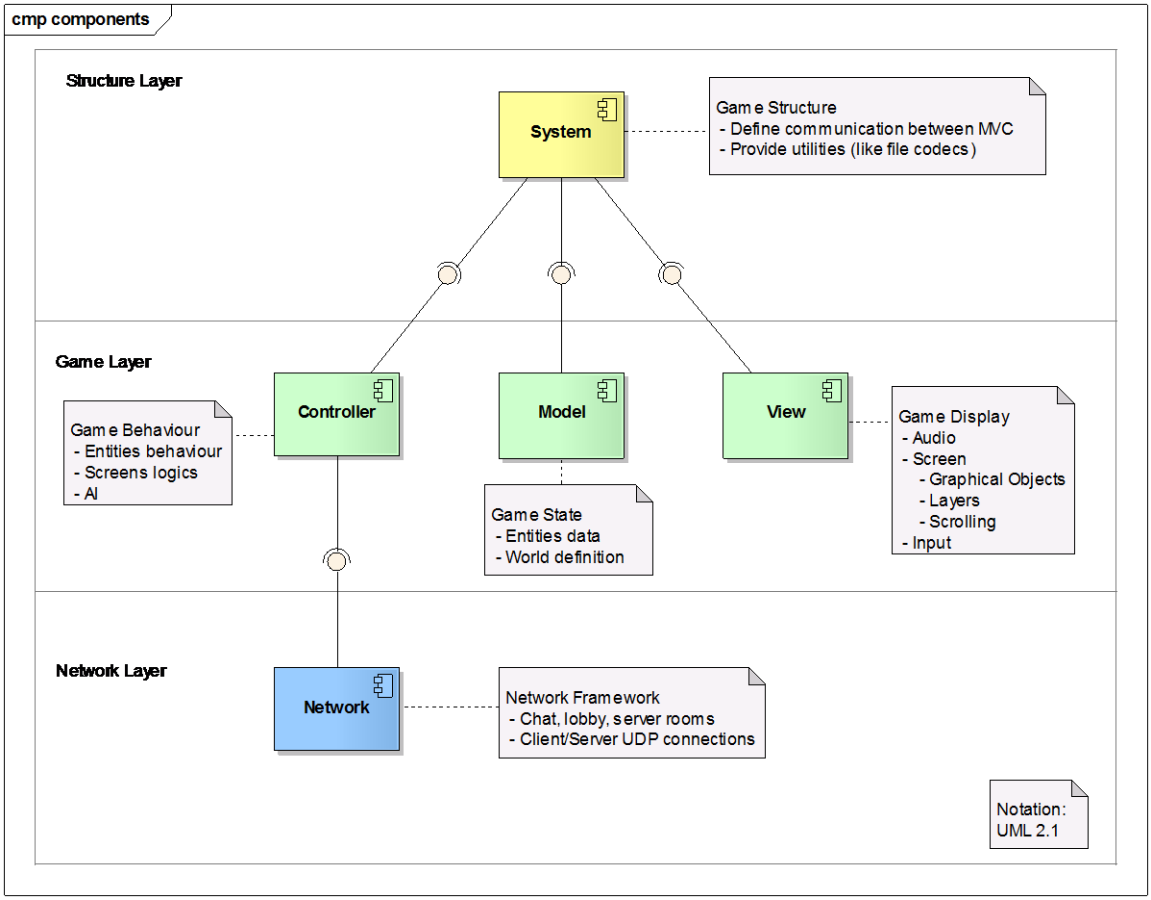

Figure 5 - UML component diagram of EvE

3.2.3 Element catalog

System	Component that links all other by defining their interfaces.
Model	Responsible for data representation and game state management
View	Responsible for the presentation of the game elements.
Controller	Decides what happens, when and how, defines the behaviour of the game.
Network	Controls all network operations.

3.2.4 Variability mechanisms

Each component with exception for System can be replaced by another that respects the same linking interface with the System component. Network is a sub-component, responsibility of the controller component. Other sub-components can be attached to others by requiring and providing the necessary interfaces.

3.2.5 Architecture background

The ease on switching components keeps the engine very dynamic. Completely different results can be achieved by connecting different components, the unique restriction is that they offer the interfaces required by the System, which links the components together and makes them work.

3.2.6 Related Views

- Modules Decomposition
- Game Organization
- Packages

3.3 Game Organization View

3.3.1 View Description

This view shows how the game components exist as DLLs and are used by an executable file.

3.3.2 Primary presentation

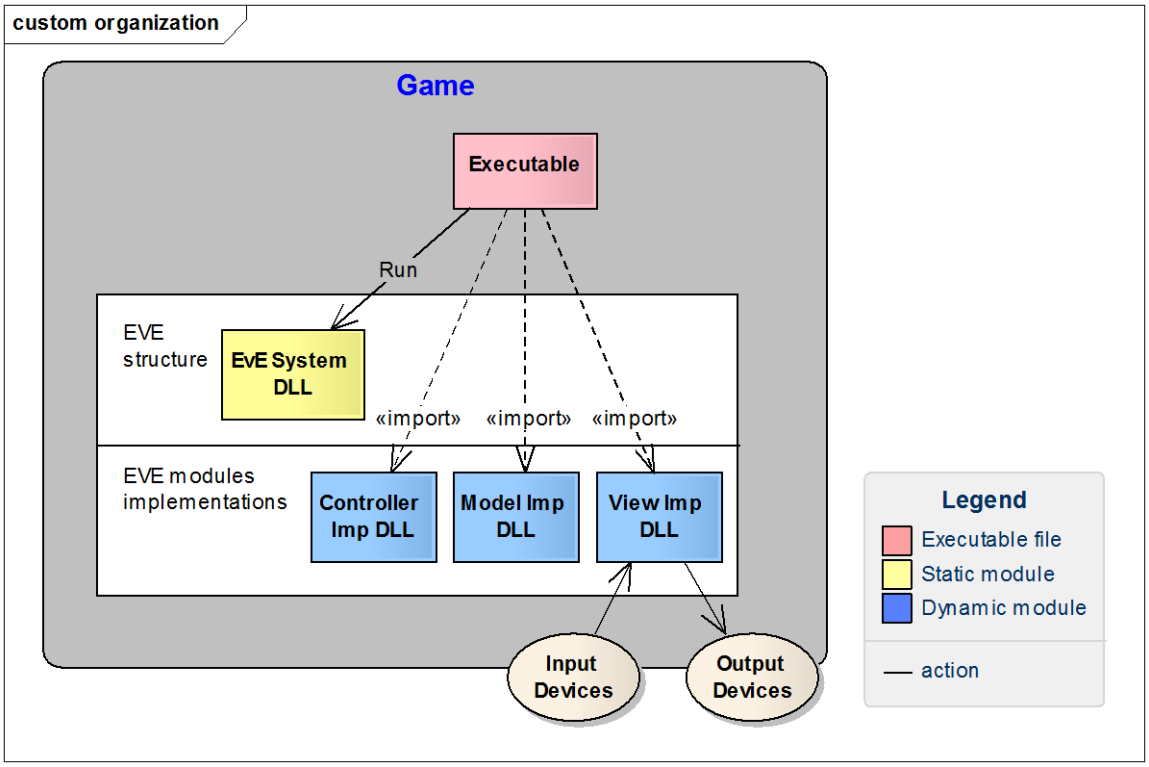

Figure 6 - EvE game organization diagram

3.3.3 Element catalog

Executable	Executable file that imports EvE and run it.
EvE System DLL	DLL of the system component, makes everything work by using the modules imported by the executable file.
Model Imp DLL	A specific implementation of the model module. Responsible for data representation and game state management
View Imp DLL	A specific implementation of the view module. Responsible for the presentation of the game elements.
Controller Imp DLL	A specific implementation of the controller module. Defines the behaviour of the game.

Input Devices	Physical input devices, produces events to be processed by the engine.
Output Devices	Physical output devices, presents the output results (audiovisual) produced by the engine.

3.3.4 Variability mechanisms

The implementations of the MVC modules can be anything, respecting the interfaces defined by the system module.

3.3.5 Architecture background

Letting the executable file decide which modules are imported allows all modules, including system, to be completely independent and unaware of each other. When the executable imports them, System receives the instances of the MVC modules and put them working together. Multiple implementations may exist, the import of different implementations may lead to different results. The executable may import the modules directly by code or dynamically via a configuration file or by user input. This may be useful for example to decide whenever run the same game with a 2D or a 3D view.

