Basic Rules for Managing Software Startups

(guide for novice investors and business people)

Author: Kamil Rafikov
ORCID: 0000-0002-7828-2488
Email: mailbox@kamil-rafik.com
Skype: kamil.rafikov

DOI: 10.5281/zenodo.2653981
Updated at September 4th, 2019.

The document has been reviewed by one well-known open source developer who preferred to stay anonymous. If you want to add your reviews here too, then feel free to contact the author.

Also, feel free to contact the author of this document for all development and managerial tasks that you may be interested in.

Table of Contents

DISCLAIMER

PLEASE, CHECK THE FULL TEXT OF CC BY 4.0 LICENSE TO UNDERSTAND DISCLAIMER OF WARRANTIES AND LIMITATION OF LIABILITY. THE FOLLOWING TEXT JUST HIGHLIGHTS SOME IMPORTANT DETAILS.

ALL INFORMATION PROVIDED IN THIS DOCUMENT IS GIVEN FOR EDUCATIONAL USE ONLY AND DOES NOT CREATE ANY BUSINESS RELATIONSHIPS BETWEEN YOU AND AUTHOR. THE TEXT OF THE DOCUMENT IS WRITTEN IN AUTOBIOGRAPHIC STYLE PARTIALLY AND WITH A PRIMARY PURPOSE TO SEARCH JOBS IN ONLINE SMALL BUSINESSES; THAT’S WHY THE AUTHOR MAY ACCIDENTALLY INCLUDE INTO THE DOCUMENT SOME INFORMATION THAT IS OUTSIDE OF HIS COMPETENCE AND/OR INFORMATION THAT IS COMPLETELY MISLEADING.

THE AUTHOR IS NOT A LAWYER, AND ALL INCLUDED LEGAL INFORMATION SHOULD NOT BE CONSIDERED AS LEGAL ADVICE.

IF YOUR PURPOSE IS TO DEVELOP SOFTWARE RUNNING OUTSIDE OF WORLD WIDE WEB, SOFTWARE CONTROLLING HUMAN SAFETY, SOFTWARE THAT PROCESSES FINANCIAL TRANSACTIONS IN AMOUNT NON-COMPARABLE WITH TYPICAL MICROENTERPRISE TURNOVER, AND SOFTWARE PERFORMING OTHER TASKS OF SIMILAR IMPORTANCE, THEN YOU ARE STRICTLY ADVISED TO AVOID RELYING ON THIS DOCUMENT AND TO OBTAIN CONSULTATIONS FROM THE QUALIFIED PROFESSIONALS.

HISTORY OF CHANGES

Almost all document has been written since June 2015th till January 2017th through multiple small fixes, sometimes one phrase in size each. After that it has been abandoned in semi-completed state. However, in August 2019th, it has been reformatted for more comfortable reading. Minor fixes in content has been performed together with reformatting, but the most part of the document still reflects author's level of professional qualification by 2015-2016th years.

Then the document has been reviewed by one open source developer well-known in professional circles and having experience of work in successful long-term projects. He asked to avoid using his name here; that's why he is mentioned everywhere as *Anonymous Open Source Developer*. Considering that some of his comments are outside of boundaries of my competence, it has been decided to leave them in **blue boxes** instead of merging and integrating into the main text.

After that, in September of 2019th, an appendix with recommendations for software developers has been added.

INTRODUCTION

This document has been written by me for my clients/investors/employers who have minimal or no understanding of startups management (and generally software development project management). For example, former designers, scientists, or businessmen coming into software development from rather different fields. It has only ***introductory*** purposes and does not pretend to cover all aspects of project management and software development. Moreover, it aims to cover technical and organizational topics only with paying minimal attention to the business side of startups management.

Generally, all management-related notes listed in this document is a mix of several software development methodologies and practices that is considered by me as the most comfortable and most efficient project management style for myself solely. That's why all this may be unacceptable in your situation. In any case all listed rules is not a dogma even for myself but just a set of continuously changing recommendations, no more.

And moreover, I wrote this document from several points of view simultaneously and gradually within a long period of time. As "developer-employee", "creator", "managerial employee", "businessman-employer", "scientist", and "science amateur". That's why some expressed ideas and opinions may look non-systematic, amateurish, or even ridiculous. It's the life. This document just reflects different parts of my professional life. But at the same time, this approach will allow you to look at the topic in more objective way.

The most part of my work experience collected since the beginning of 2000's consists of web-based startups development. However, I participated in the most part of these projects at the initial phase only. That's why by 2019th year I still have small experience in product polishing, zero experience in test-driven development, and only general understanding of development in highload projects. I

have production experience, but only in small projects. And I never participated in really well-designed development process in the way, as it is done in large corporations. However, multiple projects I worked on were rather large, up to several hundreds components in size each. And I worked in all possible environments (from remote home-based technical support up to in-office corporate environment), all possible roles (from cheap low-qualified outsourcer up to manager-architect level), and all popular technological stacks (from raw C++ up to HTML/CSS). That's why I have enough experience to provide professional recommendations. I just want to highlight here that my main strength is working with large (sometimes "dirty") codebases in the first phase startups built by geographically distributed team with unclear product requirements for the reasonable budget. If you want to get consultations for a similar project, you are reading exactly what you need.

The term "startup project" is used by me in a rather wide meaning and describes the project that meets all or the most part of the following criteria:

- development of new software from the scratch;
- absence of strict requirements and regularly changing (sometimes chaotically changing) requirements for the developed product at any stage of development process ;
- very limited budget;
- management team with limited experience in the business field of the company (and sometimes with full absence of even theoretical background in this field);
- tendency to try innovative ideas, emerging markets, and fresh tools in all parts of the business and development processes.

All listed rules are applicable to projects with a length from 2 up to 12 months (calendar months, but not human-months!) of development performed by teams with a size from 1 up to 10-20 persons. For smaller projects some of rules may be skipped. For larger projects you will need to read professional literature about project management. (However, larger projects may be broken into several phases of active development of new versions and phases of routine support between them. In such case this document is applicable to phases of active development too.)

