




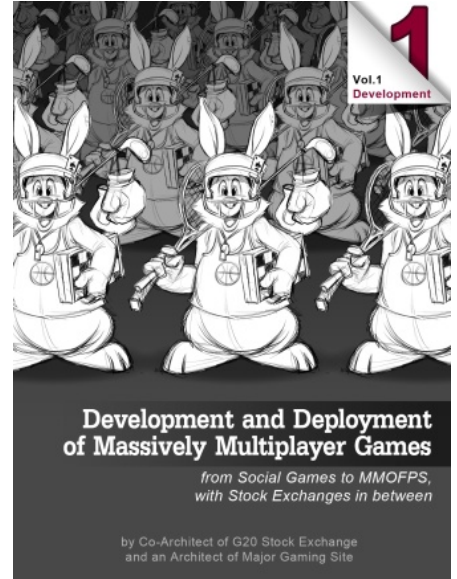
IT Hare on Software

Chapter I: “Business Requirements” from upcoming book “Development and Deployment of MMOG”

posted October 26, 2015 by "No Bugs" Hare, translated by Sergey Ignatchenko 

[[This is Chapter I from the upcoming book “Development&Deployment of Massively Multiplayer Online Games”, which is currently being beta-tested. Beta-testing is intended to improve the quality of the book, and provides free e-copy of the “release” book to those who help with improving; for further details see “Book Beta Testing”. All the content published during Beta Testing, is subject to change before the book is published.

Please note that this Chapter I may look boring for some of the developers; don't worry, there will be a lot of code-related stuff starting from Part B, but at the moment we need to describe what we're dealing with.



To navigate through the book, you may want to use Development&Deployment of MMOG: Table of Contents.]]

Preface

So, you have got a Great Idea for your Next Big Thing massively multi-player game, and know every tiny detail about gameplay and graphics which you want your game to have. Now the only tiny thing you need to do is to program it. Unfortunately for you (and fortunately for me as an architect and the author of this book 😊) game development and subsequent deployment is not that simple. There are lots of details you need to take into account to have your game released, to be able to cope with millions of simultaneous players having very different last-mile connections, and to make the game work with 0.01% of unplanned downtime while being able to add new game features twice a month.

Part A. Conception: Before the Very Beginning

You don't “make” a violin. It is barrels and benches which are “made”. And violins, just like bread, grapes, and children – are born and raised.
— Nicola Amati character from “Visit to Minotaur” movie —



“A game being developed is pretty much like your baby.