View module is the one responsible for communicating with the outside and may be the unique dependant of the OS and physical platform.

3.3.6 Related Views

- Modules Decomposition
- Components
- Packages

3.4 Editor Interfaces View

3.4.1 View Description

This view presents the interfaces of the editor that will use the first implementation of EvE modules. The primary presentation was produced with the beta version of [Mockingbird 10].

3.4.2 Primary presentation

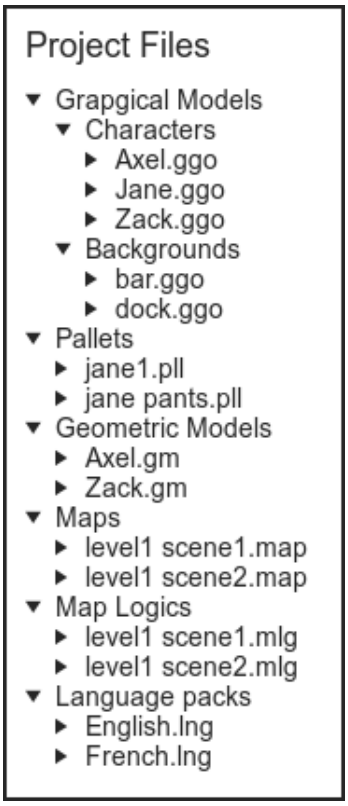

Figure 7 - Files organization example

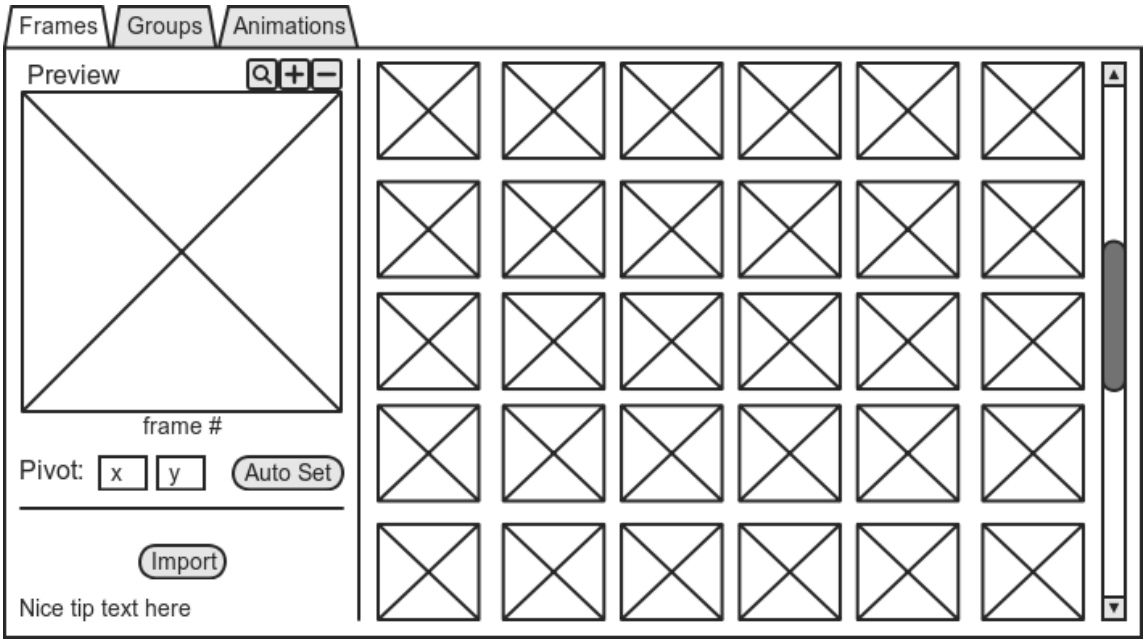

Figure 8 - Editor - GM frames

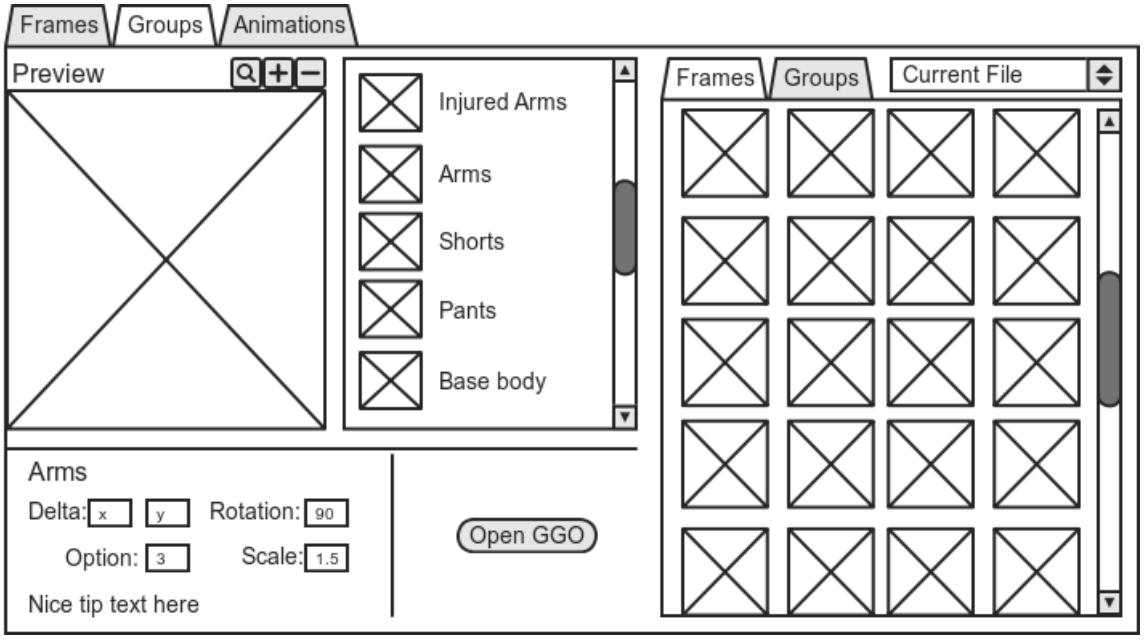

Figure 9 - Editor - GM groups of frames

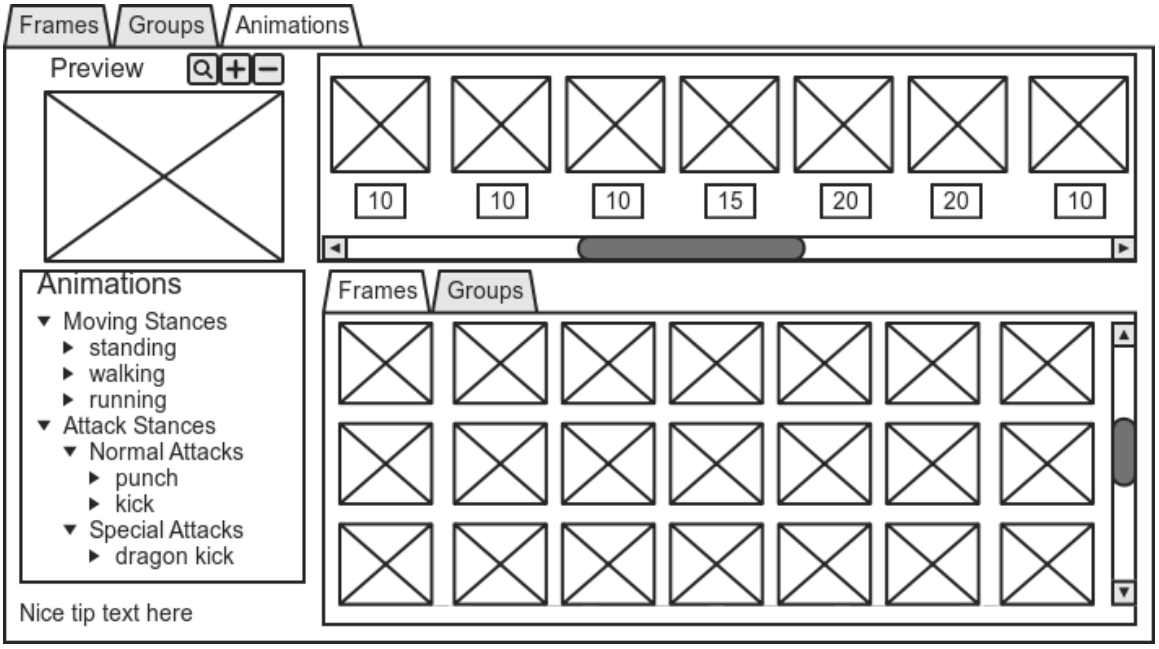

Figure 10 - Editor - GM animations

3.4.3 Element catalog

Files Organization	On the main screen of the editor a file management tree like this will be shown. Files will be shown separated by their type, and for each type files are grouped by user created folders. Those folders shouldn't be modified outside of the editor because paths are part of the configuration of a game project. From the files management screen it is possible to remove or move files, and to create and edit files, switching to the respective screen of the editor.
Frames	When creating or modifying a GM, this page allows to manage the frames of the GM (import, edit, remove). The import operation allows to open images from the user's computer, in the most common formats like png, jpg, bmp, etc. Those images are copied and turned into frames of the GM. On the View module the frames pivot will coincide with the object coordinates when drawing a graphical object on the screen, so basically the pivot is the base reference of the model (for example the feet of a character, the hold point of a weapon, etc).
Groups of Frames	When creating or modifying a GM, this page allows to manage groups of frames (create, edit, remove). A group is composed by several frames or grouped frames in a layered way. Each layer can be transformed by basic geometric transformations (translate, scale, rotation). The pivot is determined by the pivot of the first layer, plus it's translation. Grouped frames can be composed of frames and grouped frames of the self GM, and also from external GMs. Frames can be dragged and dropped on the layers list to create new layers. Each layer has an additional field named option. Option will be used by the game View (perhaps by scripts on the view) to determine what layers are shown and with what characteristics (for example on what layers should a specific pallet be applied, or what level of transparency should be applied to specific layers of a grouped frame). Individual layers may be manipulated by View scripts (for example to produce special animations based on the layers geometry).

