**Understandability (Nikolaj)**

To ease the development process of Rooms we will need to keep in mind the importance of internal understandability. Each team member will have a main responsibility such as working on the client side of the program versus the server side. Occasionally a team member may need to help code another section of the program. In this case the code will be unfamiliar and very hard to understand unless it is well structured and documented. The programming languages we will use to implement Rooms are Haxe and PHP. The knowledge and familiarity with these language will vary considerably among team members. A team member with extensive knowledge of Haxe for example will need to avoid writing cryptic code that others may not understand or at least provide comments to explain its exact functionality.

External understandability is especially crucial when it comes to video games. While a good game is challenging and difficult to master it should always be easy to learn. The modern player is very unlikely to put up with unintuitive controls or confusing gameplay since the free-to-play game market is highly competitive these days and offers countless alternatives to Rooms. How easily and quickly a player can grasp the basic game concept and emerge into the game world is what ultimately determines a game’s potential for success. We need to give players a good idea of what Rooms is about without overwhelming them with walls of text. Instead of extensive instructions and tutorials the actual gameplay has to speak for itself. We are relying on the fact that the average player is familiar with conventional mouse controls from other top-down games similar Rooms. Designing the UI in close resemblance to already existing titles will place the players in an instantly familiar and thus intuitive environment. Tutorials concerning in-game navigation on the desktop therefore will be minimal. We may however need to elaborate on the controls on mobile devices since touch screen controlled games do not have too many established conventions yet. Communicating the game concept without sacrificing too much of the uncertainty that lends Rooms its atmosphere will be our greatest challenge in terms of understandability. A possible approach we could take is integrating non-player-characters that reveal bits of information concerning the game mechanics, such as the way in which artifacts affect the gameworld, as well as the backstory of the world of Rooms. As Rooms features and relies on custom content, created and uploaded by the game community the ease of use of the editor is also crucial. Since on average players aren’t nearly as experienced in creating in-game content as they are in playing the game the editor will require more in-depth tutorials.

**Reusability (Nikolaj)**

Rooms consists of four major components: the client, the server, the website, and the editor. Reusing the server-side code may not be possible in our case and thus will be developed with a much smaller focus on reusability than the client and editor.

Reusability of the client code as well as the editor however will be crucial since we are planning on developing a single player campaign after the development of Rooms as an MMO is completed. The single player campaign will have altered game mechanics since it will be plot driven. In particular room generation will employ a different algorithm. We will be able to reuse most of the client code to develop the single player campaign without making too many changes to the overall code structure. Developing all code in a modular way will help facilitate reusability. Taking the single player campaign as an example, we will be able to reuse most of the game mechanics (pathfinding, graphics, monster and item base) and only alter the module responsible for generating rooms and removing the module responsible for client-server communication and multiplayer functionality without affecting the rest of the code. Since the client side of the game will be written in Haxe targeting Flash, we will need to avoid using any deprecated features to ensure compatibility with future versions of Flash. Same goes for the mobile version of the game. In the long run, we probably won’t be able to avoid releasing patches for the game in case it becomes incompatible with future versions of the Android OS and iOS. Haxe is a write once deploy anywhere language. In the ideal case we should thus able to reuse the client code on multiple platforms without writing any platform specific code. How this premise holds up in practice remains to be seen.

**Evolvability (Ashley)**

The evolvability of Rooms is very important for the game and it’s long term enjoyment. Since the basic system is designed to be random, all items, monsters, etc, are encountered randomly. We plan to ensure a constantly evolving system by implementing the custom content editors. As more players upload content through the editors, this content will have the opportunity to be encountered during gameplay. This allows the content and therefore the possible game scenarios to evolve as you play.

Not only do the editors provide the growth of the items and monster bases, but they also offer the ability to add complete rooms that could be encountered instead of the random rooms. These rooms, because designed by players, are more likely to contain a coherent theme or a systematic set of tasks. This will potentially make the game play they offer more interesting.

By relying on the editors for an inflow of content, rooms will be freed from the burdens that plague many game systems. Most systems require developers to implement all of the new features or levels that users can encounter. This can result in predictable or delayed features. Because the editors let any user create content, there will be a constant fresh supply of new and imaginative content, thus eliminating the issue of evolving the game.

Beyond the editors, we plan to implement several strategies in our code architecture that will allow the whole system to be evolvable. Our system is going to be designed with the entity-component design pattern. This will allow us to add additional features by adding new components. This means that existing components, and anything depending on existing components, will not be affected. This means our design will be flexible without needing to worry that adding new code will break existing code.

Additionally,we plan to mirror this evolvability in our database by using an attribute table. This lets us add new possible attributes without needing to add additional columns to, for example, out item definition table.

