Agile Game Development

Brandon Koepke, Brett Pelletier, David Adair, Rajvir Jhawar, Ian Macaulay, and Tom Bielecki

***Abstract*—TODO.**

***Index Terms*—game, development, software.**

I. Introduction

ALIQUAM TODO. [**?**].

II. Requirements

A. Background

Requirements elicitation is a very difficult part of developing software systems. In order to understand what requirements elicitation is, we first have to further define requirement. A requirement can be defined as

> (1) a condition or capability needed by a user to solve a problem or achieve an objective; (2) a condition or capability that must be met or possessed by a system or system component to satisfy a contract, standard, specification, or other formally imposed documents; (3) a documented representation of a condition or capability as in (1) or (2) [**?**].

The exact breakdown of what constitutes requirements elicitation is still being debated. Once source separates requirements into function and nonfunctional requirements [**?**]. They then further classify the nonfunctional requirements into performance/reliability, interfaces, and design constraints.

The process for requirements elicitation is decomposed into three activities [**?**]. The first is the elicitation of requirements from various individual sources. Then you need to insure that the needs of all users are consistent and feasible, and finally you need to validate that the requirements are an accurate reflection of the users needs.

There are many general problems with elicitation including [**?**]:

1) Problems of scope
   - The boundary of the system is ill-defined
   - Unnecessary design information was given
2) Problems of understanding
   - Users have incomplete understanding of their needs
   - Users have poor understanding of computer capabilities
   - Analysts have poor knowledge of problem domain
   - User and analyst speak different languages
   - Ease of omitting "obvious" information
   - Conflicting views of different users
   - Requirements are often vague and untestable, e.g. "user friendly" and "robust"
3) Problems of volatility
   - Requirements evolve over time

The understanding and volatility issues will still be present in game design, but scope issues may not be as large of a problem as they are in other software systems.

B. Emotional Requirements

Now that we have discussed requirements elicitation generally, we can analyze the differences between normal software requirements elicitation and video game requirements elicitation. Video game designers are concerned with the players experience which is rather difficult to define. One definition [**?**] describes the player experience as "the means by which the player's consciousness is cognitively engaged while simultaneously inducing emotional responses". The emotional requirements can then be further broken down into the emotional intent of the designer and the means by which the designer expects to induce the target emotional state [**?**]. This makes the nonfunctional requirements much more paramount to the success of the software system. In order to successfully capture the emotional requirement, it must capture [**?**] "the intent of the designer, and the means by which the designer expects to induce the target emotional state".

III. Design

A. Background

As with most software industries, when the paradigm of programming of software designs shifted from imperative to object-oriented programming, the game development industry adopted this change to take advantage of the many benefits that it offered. These benefits included data encapsulation, easier mapping of concepts to programming structures, and the ability to group and share behavior amongst objects of the same functional type. These benefits allowed additional functionality to be inserted without the need to duplicate a lot of code, thus greatly reducing the time, effort, and cost of developing large-scale software products.

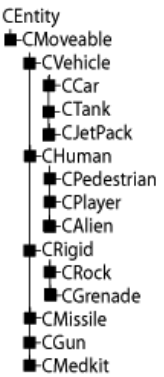

Fig. 1. A typical game object hierarchy. [**?**]

In a typical object-oriented software product, the objects form a hierarchical tree that has width (many objects with the

same direct ancestor, such as there are many weapon-types that are each modeled as an object, and they would all share the same ancestor of 'Weapon'), and height (set of objects that form a list of direct ancestors for each other, such as how a 'DwarfMale' is a 'Dwarf' that is a 'PlayableCharacter' that is a 'Character' that is an 'Entity', and so on). This tree-like structure describes the term inheritance, where one object such as 'DwarfMale' may inherit functionality from 'Dwarf'. In small-to-medium sized products, inheritance can make development both simpler and much more efficient. However, as projects grow and changes are made, designing objects around inheritance can lead to problems due to relationships between objects in the tree. Changing or adding code at the root of the tree may cause the changes to cascade down through the rest of the tree, resulting in undesired behavior, as described by Herb Marslas, a developer for Age of Empires II:

> ...functionality can be added or changed in a single place in the code to affect many different game units. One such change inadvertently added line-of-sight checking for trees [**?**].