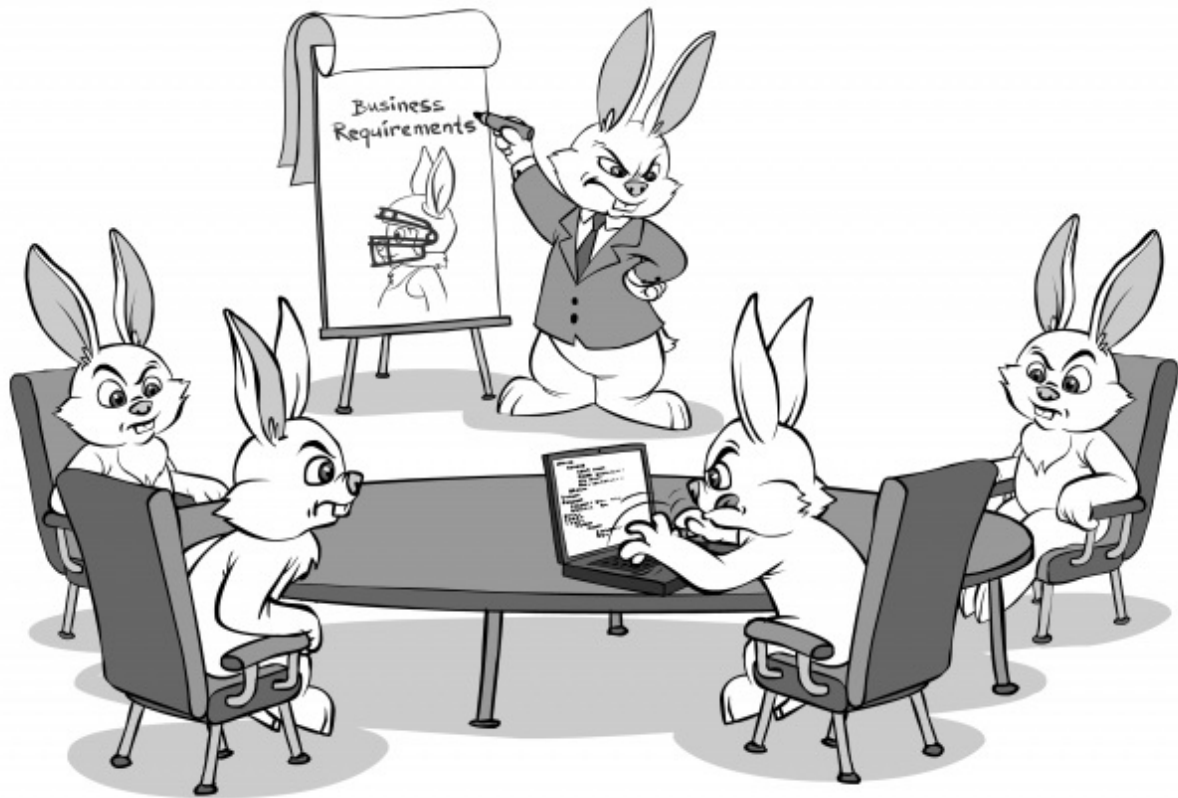
A game being developed is pretty much like your baby. It will go through all the stages which are typical for baby development, from conception to a newborn and then to a toddler. While development of your game certainly doesn't stop at that point, in this book we won't discuss how to raise your game beyond toddler; child and teen issues (both with games and with real children) are too often of psychological nature, and are beyond mostly-physical issues which we're about to discuss.

“You” as used throughout this book, actually means “parents of your game baby”; it will usually be a small team, but can be a 100-developer team on one side of the spectrum, or a single developer on another one. What is really important is not the size of the team, but how the team feels about the project.

If you (as a future parent) don't feel that your future game is *your baby* – think twice before conceiving it. Doing such a challenging development with only money in mind might not be the best decision in your life. If you're starting to develop only for money without any feelings for the project – then there are two possible outcomes. In the first case you will gradually become attached to the project, and eventually will have certain positive feelings about its development, greatly increasing the chances for success. In the second case, you keep doing it for money; while making a great game is still possible this way, it is much much more difficult to achieve. Success if doing-it-for-money-only becomes even more elusive if this is your first massively multiplayer game project, and you need to keep this in mind.

In Part A, we will discuss activities which need to be performed even before the coding can be started; let's name this stage project conception. It includes many things which need to be done, from formulating business requirements to setting up your source control and issue tracking systems, with lots of critical decisions in between.

Chapter I. Understanding Business Requirements



As mentioned above, we're working under assumption that you've got a Great Game Idea (with as full understanding of planned user experience as it is possible at this time), you're really passionate about it, and are really eager to start development.

What should be your very first step on this way? Start coding? Nope. Choose the programming language? By the tiniest of the margins closer, but still no. The very first step should be to understand what exactly you're going to achieve.

With any game, there are quite a few things which are dictated by your future players (and other project stakeholders). Even if your project is entirely non-commercial, just for the sake of being consistent with the rest of the world, let's name these things "Business Requirements".

Project Stakeholders

Every project has project stakeholders. A stakeholder can be an investor, a manager, and/or a customer. For games, it is often translated into game designers, producers, marketing/monetizing guys, customer service representatives (CSRs), and, of course, players. If you're developing the game in your spare time, it can even be yourself. In any case, every project out there has project stakeholders. For games, one extremely important type of the stakeholder is the future player (usually as a "focus group").

One thing which is paramount for the game to be successful, is to



"The very first step should be to understand what exactly you're going to achieve."

Have Project Stakeholders, including Future Players, Take Part in Development Process

If your project stakeholders don't participate in your development process (this should apply to all the development stages, from specifying requirements to alpha/beta testing) – the project is doomed almost for sure. And for games, project stakeholders MUST include future players of your game.



