NTNU

Det skapende universitet

Department of
Computer and Information Science

A Component based Game Architecture for Unknown Horizons

Thomas Kinnen

TDT4570 - Game Technology, Specialization Project

Table of Content

0.1 Introduction

0.1.1 Motivation

Unknown Horizons[1] is an open-source real-time strategy game developed by a team of programmers, artists, game designers and many more around the globe. The first revision was committed in late 2007[2].

As the project evolved the game's code architecture grew dynamically, without much planned structure or designed architecture. This resulted in a very tight coupling between the different components inside the game, making it difficult to add/change certain functionality in the game. This became clear when adding the boat builder building a while back, which resulted in months of fixing introduced bugs.

Unknown Horizons uses the outdated idea of making use of multiple inheritance to compose its in-game objects. Besides introducing very tight coupling between the different classes the current approach also does not allow non programmers to add new assets to the game. For an open-source project this is clearly not ideal, as user contributions would add great value to the project and save valuable programming time.

The idea for this project is to research how this problem is solved in similar open-source games and to transfer the results to the *Unknown Horizons* source-code.

[1] Unknown Horizons website: `http://www.unknown-horizons.org`

[2] First commit to *Unknown Horizons*: `https://github.com/unknown-horizons/unknown-horizons/commit/53eec12fd8bb52ac1a6ccfdb097296c479499dfd`

The following games have been chosen to be researched:

- *Battle for Wesnoth*[3]
- *0 A.D.*[4]
- *Mega Glest*[5]

0.1.2 Problem Statement

Three main questions should be answered by this project:

- Which architecture do open-source games similar to *Unknown Horizons* use to model their in-game objects?
- Can users add objects without modifying the game's code and if yes – how?
- Can the *Unknown Horizons* code-base be ported to a component-based architecture?

0.1.3 Project Context

This project is conducted for the course *TDT4570 - Game Technology Specialization Project*[6] which is part of **NTNU**'s computer science master program.

0.2 Research Methods and Questions

In this section we present our research questions and methods used in this work.

0.2.1 Research Questions

We work on a set of four main research questions:

- RQ1: Which architecture is used to describe objects in-game?
- RQ2: How are new objects added to the game?
- RQ3: Can existing objects easily be modified?
- RQ4: Are tools available to help with adding/modifying objects?

[3]Battle of Wesnoth website: `http://www.wesnoth.org`
[4]0 A.D. website: `http://wildfiregames.com/0ad/`
[5]Glest website: `http://megaglest.org/`
[6]TDT4570 Project description: `http://www.idi.ntnu.no/emner/tdt4570/`

0.2.2 RQ1

Which architecture is used to describe objects in-game?

The goal of this question is to find out if the game uses an inheritance based approach, a component based approach or some other design to describe objects in game.

0.2.3 RQ2

How are new objects added to the game?

With this question we want to find out if many changes have to be made to the code to add new objects. We also want to know if the objects are data or code driven. If they are data driven, we research which technology is used. Our goal is to find the easiest method of adding objects to the game.

0.2.4 RQ3

Can existing objects easily be modified?

Our goal is to assess whether existing objects are easily alterable or code has to be changed to modify them. We research what the possible implications of changing objects are.

0.2.5 RQ4

Are tools available to help with adding/modifying objects?

As creating game content is usually done by non-programmers, we want to assess how easy it is for them to add content to the game and if there are tools to support them in the process.

0.2.6 Research method

The first part of this work is four case studies in which we research four existing open source games. All games are mainly real-time strategy games, so their implementations face similar problems and are comparable to some degree. We will then use the gained knowledge to improve the handling of objects in *Unknown Horizons* by designing and implementing a system combining the best practices we found in the case studies.

0.3 State-Of-The-Art

In this section related work and literature on the topic is discussed.

0.3.1 Related Work

While there is some literature available on component-based architectures, I am not aware of works that researched moving from an inheritance based approach to a component based approach on the same product. I think this is the first work describing the actual

process. I am also not aware of any case studies of open-source real-time strategy game architectures and object representation.

0.3.2 Literature

A small number of papers and books on the topic is available:

1. A Generic Framework for Game Development - Fh et al. [2002]
2. A Software Architecture for Games - Doherty [2003]
3. A flexible and expandable architecture for computer games - Plummer [2004]
4. Game Architecture and Design: A New Edition - Rollings and Morris [2003]

Most literature titled component-based embraces the use of components in a high level view. They define a component as a library of classes that fulfilles a certain functionality (such as physics, AI or rendering), whereas I am interested in the low level component design, where a component maps to a single class of functionality.

Rollings and Morris [2003] seperates between a hard and soft architecture. The hard architecture is the part of the architecture that is platfrom or domain specific, whereas the soft architecture is usually mostly the game-logic itself. In Folmer [2007] Folmer gives a "Reference Architecture" for games, which can be seen in Figure 1. The soft archticture described in Rollings and Morris [2003] is only the *Game Interface* and *Domain Specific* layers in this reference architecture, the rest is considered hard architecture. I am interested in the components of the *Game Interface* layer of the of soft-architecture, whereas most research is done on the broad topic of the hard architecture or the *Domain Specific* layer.

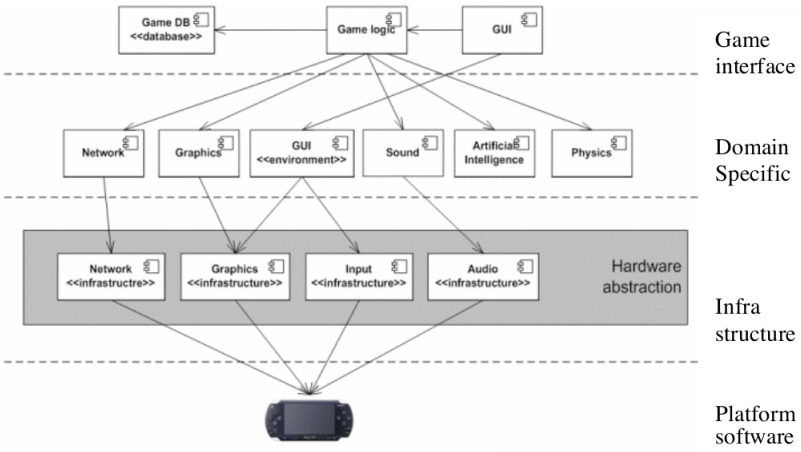

Figure 1: The reference architecture for games as given by Folmer – graphic taken from the paper.

Rollings and Morris [2003] emphasizes the need for well thought-out interfaces between the components. So that for example a route finding library/class can easily be exchanged for another one. Thus avoiding tight coupling between the high level components where possible. He does not go into any deep details concerning components on the soft architecture side, but he gives many ideas for design patters that can be used in game programming, such as the *Singleton*, *Factory* or *Delegate* patters.

I found Fh et al. [2002] particularly interesting, as it describes a system based only on components in great detail. Haller not only explains the use of components, but also gives a very detailed proposition on how to handle inter-component communication using a message system, based on and extending the QT GUI Framework[7].

Haller explains that components can be connected to each other using strictly typed in and outputs, making it possible to connect components using an external (graphical) tool, easily usable by non-programmers. Having well defined in and outputs allows the creation of component networks where as set of components are connected with each other over their in and outputs. These networks can be looked at as a "meta-component" as the network has well defined in and outputs and can thus itself be viewed as a component. Each component has a state in which it is in, when this state is changed a message is sent through the outputs. The basic interface is given in Figure 2.

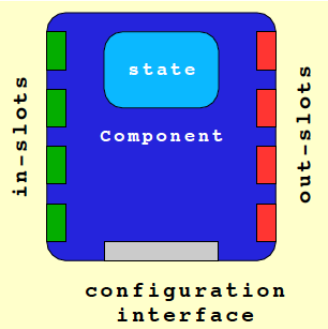

Figure 2: The component interface as described by Haller Fh et al. [2002] – graphic taken from the paper.