With the development archetype shifting from the waterfall method, where projects are defined at the beginning and are less adaptable to changes, to a more agile and iterative-based approach, the ability to change the project on both small and large scales becomes a requirement of the design of the project. As teams in the game development industry began to conduct agile development processes, they began to realize that the object-oriented design paradigm was not as flexible and adaptable as it needed to be, largely in part due to the inheritance structure and requirement of defined interfaces between objects.

To solve the issues with large inheritance trees, architects, designers, and programmers discovered [**?**] the benefits of composition over inheritance, where it is better and/or easier to consider a has-a relationship over an is-a relationship between objects. An example of this can be described as the difference between a 'Character' object having a position in the world, rather than a 'Character' object being a 'PositionalObject'. The distinction is small conceptually, but rather large in terms of design and implementation of the project. In order to change the 'Character' to not having a position, a developer would simply have to remove the position with respect to the composition method, while the inheritance method would require the object to be placed somewhere else in the tree, potentially leading to side effects and/or undesired behavior.

The realization of the benefits of composition over inheritance began a movement of the game development industry to a design pattern known as 'Entity-Component Systems', where objects are modelled by what they are composed of or contain, rather than their position in the hierarchical inheritance tree.

B. Entity-Component Design

The entity-component design paradigm is primarily focused on three primary objects in the application: entities, components, and system-managers (or component-managers). Entities are the primary conceptual objects in the game, such as 'Player', 'Target', and 'Map', among others. Components, on the other hand, are essentially atomic parameters that can be attributed to an entity. For example, a 'Player' entity may have a 'Position' component, a 'Render' component, a 'Velocity' component, a 'Physics' component and anything else that describes the behavior of a 'Player' entity. The other object type, system-managers are objects responsible for the management of entities and their components, and the coordination and synchronization of each respective type. For example, a 'RenderSystemManager' may be responsible for iterating through every entity with a 'Render' component, and rendering that entity to the display.

Fig. 2. Entity component modelling in Crysis. Source: http://piemaster.net-content.s3.amazonaws.com/wp-content/uploads/crysis-components.jpg

There are many benefits that this type of design provides. The primary benefit, however, may be the fact that this type of design is inherently agile in nature. Changing the system is simply the process of adding, removing, or editing the desired entities and/or components. For example, to create a hidden trap object, an invisible object in the game that reacts in some way to the player, a developer may simply have to create the 'Trap' entity, provide it with a 'Position' component, and a 'Script' component that provides some behavioral functionality based on the 'Position' of the entity relative to the player, and register the entity with the correct system-manager that can update the 'Script' component whenever the player moves. Or to make the player invisible, a developer could simply remove the 'Render' component from the 'Player' entity. Additionally, once a component has been created, it is possible to reuse the component across the rest of the system or in future projects as it is simply a set of parameters containing no side-effects or requiring any interfaces for any other objects.

Looking at this type of design, it is clear that the system and all of its entities are data-driven in nature. Rather than an object inheriting from some 'Renderable' object, the desire to render the object is encapsulated into whether or not the entity has a 'Render' component. With this in mind, the complimentary relationship between agile development and entity-component design actually increases due to the fact that rather than having to receive requirements from a customer (the game designer, in respect to the game developer), the developer may actually allow the customer themselves to implement the desired functionality through a scripting engine,