Animations	When creating or modifying a GM, this page allows to manage animations (create, edit, remove). An animation is composed by a sequence of frames and/or grouped frames, each with a delay associated, measured in game frames. For example, if a game runs in 60fps, a frame with a delay of 60 will play for one second. Animations are organized in a tree structure with user created sections to simplify the user work.

3.4.4 Variability mechanisms

N/A.

3.4.5 Architecture background

The edition of all kind of files using the same unique editor simplifies the work to users, they don't have to switch from program to program to produce the material of their game. They manage the whole project in one program. This also helps keeping integrity and safety of the project files. Only the owner(s) of a project can access their files though that project.

This view is very incomplete, it should be updated as soon as the editor interfaces are better planned.

3.4.6 Related Views

- Modules Decomposition
- Game Organization

3.5 Scripting API View

3.5.1 View Description

This view presents the scripting API that allows users to call EvE internal functions and to respond to EvE events, in order to produce the desired behaviour for characters, levels, view effects, etc.

3.5.2 Primary presentation

N/A

3.5.3 Element catalog

N/A

3.5.4 Variability mechanisms

N/A

3.5.5 Architecture background

This view isn't enough planned yet, but it is here for convenience.

3.5.6 Related Views

- Modules Decomposition
- Game Organization
- Editor Interfaces
- Packages
- System Classes

3.6 Packages View

3.6.1 View Description

This view presents the organization of the code in packages.

3.6.2 Primary presentation

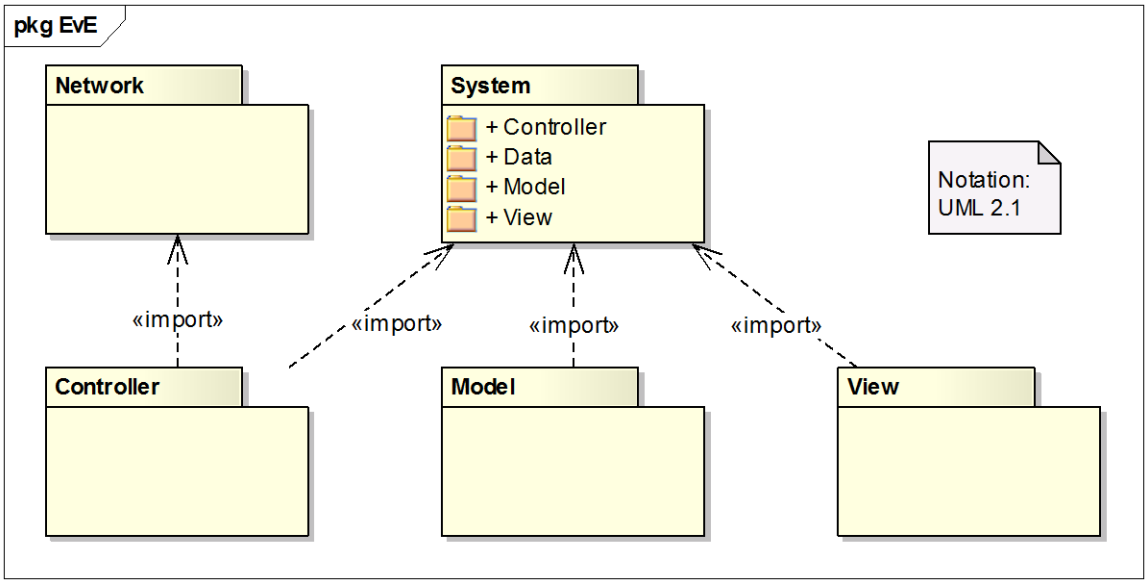

Figure 11 - UML package diagram of EvE

3.6.3 Element catalog

System	Package that defines the interfaces of the MVC packages.
Model	Responsible for data representation and game state management, imports the System package to know the interfaces and to access utilities.
View	Responsible for the presentation of the game elements, imports the System package to know the interfaces and to access utilities.
Controller	Decides the behaviour of the game, imports the System package to know the interfaces and to access utilities.
Network	Package responsible for all internet communications, it is an individual package.

3.6.4 Variability mechanisms

N/A

3.6.5 Architecture background

System package looks as a mockup of the MVC modules because of having subpackages for MVC which defines their communication interfaces, the skeleton of the whole engine. The System subpackage Data is about codecs and files access.

This view is incomplete because the MVC and Network packages, corresponding to the first implementation of the respective modules, aren't defined yet. They may have several subpackages as needed.

3.6.6 Related Views

- Modules Decomposition
- System Classes

3.7 System Classes View

3.7.1 System Package Classes

3.7.1.1 View Description

Classes of the System package refers to general utilities and base classes.

