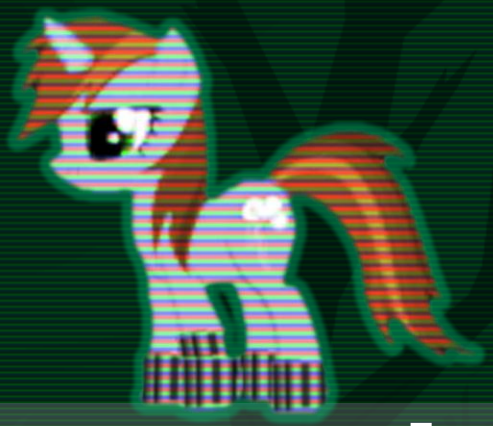
5/20/2012



Group 19

System design document for Fallout Equestria

Version: 2.0 | Lukas Kurtyan , Pontus Pall, Gustav Alm Rosenblad, Joakim Johansson

System design document for Fallout Equestria
Version: 2.0

Date: 5/20/2012

Authors:

Lukas Kurtyan

Pontus Pall

Gustav Alm Rosenblad

Joakim Johansson

This version overrides all previous versions.

Table of Contents

1 Introduction

1.1 Design goals

The project should be loosely connected in order to make anything possible.

1.2 Definitions, acronyms and abbreviations

- Archetype – A blue print for entities.
- Assets – Things that are loaded from hard drive, e.g. sound, textures, maps etc.
- Behavior – Defines how an entity behaves.
- Client – A computer connected to a host.
- Component – A class defining a property of an entity.
- Entity – A public key to a database containing components.
- Game world – A world containing entity systems, an entity database and entities.
- GPU – Graphics Processing Unit
- GUI, graphical user interface
- Host - The computer, to which the other computers are connected. Acts as a server. The place where logic is done, and the game is run.
- Java, programming language
- Look and feel – A XML-file and a large texture containing all the normal backgrounds, highlights, etc. that make up the GUI.
- LWJGL – A graphics library for java enabling access to Open GL.
- RGBA – A color format
- Screen – A layer on the display.

2 System design

2.1 Overview

The application uses a mix and match of different programming patterns and philosophies. The following headings will cover the most important of them.

2.2 Screen systems

It is a layered system able to show several screens at once on top of each other. It uses different screens for ***GUI*** and ***game worlds***. These screens transition between each other according to player input. This system is based on a C#-project, see reference. The important classes here are:

- The ScreenManager class, which is the container of the screens, managing the rendering and updating as well as making it possible to transition between screens.
- The GameScreen class, `which` is the base of all screens. It contains update and rendering logic.
- The GUIScreen class, `which` is the base of all screens containing GUI.
- The EntityScreen class, `which` is the base of all screens containing entity worlds.

2.3 Graphics system

All the graphics are done through ***LWJGL***. The main classes are:

- The SpriteBatch class, which batches sprites in order to draw them together instead of drawing them one by one. This increases performance significantly.
- The RenderTarget2D class, which makes it possible to use a SpriteBatch to render to a texture. This is done by using one of the specialized SpriteBatch begin method overloads. The texture can then be used again by using the *RenderTarget2D.getTexture()* method.
- The ShaderEffect class, which makes it possible to add shader effects to the drawing of the SpriteBatch. This is done by specifying the shader in a SpriteBatch begin method call.
- The Texture2D class, which is representing images on the ***GPU***.
- The Color class, which defines a ***RGBA***-color in the float format.

2.4 Entity Framework

The Entity Framework is core of everything dynamic in the application. I.e. everything that is inconsistent in the game world. This framework is inspired by the Artemis framework. It contains a number of classes, out of which the most important are:

- The IEntity class, which defines an ***entity***. This class has a direct connection to an entity database and can only be created through a IEntityFactory. This class is used throughout the entity systems in order to gain access to its label, groups and components. Furthermore, the network uses the ID of this class to synchronize the databases of the different players.
- The IComponent class, which is the base for all ***components***. To understand what a component is, examples are the best way to go:
  - PhysicsComp, adding this component to an entity gives it properties of mass, velocity etc. and makes it susceptible to gravity.
  - AnimationComp, adding this component makes it possible to animate and render the entity.
  - HealthComp, adding this component makes it possible to get injured and die.
- The IEntityDatabase class, which is implemented as a relational database. See figure below.