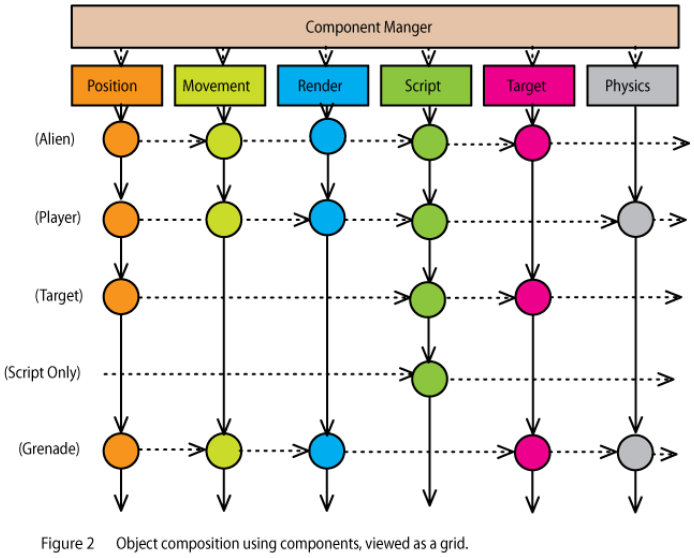

Fig. 3. Entity-component relationship. [?]

such as Dungeon Seige's scripting language Skrit.

```
[t:chicken_red,n:0x837FD928]
{
    [placement]
    {
      p position = 1.3,0,1.8,0x1738FFDB;
      q orientation = 0.3828,0.2384,-0.7772,0.98;
    }
    [common]
    {
        screen_name = "Super Chicken";
    }
    [body]
    {
        avg_move_velocity = 18.000000;
    }
}
```

Fig. 4. The Skrit scripting language for describing components in Dungeon Seige II [?].

Incorporating scripting languages allows the game designers–that is, the department whose job it is to define the functionality of the game and its entities–to actually define the functionality of an entity. Not only does this align perfectly with the agile manifesto, where interaction with the customer is paramount, but it also (if implemented correctly) allows different entities and components to be injected, removed, or edited at run-time to provide instant feedback to both the developer and the designer. This allows for an efficient iterative approach to be taken when developing the game as a whole.

The combination of entity-component systems with built-in scripting engines has become the primary design paradigm of modern game development, as showcased by the primary free, open source, and production game engines in the market today such as the Unreal Development Kit with Unrealscript scripting, and Unity3D with Mono, C#, and Javascript scripting.

C. Future of Game Design

With the increased practice of agile development methodologies, and the realization of its complimentary nature with entity-component design with embedded scripting, it is likely that entity-component systems will continue to be the primary design archetype for the foreseeable future. So where does the industry go from here? A good modification candidate to the current methodology appears to be incorporating functional programming considerations within the game design. Functional programming is a programming method that strives for pure, non-state mathematical functions. Simply speaking, a pure function is a function that receives some input, performs some calculations, and returns a result or a modified copy of the input. A pure function is never able to access any state outside of the function's local scope. This is in stark contrast to the current paradigm of imperative programming, where typically it is the state of the application or an object that determines how the application behaves, with the goal of most functions to put the object or application into some known state.

Functional programming was originally introduced with LISP in the 1950s, but was slowly overshadowed by imperative programming due to the lack of computational resources available to facilitate pure functional programming in performance-critical applications. However, as innovations in the number of transistors able to be packed onto a single processing chip have begun to slow, processors have begun to incorporate additional parallel cores to increase processing power capabilities. With this plateau in processing power per core, and an increasing number of available cores, it is necessary for software to take advantage of parallel, multithreaded computing. However, the current method of imperative programming where application behavior is primarily driven by objects modifying state begins to break down in parallel environments as you begin to require transaction-based state modifications which can impose performance-limiting overhead. Thus, with parallel architectures becoming much more common, a paradigm-shift back to functional programming begins to provide tantalizing benefits.

While functional-versus-imperative programming is primarily an implementation-level decision, understanding the advantages and limitations of functional programming at a design-level may provide a better design in which to implement the specific programming archetype. Additionally, understanding that pure-functions do not interact with program state and do not produce any side effects allows functionality to be created iteratively in an agile approach. It is simply a matter of replacing a pure-function with a new implementation–say a better algorithm or different behavior altogether–and the rest of the code (in terms of program state) is unaffected by unsuspected side effects. With the nature of the code being simpler due to the reduced complexity of program state, the application becomes much easier to maintain and extend.