Doherty [2003] gives recommendations on how to structure a complete game-engine. He discusses the options of using a component-based approach or a object-oriented design. He claimes one important advantage of using a component-based approach is:

> It enforces a data-driven design philosophy. Since objects cannot be anything but data, all behavior must be defined in terms of data. – [Doherty, 2003, Page 4]

He argues this should be a favorable goal in the programming of a game, as it seperates the data from the code. By doing this it ensures that game-designes can work in parallel

[7]QT website: `http://qt.nokia.com/`

with the programmers, but no recompiling of the software is needed to change basic game attributes. Thus the teams productivity is improved.

He also brings arguments that favor the object-oriented system:

> The internal representation matches the way that people think about the world. – [Doherty, 2003, Page 4]

From reading his paper, I think he clearly favors the component-based approach. While acknoledging that it is more difficult for the programmer to implement, it gives great benefitis for the game-designer. He claims that many problems occuring with component-based systems can be solved using database proven concepts.

0.4 Own Contribution

In this section I present four different case studies to answer our research questions. I begin with presenting *Unknown Horizons* as it is the project which I focus my efforts of improvement on. It is followed by *Battle for Wesnoth*, *Mega Glest* and *0 A.D.*. The results are then evaluated and transferred to *Unknown Horizons*.

0.4.1 Unknown Horizons

Unknown Horizons as described on the project website:

> *"Unknown Horizons is a 2D realtime strategy simulation with an emphasis on economy and city building. Expand your small settlement to a strong and wealthy colony, collect taxes and supply your inhabitants with valuable goods. Increase your power with a well balanced economy and with strategic trade and diplomacy."* – from `http://www.unknown-horizons.org`

RQ1

Unknown Horizons uses a largely inheritance based approach to describe in-game objects. As the game is programmed using the Python[8] programming language it is possible to use multiple inheritance. The project makes great use of this ability, resulting in large

[8]Python website: `http://www.python.org`

inheritance trees. To illustrate this we have generated an inheritance diagram for the *Settler* class in Figure 3. The tree consists of 16 classes including many cases of multiple inheritance.

Experience in working on this project has shown that making changes to any of the classes included in this tree is often a very big task and comes with a great risk of introducing bugs into the code. It is also very difficult or even impossible to write unit tests for these classes, as they are so dependent on each other and the game core, that it is almost impossible to create the needed environment synthetically.

Settler Explained The *Settler* class is comprised of 4 basic classes: *BasicBuilding*, *SelectableBuilding*, *BuildableSingle* and *CollectingProducerBuilding*. This is how most buildings in *Unknown Horizons* are constructed.

BasicBuilding is a base class for every building, it loads graphics and provides basic information like the name, position, owner and functionality for running costs and level upgrades.

SelectableBuilding is a decorating class, that implements functions for selecting the building in-game. It manages showing in-game menus and outlines. If a building is not supposed to be selectable, this class should not be inherited.

BuildableSingle is a decorating class which is used when building new buildings. It tells the game that it can only be built as single instance, so there is no building of multiple instances at once. For this purpose the code provides the *BuildableLine*, *BuildableRect*, etc. classes which can be used if needed.

CollectingProducerBuilding is a collective class to make the Settler have collecting units which pick up resources for usage and then produce something out of them. This is easier to demonstrate on a *Lumberjack* for example, he picks up trees and produces planks from it. The Settler consumes resources (food, textiles, etc.) and in turn produces the abstract resource happiness.

Data-driven? *Unknown Horizons* uses a SQLite[9] database to save parts of the object's attributes. For example the size, health and name are saved in the database. This is necessary to make the higher level classes in the architecture reusable for sub-classes. All buildings have a size, but it may be different from building type to building type. It is saved to an external file to make it easily editable by non-programmers.

In summary I can say that the objects are partly data-driven, but usually it is not possible to add new buildings without writing new code.

RQ2

In order to add a new building to *Unknown Horizons* one has to look at the characteristics the building should have and then find the appropriate classes from the *Unknown Horizons* building classes collection. Those can then be combined to form new buildings.

[9]SQLite website: `http://www.sqlite.org/`

For example to create a settlement wall one could use the classes *BuildableLine* and *BasicBuilding*. This is a very simple example which does not need to inherit many classes, as its functionality is very limited. All attributes of this building can then be added in the database by using any SQLite database manager.

RQ3

Modifying existing in-game objects in *Unknown Horizons* can be easy and very difficult. This depends on the degree of change that is to be made. If only basic attributes like health, production time or similar are to be changed, then it can easily be done by someone who knows his way around the database. If however new functionality is required, for example a building which previously did not collect resources needs to collect resources, a change in the games code is most certainly required. If the desired functionality exists, this can either be an easy task by just adding another class to the hierarchy of the building or it can be very difficult if new functionality in the existing classes is required.

A good example for this is the boat builder, which is mainly a *CollectingBuilding* producing units instead of resources. The building has been implemented for over a year now and the team is still not certain if it works bug free or not, as it required huge modifications to the production classes to be able to produce units instead of resources.

RQ4

There are no tools available to help with the addition/editing of content at this point. A map editor is being worked on by two students of the *Technische Universität München* in Germany. It will be available in future versions, but it is not yet in a working state. Work is being done on the "*editor*" branch of the official *Unknown Horizons* git repository[10].

0.4.2 Battle for Wesnoth

Battle for Wesnoth as described on the project's website:

[10]Link to the editor branch: `https://github.com/unknown-horizons/unknown-horizons/tree/editor`

> *"The Battle for Wesnoth is a Free, turn-based tactical strategy game with a high fantasy theme, featuring both single-player, and online/hotseat multiplayer combat. Fight a desperate battle to reclaim the throne of Wesnoth, or take hand in any number of other adventures..."* – from `http://www.wesnoth.org`

RQ1

Battle for Wesnoth comes with its own markup language, *WesnothMarkupLanguage* - *WML*, to describe units, campaigns, AIs, missions, maps, sounds, etc. *WML* is similar to other markup languages like *XML*, but more human readable and provides some basic functions to create logic - like basic if-clauses and variables.

Entities are described in a component like way, but not all tags are components. For most tags like skills, attacks and races components are used in code. Other attributes such as *[portrait]* are not mapped to their own component, but are just read as data for the basic unit class.

Battle for Wesnoth parses all the tags in a config file into the config class where the single tags can be easily accessed by code. The actual use of the tags is left to the code using the config. Classes like unit and race read values from the config class.

To ease development it is also possible to access many internals using a LUA API. This is used for reading in tags for maps and units. According to the development team the LUA API can be used for modding as well, this behavior is not documented anywhere though [11].

Data-driven? As all game content is described using *WML*, *Battle for Wesnoth* can be seen as completely data-driven. Wesnoth contains an in-game option to download other mods, containing new units, campaigns, etc. Within the concept of a round based strategy game the engine is completely independent of the content.

RQ2

New objects are added to the game by adding a new file containing basic *WML* with the description of the unit. Since the game is fully moddable all units, maps and campaigns can be replaced just by editing and adding new files into the basic directory structure Community [2011].

In order to add new *WML* attributes, the c++ source-code has to be changed to recognize them. The *Battle for Wesnoth* parser does not know anything about the tags it loads, therefore the only constraint is that the new tag should be used/accessed in another c++ class, like the unit class, or LUA helper code.

Listing 1: A basic (shortened) *Battle for Wesnoth* unit definition in WML

```
1 [unit_type]
```

[11] IRC logs regarding LUA API: `http://www.wesnoth.org/irclogs/2011/12/%23wesnoth-dev.2011-12-08.log`

```
2     id=Elvish Lady
3     name= _ "female^Elvish␣Lady"
4     gender=female
5     race=elf
6     image="units/elves-wood/lady.png"
7     profile="portraits/elves/lady.png"
8     {MAGENTA_IS_THE_TEAM_COLOR}
9     hitpoints=41
10    movement_type=woodland
11    movement=6
12    experience=150
13    level=3
14    alignment=neutral
15    advances_to=null
16    {AMLA_DEFAULT}
17    cost=10
18    usage=null
19    description= _ "Elves␣choose␣their␣leaders␣for␣their␣wisdom␣and␣sensitivity␣to␣
          the␣balance␣of␣universal␣forces;␣foresight␣is␣what␣has␣protected␣them␣in␣
          times␣of␣uncertainty.␣Their␣just␣reign␣is␣rewarded␣by␣the␣unflagging␣fealty
          ␣of␣their␣people,␣which␣is␣the␣greatest␣gift␣for␣which␣any␣ruler␣could␣ask.
          "
20    [portrait]
21        size=400
22        side="right"
23        mirror="true"
24        image="portraits/elves/transparent/lady.png"
25    [/portrait]
26 [/unit_type]
```

RQ3

To edit a unit in *Battle for Wesnoth* the *WML* files have to be changed, nothing else has to be done. In order to change the units behavior the c++ code has to be edited.