Such temporal limits are given due to the following reasons:

- in according to my experience, significant changes of requirements happen usually after approximately 2 months of work;
- projects that take less than 2 months can be successfully implemented and managed even by non-experienced persons due to relatively low complexity;
- 12 months is the period when "developers-stayers" (see definition of the term in "Hiring" section) usually change the employer, so you have a significant risk of losing all your initial development team;

- 12 months is the period when business/financial environment is changed significantly, so your initial product concept may meet natural obstacles and needs to be significantly revised.

I use the term "manager-architect" in many places in the document (because significant part of the text is written by me from the point of view of a candidate for this role/position). The term is used to describe clearly work responsibilities of a person who performs the appropriate role in the company. In different companies a position of a person performing this role may have such titles as "project manager", "tech lead", "software development manager" etc. You should understand the difference between *role* and *position*. Usually, only single person holds such official position in a small software development company. However, the role of manager-architect should be performed together by several persons in the team: one who holds that official position, several most experienced developers in the team, and several external part-time consultants. You should strictly avoid situations when the role and position of manager-architect overlap completely and held by single person only, because it may cause complete failure of the project. Further in this document the term "manager-architect" is used in both forms singular and plural; however, mainly it means a group of the most experienced persons in the project, but not a single person. Also, this term is used to highlight significance of some members of your development team and give you understanding that they are not "just technical staff". (Note, that term "architect" has completely different meaning in software development business in comparison to construction business. The architecture design activity is not isolated from the work of junior technical staff and even junior technical staff should have some skills in designing architecture.)

Other important details:

- as I noted already, the most part of this document is devoted to working with a remote geographically distributed team of developers; in office environment many tasks should be done in different way;
- in text of the document, I use slang terms like "junior", "mid-level", and "senior"; these terms just highlight relative experience of developers inside particular team, and do not correlate with their age or intellectual/educational background;
- the term "engineer" that is actively used in software development job postings in the modern world does not correlate with real engineering tasks anyhow; it is just buzz word used by hiring managers; in this document, I use the term "engineer" exclusively for people who actively work with hardware;

> **In my experience, it does and a lot. There is a huge gap between engineer and coder though.**
>
> ***Anonymous Open Source Developer***

- as you can guess, “Technologies and tools” section becomes outdated very quickly; much faster than any other section of this document; that’s why check links in “Useful online resources and books” section to get the most recent technological reviews.

Inaccuracies and inconsistencies:

- this document has been titled “Basic Rules for Managing Software Startups” but in fact the most part of text is devoted to development of software in general and particularly to development of web software; however, in the process of final edits, it has been decided to keep original title intact due to personal and technical reasons.

There are alternative ways to develop software and manage projects that are not covered in this document. Moreover, they partially contradict the described approach; so, it is recommended to avoid combining them with the most part of listed recommendations. I will list just some of these ways here for your information.

- If you cannot or if you don’t want to hire the best developers on the market, or if you cannot manage them efficiently, then you should pay attention to test-driven development concept: https://en.wikipedia.org/wiki/Test-driven_development and start to use it since the first days of the project.

> **Do you assume the best developers do not make mistakes or do not need tests? If so, that is plain wrong. In my experience, tests reduce long-term complicated project development significantly.**
>
> ***Anonymous Open Source Developer***

- If you are a person with completely creative background and you cannot perform long-term planning in your regular work (or have not a habit of performing such planning), then you should rely on hiring several external part-time consultants that will validate work of all team members and will guide you through difficult situations.
- Some companies (like Valve, search “Valve management model” online) use organizational model of open source community in commercial environment: developers work without management and they do all (or almost all) they really want to do. Such style of organization of development may be very beneficial in case if you plan to develop something that meets the following criteria: creativity is extremely important for reaching the business goals; the developed product is “something cool” that may inspire a lot of people to put their energy into work every day without any direction from company’s management; you are ready to rely completely on developers in determining where your business will go. Obviously, in such case you should hire only developers with noticeable background in open source development in a relevant field.

SOFTWARE DEVELOPMENT IN YOUR BRAIN

Before writing organizational and technical stuff I would like to describe quickly how the process of software development looks from the point of view of the human brain. Such description will give you a better understanding of the organizational sections of this document.

1. It is possible to say that software design process for the complex product consists of 3 parts:
    - defining top-level conceptual design of the systems and processes that is usually represented in schematic graphical form; this task is purely engineering task, business analysis task, and graphic design task;
    - defining new project-specific and product-specific metalanguage (see https://en.wikipedia.org/wiki/Metalanguage) that consists partially of terms from the used business domain and partially of basic constructions of the used programming languages and other technologies; this part of the work has more relation to linguistics although significant technical knowledge and business knowledge is required; note, that usually the term “metalanguage” is not used by specialists in the process of writing specifications and basic structures in code, but it is exactly what is done from the brain's point of view;

> **Usually, it is called "ubiquitous language".**
>
> ***Anonymous Open Source Developer***

    - compiling organizational rules that are required to be followed for the correct implementation of the conceptual design and correct using of the developed metalanguage; this task requires general project management skills.

> **The term “team leading skills” fits better. Project management is close and often these roles overlap.**
>
> ***Anonymous Open Source Developer***

2. The process of development consists of translating the conceptual design to code with use of the developed metalanguage. This activity is more similar to the work of translator or creative writer than to any engineering work.
3. The process of quality assurance involves analysis of the written code (which is in fact the text written in metalanguage) and analysis of behavior of the completed system. It requires skills of researcher and editor in conjunction with technical knowledge and some artistic taste.