**User Friendliness (Ashley)**

The UI and user experience is a very important part of Rooms and it offers a unique set of challenges that we have to face. The first of these challenges is the multiple platforms we intend to target. We intend to offer rooms as a playable game in both a web browser, iphone, and android. This means that we will want to design a game UI that can be effectively implemented in each of these platforms. It will need to be very simple and intuitive so that it is easy to learn and similar between the platforms. The first concern related to this was the actual game controls. Because iPhone & android will both be touch platforms, the game controls will be implemented in touch. To move the player, the user will touch the screen where the player is supposed to walk. To attack, a player on a mobile device will be able to quickly tap between their set of main weapons and then tap monsters or players to attack with the currently selected weapon. Unlike the mobile platforms, the web implementation will not be able to rely on a touch interface to control gameplay. To make it similar to the mobile app, we plan to use a combination of point & click and keybindings to control player movement and interaction. To aid in the use of items and the change between primary weapons, we will have a quick easy to access item menu that will allow you to access to the supplies & items you have collected.

The next major issue with the UI was the game editors. They will need to be both easy to use and allow a large range of options. We decided that we will only implement them on the website. To make them simple to use, we plan to let you select a ‘base’ existing object, from which you can inherit it’s attributes.

Additionally, we plan to make all of the information on the website highly accessible. When you first enter the site, if you have a valid session then your game will automatically load on the front page. If you are not currently logged in, then you will need to log in before you can access the game. We will implement a simple navigation system across the top that will allow one-click access to your profile, forums, custom content creators, and information about Rooms.

**Performance (Thom**)

There are two related (but separate) issues which must be addressed in terms of the performance of ROOMS: Perceived performance (e.g, game response time, load times, etc), and actual performance (e.g. frame rate, server latency, etc). If either is poor, then ROOMS will seem to have poor performance overall.

In order to have the best possible perceived performance, we must address the issue of synchronization in an low-error and efficient manner. A zero error synchronization strategy would involve getting the server to accept any action before allowing a client to make that action. This would destroy response time (and the game’s perceived performance) requiring time on the order of 2 \* server latency + server processing, which is unacceptable. ROOMS opts to allow the client to simulate forward independently of the server, transmitting a time-stamped message including the necessary data for the server to calculate the players actions. The server will transmit this data to other clients, so that they may accurately simulate the game. When a synchronization error occurs (which given that all information is timestamped, should be rarely), the server will inform the client of the actual current state of the world, and it will revert to that state. This way, the perceived performance of ROOMS will be high, as the clients will not have to wait any time to see their actions, and the occasions when a client is moved into a different position due to a synchronization problem will be rare.

To further improve perceived performance, loading times will be mitigated or entirely removed by the server preparing (that is, generating or choosing) all rooms adjacent to an occupied room, starting from the moment that room becomes occupied, after they are prepared, they are transmitted to the clients occupying the room, so that the client does not have to experience a loading screen when traveling between rooms.

To a large extent, the “actual performance” of the system is beyond our control. ROOMS will run on a large range of devices, and run well on a smaller range. Despite this, having the gap between those be large is unacceptable. The single largest decision ROOMS makes to improve its overall performance across these devices is to render the world in two dimensions. Additionally, we will be caching images on the client side when possible, and using profiling information to identify code hotspots and replace them with optimized platform specific versions if necessary.

**Portability (Thom)**

ROOMS is designed from the ground up to be extremely portable. We chose Haxe as our implementation language for exactly this reason. Our plan is to release our game on the iOS App Store, and the Android marketplace, in addition to our flagship Flash version, located on our website. However, we do not want to restrict ourselves from porting ROOMS to other platforms in the future.

The promise of “write-once run-anywhere” that Haxe, among others, gives is alluring, however it rarely works perfectly . Often, different platforms do things in fundamentally different ways, and you’re stuck writing different code for different platforms, and sometimes this can threaten to take over your codebase. In order to avoid this with Haxe, we will be designing a platform neutral abstraction layer that sits between the Networking and UI portions of our code. This way, should we identify an inconsistency in an api between different platforms, we will be able to normalize it and keep the platform-specific code localized in these areas. Additionally, we leave open the chance of porting these platform-specific abstractions to other platforms, and as a result, ROOMS itself. It is important to note that this requires that the game logic of ROOMS must be entirely platform neutral in this case. In fact, it must not only run on each client, but must also run on the server (as the server simulate the game logic as well in order to synchronize the clients).

A very large and important part of ROOMS is a web application, and the web has an entirely different set of portability considerations than a game. A portable web application is one that looks acceptable at all screen sizes, and that that degrades gracefully when run in older or less featureful, browsers. This is a large reason that we chose Flash as our target platform for the game and editor instead of HTML5/Javascript, because it works better in more browsers. The fact that Flash tends not to work on mobile devices is largely inconsequential, as we are developing versions of the game for iOS and Android as well. We believe the lack of a mobile version of the editor is acceptable, due to substantial UI paradigm differences.