RQ4

Battle for Wesnoth comes with a set of tools to help developers and content creators. Wesnoth comes with a map editor, tools to validate user created *WML* and provides an *Eclipse*[12] plug-in, called *"The Battle for Wesnoth UMC Development IDE"*[13].

It can setup campaigns, races and more for the user, provides syntax highlighting for the WML and also some means of auto-completion. It can also launch the map editor and game with the specified campaigns and maps the user is working on and provides means of using the WML validation tools. A screenshot of the WML editor is provided in Figure 4.

[12]Eclipse Project Homepage: `http://www.eclipse.org`
[13]UMC Plugin Website: `http://eclipse.wesnoth.org/`

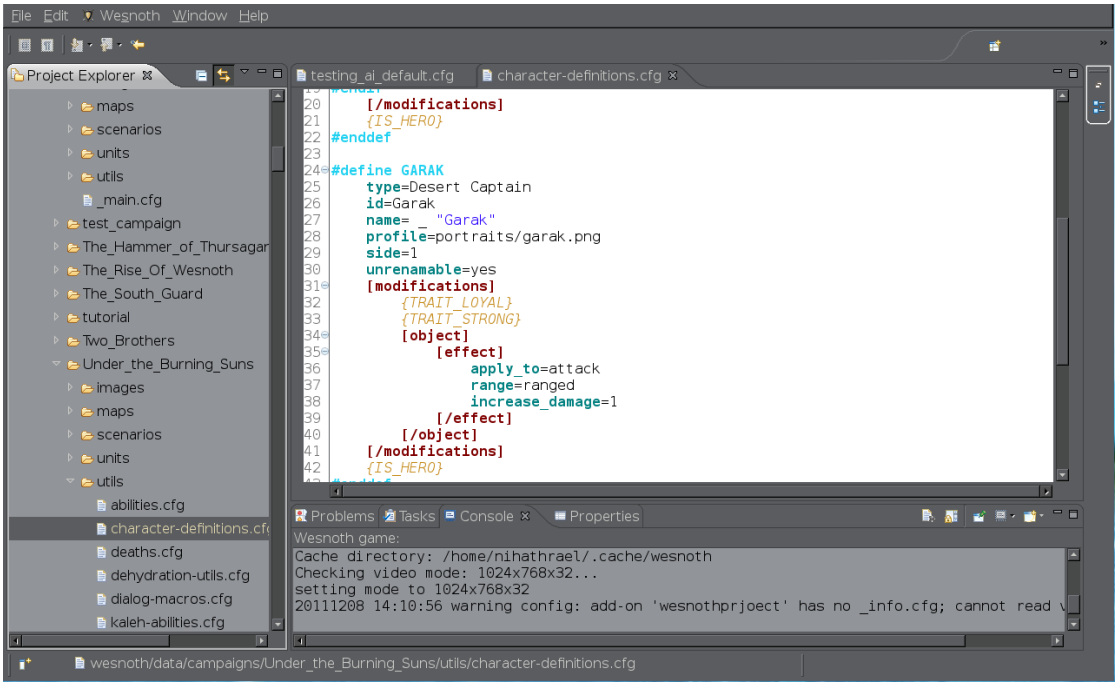

Figure 4: Wesnoth UMC Plugin WML editor

0.4.3 Mega Glest

Mega Glest as described on the project's website:

> *"MegaGlest is a free and open source 3D real-time strategy (RTS) game, where you control the armies of one of seven different factions: Tech, Magic, Egyptians, Indians, Norsemen, Persian or Romans. The game is setup in one of 16 naturally looking settings, which -like the unit models- are crafted with great appreciation for detail. Additional game data can be downloaded from within the game at no cost."* – from `http://www.megaglest.org`

RQ1

Mega Glest uses a mixture of inheritance and component-based object description. Basic things are set using inheritance, for example the *UnitType* class inherits from the

ProducibleType class, as every unit in the game is producible. More advanced things are added to the unit as components, for example the *UnitType* has *Level*,*SkillType*, *Resource*, *CommandType* and *UnitParticleSystemType* components. Units themselves are part of a bigger component hierarchy: Units are part of a *FactionType*, which is part of a *TechTree*. See Figure 5 for a detailed structure analysis. A class ending in **Type* is used to represent prototypes for the actual instance classes. For example the *UnitType* class loads all necessary data from the XML definitions. In-game a *Unit* instance is used, which itself contains a *UnitType* as information base.

Data-driven? *Mega Glest* is fully data-driven. All information needed for in-game objects, scenarios and campaigns is stored in XML files. This makes *Mega Glest* more of a game engine with a focus on real-time strategy than only a game. Several[14] mods exist, proving that it is indeed possible to create new games using *Mega Glest* as a game engine.

0.4.4 RQ2

In order to add new objects to *Mega Glest*, new XML files have to be created. It can contain the definition of a unit, campaign, tech tree or similar. See Listing 2 for a shortened example of a basic unit definition. Every class has many parameters which allow the user to specify the details for every unit.

Listing 2: A basic *Mega Glest* (shortened) unit definition in XML

```
<unit>
  <parameters>
    <size value="1"/>
    <height value="2"/>
    <max-hp value="450" regeneration="5"/>
    <max-ep value="3000" regeneration="30"/>
    <armor value="15"/>
    <armor-type value="leather"/>
    <sight value="12"/>
    <time value="200"/>
    <multi-selection value="true"/>
    <cellmap value="false"/>
    <levels>
      <level name="expert" kills="5"/>
      <level name="master" kills="15"/>
      <level name="legendary" kills="30"/>
    </levels>
</unit>
```

RQ3

In *Mega Glest* editing objects is easy: Simply change the XML files to match the new requirements. Changing things in the source code seems nicely doable as there are not too many inheritance based objects, instead composition is preferred.

[14] List of some mods: `http://www.moddb.com/games/megaglest/mods`

RQ4

Mega Glest comes with a map editor(Figure 6) to help with creating new maps to use in-game. It allows editing every possible detail of the map and is thus of great help for content creators.

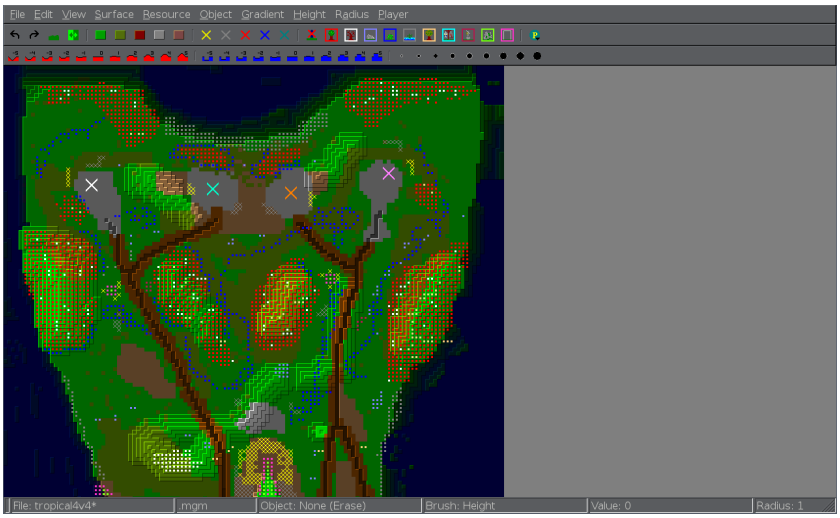

Figure 6: *Mega Glest* map editor

Mega Glest also comes with a model viewer, with which the custom *g3d* 3D model file format can be opened. It allows to view basic models and particle effects.