> **I don't think so. There are strict rules you can apply to make an opinion about technical debt level and overall correctness of the software developed.**
>
> ***Anonymous Open Source Developer***

So, here is the summary of relatively basic intellectual activities involved in the process of the software development in the large project:

- business domain analysis in the general meaning of the word "business" (an activity that is completely natural and innate);
- engineering, although in many cases it is not real engineering but just creation of tools (which in its turn is an activity performed in basic form even by apes, that's why it may be considered as partially innate);
- graphic design (it is an activity that was practiced even by humans of the Stone Age when they decorated walls of their caves);
- creation and analysis of languages (set of skills that are innate for every human);
- project management (hunting a herd of mammoths or any other large game is a much more complex project than development of a CRM system, but such projects were successfully performed regularly tens of thousands years ago);
- translation (an activity that was practiced even by early humans in the process of contacting neighbor tribes);
- creative writing (an activity that is practiced by humans within the latest several thousands years);
- editing (an activity of the same age as writing);
- research (an activity that is completely natural and innate).

You should understand that the order of actions listed above is not strict and everything may be done in any order. And with multiple steps back and forward made from every point. In different projects each activity may take different percentage of the whole work.

As a result of the complexity in the required intellectual processes and their combinations we have the following consequences.

1. Some people may be more skilled from the birth in doing different intellectual processes and extremely unskilled in others. It explains, for example, why some great engineers may be very slow in the process of writing code and completely useless in quality assurance.
2. Even junior-level developer need to have some software architecture design skills, because all parts of work may be easily rearranged in any order at any time in the project.

> **The outcome of doing so is usually a failure.**
>
> ***Anonymous Open Source Developer***

3. Very experienced developers may have lack of knowledge of software development theory and basics of their main programming languages. Why? Because all main work is done in metalanguages that are project-specific and product-specific, only limited amount of each technology's features is used in each metalanguage, and usually some features of each technology are used in the very beginning of the project only (with a frequency once per year, for example).

> **While that is true about concrete languages, it is completely false about theory. It does not change from language to language much so developer not knowing theory basics can not be called experienced.**
>
> ***Anonymous Open Source Developer***

4. In some projects a specific metalanguage is not required at all because the whole business domain is extremely simple and/or completely covered with existing ready technologies. In other projects definition of the metalanguage may require significant intellectual and financial input and long period of onboarding for new team members.
5. Classic industrial engineering has rather weak relation to software development. (Note, that some old management literature describes situations when software projects staff was built completely of industrial engineers; but authors did not understand this. That's why some conclusions made by authors of this literature are not relevant to the modern world.)
6. Graphic design, translation, creative writing, and research are operations with a non-deterministic result. As a result of this every software contains bugs: only quantity of bugs can be decreased, but all bugs cannot be completely eliminated.
7. Graphic design, writing (and partially engineering and translation) can be considered as so-called "creative" activities. Many "creative" works and works that pretend to be named "creative" are built on ideas taken from multiple sources, through multiple copying, rewriting, merging, and rethinking of these ideas. And your workers in the process of taking these ideas may cross the border between copying ideas and copying implementation (either accidentally or with a purpose), or they just may copy patented ideas. So, it may happen that your software product is walking on the edge of copyright infringement or even crossing this edge. That's why in some cases you as a business owner should strictly check not only final results created by your workers but also origins of used ideas to be sure in their legal cleanliness, or you will get into big problems.

MARKETING NOTES

Do not start any development before doing all steps in the following list.

1. Determine 1-5 “killer features” of your future product. “Killer feature” is a feature that makes the product clearly distinguishable on the market and/or puts it ahead of competitors. (You may use the term “breakthrough feature” if the term “killer feature” sounds too gloomy for you or if it is not appropriate for your business/product. Or “main feature” if you just implement something standard.)

> **In my experience, there are successful products without killer features. They are just doing standard features better than others. Or charging less.**
>
> ***Anonymous Open Source Developer***

2. Limit target customer group of the product as narrow as possible for the first version of the product.
3. Determine clearly what is “final product” of your business: for example, it may be software that you develop, some files or visual content generated by the software, services performed through use of the software etc.
4. If your software will perform several large business functions, it is recommended to describe proportions and priorities in the following style: *“1. Project management software for 10%, 2. CRM software for 60%, 3. E-commerce software for 30%”*. It will allow developers to concentrate efforts at the most important parts. Even if software performs a single function, this function may be broken into subtasks ordered in the same style.
5. Check that every developer in the team understands completely all that you determined on the previous steps.

In other words, 500-page marketing plan is generally not required. However, such approach to marketing works only if you will hire experienced staff as described further.

PLANNING

Perform as much planning as possible. The most part of startup projects do not create anything “really innovative”. I suppose, 80/20 (or even 90/10) ratio may be applicable, where 80%-90% is the part of functionality that is well-known for everybody on the market, but cannot be obtained in a ready form through the use of 3rd party’s libraries. You can easily perform detailed planning for these 80%-90% of the work.

Documents to be written

By the beginning of the project it is recommended to have the following documents:

- general functional and technical specifications;
- low-level specifications of internal and external APIs;
- input/output formats of data for the largest functional blocks in the software;
- list of all possible technical bottlenecks and technical/organizational problems that may arise in the process of development and after launch of the product;
- logical and physical schemas of data;
- mockups of UI and diagrams of UI workflow;
- general diagram of interaction/inheritance of modules and components (some components may be grouped and only main relationships may be displayed);
- document with maximal possible numerical limits of the software for the first and all subsequent versions – number of users, number of records in the database, amount of processed data etc;

> **Worth saying that it is usually called SLA.**
>
> ***Anonymous Open Source Developer***

- document with development standards.

In the process of writing all this technical stuff don't be afraid to show to architects/developers all your business documents concerning marketing, investments, business plans etc. Because all this is critically important for giving the technical team full understanding of business goals of the project. Without full understanding of business goals developers may choose incorrect development platform, framework, libraries, language, approach etc that cannot be changed later at all. So, you should be very careful at this point.

After completing all documents listed above, your architects should break down all work into multiple short tasks with a size of 0.5-3 days each and put these tasks into task management system with grouping them by iterations and setting dependencies between them. Hiring of pure technical staff should be started only after that!

In this phase you should use as many as possible ready 3rd party's libraries, frameworks etc. Try to avoid reinventing the wheel.

Some authors of managerial literature recommend to avoid overplanning; but the fact that you have built the planning documents described above does mean that you will follow them in all details

further; in fact, the main purpose of these documents is just to polish ideas in your head and throw out completely unacceptable options.