**“For your
game to
survive, most
likely you will
need some kind
of
monetization.**

On the other hand, having only future players as project stakeholders is not enough. For your game to survive, most likely you will need some kind of monetization. And those people who're responsible for monetization (marketing etc.) are also very important project stakeholders, and MUST be involved in game development.

The reason behind can be roughly described as follows. Each game tends to create a separate world, with its own rules, which are not obvious to the outsiders (and developers are outsiders for the game world despite intimate involvement with game mechanics). While we as developers can try to guess what is the best from the stakeholder's point of view – these guesses are usually way off, that makes the game unplayable (if players' opinions are not asked for), or non-monetizable (if other stakeholders are not asked). For the project to be successful, we DO need to have a stakeholder available during all the stages of the game development process. In other words, if we (as developers) have any doubts on any issue related to business requirements – we SHOULD have somebody on hand to ask for their authoritative opinion.

BTW, don't think that if you're going to play the game, your opinion as a future player's will be sufficient. Unfortunately, when we (as developers) are writing code, it affects our judgments about the game a lot; in other words, we know too much about the game internals (and on efforts we need to spend to develop this or that particular feature) to represent opinion of “an average player out there”. While our suggestions (based on this knowledge) can be very valuable, all the decisions about gameplay SHOULD be made by those future players who are not developers.

To summarize:

**Participation of both future-players and other
stakeholders (such as people responsible for
monetization) in developing (and later in amending)
Business Requirements is absolutely necessary.**

No stakeholders – no Business Requirements – no development, it is that simple. Doing it any other way is a foolproof way right into disaster.

Stakeholders and Business Requirements

In some cases your stakeholders will give you a specification which says what you need to do. More often than not, however, you will just get a vague description of the Great Game Idea. It's fine as a first step, but to get a clearer understanding of what is needed, you have to get your project stakeholders to sit down together with you and to write down your real Business Requirements.

This will involve at least one session dedicated just to this purpose (and probably much more than just one such session); ideally, these sessions should be in person rather than some kind of a conference (video)-call. I do know that in the XXI century there are ultra-cheap conference calls and video conferences available, but they still fall short compared to in-person meetings. While most of ongoing communication can be made over the phone/Skype/chat/email/..., at a few important points during game development process, such IRL meetings are necessary, and one of these points is certainly those Business-Requirements-writing sessions.

Much more important, however, is to make sure that

Business Requirements are written by Project Stakeholders (and not by Developers)

During the session, by all means, note down all the things-you-think-are-stupid and raise concerns (preferably in a bit more polite form than “are you guys crazy or what?”), but be ready to accept decisions by stakeholders when they insist (as long as they're staying away from implementation details, see below).

During these business-requirement-writing sessions, our role as developers is generally not to suggest business requirements, but to make sure that all our questions to project stakeholders are answered. Also it is very important to remember that Business Requirements is *not* about “*how* we will do it”, and to concentrate on “*what* is the thing which we will do”. While it is perfectly ok to say “implementing this feature will take us extra 3 months” (which in turn does need you to understand – but not explain – how to do it), a decision “if having this feature worth these extra 3 months”, lies entirely in stakeholder's domain.

It should be also understood that (exactly because the session is about *what?* and not about *how?*) it is entirely possible that at later stages (but still before the coding is started) it may happen



“During these business-requirement-writing sessions, our role as developers is

that Business Requirements cannot be satisfied. It is even more important to emphasize that this is a part of normal iterative development process, and in this case another Business Requirement session may be necessary (though the second and subsequent sessions usually simple enough and don't normally need to be in-person; that is, if you're not too unlucky and didn't skip too much of this book 😊).

generally not to suggest business requirements, but to make sure that all our questions to project stakeholders are answered.

Requirements vs Implementation Details

What are these “Business Requirements” for a typical game? Basically, they include everything your players will be able to observe. However, we need to distinguish between the things that the player cares about, and their respective implementation details.