IV. Testing

The video game market is a multibillion dollar business. Game developers spend thousands of dollars into game testing to have a product that is designed and tested well enough it doesn't frustrate the players who buy the game. They do this so that the developers can successfully penetrate the market

and get your name known. Testing in a video game is quite different from testing a conventional piece of software. The developer wants the user to get immersed into their game. This can easily be ruined by small bugs in the software or hardware. Described below are some of the most important tests conducted throughout the design cycle of the product.

A. Functionality Testing

The most common type of testing and as such can be done by a person with little or no technical knowledge. This, like beta testing, can be released to the public or individuals to do the testing for the publisher. These test check for general problem in the game or the user interface. The main problems these tests looks for, is any issues with the stability, game mechanic, and integrity of the game. This is used when the game is playable in some form before being release.

B. Compliance testing

This is the general standards for each platform that a game would like to be released on. Such standards include handling of error messages, copyright material, trademarks, and save data handling. If the game does not meet the standard guidelines as laid out by the platform developer then game will be delayed and cost will increase as a result.

C. Compatibility testing

This tests the game to see if its compatible with the software and hardware supplied by the publishers. Compatibility testing is used early in the beta version and at the end of the production of the game. This ensures that the hardware includes brands of different manufactures and assorted input such as joysticks. They also test the performance and results when the hardware is ran at the advertised minimum requirements.

D. Multiplayer testing

Used when testing a game with a some type of multiplayer feature. Testing is done to ensure all the connectivity methods, such as modem, LAN, and internet, are working efficiently and correctly. Online player interaction and proper game performance while online contributes to the majority of this type of testing.

E. Localization testing

Deals with checking proper translation of in-game text or dialogue. These testers are usually person's who speak multiple languages and understand regional differences between countries.

F. Soak testing

Tests the how the game is running on the system for prolong periods of times in different modes of the operation or when repetitive actions. The modes of operation include pausing the game in various modes of play, remaining on the title screen, or just idling. Usually since this type of testing requires no interaction besides the initial set-up automated tools are used to do any repetitive tasks such as mouse-clicking or button pressing. Used to find any memory leaks or rounding errors.

G. Regression testing

Once a bug has been found and potentially fixed QA testers must verify that the bug has actually fixed the problem. They must also determine if the fix had a detrimental effect on the game or caused a new bug to suffer.

H. Beta testing

Initial versions of the software are released to the general public or select individuals so that they can physically play the software and find any bugs the testers have missed. This increases the chances of finding bugs in the game itself and is a cheap and efficient way to create hype for the game. Since the game is in beta the players understand that there will be bugs in the game.

V. Creative Interaction

VI. Frameworks

A. Background

In the game development industry today, Game Engines provide the necessary frameworks, API's, and SDK's for developers to build games for various platforms. The purpose of a game engine is to abstract out the details of commonly used tasks and separate them into reusable components for the developers. Some of the functionality provided by game engine's include Rendering, Physics, User Input, Collision Detection, and Animation. As there are hundreds of different game engines available on the market, functionalities provided by each game engine will differ depending on the type of engine the developers choose to use. There are both commercial and open source game engines but the majority require a commercial license but may include a free non-commercial version.

B. Types of Game Engines

There are various types of game engines available to developers. At the lowest level, developers will combine many different API's to develop their own game engines. While this gives developers the flexibility of choosing their own components, it usually comes with a drawback of high costs. There can also be technical issues that can arise when trying to use many different libraries together for the first time. The drawbacks associated with this approach make it less attractive in the gaming industry.

The mid-level game engines include many features out of the box. These game engines are usually ready for rendering, physics, user interface, collision detection, and much more with less setup required than that of the low level approach. Developers will still need to do a bit of programming to have a complete game up and running. The Unreal Engine 3 is an example of mid-level engines.