Notes about software architecture

Please, read the following articles (with all other articles linked to them):

- https://en.wikipedia.org/wiki/Software_architecture;
- https://en.wikipedia.org/wiki/Software_design_pattern;
- https://en.wikipedia.org/wiki/Model–view–controller;
- https://en.wikipedia.org/wiki/Microservices and http://microservices.io/patterns/microservices.html;
- https://en.wikipedia.org/wiki/Representational_state_transfer.

Further I will add just some practical notes that may be useful for you.

- Many developers like the job title "architect" because it gives them a feeling of significance. But you should understand that in small software development companies (hiring less than 20 persons) up to the half of the personnel participate in making architecture-level decisions. So, this job title may be freely shared between all of them: it will be the role, but not a job title.

> **Usually, that leads to a huge mess in half a year or, if you are lucky, later.**
>
> ***Anonymous Open Source Developer***

- In a large complex monolithic application that consists of several hundreds components up to 50% of them may be implemented as plugins. In other words, they may be outsourced for the minimal rate to external junior-level developers. From the very beginning of such project you may bravely plan development of 10-20 types of plugins with up to 2-20 plugins per each type.
- In your timeframe estimates for projects with a size of several hundreds components you should include time for developing separate internal tools for building and installing/deploying the software, and time for developing utility scripts used by developers in the process of development only but not in software itself.
- If your final product consists of some output data files or if your software saves data in some files, they should be stored in well-known readable format (for example, XML, JSON etc). It may happen that in the process of defining what is your final product you may discover that developed software is not your final product, but it is just an internal tool for creating a product. Document this fact, it affects your business significantly: for example, some

developers don't like to write "internal tools", because it puts them into non-significant position inside the company (in such situation they are not "creators" anymore, but just "technical personnel"). So, you may meet obstacles in finding highly qualified people.

- Active usage of design patterns terminology by your architects/developers may in fact be a mask hiding lack of qualification. It is nice, if all architecture decisions are documented with use of design patterns terminology, it will improve readability of specifications. But it is possible to design and develop a large project successfully without knowing anything about design patterns.

> **I can agree that talking with design patterns only is a bad sign but I don't agree that without knowledge of basic design principles architecture will be OK.**
>
> ***Anonymous Open Source Developer***

- In the significant part of web-based projects Model-View-Controller (MVC) is the only architectural solution that you will notice at all. If you have in your head an idea to hire many junior-level and mid-level developers with assigning to them directly tasks to rewrite many parts of software in a non-thoughtful manner, then it is better to keep all code in MVC only with minimizing the number of abstractions. Yes, such approach may cause too large amount of duplicating similar code or even dirty copy-pasting. But it will allow you to avoid bigger problems, and will give you the feeling of full control over the development process.

> **Feeling of full control, but not control. :)**
>
> ***Anonymous Open Source Developer***

Example of gradual design of software architecture

An example of gradual design of software architecture in complex web application performed simultaneously with development may look like the following (with spending up to several months for each phase).

1. One or several basic MVC applications and one or several basic services are designed and developed. It may look that such work can be performed even by junior-level developers. And it is true, but only in case if we will stop at this phase and will not go further! Because if we go further, some extremely important decisions need to be done even on this step.
2. Functionality of each application and service is extended in different ways. All logic developed at this phase is placed into multiple components grouped just by different folders with minimal isolation from each other. Amount of "dirty" code and incorrect architectural solutions may grow up significantly. Don't be afraid of this, just document future solutions for each "dirty" place. Active commenting in "todo:" style and writing documentation even

in the format of messy text files will allow to save the situation with minimizing the unnecessary work.

3. Different parts of code are either grouped into isolated modules or extracted into plugins. All APIs of plugins and services are stabilized and documented. It is the period of maximal refactoring and documenting, and the growth of software functionality is minimal. But you should never skip this phase! It may happen that all development from the phase #1 up to the end of the phase #3 is performed just by 1-3 persons whom you hired in the beginning of the project.
4. The system is significantly extended with use of cheap junior and mid-level developers who develop additional plugins, services, and MVC inside the stabilized architecture. It is the period of maximal growth of software and development team. By the end of this phase the software may reach size of several hundreds components and the total length of the project may reach about one calendar year, depending on your budget. Also it may happen that the whole initial development team has left the company already by this time, and you are working with completely new people.
5. All parts of the system are polished to make naming and software design to meet standard descriptions of different design patterns and architectural patterns, if they do not meet yet. All APIs are updated and stabilized again. In practice these actions may be completely skipped, if no significant vertical growth of the system is planned in the future. Usually significant amount of resources are spent in this phase for optimization-related architectural changes, usability-related architectural changes, and QA.

Unobvious advantages of preliminary planning

As you can understand initial planning for large project may take up to several weeks and require several thousands USD to be paid to architect. That's why many small seed investors prefer to start development immediately and perform planning "on the fly" with making "rough" version of software firstly and then polishing/rewriting it. Such style of work is more comfortable for them. Especially considering the fact that in startup environment all plans are usually changed several times in the process of implementation, and all initial planing looks useless for them. But in relatively large projects absence of planning may cause additional significant expenses.

Here are several benefits that you may obtain from preliminary detailed planning:

- you can hire 3-5 or even more developers starting from the first day because you can load them with well-described small tasks that don't intersect with each other and don't depend on each other (without planning you can hire only 1 developer at the first day and then gradually attach more people to the project after 1-2 months);
- as a result of hiring 3-5 developers together you will be able to determine easily (in 1-3 weeks) who of them is the best and who is the worst, and replace the worst developers with new candidates immediately (with gradual hiring such process of evaluating qualification of developers may be stretched up to several months and may cause additional problems);

- if 3-5 developers start working together from the first day, they are placed immediately into the equal positions so they don't need to teach each other and compatibility issues will not arise (in case of gradual team building all hired developers should be compatible with the first 1-2 developers, so it will make the process of team building/management more complicated or even impossible if your first 1-2 developers are not good enough).