For example, players do care about the platforms where they will be able to run your game, so “which platforms are to be supported?” is certainly one of your business requirements, but on the other hand players don't care about the programming language you will be using (as long as it can run on all those platforms). In another example players do care about response times and may care about how-your-app-works-over-firewalls, but they don't care if you achieve those response times and working over firewalls via TCP or via UDP, as long as the whole thing does work.

It can be summarized as follows:

Business Requirements SHOULD be expressed exclusively in Player's Terms

Or the other way around, using terms which are not familiar to the majority of players (or monetization people) SHOULD be prohibited for your business requirements document.



“If you write down a Bad business requirement “We MUST write our app in Java”

Why is this so important? Because writing requirements down in implementation terms rather than in player terms may severely hurt your ability to choose an optimal way to implement your game. Just as an example, if you write down a Bad business requirement “We MUST write our app in Java” (instead of the Good one “Our app MUST run on Windows, iPhone, and Android”), you won't even start to think about writing your app in C++ and porting it to Android using NDK (with a rather minimal Java UI, as described in Chapter [[TODO]]).

In another example, if you write a Bad business requirement “We MUST use UDP” (instead of Good one “In 99.99% of cases, we need at most 3sec delay between the user pressing a button and it showing up to the other users”), you won't even start to learn

(instead of the Good one “Our app MUST run on Windows, iPhone, and Android”), you won't even start to think about writing your app in C++ and porting it to Android via NDK.

about the ways to improve TCP interactivity (described in Chapter [[TODO]]), and may miss on an opportunity to make your app more firewall-friendly and to simplify your development by using TCP. Or the other way around, you may write a Bad business requirement “We MUST use TCP” (instead of a Good one “We MUST have TLS-class security”), and may miss on an opportunity to make your app more responsive via implementing it over UDP (using DTLS and/or TLS-over-reliable-UDP for security purposes, as described in Chapter [[TODO]]).

In short:

Writing Business Requirements in Player Terms allows you to Keep your Options Open

and keeping your options open is a Good Thing in general.

This separation between business requirements and implementation details means that if your project stakeholder (future player, marketing guy, manager, investor, etc.) says “we have a business requirement to write it in Java” (or “to use TCP” etc.) – you need to explain that this is an implementation detail, and to ask for a definition in terms which are obvious to the player.

Moreover, if the stakeholder is a manager and after all the explanations he is still insisting that using UDP is a business requirement – you really need to think if you want to work on this project, as such a deep misunderstanding is often a symptom of super-micro-management and upcoming deep conflicts with this specific manager.¹

¹ While “we need to use UDP” (or TCP for that matter) may be a valid business requirement in some cases (for example, when you’re writing a communication library, and your user is a programmer, so she knows about UDP), it doesn’t apply to games. You MAY need to use UDP for your game – it is just not a business requirement, but a technical decision on “how to implement these business requirements”

Subject to Change, Seven Days a Week

It is to be understood that in the real world Business Requirements tend to change very often, and are certainly not carved in stone. This is to be expected for most software projects out there, and applies to game development in spades. Therefore:

Expect Business Requirements to Change and Leave

Lots of Room for These Changes

Even if you're told that a certain thing will "never ever" change, keep in mind that "never ever" can come up much earlier than you expect. This is not to tell that you should write an "absolutely universal" system able to deal with *any* change (see about dangers of being over-generic below); this is to tell not to be too upset when you're forced to rewrite 50% of the system when a thing-that-you-were-told-will-never-change does change overnight. Oh, and do keep records of these assurances, so when the requirement changes, you can explain why such a simple thing (from the point of view of stakeholder) requires rewriting half the system.

One important thing to understand is that business requirement being agile doesn't imply that you don't need to write them down. While each of requirements may change later, at every point it should be very clear (and agreed by both stakeholders and developers) what you're trying to achieve *right now*. When (not "if"!) business requirements change – fine, you will update them.

Treat business requirements as one of the documents under your source control system (whether you really put business requirements document under source control – is up to you, but IMHO it is a good thing to do). In any case, business requirements tend to have effects similar to those of an extremely high-level header file in C/C++: as with changing a high-level header file, changing business requirements can be very expensive, but in a majority of cases it doesn't mean rewriting everything out there – *especially if you have prepared for it (see Chapter [TODO] for discussion on how to do it)*.