EntityID	IComponent A	IComponent B	IComponent C	…
0x000000	Null	B	C	
0x000001	A	B	Null	
0x000002	Null	Null	C	
0x000003	A	B	C	
…	…	…	…	

- The ComponentMapper, which maps a component type through the ComponentTypeManager to an index for fast retrieving of components from the database.
- The ComponentTypeManager, which maps a component to the specific index that is represented by the column in the IEntityDatabase. For instance the IComponent A in the example figure above is mapped to the column index 0.
- The EntityArchetype class, which is a class that the IEntityFactory can use to create an entity containing specific components. It is basically a runtime blueprint of an entity type.

- The EntityFactory class, which is the place where the entities are born, as well as the place they go to when they die. In other words, it is a factory for creating and recycling entities.
- The EntityGroupManager class, which enables entities to be placed in different groups, identified by a string group name. An example of usage of this is the CameraControlSystem, in which the camera is affected by all entities contained in the group “Camera targets”.
- The EntityLabelManager class, which is similar to the EntityGroupManager, but this class enables entities to be identified by one unique string label.
- The EntityManager class, which provides a simple interface for interacting with all the other features of the entity framework; e.g. EntityFactory, IEntityLabelManager etc.
- The EntitySystem class, which is the base of all entity systems. To run the game entity systems are required. For example, the following classes are entity systems:
    - AnimationSystem – This handles the animation components.
    - HealthRegenerationSystem – This handles the health components, and thereby regenerates health.
    - DeathSystem – Kills entities when they have zero health or less.
- The EntitySystemManager class, which manages entity systems, making them update and process as they should.
- The EntityWorld class, which contains an IEntityManager, IEntityDatabase and IEntitySystems. This class makes it easier to use the entity framework, by providing an easy interface to interact with.

These entity classes all implements respective interfaces, so that anything can have other custom implementation and still work. An example of this is the NetworkedEntityFactory that extends EntityFactory and thereby replacing it as the used IEntityFactory.

2.5 GUI

The GUI is based on the class GUIControl, as well as LookAndFeel, GUIRenderingContext and GUIRenderer<T>.

- The GUIControl is the base class of all GUI controls. It is responsible for the things common for all controls; like resizing, colorization, and event management.
- The LookAndFeel makes it possible to implement a wide range of different looks to the GUI in an easy way. The LookAndFeels can be loaded through the ContentManager.
- The GUIRenderingContext is what makes the GUI render. This by using the GUI render-targets and rendering things in the correct order. The rendering is done by GUIRenderers.
- GUIRenderer<T> Makes it able to render the GUI control of the type T. Controls that needs to be rendered in a significantly different way uses different renderers.