> **Compatibility issues may arise even if two developers are super-great. That's not the problem with technical skills level in general.**
>
> ***Anonymous Open Source Developer***

UI/UX design

Just several notes:

- any change in UI/UX design in a ready project may require up to 10x (or so) changes in code; that's why try to stabilize UI/UX design as early as possible, or if such stabilization is impossible, try to stabilize at least the ways how you plan to change design in the future to allow developers to automate these procedures;
- considering the previous item do not look at UI/UX design refactoring tasks as "junior-level job"; sometimes it may be really so, but in many cases you may break a lot by assigning these tasks to junior developers; firstly, you should discuss with your lead developers and architects the range of tasks that may be assigned to junior developers;
- if you want to attract a design studio to design of your project instead of hiring a designer working inside the team, then be ready "to pay for the label"; studio rates and hired designer rates may vary up to 10 times (or so) for the work of the same quality.

Explanation of some complexities of software development process

You may be very interested to understand how software developed by 1 person during 5 months may become much more complex and much more large than software developed by 5 persons during 1 month. Especially in relation to project size limits declared in the beginning of this document. It may happen due to the following circumstances.

1. Communication and managerial time expenses may take up to 20%-30% of the project budget. In case of having only 1 developer in the team, these expenses may be minimized to 0-5%. So, all time will be spent for development.
2. Inconsistency of internal documentation has a tendency to grow over time. Even if the best efforts are applied to keeping the documentation valid. That's why after 5 months you will have worse documentation than after 1 month even if amount of developed code in software is comparable.
3. The risk of changes in the product concept (with appropriate significant changes in software code and architecture) caused by market influenced circumstances or decisions of top

managers corresponds to the length of calendar period of software development. So, during 5 months you will definitely have 1-3 significant changes in product concept that may cause many inconsistencies in the developed code.

4. Lack of qualification of 1 person has higher influence on software quality. Also, a person working on software alone may have a tendency to make more risky technical decisions and to use more exotic tools than a team of 5 persons.
5. As a result of all items listed above, the number of abstraction layers in the longer project may be higher in each particular place in software, and the number of places where multiple abstraction layers are used may be higher too.

And as the final result of all this, the software written by 1 person during 5 months may be 1.5 times more large (which is not bad) and 2-5 times more complex (which may be not very good sometimes) than software written by 5 persons during 1 month.

> **That easily could be the opposite.**
>
> ***Anonymous Open Source Developer***

However, I think you understand that paying, for example, 5,000 USD per month to one person during 5 months is much more comfortable financially than putting on the table 25,000 USD at the first day for the team of 5 persons. That's why in many cases you may prefer to work with single developer than with a team on initial steps of the project development.

How to estimate the budget?

It is difficult (or maybe even impossible) to give you a universal algorithm for estimating the budget of your project. However, it is possible to give you some values for several variables in the equation that you may use as a general guidance.

All values that are provided further are given for the code of rather high quality developed with use of “good enough” approach by senior level staff with a cost of 30 USD/hr on the international market of remote work. Also, please, note that all values are based on my personal experience and on data obtained from informal communication with other professionals. That’s why some things may be completely incorrect.

Any system may be broken down into multiple functional components; some functional components may correlate with components finally implemented in code, but it will not be always so. These components can be roughly classified into 3 groups (number of lines is given without comments, only for pure code and spacing):

- small, 50-100 lines of code per each;
- mid-size, 300-800 lines of code per each;

- large, 1000-1200 lines of code per each.

> **That is very bad estimation technique:**
> **1. You have no idea of the number of lines required to build a component before it is built.**
> **2. Writing 1000 lines could be fast and easy and could be slow and hard. Depends on what these 1000 lines are doing and the experience / expertise of the one doing the job.**
>
> **Components consisting of 1000-1200 lines should not exist.**
>
> ***Anonymous Open Source Developer***

Gaps between ranges of sizes have been left because in practice all components tend to fit one of each these groups instead of being somewhere between of them. (It is just my personal opinion.)

For each of these groups of components you may take the following rough estimates for development:

- 0.5 day for each small component;
- 2 days for each mid-size component;
- 5 days for each large component.

One day is equal to 6-8 hrs of developer's work (all work, without breaking it down into coding, design, communication etc).

For the components that represent automated tests you may apply the coefficient 0.6-0.8 to the aforementioned estimates.

> **That is not true. If we consider component "done" when it works and works correctly, the number of iterations is less when using tests right from the start.**
>
> ***Anonymous Open Source Developer***

After this code has been developed you may bravely take into account the following estimates for the quality assurance (the term “easily detectable” used further means “easily detectable in code review, but not in standard development work”):

- each 1-3 components contain 1 easily detectable minor bug that takes 0.5-4 hrs to be fixed;
- each 5-10 components contain 1 easily detectable major bug that takes 2-8 hrs to be fixed;
- about each 20 components contain 1 place that strictly requires serious refactoring for better speed, usability, scalability etc, such refactoring takes 4-40 hrs.

“Easily detectable bug” means a bug that can be found upon quick UI review performed by non-qualified personnel or code review performed by experienced developer. In other words, a bug that does not require automated tests for being detected.

For each of the following situations you may apply to the pre-calculated estimates the coefficients with a size from 1.2 up to 3 each:

- exploring legacy code on the fly;
- onboarding new team members into the project with large amount of already developed code;
- learning new business domain on the fly;
- using new technology/tool;
- communication (above standard work daily communication);
- stressful situation;
- loss of motivation caused by personal or work-related issues.

Obviously, all listed above is given for the technical side of your project mainly. Organizational issues may add many more variables into the estimation equation.

LEGAL NOTES

Licensing and copyright

Many startups are initially driven by the idea of cloning some existing products available on the market. (Some of them even market themselves in this way, like *“We are doing ‘Label A’ + ‘Label B’.”*) But it may cause copyright-related issues and large penalties in the future. Even if in fact cloned functionality does not contain something really valuable. Punishment for copyright infringement in US may consist of penalties with a size of several hundreds thousands USD or even imprisonment! Note, that in terms of service of some software products/services you may find an item that you cannot even look at this software if you have in mind plans about competing with it (surprise! did you ever read terms of service?).