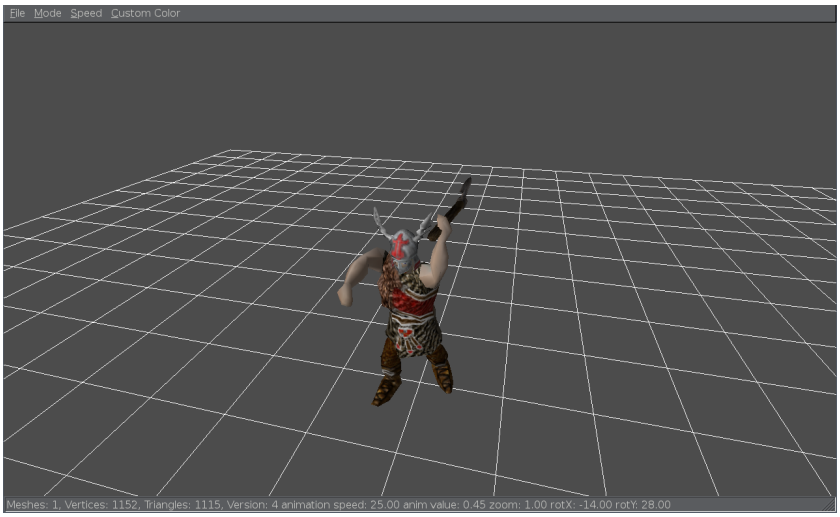

Figure 7: *Mega Glest* model and particle viewer

0.4.5 0 A.D.

0 A.D. as described on the project's website:

> *"0 A.D. (pronounced "zero ey-dee") is a free, open-source, cross-platform real-time strategy (RTS) game of ancient warfare. In short, it is a historically-based war/economy game that allows players to relive or rewrite the history of Western civilizations, focusing on the years between 500 B.C. and 500 A.D. The project is highly ambitious, involving state-of-the-art 3D graphics, detailed artwork, sound, and a flexible and powerful custom-built game engine."*
> – from `http://wildfiregames.com/0ad/`

0.4.6 RQ1

0 A.D. uses a completely component based approach to describe entities in the game. Units are described using a simple XML based format, making up single units of many components. The engine provides components like *Attack*, *Cost*, *Position* or *VisualActor*. See 0A.D. [2011a, Entity Component Documenation] for a complete list and details.

It is possible to implement components in c++ and JavaScript. The general idea is to use JavaScript where possible and only use c++ if necessary for performance or communication with the game engine, for instance for rendering. The components communicate with each other using a message based system or calling methods on other components directly. The first approach is to be preferred, as the implementations remain separated from each other this way. A component can send messages directly to a specific component type, or broadcast it to everything listening to the specific message. A entity in the game is basically a number associated with a set of components. There is no inheritance involved in creating an entity, other than inside a single component.

The engine allows to *hot-load* components implemented in JavaScript, meaning changes in the JavaScript files will be detected while the game is running and loaded into the engine. This enables the developer to make changes to the components behavior while the game is running and directly seeing his results in-game.

Data-driven? *0 A.D.* is completely data-driven. All information needed to construct a unit is saved in the basic XML format. The XML format is specified by the components directly, making it easy to extend the markup language by new components. As the game logic is implemented using these external components, it is easy to exchange the complete gameplay logic, thus enabling very wide modding support and separating the game-play logic from the main engine.

0.4.7 RQ2

New objects can easily be added to the game by adding new XML unit entity definitions. No extra source-code is necessary. A shortened sample unit definition is given in Listing 3.

Listing 3: A basic *0 A.D.* (shortened) unit definition in XML

```
<Entity parent="units/cart_cavalry_spearman_b">
  <Attack>
    <Melee>
      <Hack>6.0</Hack>
      <Pierce>16.0</Pierce>
    </Melee>
    <Charge>
      <Hack>18.0</Hack>
      <Pierce>48.0</Pierce>
    </Charge>
  </Attack>
  <Health>
    <Max>140</Max>
  </Health>
  <VisualActor>
    <Actor>units/carthaginians/cavalry_spearman_a.xml</Actor>
  </VisualActor>
</Entity>
```

0.4.8 RQ3

All units are defined using basic XML based definitions, making it easy to edit them with a normal text editor. No changes to the code are necessary to edit any unit.

0.4.9 RQ4

0 A.D. comes with an editor called *Atlas*. It is a map and scenario editor and is not meant for editing unit definitions. The editor is very similar to *Mega Glest*'s editor in terms of features. It allows the user to edit any detail of a map and comes with a unit viewer, that previews animations. A detailed user-manual is provided on the project's developer pages 0A.D. [2011b].

Tool support for editing unit entity descriptions is not provided.

0.4.10 Evaluation

All four games provide data-driven approaches in varying degree.

Unknown Horizons provides an SQLite driven approach, which usually requires editing some code to add new entities to the game. The code is made up of huge inheritance trees, making it difficult to change the code or add new features. Storing the data in an SQLite database makes it more difficult for contributors to change values, as they first have to learn how to use an SQLite browser and understand how databases work. The plus side for this is, that relational queries are very fast.

Battle for Wesnoth and *Mega Glest* allow the addition of units using basic markup languages like *WML* and *XML*. *Battle for Wesnoth* provides basic means for scripting events and behavior using the *WML* and possibly the LUA extension. Both games use basic classes to which the data stored in the entity files is mapped, changing fundamental things about the way the game behaves requires work on the code. Both games have separated the concerns by using a component similar approach, making editing the source code easier than in *Unknown Horizons*.

0 A.D. is the only game in this study that uses a purely component based system. An entity in the game is represented by a set of components tied to an ID, there are no base classes for units or buildings. They are entirely made up of components. The engine allows scripting the components in JavaScript to ease development, but provides the possibility to easily port components to c++, should speed be a problem. Components written in JavaScript can be hot-loaded during the run-time of the game, making it easy to debug and work on single parts of the game and directly seeing the impact on the game. The implementation of *0 A.D.* is very similar to the system described in Fh et al. [2002], but it does not use the enhanced component in and output system. Instead it uses the QT like described version of messages between components.

0.5 Transferring the Results to *Unknown Horizons*

Looking at the ease of content creation of *Battle for Wesnoth*, *Mega Glest* and *0 A.D.* it becomes clear that *Unknown Horizons* is not nearly as flexible and easy to use as the other projects. While *0 A.D.* has the cleanest component based architecture, it is not possible to transfer the design completely to *Unknown Horizons*.

As the current code is not a component driven design, it is very difficult to convert the entire code structure to a component based architecture in one big refactoring. Therefore a middle ground has to be found here, where the code can be refactored and changed to components in small parts. The difference between c++ and JavaScript components is not suitable for *Unknown Horizons* as it is written entirely in Python, so there is no possibility to differentiate between compiled and scripted components.

An implementation similar to *Battle for Wesnoth* or *Mega Glest* seems useful, keeping in mind the specific requirements for *Unknown Horizons*.

0.6 Requirements, Design & Implementation

In this section I discuss the requirements and design of the new component system introduced into *Unknown Horizons* and give details about the new implementation.

0.6.1 Requirements

I will first define the *must-have* requirements, which need to be achieved to make the project a success.

As the code-base of *Unknown Horizons* is already quite big and evolved it is not possible to create a design that requires a complete rewrite of the code or which requires all the implementation work being done in one step. Therefore I aim at creating a design that can be implemented in smaller steps and is compatible with the current code design.

I also aim at making it easy for content-creators to contribute new objects to the game and make it easy to edit current object descriptions. The lack of an easily editable format can be considered of *Unknown Horizons*'s major weaknesses compared to other open-source games and has to be corrected. Therefore it is important to move away from the SQLite based data-storage to a more humana readable and easily editable file based format. This file based format should be easy to read and edit without special knowledge of computer programming.