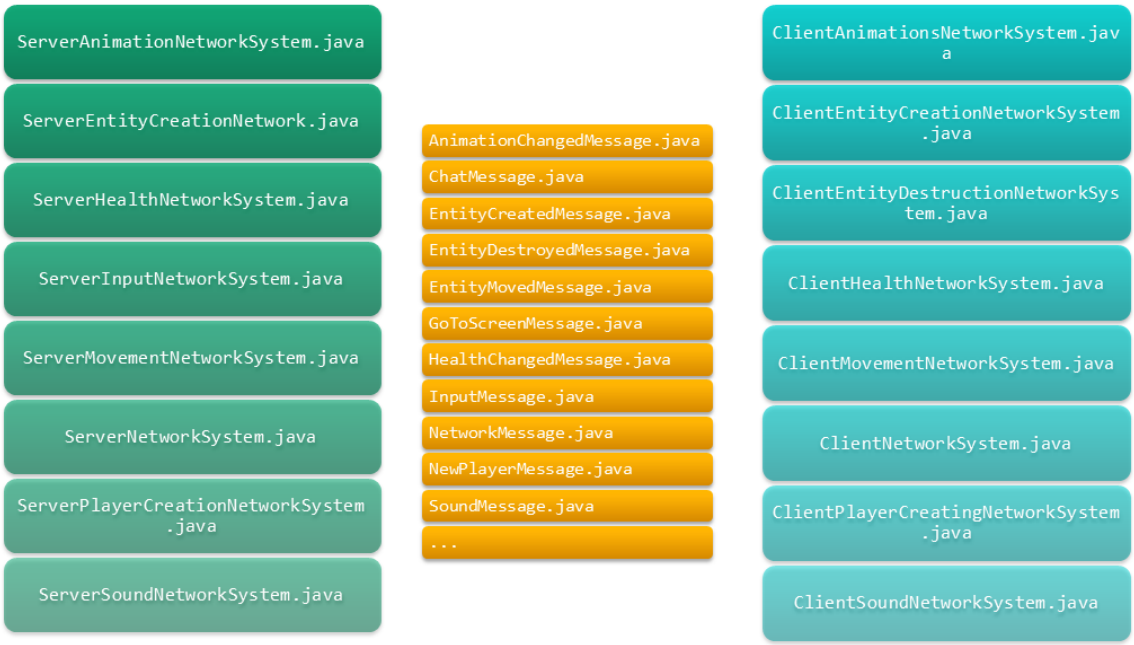
2.6 Network

The network uses the KryoNet library in order to send objects of the type NetworkMessage between the server and the clients. Both the server and

the clients have network systems handling the incoming messages and sending new messages.

The architecture used is *Authoritative Server*, i.e. all the logic is done on the server side and then sent to the client side for rendering. The clients only send their input and the server does the rest.

To enable networking, use NetworkSystemBuilder and specify the world and other arguments needed by the the networked entity systems. Examples of functioning parts of the system is ServerMovementNetworkSystem (SMNS), ClientMovementNetworkSystem (CMNS), EntityMovedMessage (EMM). Each update, the server sends the position of all entities through an EMM. The client then receives the EMM and uses it to update its entities.



2.7 Content

The framework is currently able to load files of the following formats:

- .anim (Format for animation)
- .animset (Format for a set of animations)
- .archetype (Format for entity archetypes)
- .effect (Format for vertex and fragment shaders)
- .font (Format for character information on the font texture)
- .ogg (Format for sound effects and music)
- .pchar (Format for player characteristics)
- .png (Format for textures)
- .scene (Format for the scenes in the game)
- .tdict (Format for look and feel, as well as body part information on animation textures)
- .XML (Format for things we didn’t come up with a good format name for)

Each format has one or more loaders derived from IContentLoader<T>, which converts the content into the type T. For XML-based content, JDom and XStream are used for the conversion. For loading of content, *contentManager.load(String path, Class<T> classToLoad)* is used. For example:

```
Texture2D texture = contentManager.load("foo",Texture2D.class);
```

2.8 Behaviors

The behaviors are inspired by the MonoBehavior class in Unity[1]. They are used to define very specialized behaviours, differing from the larger events handled by the IEntitySystems. For example; AI, input response, etc. are handle in these classes

2.9 Animation

This system makes it possible to animate. The animations are bone based and the animationsystem is based on a port from a C#-project named Demina. The Demina animation editor is used to make the animations.

2.10 Scene

Scenes are made through a tile editor previously implemented by Lukas Kurtyan in a hobby project which has been customized to fit this project as well as possible.

2.11 Utils

The package Utils contains various useful classes such Camera2D, Clock, Timer.