Check legal cleanliness of all 3rd party’s libraries used by developers. Generally speaking, LGPL, MIT, and some other licenses are fine to use in a commercial project.

If you create a lot of graphical 2D/3D stuff, please check that artists do not perform scanning/copying of existing 3rd party’s works, because it may be considered as copyright infringement.

Also check legal cleanliness of all 3rd party's 2D/3D stuff that may be loaded by developers into the project for testing purposes.

Please, note that in many poor countries developers and designers hardly understand the concept of copyright at all. So, you may need to educate your personnel.

Patents

For the beginning the only you should know is the following:
https://en.wikipedia.org/wiki/Patent_troll.

Contracts

All your contracts with clients, subcontractors, and employees should contain the following sections. Details of content for each section you may obtain from your lawyer.

- Definitions in the context of the agreement.
- Explanation.
- Scope of services.
- Payment.
- Acceptance.
- Indemnification.
- Termination.
- Ownership of intellectual property.
- Confidential information.
- Miscellaneous.
- Warranties.
- Limitation of liability.
- Force majeure.
- Arbitration.
- Signatures.

Key phrases and terms used in the contract that you should pay the maximal attention to:

- "*services* for development of products, but not *products*";
- "work made for hire";
- "moral rights";
- "substantial conformance".

Non-disclosure agreements and non-compete agreements may be required in some circumstances.

REPOSITORIES

Software should be stored in repository of version control system in the process of development. There are many hosting services of version control systems. Some of them have additional features like task tracking, internal wiki, auto-deployment etc. Other may provide only basic functionality. It is recommended to check the following (just register a free account there and play with features to understand what you will receive):

- https://gitlab.com
- https://www.assembla.com/
- https://bitbucket.org/

It is recommended to use only Git as version control system; you may easily find different Git GUI clients for Windows OS. You, as a business owner, need to know only basic operations for downloading code to local machine to be able to check the work performed by developers (although in fact it can be performed online through web interface too).

It is recommended to store in repositories not only code, but documentation too. So for any relatively complex project you may need several repositories:

- repository with code;
- repository with source code of 3rd party's libraries/frameworks (usually they contain many files, sometimes projects are linked to specific versions of libraries/frameworks, and sometimes in the process of development you may customize them slightly, so it is better to keep all source code of this 3rd party's software by yourself);

> **Why? You can get all that from official repositories.**
> **Moreover, using forks will make development a hell.**
>
> ***Anonymous Open Source Developer***

- repository with internal documentation (in case if hosting does not provide built-in wiki tools or if you don't like them);
- repository with end-user documentation (it may be required even in case if internal wiki is present, because end-users may need documentation in different format, and it is written by technical writers who don't need access rights to other repositories);
- repository with notes and different auxiliary stuff, like templates from designers, unused old pieces of code, distributives of 3rd party's libraries used in the project;
- repository with secret data that should be available to several trusted persons in the team only (for example, full database of CVs collected in the process of hiring, some important

business documents, logins/passwords to some resources etc); note, that in fact it is not a really secure way of keeping this stuff, so, please, consult with your system administrator and your lawyer; also you should know that laws of some countries, like Russia, prohibit collecting personal data in databases even for foreign business entities; so, by keeping collection of CVs of Russian developers in repository, you may break Russian laws (please, talk to your lawyer about it);

- repository with plugins (optionally, for example in case if you want to keep main code of software in secret with giving access solely to local development team and outsource development of plugins to foreign freelancers).

If your software consists of several applications/services you may keep each application/service in a separate repository. Also you may separate documentation of plugins APIs from other documentation.

DOCUMENTATION

It is strictly recommended to store in internal documentation all business documents too. In such case all developers will have the correct vision of the project goals. You cannot even imagine a size of the problems that may happen if your lead developers do not know business goals of the company!

Documenting everything is a very good habit. In case if you will lose all your core team members and/or you will want to sell your project due to lost interest, you will not be able to do anything without internal documentation. The quality of internal documentation should be regularly checked by external part-time consultants once per 1-4 weeks. Code of the large project does not cost almost anything without internal documentation!

If you don't have or don't want to use wiki tools built-in into repository hosting, you may pay attention to some tools that allow writing structured documentation with a possibility to generate output in HTML, PDF, and other formats; for example, http://sphinx-doc.org/.

Here is a recommended minimal set of sections for internal project documentation:

- rules for writing internal documentation;
- package of initial business documents and technical documents (listed in "Marketing notes" and "Planning" sections);
- list of non-obvious technical solutions implemented in the project in the process of development after initial planning period;
- list of the most fragile, the most costly, and the most transparent parts of the software where each change may cause significant consequences;
- local and server-side software installation instructions for developers;

- system administration documents (listed in “System administration” section);
- list of links to all repositories used in the project;
- debugging instruction for developers;
- basic manual QA instruction for checking status of “killer features” only;

> **All test cases for manual testing should be written down. Not only killer features.**
>
> ***Anonymous Open Source Developer***

- documents and notes about relationships with clients;
- full list of current and former personnel of the company with all contacts;
- hiring and dismissal protocol with access control management procedures.

In addition to business-related and technical information internal project documentation may include the following:

- theoretical and scientific background of the project;
- documentation of all artistic works, if many such works are used in the project (note, that you may need to use some special inventory management software for this purpose).

Please, note that full internal documentation required for developers may contain a lot of minor technical details, including information about multiple problems in different parts of software, names of developers who left the company much time ago, excerpts from email correspondence written in free format etc etc etc. All these details may be unacceptable for presenting documentation to existing or new investors. So you may come to the requirement to create 2 versions of internal documentation – a polished one for presenting it to investors, and a “dirty” one for real daily use by developers. Such documentation may be implemented in 2 ways – either by keeping “dirty” details in separate storage or by using special formatting markup around “dirty” details that allows compiling documentation into 2 output formats (with or without these “dirty” detais).