It should be possible to add new components to entities by editing these files, changing the code must not be necessary.

An additional *nice-to-have* requirement is that we do not add another dependency to the project, as in adding a new library for the file format for example. This is not mandatory though and can be dropped if necessary. The usability of the code and file format has a higher priority.

Gathered Requirements

1. Porting the old code to the new structure should be possible in small steps
2. Move object data from SQLite to a file based format
3. The file format should be easy to understand by non-programmers
4. Add/remove components by editing the files, not code
5. No extra dependency (nice-to-have)

0.6.2 Design

To ensure compatibility with the old inheritance based code, I will use a *ComponentHolder* class, which manages all the components an object has. This class can be included into any inheritance tree and thereby extending the current code with the possibility of containing components. By using this approach I can slowly extract single classes from the code and move them into components. A UML showing the planned implementation is shown in Figure 8.

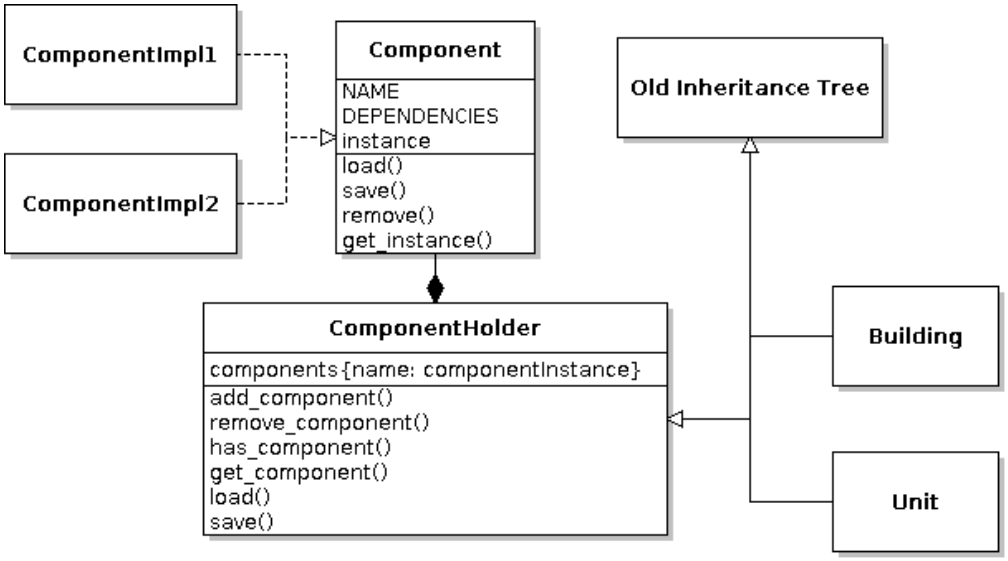

Figure 8: The code design for the *ComponentHolder* and *Component* classes

A component should be a class to handle a specific task only and should be as independent as possible from other components. Clearly a component can depend on other components being present, for example if I decide to add a component that manages the production process, it will most likely depend on a component which manages storage of items for this entity. These dependencies should be checked when the object is constructed to warn the content creator if there are errors in this regard. Each component has to be able to save and load its state without depending on any work being done by other components, to ensure they are loosely coupled and easy to test.

0.6.3 Implementation

In this section I will discuss the details of the implementation.

Data Format

Unknown Horizons already uses YAML[15] to describe scenarios and campaigns.

> *YAML is a human friendly data serialization standard for all programming languages.* – `http://www.yaml.org`

YAML has many similarities with Python, as it depends on the indention to be correct in order for the parser to work correctly and uses a similar syntax for lists and dictionaries. This has already proven useful for the *Unknown Horizons* code-base in my experience as all code is well formatted and should result in nicely formatted object files.

[15]YAML website: `http://www.YAML.org/`

The reason to choose YAML at the time was, that it is very easy to read and edit by humans. This comes at the cost of being a little slower when parsing. As most data can be cached and has to be loaded from disc only once, this is not a major concern.

I decided to use YAML for the object description files as well, the requirements match and it avoids adding another dependency to the project. The result of converting most of the data in the database to a YAML based file for each object looks similar as Listing 4. It contains all basic information needed for every building and a list of components that are used by this specific building. As the conversion from an inheritance based approach to a component based approach is only done in small steps, we have to specify the original base class for every object using the *baseclass* attribute.

Listing 4: A basic (shortened) building definition in YAML for *Unknown Horizons*

```
id: 24
name: Brickyard
baseclass: production.Refiner
radius: 8
cost: 15
cost_inactive: 5
size_x: 2
size_y: 4
inhabitants_start: 1
inhabitants_max: 1
button_name: brickyard-1
tooltip_text: Turns clay into bricks.
settler_level: 1
buildingcosts: {1: 500, 4: 6, 6: 1}
components:
- HealthComponent: {maxhealth: 1000}
- ProducerComponent:
    productionlines:
      33:
        produces:
        - [7, 1]
        consumes:
        - [21, -1]
        time: 15
- StorageComponent:
    inventory:
      SlotsStorage:
        slot_sizes: {21: 4, 7: 10}
actionsets:
  as_brickyard0: {level: 0}
```

Loading and Caching As loading YAML files is too slow to be repeated for every object creation in game, the data has to be cached after reading it once. We use the abilities of Python to create a special type instance for every object. This type instance can than be instantiated to a "normal" python object instance. It can be thought of as creating classes on the fly. We create a lumberjack type for example, so for every new lumberjack in the game we can create a new object instance from this type. Since the data from the YAML file is only read once during type creation, we efficiently cache all the data in the type instance for later use.

Code Layout

To achieve easy compatibility with the old inheritance based approach, a *ComponentHolder* class has been introduced, that manages a set of components. This class can be included into the normal hierarchy of any in-game object.

ComponentHolder The *ComponentHolder* has a very basic interface:

- initialize()
- add_component()
- has_component()
- get_component()
- remove_component()
- save()/load()/remove()

The add/has/get_component() methods are pretty clear, they add *Component*s to the *ComponentHolder*, check if they are available for this *ComponentHolder* or return a specified *Component*.

It is note-worthy that the *ComponentHolder* contains an extra *initialize()* method, which is separate from the normal constructor. This method has to be called after instance creation on any class that inherits from the *ComponentHolder* class. This is a work-around for the problem that certain attributes, such as the game-engine's visual instance, needed in the components are only ready after the entire constructor hierarchy has been executed to the top. This will hopefully be repaired after everything has been moved to components, but for the moment this is a necessary evil.

Component To represent a component the *Component* class has been created. Each component should take care of a certain set of functionality, which should be as independent as possible from any other code. Of course this is not always achievable, as it soon becomes clear that special components will rely on the existence of other components. For example a production component might rely on the presence of a storage component, with which it can work.

Each *Component* has to implement the basic interface:

- initialize()
- load()/save()
- remove()
- get_instance() - classmethod

As with the *ComponentHolder* the *Component* has to implement an *initialize()* method, in which all the setup should be done. This method is called automatically by the *ComponentHolder*, so no special care has to be taken here.

The *load()*/*save()* methods are called by the *ComponentHolder* and should implement loading and saving the components state into the given database. This should be done in a way, that the component does not relay on any other part of the code to correctly restore its state on loading.

The *get_ instance()* method is a class method, its purpose is to return an instance of the component, given data loaded from the YAML file. For basic components the basic implementation is likely to suffice, it will pass in the dict of data loaded from YAML as arguments to the class. For more sophisticated components this method might need to be re-implemented. An example of this is the *StorageComponent*.

Each component has to set the class's *NAME* variable to unique string. Components that inherit from other components can usually keep the name of the inherited class, as they are not used in the same *ComponentHolder* instance. Keeping the name the same, also ensures that if they implement similar functionality it is possible to call *get_ component()* with the parent class, and still receiving the correct implementation. This is used for example for names. A *NamedComponent* has been implemented. It handles giving names to objects. There are some special implementations of this, for example the *ShipNameComponent*, which uses special names designated for ships. If a *ShipNameComponent* component is present in the *ComponentHolder* class and get_componentNamedComponent is called, the *ShipNameComponent* is returned, as they both use the same *NAME* variable.