The Over-Generic Fallacy

Sculpting is Easy. You just chip away the stone that doesn't look like David.

— (Mis)attributed to Michelangelo —

When speaking about agility and taking "be ready to changing requirements" adage to the extreme, there is often a temptation to write a system-which-is-able-to-handle-everything and which therefore will never change (and handling "everything" will be achieved by some kind of configuration/script/...). While as a developer, I perfectly understand the inclination to "writing Good Code once so we won't need to change it later", unfortunately, it doesn't work this way. The issues with this over-generic approach start with the time it takes to implement, but the real problems start later, when your over-generic framework is ready. When your over-generic code is finally completed, it turns out that either that (a) "everything" as it was implemented by this system, is too narrow for practical purposes (i.e. it cannot be really used, and often needs to be started from scratch), or



“There is often a temptation to write a system-which-is-able-to-handle-everything and which

that (b) the configuration file/script are at best barely usable (insufficient, overcomplicated, cumbersome, etc.). In the extreme case of an over-generic software, its configuration file/script is a fully fledged programming language in itself, so after doing all that work on the over-generic system we need to learn how to program it, and then to program our game, so we're essentially back to the square one.²... **therefore will never change**

In fact, systems-which-can-handle-everything already exist; any Turing-complete programming language can indeed handle absolutely everything; in a sense, Turing-complete programming language,³ represents an absolute freedom. However, as writing a Turing-complete programming language is normally not in the game development scope, our role as game developers should be somewhat different from just copying compiler executable from one place to another and saying that our job is done.

What we as developers are essentially doing, is restricting the absolute freedom provided by our original Turing-complete language (just like a sculptor restricts the absolute freedom provided to him by the original slab of stone), and saying that "our system will be able to do this, at the cost of not being able to do that". Just as the art of sculpting is all about knowing when to stop chipping away the stone, the art of the software design is all about feeling when to stop taking away the freedoms inherent to programming languages.

Coming back to the Earth from the philosophical clouds:

When developing a game, it is important to strike the Right Balance between being over-generic and being over-specific

² Creating domain-specific programming language optimized for a game, may make perfect sense; the point here is not aimed against developing scripting languages where they make sense and provide additional value specific to the game domain, but against being over-generic just for the sake of writing-it-once-and-forgetting-about-it

³ and I don't know of any practical programming language which is not Turing-complete,

Keeping Quality under Time-To-Market Pressure: Priorities, MVP, and Planning

When developing a game (or any other software for that matter), it is very important to deliver it while it still makes sense market-wise. If you take too long to develop, the whole subject can disappear or at least become much less popular, or your graphics can become outdated.⁴ For example, if you started developing a game about dinosauri during dinosaur craze of 1990's but finished it only by 2015, chances are that

your target audience has shrunk significantly (not to mention that they've changed a lot).



“We will inevitably be pushed to deliver our game ASAP (with a common target date being 'yesterday'), there is no way around it.

That's why (unfortunately to us developers) we will inevitably be pushed to deliver our game ASAP (with a common target date being “yesterday”), there is no way around it. If leaving this without proper attention, it will inevitably lead to a horrible rush at the end, dropping essential features (while already a lot of time was spent on non-essential ones), skipping most of testing, and to a low-quality game in the end.

Dealing with this time-to-market problem is not easy, but is possible. To avoid the rush in the end, there are two things which need to be done.

The first such thing is defining a so-called Minimum Viable Product (a.k.a. MVP). You need to define what exactly you need to be in your first release. The common way to do it is to do about the same thing which you're doing when packing for a camping trip. Start with things-which-you-may-want-to-have, which will make your first list. Then, go through it and throw away everything except things which are absolutely necessary. Note that you may face resistance from stakeholders in this regard; in this case be firm: setting priorities is vital for the health of the project.

The second endeavour you need to undertake to avoid that rush-which-destroys-everything, is as much obvious as it is universally hated by developers. It is planning. You do need to have schedule (with an appropriate time reserves), and milestones, and more or less keep to the schedule. A bit more on planning will be discussed in Chapter [[TODO]].