3.7.1.2 Primary presentation

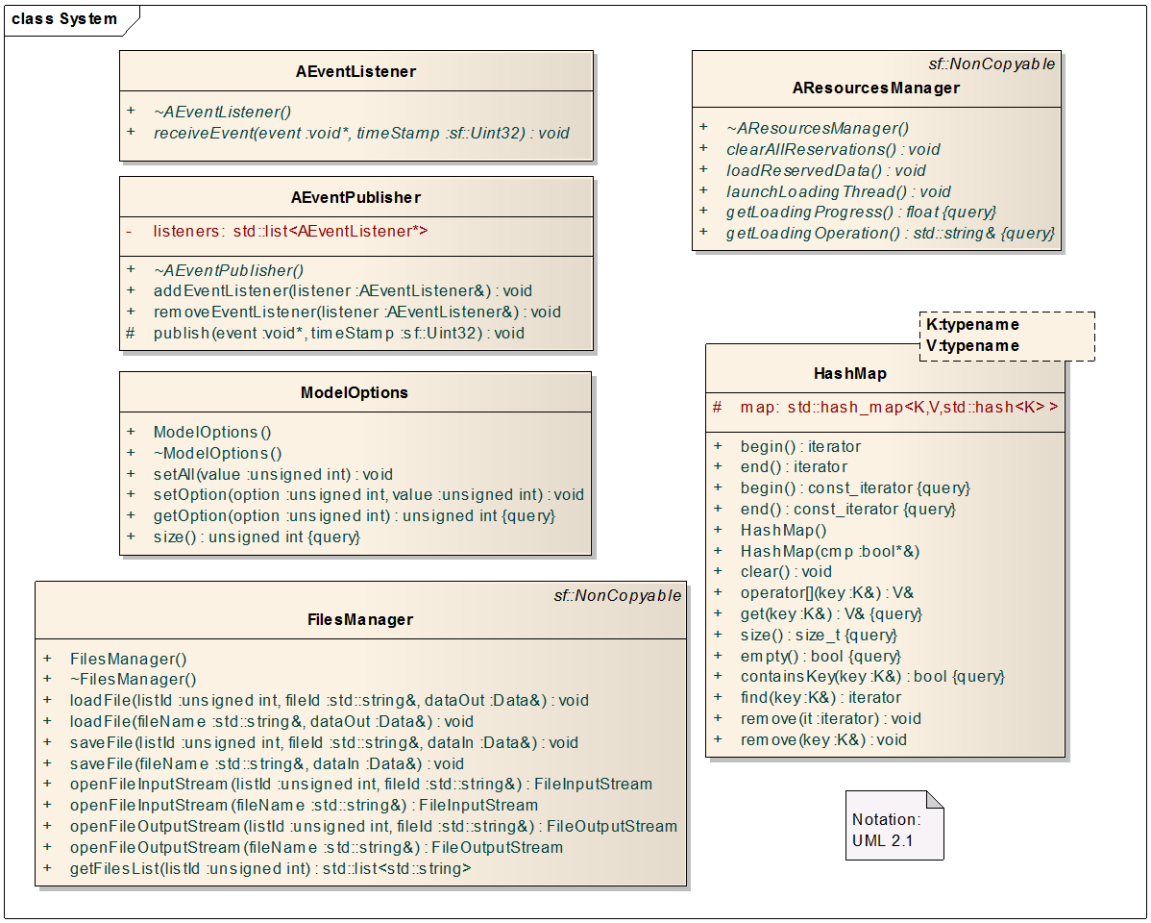

Figure 12 - UML class diagram of System package classes

3.7.1.3 Element catalog

AEventListener	Base class for event listeners. Input events may be key presses, mouse clicks or even selection of menu items. Events should be defined and agreed by all interested parts.
AEventPublisher	An event publisher regists event listeners (objects of type AeventListener). When publishing an event it is sent for all listeners.
AResourcesManager	Base class for resources loading and management. Resources are first reserved by the manager. That means resources are signalled to be loaded, but are not loaded yet. All reserved resources are only loaded on a call to loadReservedData or launchLoadingThread. On those calls, all previously loaded resources that wasn't reserved for this loading are freed and all reserved resources are loaded.

ModelOptions	Model options are useful for the model control as well as for it's visual representation. Those options may be used to select specific items to be used or shown, or to save/load only a specific collection of items, accordingly to the values of the model options.
HashMap	Provides all common hash map methods and operators. It simply uses a specific hash map implementation behind. This is needed while hash map isn't part of C++ Standard Library.
FilesManager	Class conforming to file input/output operations. It allows to easily load and save files with the standard EvE codec. Those operations includes loading/saving a file at once, or by creating a stream to it. The files needed by a game are listed in special index files. This class keeps those lists updated and allows to save/load files identified by their indexes on the lists.

3.7.1.4 Variability mechanisms

N/A

3.7.1.5 Architecture background

The event publishers and listeners fits with the publish/subscribe pattern, a good way of keeping publishers unaware of who exactly is expecting their messages.

The extensions of the *AResourcesManager* will certainly use the *FilesManager* in order to load the reserved data. Codecs and files management becomes hidden from the rest of the engine.

3.7.1.6 Related Views

- Components
- Packages
- Controller package classes
- Data package classes
- Model package classes
- View package classes

3.7.2 Controller Package Classes

3.7.2.1 View Description

This view presents the base classes that constitutes the interface that the Controller modules must implement.

3.7.2.2 Primary presentation

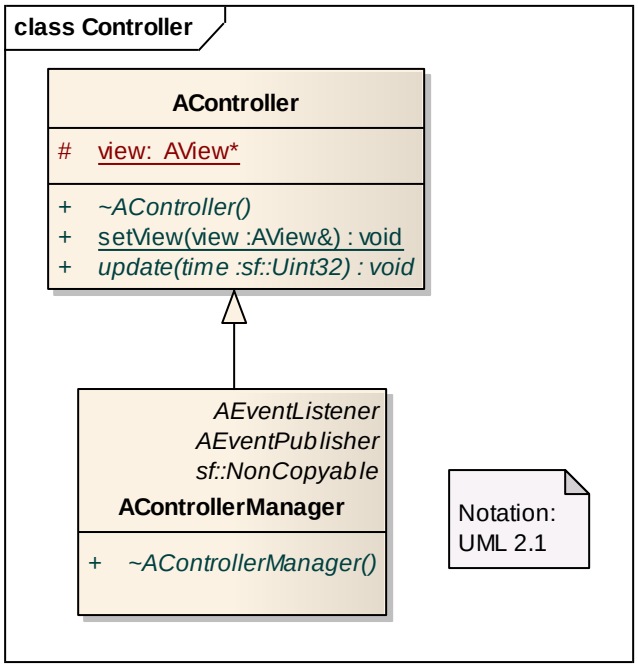

Figure 13 - UML class diagram of System::Controller package classes

3.7.2.3 Element catalog

AController	A game controller listen to what is happening on the game, and react accordingly. It may react to events like collision detection, proximity, certain position, specific variables reaching a certain value, etc. There may be controllers of controllers, and not all controllers must be updated on every game clock. Controllers of controllers may listen to events and update their controller listeners so that they can react.
AControllerManager	Base class for a controller manager. It listen to events that may be produced by a View Module, handle them, and if necessary, redirect them

	or produce and publish new events. It is also a controller itself. Possible listeners of a controller manager may be a game log, network, other controllers, etc.

3.7.2.4 Variability mechanisms

N/A

3.7.2.5 Architecture background

This package seems deceptively empty, however those are the unique interfaces required for the controller module. It only has to be able to receive and publish events. Controller modules are free of deciding how they manage the other modules.

3.7.2.6 Related Views

- Components
- Packages
- System package classes
- Data package classes
- Model package classes
- View package classes

3.7.3 Data Package Classes

3.7.3.1 View Description

This view presents the utilities provided by the System module that refers to data manipulation.

3.7.3.2 Primary presentation

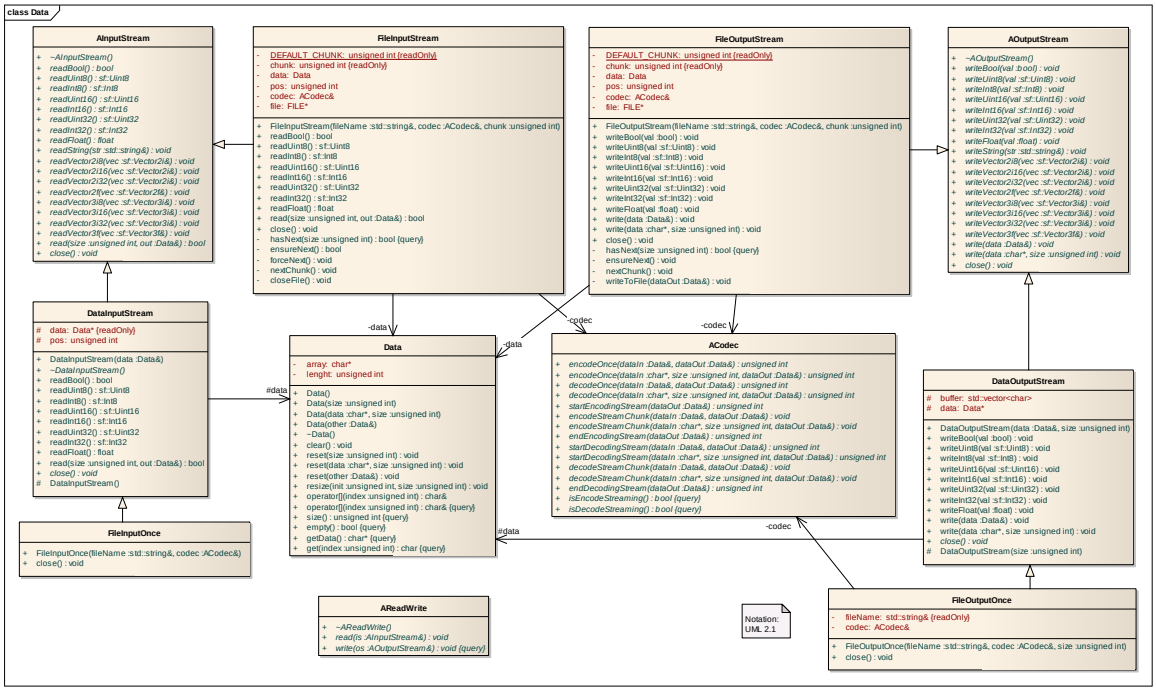

Figure 14 - Figure 15 - UML class diagram of System::Data package classes

3.7.3.3 Element catalog

Data	Useful data container. It is composed of a dynamic byte array and an integer containing it's size. It is designed to protect data integrity and avoid leaks.
ACodec	Base class for codecs. Codecs are responsible for compression/ /decompression and encryption/decryption of data files. There is two ways of encode/decode data files: – all in one step: data is all set into memory and encoded/ /decoded all in one step before being saved to file or accessible to the application. – streaming: data is loaded/saved in chunks, each chunk is encoded/decoded as needed.
AInputStream	Base class for formatted input streams. Input streams allows a nice formatted input from raw data. Input streams should be closed after use, however they are also closed on destruction.
AOutputStream	Base class for formatted output streams. Output streams allows a nice formatted output into raw data. Output streams should be closed after