Dependencies *Components* can depend on other components to exist. For this every *Component* class can use the *DEPENDENCIES* class variable. It is a list of *Component* classes. The classes specified will be initialized before the class that lists them as dependencies.

The basic dependency resolution is done by implementing the *__lt__()* (less than) method on the Component, so that a list of *Components* can be sorted by dependency order. This is of course a very basic form of dependency resolution, which can't detect circular dependencies, etc. It should be good enough for this purpose and is very simple and fast, as it uses python built-in *sorted()* method.

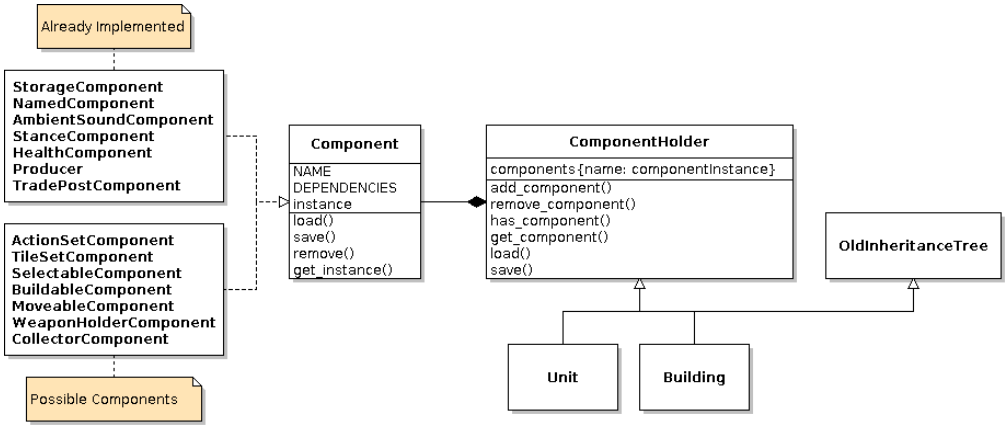

Figure 9: UML diagram for the *ComponentHolder* and *Component* classes which were implemented

Implementation Overview Figure 9 shows the basic layout of the implementation in a simple UML diagram. When compared to the first settler class diagram in Figure 3 one can see that classes like *Producer* have been pulled out of the inheritance tree and are now used by composition with the *ComponentHolder* instead.

0.6.4 Added Components

Since the scope of the project is not big enough turn the *Unknown Horizons* code-base into a completely component driven architecture, only parts of the classes have been converted to *Component*s yet. The following *Component* implementations have been made:

- AmbientSoundComponent
- HealthComponent
- NamedComponent + Sub-classes
- StanceComponent
- StorageComponent
- TradePostComponent
- Producer

AmbientSoundComponent The *AmbientSoundComponent* takes care of playing sounds in-game. It can position sounds at the current position of an entity and saves a list of all the sounds an entity can play. The goal of this component is to collect all sound handling code in one place. Since we do not have a lot of this code, this component was a good start together with the *NamedComponent*.

HealthComponent Together with the *StanceComponent* this was one of the components that had been created during the project's participation in *Google Summer of Code 2011*. This was the first try of working on a component based system. I did not make major changes to this component, other than fitting it into the new interface.

Its purpose is to provide a health counter for objects and handle everything that is connected with this, like drawing life-bars.

NamedComponent + Sub-classes The *NamedComponent* provides unique names for an object throughout one game. For this class several sub classes exist, that provide different names. Namely these are:

- *ShipNameComponent*
- *PirateShipNameComponent*
- *SettlementNameComponent*

StanceComponent The *StanceComponent* is used to set the combat stance of the object. It can be aggressive, defensive or neutral. This Component was introduced during the *Google Summer of Code 2011* as part of the combat system implementation.

StorageComponent The *StorageComponent* provides the entity with a sort of inventory where it can store different resources. This is a fairly complex component, as it can use different storage implementations internally. For this reason this component has to implement its own *get_instance()* method. An example markup for the *StorageComponent* using an inventory that can handle a fixed set of slots with specific sizes is given in Figure 5.

Listing 5: YAML representation of the StorageComponent using a SlotStorage

```
1 - StorageComponent:
2     inventory:
3       SlotsStorage:
4         slot_sizes: {28: 8, 5: 8}
```

TradePostComponent The *TradePostComponent* adds trading functionality to the entity. This component handles selling and buying resources from other players or the Free-Trader.

Producer This is most complex component in the game so far and I estimate it to stay this way, even if more components are added. It manages the production of goods in the game, but using production lines. Since this is a very complex system it is very bug-prone and extracting it into a single component and thereby removing it from the main hierarchy of buildings was not easy. The code depended heavily on other classes being in the inheritance tree and debugging this was very difficult and time-consuming.

The *Producer* component greatly eases the changing and adjusting of production lines for the content creator, as all information is now at one place. Using the old SQLite system, the content creator had to look at countless tables to be able to get an overview of even a single production line. This has greatly improved. Listing 6 provides a short example.

Listing 6: YAML representation of the Producer with two production lines

```
- ProducerComponent:
    productionlines:
      7:
        produces:
        - [10, 1]
        consumes:
        - [2, -1]
        time: 1
      47:
        produces:
        - [31, 1]
        consumes:
        - [30, -1]
        time: 1
```

0.6.5 Converting the SQLite data to YAML

Two methods were used when converting the SQLite data to YAML data:

- manual conversion
- converter script implemented in Python

The reason I used manual conversion is that *Unknown Horizons* contains only very few units, for which a conversion script would not have paid of in terms of total used time. I used the existing "print_db_data.py" to output relevant information to the console and copied data straight from the SQLite file to create the new unit YAML definitions.

As there are over 40 building in *Unknown Horizons* with many attributes, a different approach had to be taken. For this reason a helper script was implemented in Python, which automatically converted the entire list of buildings into YAML building definitions. This has the added benefit of avoiding syntax errors, as I let the YAML parser write the file instead of writing it by hand. It also allowed testing different formats for the data, to find the best suited format. The converter script is available in *development/convert_db_to_YAML.py*.

0.6.6 Testing

To ensure working code, three types of tests were used:

1. Unit Test
2. System Test
3. Smoke Test

Unit and System Tests *Unknown Horizons* has a set of unit and system tests. Unit tests are basic tests for single classes to check their functionality. System tests try to test the working of the entire game, instead of just a small function. It has a more general scope to discover bugs in game logic. After making the changes to the code, all tests have been repaired to work as expected. Also a few tests concerning the new component system have been added. Together with the community I am working on extending the tests to cover more code and to make sure my changes are stable before they are merged into the mainline of development.

Smoketests A smoke test describes a kind of test in which the goal is to run as much code of the system as possible and see if it fails Mcconnell [1996].

To emulate this, I used our artificial intelligence player. I started the game on 20 times the normal speed and let 3 AI players play for a while. This helped me find a lot of bugs, especially in the beginning. As python is not strictly typed, the interpreter discovers a lot of bugs, like missing attributes, only at run-time. Therefore it is crucial to actually run as much game code as possible.

0.7 Evaluation

In this section I will evaluate the project and the result.

0.7.1 Project

The project was executed in an autonomous way, without much interaction of the professor or other mentors. The focus was on mainly working with other people involved in *Unknown Horizons* or other open-source games discussed in this work. This way it has a direct impact on the work of *Unknown Horizons* and might have lead to some additional insights for members of other teams.

I believe the result matches the style of project execution, it had more emphasis on implementing the new system than production many pages of documentation. A lot of time has been invested to implement a stable system that can be used in the production mainline of development in *Unknown Horizons*.

The plans for the implementation were worked on and discussed with the *Unknown Horizons* team throughout the project and during the weekly meetings[16][17]. This allowed me to identify weaknesses in my design and ensure that I have the same ideas, of how it should look, as the rest of the team.