2.12 Software decomposition

2.12.1 General

Package diagram. For each package an UML class diagram in appendix

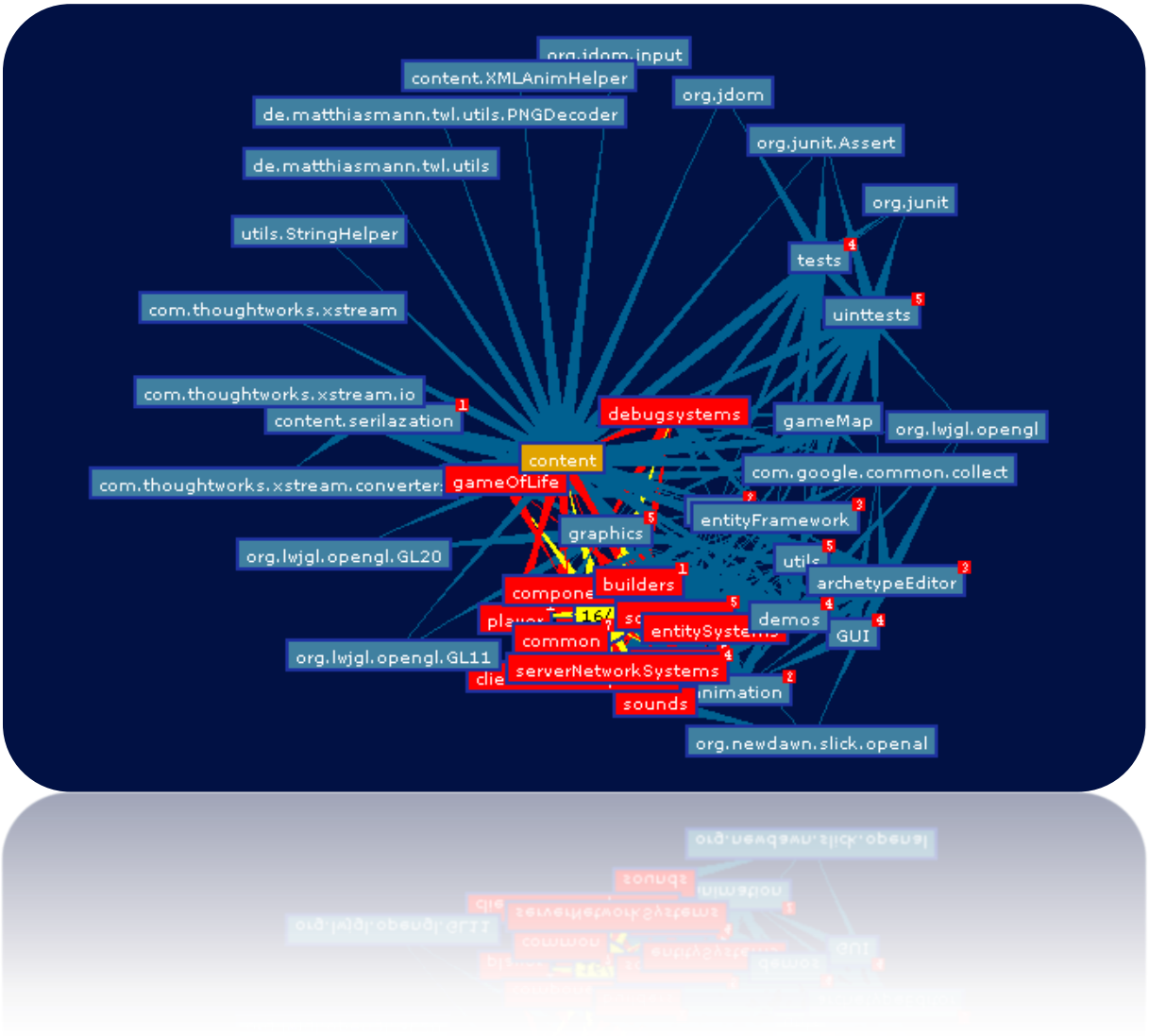
2.12.2 Layering

Even though the application is very flat in a hierarchical point of view, it can still be said to be significantly layered in the sense that is shown below. Different layers have different arbitrariness.

[1] http://unity3d.com/

2.12.3 Dependency analysis

The dependency of the application is overall good. There are some circular references between packages, but those are only between classes that should have been in the same package if not for the sake of organizing. If the classes are moved, the project would be too hard to overlook.