At the highest level are game engines that allow for point and click game development with very little programming required. Unity3D and GameMaker are examples of this type of game engine which makes the development process as user

friendly as possible. Although these are easy to use engines, they come with a drawback. Most of them are limiting in the types of games they can make or the quality of graphics they can produce. The advantage of quickly developing games have made the high level engines more common today.

C. Unreal Engine 3

The Unreal Engine 3 is a cross platform game development toolkit that was developed by Epic Games. This toolkit is widely used by many game developers including BioWare (EA), 2K Games, Epic Games (Microsoft), Arkane Studios (Bethesda Softworks), and many more. It has been used to produce hit games including Gears of War, Borderlands, and BioShock. Epic Games describes the Unreal Engine 3 to have been designed with

> ...ease of content creation and programming in mind, with the goal of putting as much power as possible in the hands of artists and designers to develop assets in a visual environment with minimal programmer assistance, as well as giving programmers a highly modular, scalable and extensible framework for building, testing, and shipping games in a wide range of genres. [**?**]

Features include tools for Animation, Artificial Intelligence, Audio, Physics, Rendering, Networking, and many more. Although the Unreal Engine 3 is a commercial use game engine, Epic Games released the Unreal Development Kit (UDK) which includes many of the features of the Unreal Engine 3 for free non-commercial use.

Fig. 5. Rendering produced using the Unreal Engine 3. Source: http://www.unrealengine.com/files/features/rendering1.JPG

D. CryENGINE 3

The CryENGINE 3 was developed by Crytek and is used to develop games for Xbox 360, Playstation 3, Windows, and Wii U. CryENGINE 3 was used to develop the first person shooter franchise titled Crysis developed by Crytek and publish by Electronic Arts. Similar to the Unreal Engine 3, CryENGINE 3 is a commercial Game Engine that offers a free version of the toolkit for non-commercial use only. It also includes a large number of tools for Visuals, Characters, AI Systems, Physics, Performance, and many more.

Fig. 6. Crysis 3 was develop by Crytek using the CryENGINE 3. Source: http://www.crysis.com/us/screenshots/crysis-3-concept-art-warehouse

E. Unity3D

Unity3D is a cross platform game Engine that can be used for Developing Games for Xbox 360, Playstation 3, Windows, iOS, Android, Mac, and more. Unlike CryENGINE 3 and Unreal Engine 3, Unity has a free version that can be used for developing games for commercial use (with certain conditions). The Pro version of Unity3D has a licensing fee but includes a larger set of features than that of the free version. The developers of Unity call it a

> game development ecosystem: a powerful rendering engine fully integrated with a complete set of intuitive tools and rapid workflows to create interactive 3D content; easy multi-platform publishing; thousands of quality, ready-made assets in the Asset Store and a knowledge-sharing Community. [**?**]

Although Unity has not been used to produced big name titles like the Unreal Engine 3 has, it is still a popular choice especially for mobile application development. Of the featured games provided on the Unity website, a majority of the titles were developed for iOS, and Android devices.

Fig. 7. "Escape Plan" Developed using Unity3D for the Playstation Vita. Source: http://en.wikipedia.org/wiki/File:Escape-plan-screenshot.jpg

F. Cocos2D

Cocos2D is an open source 2D game framework released under the MIT License making it free to use and develop for commercial purposes. It has been ported to many different

devices including iOS, Android, Blackberry 10, and Windows. There have been thousands of titles produced for mobile devices using Cocos2D. Unlike Unreal Engine 3, CryENGINE 3, and Unity3D, Cocos2D is not considered a game engine. Its goal is to abstract the details of the low level graphics packages and make game development fast and easy to use for mobile developers.

Fig. 8. "Zombie Smash" Developed for iOS and Android using Cocos2D. Source: http://www.blogcdn.com/www.tuaw.com/media/2010/03/zombie-smash.jpg

VII. Conclusion

(The conclusion is dependant on the body of our paper, which has not been completed for this draft.)