⁴ this is not to mention that you can simply run out of money for the project

Limited-Life-Span vs Undefined-Life-Span Games

One of the requirements for your upcoming game is extremely important, but is not too-well known, so I'll try to explain it a bit. It is all about projected lifespan of your game. As we will see further down the road, game lifespan has profound implications on the game architecture and design.

Starting from the times of the Ancient Gamers (circa 1980), most games out there were intended to be sold. It has naturally limited their life time (for one simple reason: to get more money, the producer needed to release another game and charge for it). This is a classical (not to say it is necessarily outdated) game business model, and massively multiplayer games which are intended to have a limited life span, share

quite a bit with traditional game development. In particular, limited-life-span games are normally built around one graphics engine. Moreover, very often such an engine is very tightly coupled with the rest of the game.

As games were developing from Ancient Gamer Times towards more modern business approaches, game producers have come up with a brilliant idea of writing a game once and exploiting it pretty much forever. Therefore, these days quite a few multi-player games are intended to have a potentially-unlimited life-span. The idea behind is along the following lines:

“Let’s try to make a game and get as much as we can out of it, keeping it while it is profitable and developing it along the road”

Indeed, games such as stock markets, poker sites, or MMORPGs such as World of Warcraft are not designed to disappear after a predefined time frame. Most of them are intended to exist for a while (providing jobs to developers and generating profits to owners), and this fact makes a very significant difference for some of the architectural choices.

Most importantly, for unlimited-life-span games, there is a risk with relying on one single graphics engine. If your engine is not 100% your own, a question arises: “Are you 100% sure that the engine will be around and satisfy your crowd in 5-10 years time?” This, in turn, has several extremely important implications, shifting the balance towards DYI (see Chapter [\[TODO\]](#)) and/or going for ability to switch the engines, severely reducing coupling with the engine (this will be discussed in Chapter [\[TODO\]](#)).



“Indeed, games such as stock markets, poker sites, or MMORPGs such as World of Warcraft are not designed to disappear after a predefined time frame.

Your Requirements List

So, after reading⁵ all the stuff above about “how to write your business requirements”, you’ve finished your business-requirements session, and got your list of business requirements for your game. While your list is unique for your game, there are some things that need to be present there for sure:

- A very detailed description of the user experience (including game logic, UI, graphics, sounds, etc.). This is what is often referred to as a “Game Design” document; it is going to take most of your Business Requirements document, but it is game-specific so we cannot really elaborate on it here. However, there are lots of much-less-obvious (and MMO-specific) things which need to be written down, see below.
- Game projected life span (is it “Release, then 3 DLCs over 2 years, and that’s it”,

or “running forever and ever – until the death will us part”?). For further discussion, see [[TODO]] section above

- Do you need some kind of “invite your Facebook friend” feature (or anything similar)?
- Is your game supposed to be 3D or 2D? Note that at least in theory, dual 2D/3D interface can be implemented, especially for those games with “forever” lifespan
- List of platforms you would like to support for the client-side app
 - List of platforms you want to support in the very first release
 - Note that the list of platforms for the server-side is normally an implementation detail, and as such doesn’t belong to the business requirements. Neither do programming languages, frameworks etc.
- Timing requirements (how long it should take for the player to see what is going on). These are very important for our network programming, so you need to insist on specifying them. “As fast as possible” is not really useful, but “at least as fast as such and such game” is much better (if you can get “at most X milliseconds delay between one user presses a button and another one sees the result”, it’s even better, but don’t count on getting it).
 - closely related to timing requirements is a question about your game being “synchronous” or “asynchronous”. In other words, do you players need to be simultaneously online when they’re playing?⁶ At least most of the time, fast-paced games will be “synchronous” (it doesn’t make much sense to play MMOFPS via e-mailing “I’m shooting at you, what will be your response?”), while really slow-paced ones (think chess by snail mail) will be “asynchronous”.
- What types of connection do you need to support? Do you need to support dial-up (hopefully not)? But what about connection-over-3G? What about working over GPRS?
- What is your target geographical area? While “worldwide” always sounds as a good idea, for some very-fast-paced games it might be not an option (this will be discussed in Chapter [[TODO]]). Also considerations “when most of the players are available” can affect some types of gameplay too (for example, if in your game one player can challenge another one, with a loser losing by default, you most likely will need to have “time windows” where such challenges are allowed, with timing of these “time windows” tied to real-world clock in the relevant time zone).
- Are you planning to have Big Finals shown in real-time to thousands and hundreds-of-thousands of observers? *NB: we’ll see why it is important technically, in Chapter [[TODO]]*
- Do you need to implement i18n in the very first release or it can be postponed?
- Client update requirements. There is a part of it that is