0.7.2 Results

This project provides two main results:

[16]Meeting logs: `http://meetings.christoph-egger.org/unknown-horizons/2011/unknown-horizons.2011-10-16-17.02.html`

[17]Meeting logs 2: `http://meetings.christoph-egger.org/unknown-horizons/2011/unknown-horizons.2011-12-04-17.04.html`

1. Case-study of four existing systems to describe in-game entities
2. Implementation of a new component system for *Unknown Horizons*

Case Study In the case study I have explained how the four project *Unknown Horizons*, *Battle for Wesnoth*, *Mega Glest* and *0 A.D.* represent their objects as in data format as well as in code. It provides insight for new projects on how a component based system can be designed and on the quality of code-architecture in open-source games.

While the last part has not been explicitly discussed, it is clear that some thought on how to design these systems is being put into the systems. Especially *0 A.D.* provides a very cleanly designed system, designed for ease of development and content creation with a very clear code structure. It provides a purely component based system, making it the best designed of the four discussed projects under this point of research.

Implementation The first lesson learned from this project is that refactoring the entire code of *Unknown Horizons* into a component based system is a lot of work, that if pursued further will require an enormous effort by the development team. Since it provides great benefits to the code-quality, as in less coupling and thus better testability, it is likely that more work in this area will be done.

Because of the amount of work this task requires, it was not possible to move the entire code to a component based system in the time of this project and only a few proof of concept components have been implemented.

The implementation allows for an easier representation of entities outside the code, making it very interesting for content creators, as they can now change the behavior of entities in the game in an easy way – by just using a simple text-editor. This has been enabled by using the YAML markup language to represent and save entity data. Simple behavioral changes can be made by adding components to an entity and/or changing their attributes.

The use of YAML for entity description does not only allow using components in an easy manner, but also the changing of data for entities, as this project involved moving most of the data from the main SQLite database into the YAML format. This again enables easier work for the content creator.

I my opinion YAML is a better choice than *XML*, because it is easier to read by humans and thus easier to handle for non-programmers. Because we already use YAML for scenarios, it also means not adding another dependency and not learning another markup language.

Settler Class To compare the old inheritance based class hierarchy, we compare the old UML tree given in Figure 3 with the new version given in Figure 10:

We removed the *AmbientSound*, *StorageHolder* and *Producer* classes from the hierarchy, these are now used as components. Not all created components are used in the *Settler* class, so not all work is visible in this UML tree. However it is clearly visible that the

inheritance tree can be reduced in size by using components – trimming it to a small size still requires considerable work, as stated previously.

In my opinion especially the refactoring of the *Producer* code into a *Component* class brings great benefits. The production code has been regarded as some of the most complex code in *Unknown Horizons*, it is now much easier to write tests for this part of the code, ensuring higher code quality on the long run.

0.7.3 Methods

Performing the research as a case-study seems to be a good idea, as there is hardly any dedicated literature available on the topic of game architectures, specifically the area of entity description. Comparing *Unknown Horizons* to other open-source projects rather than commercial products makes sense, as, judging from experience, the man-power available matches a lot better than on commercial projects. open-source projects usually don't have the time to create sophisticated tools, therefore it is crucial to rely on simple methods for content creation.

The implementation created in this project can serve as a solid foundation for further work on the area of a component based architecture. I believe that researching other projects first and then using the results gained from this study as foundation lead to a nice solution for the scope of the *Unknown Horizons* code base.

0.8 Conclusion and Future Work

In this section I present a conclusion of this project and the future work that can follow it.

0.8.1 Conclusion

I set of with the goal of introducing a component based architecture into the *Unknown Horizons* code-base. In order to complete the task properly, I first looked at the few existing literature on the topic and conducted a case study on the four open-source games *Unknown Horizons* (to see where I am at), *Battle for Wesnoth*, *Mega Glest* and *0 A.D.*.

I found out that *Unknown Horizons* does not have a component based system in-place yet, only a small start in terms of a *HealthComponent* and a *StanceComponent* was made during the last *Google Summer of Code*. I examined an example class hierarchy to understand that the hierarchy is big with 16 involved classes and multiple inheritance.

Battle for Wesnoth and *Mega Glest* provide a semi-component based system. Objects are described using an extensible markup language (WML and XML), making it easy for content contributors to add value to the game. Code wise both games don't follow a pure component based architecture, but also use some form of inheritance.

0 A.D. uses a purely component based architecture, representing an entity in code as an unique identifier to which a set of components is mapped. It provides a powerful

architecture which allows the developer to implement components in c++ or LUA, depending on the need for speed or simplicity. It also allows the exchange of all the LUA components in the form of a mod, making the core c++ code more a game engine than base code for *0 A.D.* specifically.

Based on this knowledge I designed a component system for *Unknown Horizons* that can gradually be introduced into the existing code-base. For this reason it is not as pure as the approach used in *0 A.D.*, but it provides a lot more flexibility in porting the existing code to the new system in small steps.

While implementing some basic components I quickly realized that porting the entire code base to such a new system in one step would be impossible. This work presents a solid foundation for further work on this topic, which I am planning to do in the next months. It will help *Unknown Horizons* become more content contributor and game-design friendly as it allows editing units and buildings by editing simple to read and edit YAML files instead of editing SQLite databases.

0.8.2 Future Work

On the research level it would be interesting to compare commercial games to the open-source systems described in this report. It could provide a lot of insight of how far industry standards in gaming have made their way into the open-source gaming world.

There is a lot of work remaining to finish the conversion from an inheritance based approach to the new component based architecture. Many components have to be created, thus minimizing the hierarchy. A lot of time should be allocated for this task, as even creating one component can take days of debugging and fixing to make the code run again. Because of the tight coupling of the current code-base and the dynamic nature of Python refactoring is not easy and requires thorough testing afterwords.

In order to ensure easy testing of newly created components the current collection of unit and system-tests should be extended, to cover greater areas of the code. Especially system tests help with the refactoring as they test the coordination between the components, which is likely to break when refactoring the code.

Currently the code uses a changelistener approach for class communication and to signal events. It could be an interesting project to redesign this system and port it to a message based system, as described in Fh et al. [2002]. This could provide even looser coupling, making the component system more effective at modularizing the code-base.