	use, however they are also closed on destruction.
AReadWrite	Base class for objects that need to be read/written from/to input/output streams.
DataInputStream	This kind of stream provides formatted input from a raw *Data* object.
DataOutputStream	This kind of stream provides formatted output to a *Data* object.
FileInputOnce	Input from a file, full read in one step. This class reads a file at the constructor, decoding it in one step and storing the resulting data on memory, so it is ready for formatted input operations. Ideal for small data files.
FileOutputOnce	Output to a file, full write in one step. This class uses a buffer to store the data written with the formatted output operations. At the close operation it encode the buffer data and save it to a file all in one step. Ideal for small data files.
FileInputStream	Input by streaming from a file. Every chunk read is decoded and is ready for formatted input. When the end of a chunk is reached, it tries to read the next chunk of data.
FileOutputStream	Output by streaming into a file. Data is written with the usual formatted output operations. Those operations fills an internal buffer. When the buffer is full data is encoded and written on the file, and the buffer is clean to receive more data.

3.7.3.4 Variability mechanisms

N/A

3.7.3.5 Architecture background

Codecs classes can be used for file input/output, but also for other data streaming operations, like streaming into/from the network. The system module implements it's own codec, it is private to guarantee that no one find how the codification is made. That codec is the one used to codify the game designers resources and configuration files.

3.7.3.6 Related Views

- Components

- Packages
- System package classes
- Controller package classes
- Model package classes
- View package classes

3.7.4 Model Package Classes

3.7.4.1 View Description

This view presents the base classes that constitutes the interface that the Model modules must implement.

3.7.4.2 Primary presentation

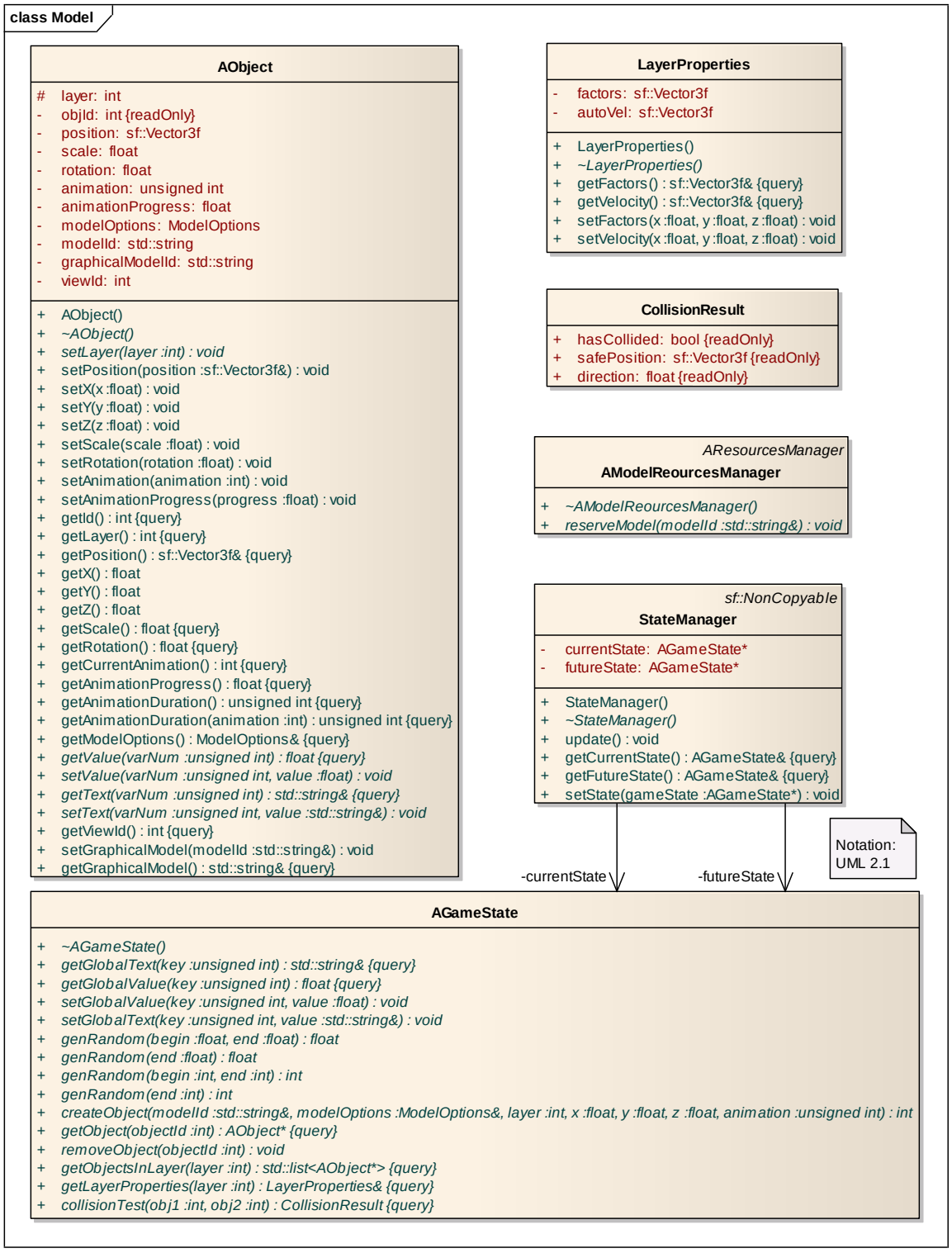

Figure 16 - UML class diagram of System::Model package classes

3.7.4.3 Element catalog

AGameState	Base class providing game state operations. Any implementation of game state should allow access to global variables, objects and their variables, random numbers generator, and additionaly to the game layers properties and a collision detection method.
AModelResourcesManager	Base class for managers of the models resources in memory.
AObject	Abstract logical representation of an object. Represents the object state, including position, animation, and all of it's free variables.
CollisionResult	Represents the result of a collision. Can indicate if there was a collision or not, what is the position immediately before the collision, and the final bouncing velocity. Objects of this type are returned by the game state. It works more as a struct, all data fields are public, but const.
LayerProperties	Properties of a game layer. This is used mainly to keep track of layers own coordinate systems, in order to perform conversions between layers. For example, when moving an object from one layer to another it may be important to convert it's coordinates accordingly to the coordinate systems of both layers.
StateManager	Controls the game state. The game state changes at every game clock. On each game clock it is possible to get the actual state, but also to request modifications to take place on the next state. That's why this class keeps two states: – current state; – future state. The update replaces the current state by the future state.

3.7.4.4 Variability mechanisms

N/A

3.7.4.5 Architecture background

The interface of the Model module may not be complete yet. State operations are all that the module has to offer to the controller. How it represents the models data are up to the implementa-

tions of the interface. All objects of a game are represented by the same object interface, so all are treated the same way on the Model. Controllers decide how they behave.

3.7.4.6 Related Views

- Components
- Packages
- System package classes
- Controller package classes
- Data package classes
- View package classes

3.7.5 View Package Classes

3.7.5.1 View Description

This view presents the base classes that constitutes the interface that the View modules must implement.