i18n
Internationalization
(frequently
abbreviated as
i18n) is the
process of
designing a
software
application so
that it can
potentially be
adapted to
various
languages and
regions without
engineering
changes.

— Wikipedia —

(almost) universal for all the games: “we do need a way to update the client automatically, simply when the player starts the app” – still, make sure to write it down. However, there are two more subtle questions:

- is it acceptable to stop the game world at all while the clients are being updated? How long this stop-the-world is allowed to take?
- Is it acceptable to force-update client apps (or at least not to allow playing with an out-of-date client)?
 - If not – for how long (in terms of “months back” or “versions back”) do you need to support backward compatibility?
- Server update requirements. Most of the server-side stuff qualifies as “implementation details”; however, whenever the server is stopped, it’s certainly visible to the players, so “how often we need to stop the server for software upgrades” is a perfectly valid business-level question. Is it acceptable to stop the game while server being updated? How often are server updates planned? With the game being multi-player, stopping and then resuming the game world may become Quite a Pain in the Neck for players. However, allowing for server updates without stopping the game world can easily become a Much Bigger Pain when developing your system (see [[TODO]] chapter for some hints in this direction), so you need to think in advance whether the effort is worth the trouble. Unless a non-stopping server requirement is Really Significant business-wise – you may want to try dropping it from the requirements list, and explicitly say that you can stop the server once-per-N-weeks (and also whenever an emergency server update is required) to update server-side software (where N depends on the specifics of your game).
- In-game payment systems which *may* need to be supported in the long run (these have implications on security, not to mention that you need to have a place for them within your architected system). Even if it is “the game will be free forever and ever”, or “all the payments will be done via Apple AppStore” – it needs to be written down. Oh, and if it is “all the payments will be done via Apple AppStore” *and* there is a “Windows” in the list of the platforms to be supported – there is a likely inconsistency in your requirements, so either drop “Windows”, or think about specific AppStores for the Windows platform, or be ready to support payments yourself (which is doable, but is a Really Big Pain in the Neck, so it’s better to know about it well in advance).



**“Even if it is
“the game will
be free forever
and ever”, or
“all the
payments will
be done via
Apple AppStore”
- it needs to be
written down.**

As we will see later, we will need all these things to make decisions about network architecture. It means that if your list is missing any of these – you need to go back ~~to the drawing board~~ to the meeting room and get them out of the project stakeholders.

⁵ And hopefully agreeing with, as blindly following the advice won’t get you far enough

⁶ I don't want to go into lengthy splitting-hair discussion whether this property should be named "temporal" or "synchronous"; let's simply use the name "synchronous" for the purposes of this book

[[To Be Continued...



published]]

This concludes beta Chapter I from the upcoming book "Development and Deployment of Massively Multiplayer Games (from social games to MMOFPS, with social games in between)". Stay tuned for beta Chapter II, "Games Entities and Interactions"

EDIT: Chapter II, "Game Entities and Interactions", has been

Acknowledgement

Cartoons by Sergey Gordeev^{IRL} from Gordeev Animation Graphics, Prague.

« *Contents of "Development and Deployment of Massively Multip.*

Chapter II: "Game Entities and Interactions" from upcoming boo.. »

Filed Under: Distributed Systems, Network Programming, Programming, System Architecture
Tagged With: business requirements, game, multi-player

Copyright © 2014-2015 ITHare.com