HIRING

If you are ready to hire remote workers globally the best resource for doing it is http://careers.stackoverflow.com/. Also, you should use technology-specific job boards for each used language, framework etc. For example, https://www.python.org/jobs/ in case of using Python. Expect to receive several hundreds CVs from all countries around the world in response to any job posting that offers remote work.

It is strictly recommended to hire only experienced people for startup projects.

Who to hire: “developers-creators” and “developers-analysts”

If you came into software development business from design business (or another creative business where portfolio is something that everybody must have), one of your first desires in the process of hiring is to check portfolio of each candidate. And you may be very surprised that some candidates that look extremely experienced by CVs and by communication do not have portfolio at all!

But it is just a particularity of software development business. It does not mean that these developers are worse than developers with portfolio. It is just another kind of developers, “developers-analysts”. (I’m a typical “developer-analyst”, that’s why I started from the description of this type.)

Here is the list of advantages of “developers-analysts” in comparison with “developers-creators”:

- usually they have corporate or research background with appropriate skillset that can be obtained in corporate/academic environment only;
- usually they are more reliable in long-term perspective (like any person with employee psychology);
- they may have experience of working on larger projects than “developers-creators”;
- they may have background in some fields that are not related directly to software development but may be extremely useful for your project depending on circumstances.

And here is the list of advantages of “developers-creators” in comparison with “developers-analysts”:

- they may have very fashionable skillset;
- they may operate and communicate with you like subcontractors but not like employees;
- you may check some of their skills by naked eye immediately before hiring, even without having knowledge of development;
- they may be better coders than “developers-analyst”.

Who to hire: “developers-engineers”

In addition to creators and analysts, it is possible to describe a completely different type of developers - “developers-engineers”. Engineering skills together with analytical skills and creative skills form the triad of skills required for every person working in software development business. In different companies and in different projects an accent in the work may be shifted to any of these three kinds of skills. I describe “developer-engineers” in a separate section because in many standard business projects they are either not required at all or required only after the project has been scaled to giant amount of customers.

Here are particularities of “developers-engineers” in comparison to analysts and creators:

- they may have background in system administration or industrial engineering;
- very often they work in fields of software development that require low-level programming of hardware or network infrastructure;
- mission critical systems that may cause multi-million financial losses or may affect human safety usually rely on complex hardware systems and custom code that directly operates with this hardware; also they require more formal approach to the software development process that is based on traditional industrial engineering; that's why you may often meet “developers-engineers” in mission-critical projects;
- they may have very exotic and significantly outdated skillset from the point of view of typical startup developer, because many fields where “developers-engineers” work may require very narrow specialization and may rely completely on ancient well-tested technological stacks.

Who to hire: sprinters, stayers, and permanent workers

Also, all software developers may be classified by the following types:

- “permanent workers”, who can work on the same project full-time for many years;
- “stayers”, who can work on the same project full-time for no more than one year (or up to 2 years part-time);
- “sprinters”, who can work on the same project full-time for no more than several months.

As you can guess “permanent workers” prefer to work in a stable corporate environment. So, in a startup or studio environment you may expect yearly rotation of the whole team (or the most part of it), including architects. That’s one more reason why writing documentation is so important!

> **If that's a startup, yearly rotation of the whole team means that it is doomed.**
>
> ***Anonymous Open Source Developer***

“Stayers” can be used for more than 1 year in case if after the first year you are ready to give them a completely different project using a new stack of technologies or managerial/design-related role.

Who to hire: developers with open source background

Some companies prefer to hire only developers with significant open source background: https://en.wikipedia.org/wiki/Open-source_software. As you can further understand, generally it is just a subtype of “developers-creators” with the following particularities:

- in addition to creative skills they are also driven by the ideology of open source movement;
- their open source background may consist of multiple minor tasks (performed together with other people) in large open source projects but not not of complete products made by themselves solely.

I never hired such developers, and I'm not one of such developers. So it is difficult for me to provide qualified opinion in this topic. However, I can provide a list of possible pluses and minuses of this approach.

Pluses:

- Performing minor tasks in large open source projects is very similar to team work in large commercial projects. So, open source experience in such projects may demonstrate not only creative skills but also skills in team work.
- If developer's open source projects intersect with main field of your company's business you can easily take developer's projects for free (certainly, if license allows) and include them into your company's product with paying to him/her only money for integrating his/her open source work into your product. However, I suppose that in fact generally the price for integration will be rather high; so, with this approach you will only eliminate risks related to development of completely new code by taking ready one.

Minuses:

- Similarity between minor participation in open source projects and standard work has a negative drawback. Every developer has limited amount of energy that he/she can spend for development (usually no more than 30 hrs per week for a long term period), so if the best part of this amount is spent for open source work (which is usually preferred by such people), the paid work may suffer (it may be performed too slowly or with a lower quality). So, it may happen that you have hired just a typical mid-level developers for a higher rate.
- I suppose, that open source developers may tend to over-polishing their work and to being very choosy in using only the most fashionable technologies. All this may be not suitable for standard commercial projects. Also, I suppose that such hiring approach allowed to filter the best developers in the second half of 2000's and the first half of 2010's. But gradually (when this approach became widespread) more and more novice developers started to participate in open source projects, even without having real passion for doing it. That’s why now presence of open source background in CV does not tell much about a candidate.

> **Tendencies to over-polishing and being choosy are typical for many developers. I can't say that open source developers are different in these regards.**
> **In case of finding open source background in CV, you actually have to check contributed code.**
>
> ***Anonymous Open Source Developer***

Current rates of developers on international market of remote work

- 15-25 USD/hr for mid-level developer.
- 25-40 USD/hr for senior-level developer or architect.

Filtering of candidates

Here is the best and simplest approach for filtering the best developers. You can do it even without having minimal technical background!