3.7.5.2 Primary presentation

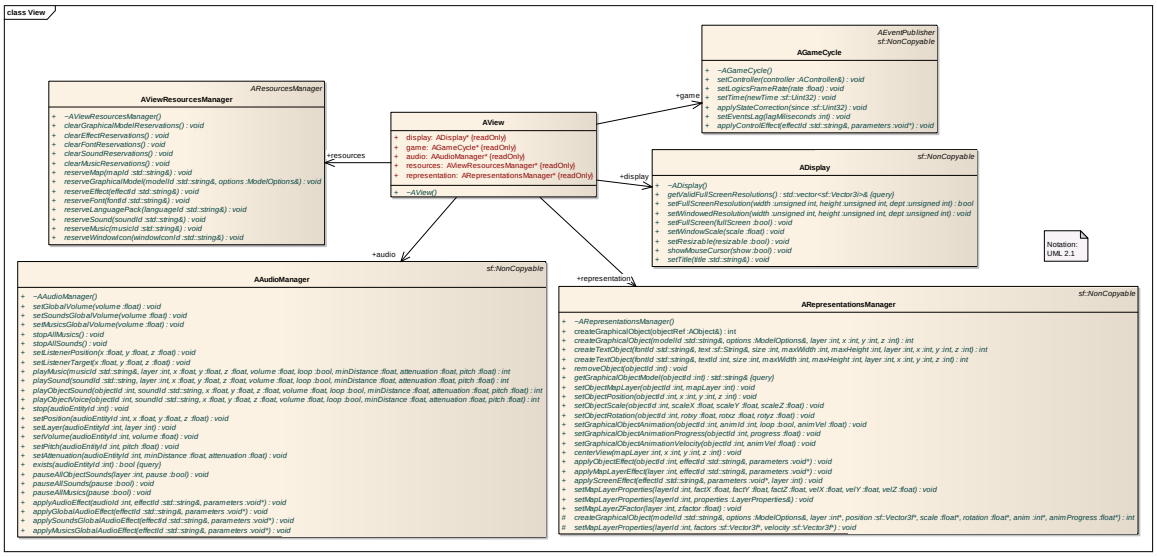

Figure 17 - UML class diagram of System::View package classes

3.7.5.3 Element catalog

AAudioManager	Base class for audio operations. Audio operations includes play sounds and musics, spatialization, audio effects, etc.
ADisplay	Base class for control of the display. It is responsible for displaying the representations of the objects on an output medium.
AGameCycle	A game cycle updates the game controllers and view internal scripts, and display the objects representations on the output medium by making use of a *ADisplay* object.
ARepresentationsManager	Controls everything that is displayed on the screen.
AView	Provides all abstract objects of a specific View implementation. External modules will use the implementations of this class to access the view module operations. On the View implementations, one derivation of this class should be the unique public class, all others should be private so that encapsulation and implementation privacy are kept in good stand.
AViewResourcesManager	Base class for loading/unloading resources of the View module. It keeps track of: – graphical models – special effects – text fonts – sound sources – music sources – language pack – window icon

3.7.5.4 Variability mechanisms

N/A

3.7.5.5 Architecture background

The most basic element of the View is the representation of an object. It uses a graphical model and is displayed on a certain position, accordingly to a certain animation. It's also possible to create and display text objects. It is important to remember that a scene is composed of layers. Each layer can have several text and representation objects. It is also possible to apply special effects to

singular objects, layers, or to the entire screen. Not less important is to control the scrolling by telling where we want the screen to be centred on.

On the audio side, the sounds plaid on objects can be of two types:

- normal sounds: they accompany the object and multiple sounds of this kind can be plaid for the same object.
- voice sounds: acompain the object, but unlike normal sounds, only one voice can be plaid at a time. If a voice is playing, a new voice interrupts the previous one.

3.7.5.6 Related Views

- Components
- Packages
- System package classes
- Controller package classes
- Data package classes
- Model package classes

3.8 Runtime View

3.8.1 Create Object Scenario View

3.8.1.1 View Description

This view illustrate the scenario of the creation of an object.

3.8.1.2 Primary presentation

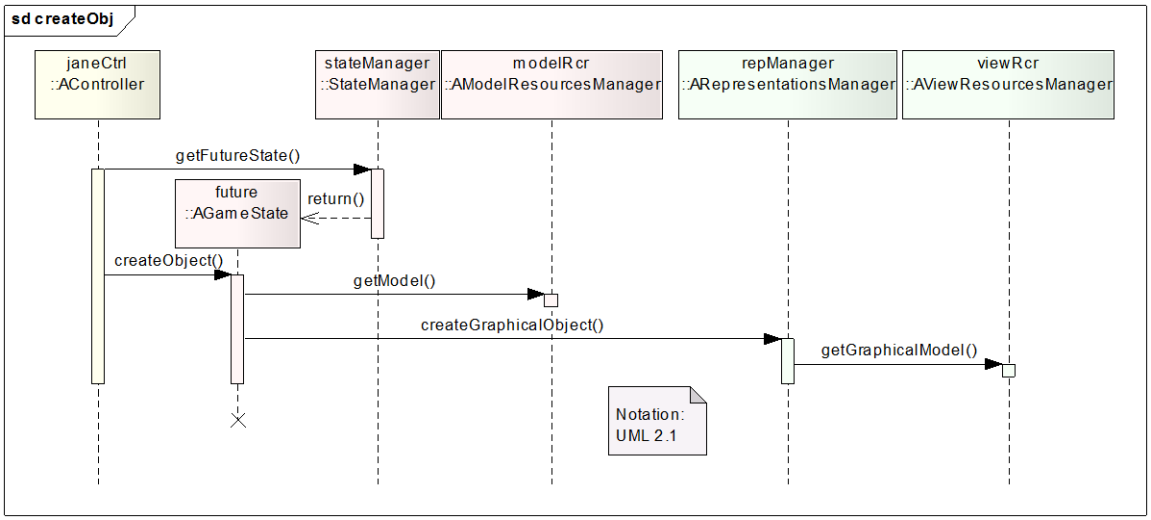

Figure 18 - UML sequence diagram of create object scenario

3.8.1.3 Element catalog

janeCtrl ::AController	Example of a controller.
future ::AGameState	Future game state.
stateManager ::StateManager	Object that keeps track of the game state.
modelRcr ::AModelResourcesManager	Manager of the Model resources.
repManager ::ARepresentationsManager	Object responsible for the View representations management.
viewRcr ::AViewResourcesManager	Object responsible for the View resources management.

3.8.1.4 Variability mechanisms

Instead of automatically creating the graphical representation, the game state can leave that for the controller.

3.8.1.5 Architecture background

To create a new object the controller must request the future state from the state manager, and create on it the desired object. This operation will automatically search on the model resources for the model to be used by that object. It can also request on the View module for a suitable representation. In that case both model and view objects get linked together. The View module searches for the right graphical model to be used by the representation object.

3.8.1.6 Related Views

- Components
- Packages
- System Classes
- Events propagation scenario
- Game update scenario
- Load data scenario

3.8.2 Events Propagation Scenario View

3.8.2.1 View Description

This view illustrate the scenario of the propagation of a player input event.

3.8.2.2 Primary presentation

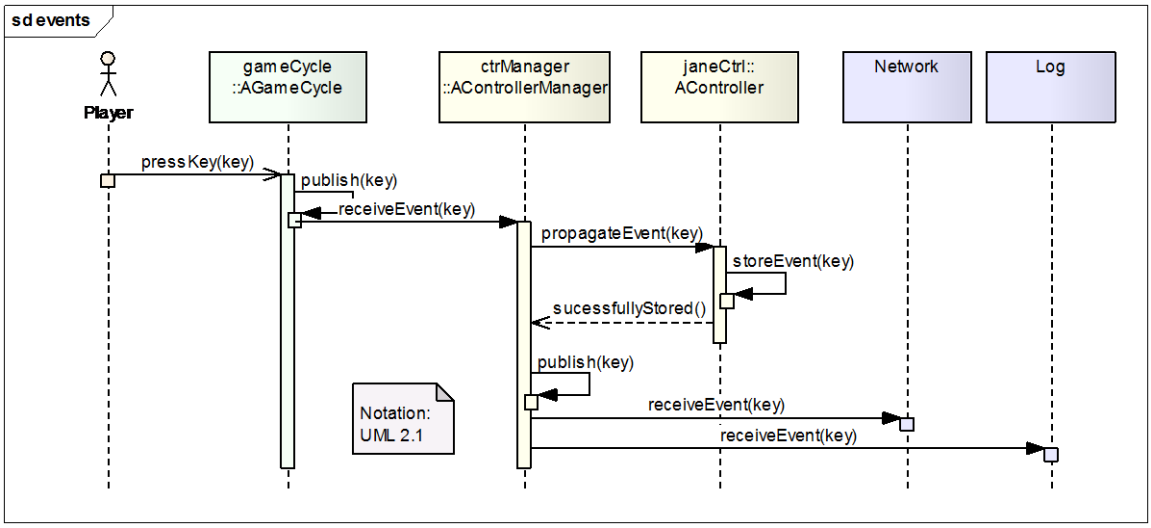

Figure 19 - UML sequence diagram of events propagation scenario

3.8.2.3 Element catalog

Player	User playing an EvE based game.
gameCycle ::AGameCycle	Object responsible for making the game update and display, and to propagate events to listeners.
ctrManager ::AControllerManager	Controller manager, controls all controller objects.
janeCtrl ::AController	Example of a controller.
Network	Network framework.
Log	Event logging system.