**Maintainability (Nhat)**

Maintainability is important to consider for ROOMS because it will have a very fast development pace as well as a fast updating pace. We will be using version control (such as Git or Subversion) to maintain our code base. This will allow for rapid peer review which will decrease the amount of duplicate code generated. Since we will be deploying the game client to multiple platforms, we will maintain a single codebase using HAXE. This will be key since it will essentially eliminate any duplicate code across multiple gaming platforms such as mobile, web-based, or desktop gaming platforms. We also acknowledge that there may be some platform specific code to fix platform specific quirks.

Specifically for ROOMS, we need to embrace concepts like modularity and changeability in order to keep the amount of time spent bug fixing and adjusting to the learning curve for our development team low. This means we will need to break down almost any functionality of the game, like custom content, networking, and artificial intelligence into very small pieces of code which are strictly modular. For example, when we are defining the AI system, we should create a standard type of output will can be thoroughly tested. Actions such as attacking or moving will need to be easily simulated by test cases to check for correctness in our implementation.

Creating a bug-free product is not the goal, since it is impossible. We are aiming to minimize the pain of having to understand the whole system to be an effective developer. Maintaining best practices such as when developing will allow for any member of the development team to contribute to ROOMS. Having a clean code base with sufficient comments as well as documentation is critical the project’s success. It will encourage further rapid development allowing us to focus on the gameplay and add new features easily. Unlike traditional software, games like ROOMS should never be a complete or finished product once it has reached production.

**Robustness (Nhat)**

Robustness is important to ROOMS since it needs to be relatively bug free. Robustness is something that should be designed from the beginning of a project, not an afterthought. We will be using an entity component system, which is commonly used for games. It will allow us to break down each attributes of monsters, items, and room objects into atomic parts. This system also allows us to accommodate new features between our game entities (like monsters and items) by adding new combinations of components and their attributes. Very little change to the game engine will be needed to implement new ideas and custom content created by our users.

Our code base should be designed to be transparent and simple. Compared to software which need to be thoroughly tested to guarantee a predictable outcome, a game like ROOMS has a little more leeway. However, this should not be overlooked just because ROOMS does not have any significant impact if it fails. In order for our game to be considered robust, its code must have complete test coverage.

Since our project will be large and complicated, we need to deploy coding practices which will make it easier to detect subtle errors. An important part of robustness is writing generalized code. We will need to avoid having to write additional code to handle special cases. This implies that our code must be modularized into very small, testable pieces. To aid in maintaining robustness, we can make use of continuous integration to test every commit or build of our code. Every time new code is introduced to the repository, the test cases will automatically be ran and reports will be generated for developers to analyze the changes. Having complete test coverage in this case will guarantee that we will have less issues.

**Productivity (Mevludin)**

Productivity is a general problem for all projects. Without productivity, the project may be more difficult to deliver and work with. To increase the productivity for ROOMS: we will be making ROOMS object oriented, coding the server side and client side in HAXE, coding in PHP for the website, as well as using a framework for the website.

Making ROOMS object oriented will make the project easier to work with and easier for every member to understand, since all members know object oriented programming. Because of this, each member will be able to program a section of the game and still be able to understand the system as a whole.

Another decision which increases productivity is coding server side and client side in HAXE. Coding both in HAXE decreases code duplication, especially for the game logic because game logic has to be exactly the same on the server and client. If the server and client side were done in a different language, it would require more work to rewrite the game logic, and it wouldn’t guarantee that they will run exactly the same because each language may execute and handle the code differently, although they were written to do the same thing.

Also, using a model-view-controller architecture type framework and coding in PHP for the website will increase productivity. There are multiple team members that know how to code in PHP, therefore decreasing time in learning another language and in debugging. The reason it will be easier to debug is because the team members have written in PHP and are familiar to various errors. And using a framework will make the website development part of the project faster to finish because the framework will provide the basic structure for the website and allows us to use built in features, such as security, a template system, database access and much more. The MVC framework also promotes code reuse, decreasing time spent on coding.

**Reliability (Mevludin)**

The reliability of ROOMS will depend on the software used for and by ROOMS. They are the server, the website model-view-controller framework, and HAXE’s Windows and Macintosh support. Since ROOMS is using HAXE to support Windows and Macintosh operating systems. To ensure the game works on these operating systems, ROOMS will need to avoid any known HAXE code that causes bugs for any OS. As well as, ROOMS will be played and tested on each operating system.

Since ROOMS uses a framework for the website, the reliability of the website depends on the quality of the frameworks functions and features. The framework components that are used to build the website will need to be tested to ensure that they work correctly and do not contain any bugs. More importantly, the server will need to be stress tested to ensure the game and the website will be able to function with high traffic. If the game constantly crashes, no user will want to play.

Along with having to test all the tools used to build ROOMS, the main issue is to avoid bugs within the developer code when building the basic structures for the game. ROOMS will be using a creational prototype design pattern and a structural composition design pattern. Since these design patterns define how the game works, the development team will need to ensure they work correctly by doing code review.