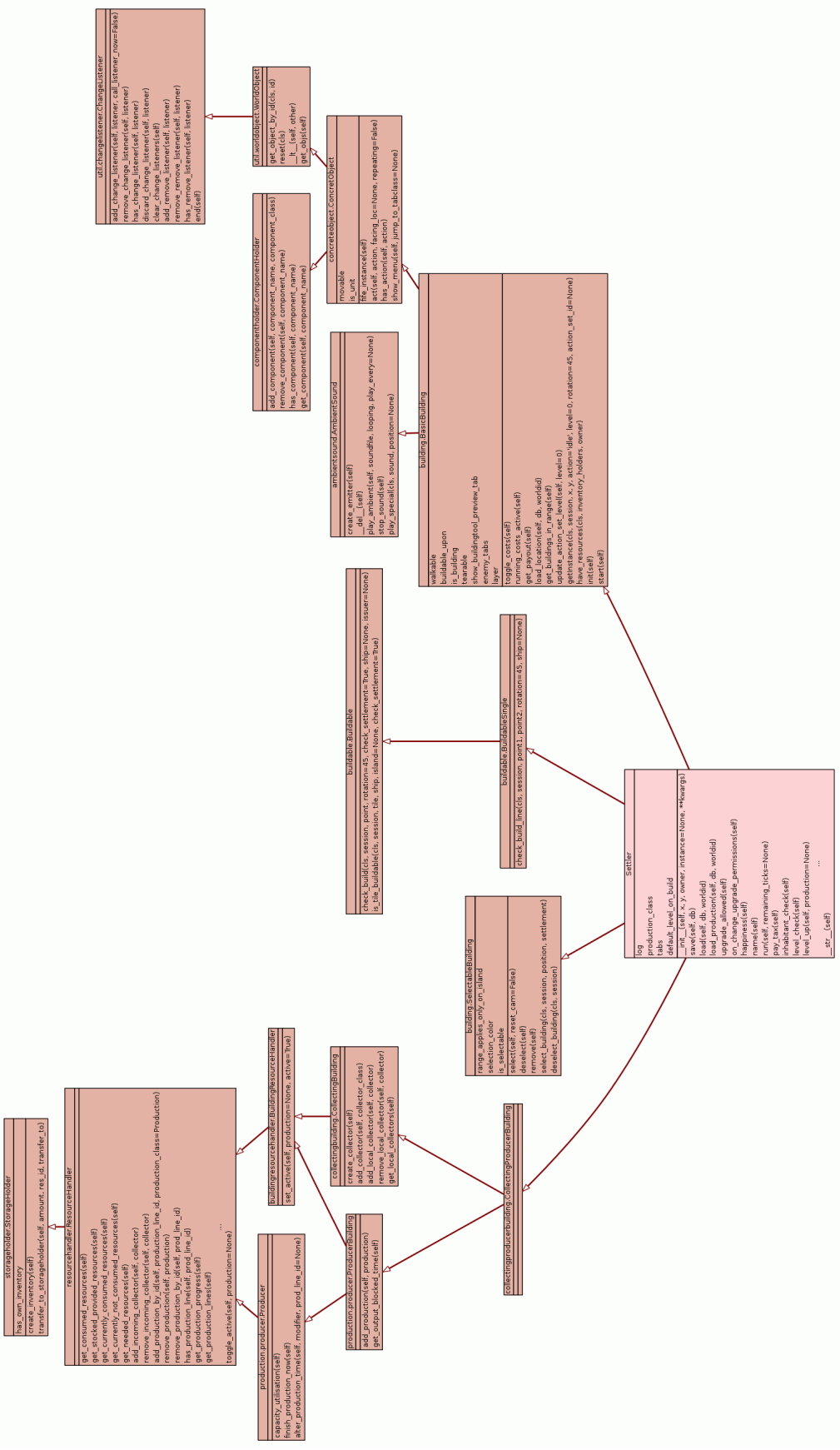

Figure 3: Inheritance tree for the *Settler* class in *Unknown Horizons*

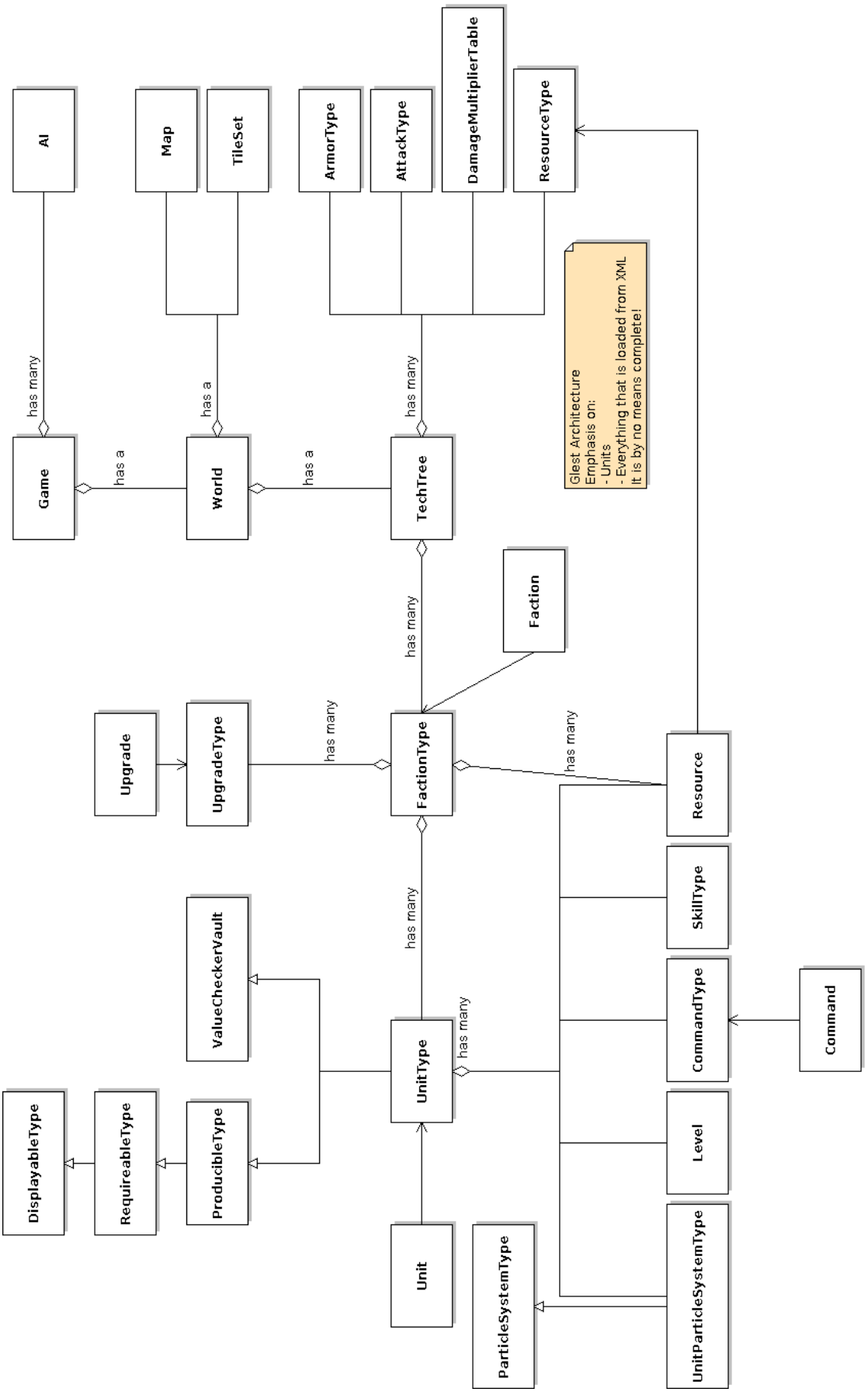

Figure 5: MegaGlest class hierarchy diagram

Figure 10: Inheritance tree for the *Settler* class in *Unknown Horizons* after the addition of the ComponentHolder

Bibliography

0A.D. Entity component documentation, 2011a. URL http://svn.wildfiregames.com/entity-docs/#component.VisualActor. [Online; accessed 08-November-2011].

0A.D. Atlas (scenario editor) manual, 2011b. URL http://trac.wildfiregames.com/wiki/Atlas_Manual. [Online; accessed 08-November-2011].

Wesnoth Community. Editingwesnoth, 2011. URL http://wiki.wesnoth.org/Editingwesnoth. [Online, accessed 08-December-2011].

Michael Doherty. A software architecture for games. *University of the Pacific Development of Computer Science Research and Project Journal RAPJ*, 1(1), 2003.

Michael Haller Fh, Michael Haller, and Werner Hartmann. A generic framework for game development. In *In Proceedings of the ACM SIGGRAPH and Eurographics Campfire*, pages 3–8322, 2002.

Eelke Folmer. Component based game development – a solution to escalating costs and expanding deadlines? In Heinz Schmidt, Ivica Crnkovic, George Heineman, and Judith Stafford, editors, *Component-Based Software Engineering*, volume 4608 of *Lecture Notes in Computer Science*, pages 66–73. Springer Berlin / Heidelberg, 2007. ISBN 978-3-540-73550-2. URL http://dx.doi.org/10.1007/978-3-540-73551-9_5.

Steve Mcconnell. Daily build and smoke test. *IEEE Softw.*, 13:144–, July 1996. ISSN 0740-7459. URL http://dl.acm.org/citation.cfm?id=624614.625626.

Jeff Plummer. *A flexible and expandable architecture for computer games.* PhD thesis, 2004. URL http://www.gamecareerguide.com/education/theses/20051018/plummer_thesis.pdf.

A. Rollings and D. Morris. *Game Architecture and Design: A New Edition.* New Riders, 2003.