3.8.2.4 Variability mechanisms

Network and Log are optional, they are examples of other listeners that can be listening to events propagated by the controllers manager.

3.8.2.5 Architecture background

The pressKey event from the player is illustrative, other kind of events may occur, like from other input devices, from menu selections or other View internal controllers and scripts.

When the controller manager receives an event, it won't automatically publish it to it's listeners. First it propagates the event for the controllers that are interested on that kind of event, and awaits their feedback. If the event was accepted and produced any effect, then the main controller publish it. This way only relevant events are published to the network, logging system, etc.

3.8.2.6 Related Views

- Components
- Packages
- System Classes
- Create object scenario
- Game update scenario
- Load data scenario

3.8.3 Game Update Scenario View

3.8.3.1 View Description

This view illustrate what happens on a game update.

3.8.3.2 Primary presentation

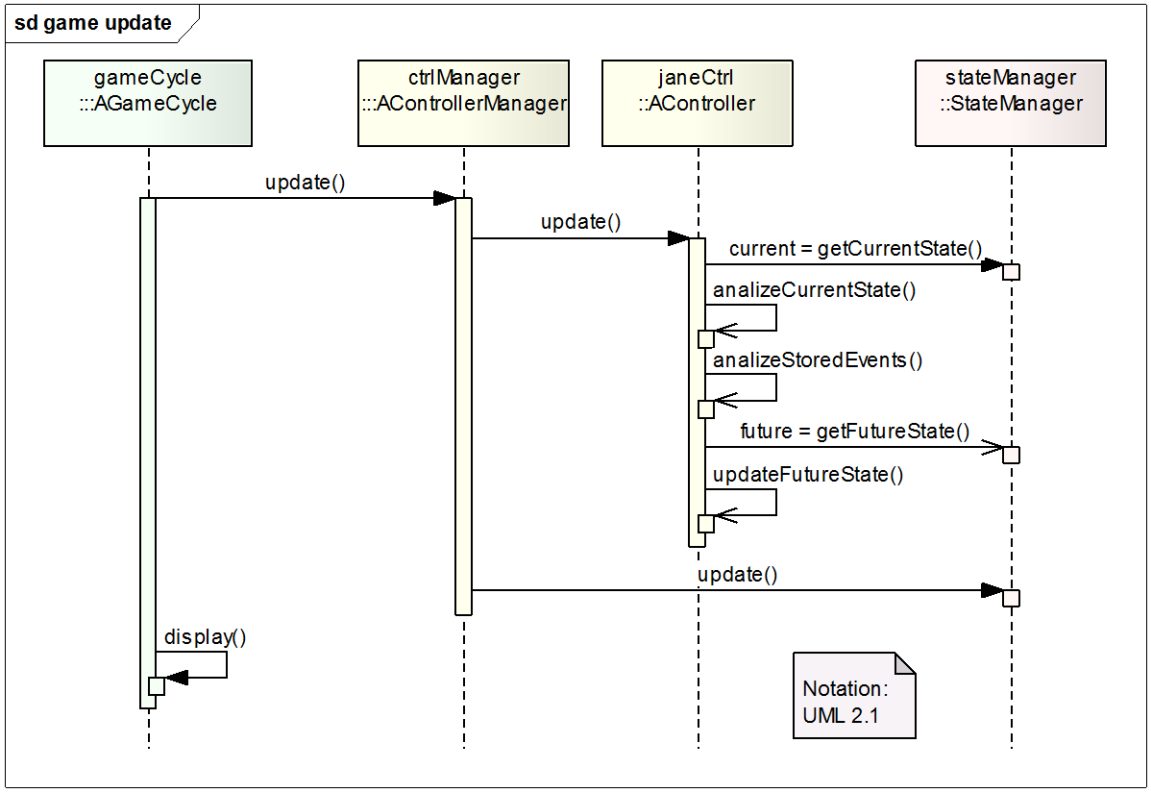

Figure 20 - UML sequence diagram of game update scenario

3.8.3.3 Element catalog

gameCycle ::AGameCycle	Object responsible for making the game update and display, and to propagate events to listeners.
ctrlManager ::AControllerManager	Controller manager, controls all controller objects.
janeCtrl ::AController	Example of a controller.
stateManager ::StateManager	Object that keeps track of the game state.

3.8.3.4 Variability mechanisms

Controllers may be updated on events and not directly from an update call from the controller manager.

I the update takes too long, the display may not occur to recover some or all of the time lost.

3.8.3.5 Architecture background

A controller update resumes on analysing the current game state and the events that are waiting to take effect, in order to take decisions about what should happen on the next state. The controller manager only updates the game state after all controllers has been updated.

3.8.3.6 Related Views

- Components
- Packages
- System Classes
- Create object scenario
- Events propagation scenario
- Load data scenario

3.8.4 Load Data Scenario View

3.8.4.1 View Description

This view illustrate the scenario of a full loading operation, perhaps when setting up a game scene.

3.8.4.2 Primary presentation

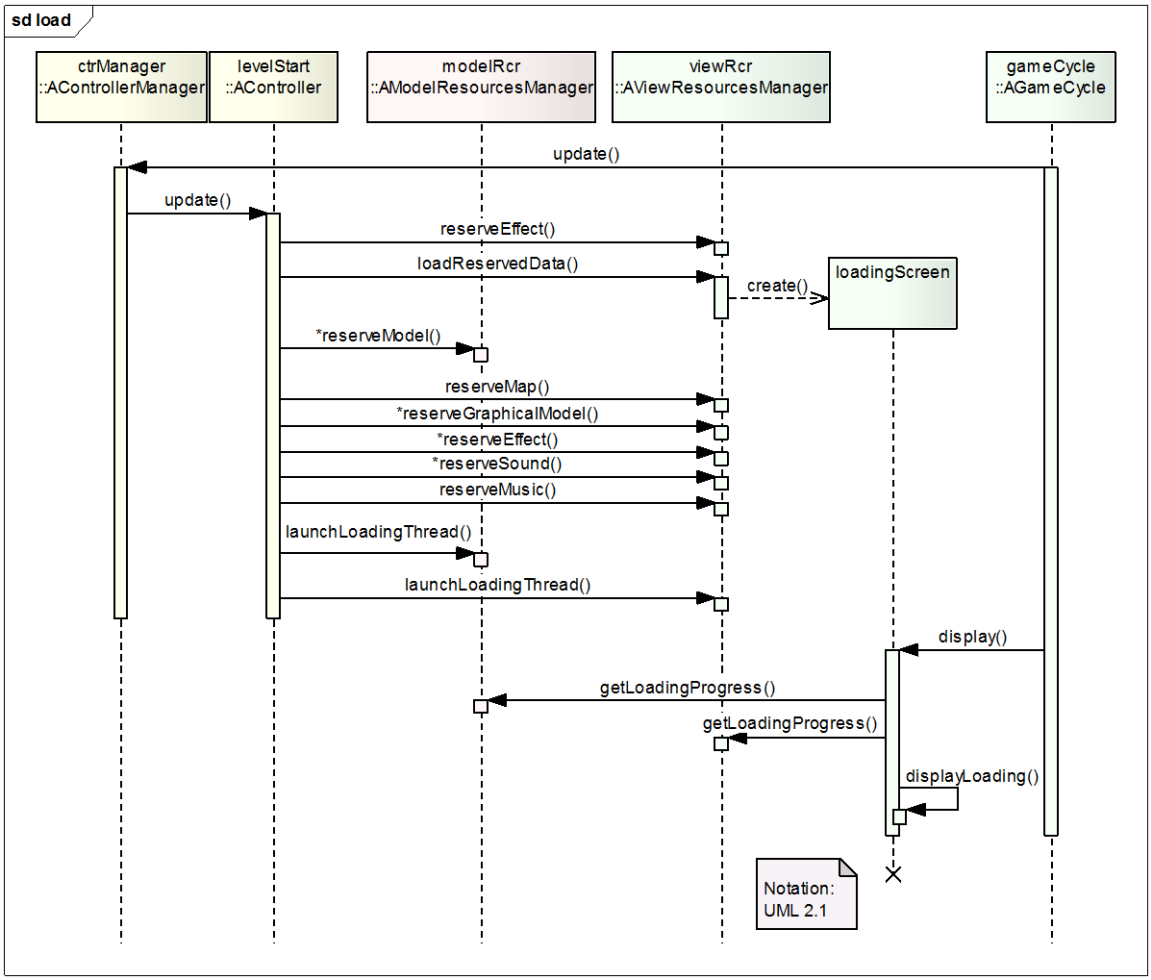

Figure 21 - UML sequence diagram of load data scenario

3.8.4.3 Element catalog

ctrlManager ::AControllerManager	Controller manager, controls all controller objects.
levelStart ::AController	Example of a controller.
modelRcr ::AModelResourcesManager	Manager of the Model resources.
viewRcr ::AViewResourcesManager	Object responsible for the View resources management.

gameCycle ::AGameCycle	Object responsible for making the game update and display, and to propagate events to listeners.
loadingScreen	Example of View internal controller (perhaps based on a script).