2.13 Concurrency issues

There were some issues mainly during the setup state of the network. Those were however resolved by locking with *synchronized*.

2.14 Persistent data management

All persistent data is stored inside the "resources" folder, and is accessed by using the different loading methods defined in ContentManager. Depending on the type of data, the ContentManager will in turn call the methods of filetype-specific loaders. The types of data that can be loaded has already been described in 2.7 Content.

2.15 Access control and security

N/A

2.16 Boundary conditions

N/A

The application may well be started through Eclipse. It should run as a desktop application.

3 References

APPENDIX

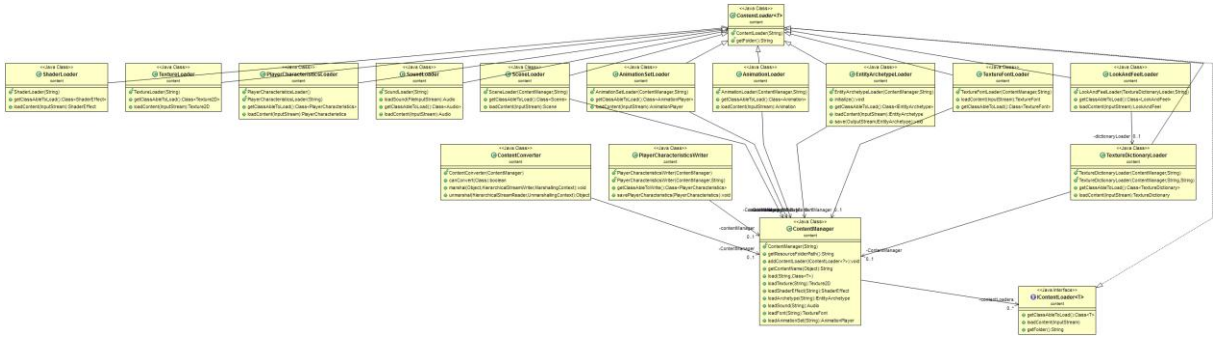


Figure 1 Content package

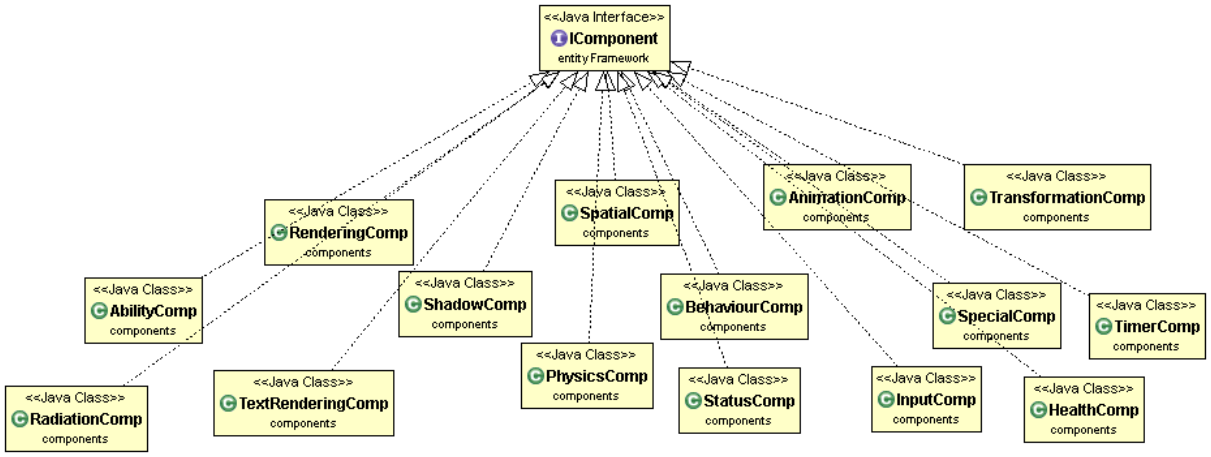


Figure 2 Components package