3.8.4.4 Variability mechanisms

This view is an example of a loading process, many other ways of doing it are possible. However this should be the most common one.

3.8.4.5 Architecture background

To present a loading screen for a new scene, first of all the resources needed for that loading screen must be loaded. Then the screen is finally ready to show the loading progress. That's when all the resources needed for the new scene are reserved and then the loading threads starts. While those threads are running, the game is still running, calling updates to the controllers of the Controller module and to the internal controllers of the View module (by a display call), in which the loading screen is. On those updates the loading screen looks into the loading threads progress and displays it on the screen.

3.8.4.6 Related Views

- Components
- Packages
- System Classes
- Create object scenario
- Events propagation scenario
- Game update scenario

4 Relations Among Views

Each of the views specified in Section 3 provides a different perspective and design handle on the system, and each is valid and useful in its own right. Although the views give different system perspectives, they are not independent. Elements of one view are related to elements of other views, and we need to reason about these relations. This section describes the relations that exist among the views given in Section 3. As required by ANSI/IEEE 1471-2000, it also describes any known inconsistencies among the views.

4.1 General Relations Among Views

The views of modules decomposition, components, game organization and packages gives a general view of the elements of they in different perspectives.

- Modules shows how the main blocks of the system are connected.
- Components specifies how the main modules of EvE are connected though interfaces.
- Game organization shows how the modules are imported and called on a game.
- Packages specifies how the classes that implement EvE are aggregated.

All of those elements represents the main layers of the system and how they communicate.

Classes views shows the structure of the code, which are organized in packages. Those packages are described at the packages view, so those views has a direct relation. Each classes view refers to the classes of one package.

The editor interfaces view has any relation with the others. It shows visual elements only, how the editor would look like. But it shows something about the data representation (what data is needed for each file type). There isn't a view for the data representation yet, but when it becomes more clear on EvE development it should be described on this SAD.

The scripting view has direct relations with the classes views, but since it isn't defined yet, it can't be relate yet.

The views referring to running scenarios uses objects of the classes specified on the classes views. Those classes are discriminated in front of the name of each object of the runtime views.

4.2 View-to-View Relations

The relations of the elements of the different views are mapped as follows:

4.2.1 General views mapping

Modules	Components	Game Organization	Packages
N/A	System	EvE System DLL	System
Model	Model	Model Imp DLL	Model
View	View	View Imp DLL	View
Controller	Controller	Controller Imp DLL	Controller
Network	Network	N/A	Network
Log	N/A	N/A	N/A
N/A	N/A	Input Devices	N/A
N/A	N/A	Output Devices	N/A
N/A	N/A	Executable	N/A

4.2.2 Scenarios

Create object	Events propagation	Game update	Load data
janeCtrl ::AController	janeCtrl ::AController	janeCtrl ::AController	levelStart ::AController
N/A	gameCycle ::AGameCycle	gameCycle ::AGameCycle	gameCycle ::AGameCycle
N/A	ctrManager ::AControllerManager	ctrlManager ::AControllerManager	ctrlManager ::AControllerManager

Create object	Events propagation	Game update	Load data
future ::AGameState	N/A	N/A	N/A
stateManager ::StateManager	N/A	stateManager ::StateManager	N/A
mod-elRcr::AModelResourcesManager	N/A	N/A	mod-elRcr::AModelResourcesManager
viewRcr::AViewResourcesManager	N/A	N/A	viewRcr::AViewResourcesManager
repManag-er::ARepresentationsManager	N/A	N/A	N/A
N/A	Network	N/A	N/A
N/A	Log	N/A	N/A
N/A	Player	N/A	N/A
N/A	N/A	N/A	loadingScreen

5 Referenced Materials

Bass 03	Bass, Clements, Kazman, *Software Architecture in Practice,* second edition, Addison Wesley Longman, 2003.
Clements 02	Clements, Bachmann, Bass, Garlan, Ivers, Little, Nord, Stafford, *Documenting Software Architectures: Views and Beyond*, Addison Wesley Longman, 2002.
CA 10	Coding the Architecture - Software architecture document guidelines. http://www.codingthearchitecture.com/pages/book/software-architecture-document-guidelines.html, accessed on April 2010
SEI 10	Software Enginnering Institute - SAD template 05-Feb-2006. http://www.sei.cmu.edu/architecture/SAD_template2.dot, accessed on April 2010
CMU 10	Carnegie Mellon University - wiki-based software architecture documentation, https://wiki.sei.cmu.edu/sad/, accessed on April 2010
Goulão 10	Miguel Goulão - Software Architectures 2009/2010 Template for the KWIC assignment report
IEEE 1471	ANSI/IEEE-1471-2000, *IEEE Recommended Practice for Architectural Description of Software-Intensive Systems*, 21 September 2000.
OMG - UML	Object Management Group - Unified Modelling Language specification, http://www.omg.org/spec/UML/, accessed on April 2010
Mitra 08	Tilak Mitra - Documenting software architecture, http://www.ibm.com/developerworks/library/ar-archdoc1/, accessed on April 2010
Mockingbird 10	Website wireframes: Mockingbird, http://gomockingbird.com/, accessed on August 2010

6 Directory

6.1 Glossary

Term	Definition
beat'em up	Video game genre featuring melee combat between the protagonist(s) and large numbers of enemies.
codec	Software program capable of encoding and decoding a digital data stream.
game engine	Software system designed for the creation and development of video games.
game entity	A game element with some properties and that can take some behaviour.
interface	Structure that defines the communication between two or more software systems or subsystems.
layer	A graphical part of a composed object or scene. Sidescrolling games usualy uses a parallax effect within various layers.
side scrolling	Video game genre where action is viewed from a side view and scrolls across the screen.
software architecture	The structure or structures of that system, which comprise software elements, the externally visible properties of those elements, and the relationships among them [Bass 03]. "Externally visible" properties refer to those assumptions other elements can make of an element, such as its provided services, performance characteristics, fault handling, shared resource usage, and so on.
stakeholder	An individual, team, or organization (or classes thereof) with interests in, or concerns relative to, a system. [IEEE 1471]
view	A representation of a whole system from the perspective of a related set of concerns [IEEE 1471]. A representation of a particular type of software architectural elements that occur in a system, their properties, and the relations among them. A view conforms to a defining viewpoint.
viewpoint	A specification of the conventions for constructing and using a view; a pattern or template from which to develop individual views by establishing the purposes and audience for a view, and the techniques for its creation and analysis [IEEE 1471]. Identifies the set of concerns to be addressed, and identifies the modelling techniques, evaluation techniques, consistency checking techniques, etc., used by any conforming view.

6.2 Acronym List

API	Application Programming Interface
CFR	Core Functional Requirement
CMU	Carnegie Mellon University
EFR	Editor Functional Requirement
EvE	Evolution Engine
FR	Functional Requirement
GM	Graphical Model
HUD	Head-Up Display
IEEE	Institute of Electrical and Electronics Engineers
LM	Logical Model
MMORPG	Massively Multiplayer Online Role-Playing Game
MVC	Model View Controller
N/A	Not applicable
NFR	Non Functional Requirement
OME	Organization Modelling Environment
OS	Operating System
QAS	Quality Attribute Senario
RPG	Role-Playing Game
SAD	Software Architecture Document
SEI	Software Engineering Institute
SoR	Streets of Rage
SoRE	Streets of Rage Evolution
UML	Unified Modelling Language