1. Check readability of developer's CV. If a person cannot write clearly 2 pages of text, giving him/her coding work is not a good idea.
2. Check that person has educational awards, degree in Computer Science (Physics, Mathematics, Life Science and any Engineering degree are fine too), knowledge of more than 2 foreign languages, experience of active participation in open source projects, online commercial portfolio. Having any of these items is a good sign. Ideally, at least 2-3 items should be present.
3. Check that person has experience with all required technologies by checking presence of names of these technologies in his/her CV. Note, that different technologies have different complexity, and if a person had experience of working with technologies *X* and *Y* that are more complex than technology *Z*, then this person may be suitable for working with *Z* technology too. So, if hiring is performed by technically proficient managers, it may be possible to extend the circle of candidates by hiring developers with background that does not match positions requirements exactly.
4. Ask number of components in the largest project developed by this person. Ideally, it should be more than 100. If number of components is not a good metric in your case, you may ask about other numeric attributes of the largest project – number of users, size of databases etc. (Yes, people may lie, but with the first 3 steps you have limited the pool of candidates with rather smart and intelligent persons who usually tell truth.)
5. If you want to limit rates in your company with the range of 15-30 USD/hr, then usually only about 10 of 100 candidates meet criteria listed above. However, such filtering is not enough. Further you need to do one or both of the following:

- give to each developer a test task (see the section “Test tasks” to understand what types of test tasks exist);
- ask each developer to send you samples of code from some of his/her previous projects: ideally at least 100 KB in several files; you can check the quality of code even if you don’t know programming language that was used – you should just check that there is a lot of comments, that all naming of classes/methods/variables is done in readable English to make sense of code clear, and that methods have reasonable size of no more than several hundreds lines each without too deep nesting of logical blocks.

> **Much comments may mean that code is overcomplicated. Also, comments may be useless. That's a very bad sign.**
>
> ***Anonymous Open Source Developer***

After doing all this you may be rather sure that you have found the right person. Usually only about 3 of 10 prefiltered candidates are able to pass the step #5.

Obviously, if you need to hire junior or mid-level candidates then you may skip the steps #4 and #5, and also, filter candidates at the steps #1, #2, and #3 less carefully.

Different managers use different approaches in hiring remote workers:

- some prefer to hire people from their own geographical/political region only (to guarantee legal protection against theft of intellectual property and to be able to control developers with informal relationships);
- some prefer to hire people from some specific region only even if they by themselves are not from this region (this approach is based on the assumption that people of the same ethnic/cultural/political origin will find a common language more easily and will provide better productivity in team work);
- some prefer to hire ethnically diversified teams globally with use of professional criteria only (this approach is based on the global vision of manager and on assumption that diversified teams built on professional criteria will provide solutions of higher quality).

If you will hire a team of about 5-10 developers with use of professional criteria only and plan to pay them from 15 USD/hr up to 30 USD/hr you should expect to have in the team 1-2 persons from each of the following regions listed further alphabetically:

- Brazil (and other Latin American countries);
- Middle East, South Africa, Southern Europe;
- Poland, Romania (and other Central European countries with former Yugoslavia);
- Russia, Ukraine (and other European ex-USSR countries with Israel);

- South Asia, East Asia, and Central Asia.

It will happen because currently all aforementioned regions/countries are primary sources of highly qualified and relatively cheap workforce on the international market of remote work. If you plan to hire more developers you may expect that national compound of the team may vary depending on economic/political situation in each region and size of professional networking of developers whom you have hired initially.

Many companies use so-called *technical interviews* in the process of hiring. Technical interview is a process of interviewing developers mainly with theoretical academic questions that are not directly related to ongoing work. Sometimes it also includes writing code of different rarely used algorithms exactly in the process of interview; maybe, even writing code on paper. You should understand that this way of filtering developers was invented by US-based corporations where laws required completely formal procedure of filtering candidates. But in fact this way of filtering candidates does not work well, and as it was told in one public talk with a journalist by CEO of Google, the statistics collected internally found correlation between results of technical interviews and subsequent work performance for recent graduates only. In other words, if you need to hire really experienced staff that has significant background not only in technical part but in business part too, then technical interviews is not your way. In practice, technical interviews are often used just a way to say politically correct "no" to people who meet the professional criteria but whom you don't want to see/hear to every day.

> **Presence of mainly theoretical academic questions in interview means that interview is conducted in bad way. In my experience, interviews give the best results, if they are conducted correctly.**
>
> ***Anonymous Open Source Developer***

Note, that many startups prefer to hire 1-2 developers at the beginning of the project with a desire to extend the team later up to 5-10 persons. But this idea may fail due to low quality of initial version of software developed by the first 1-2 developers and human factor (these people may start behaving egocentrically and prevent adding more team members). So, to be able to do it you may do the following:

- in the contracts with all initial developers you may include an item that they are obliged to perform active paid mentoring of all new team members or penalties may be applied (although it will not help you much in fact);
- you should hire external part-time consultants who will regularly perform checking of quality and readability of code and internal documentation, and you should fix all detected problems immediately.

In case of hiring remote developers, you should expect that the most part of them will work part-time, so you will need to hire 1.5-2 times more people than it was planned. Also it is better to expect

that even full-time workers will work no more than 30 hours per week, even if you pay them for 40 hrs per week.

Test tasks

As it was explained earlier if you have met an experienced “developer-creator”, then this person has visible touchable executable portfolio that you may play with to understand developer's qualification.

So, when do we need test tasks in such case? We need them for the following types of candidates:

- experienced “developers-analysts” who have not portfolio;
- mid-level “developers-creators” who have not created solid portfolio yet;
- junior-level developers whose psychological and professional type is not clear yet.

In fact these types of candidates may represent the majority of people who sent you CVs. You will not always have a luxury to hire only “developers-creators”. Moreover, in many cases you don't need them at all.

There are several kinds of test tasks that I have met in my professional practice.

1. Coding of pure algorithmic tasks for 0.5-3 hrs. Like those, that are provided by https://codility.com/ service. Or those that are invented by project managers of different companies.
2. Development of some functionality for use in the real world. Usually such tasks take from 3 hrs up to 8 hrs, or so. (Some non-competent or ridiculous employers may give tasks that take up to several days of work.)
3. Development of some functionality for a particular project your company works on. Such tasks take the same amount of time as tasks in the item #2. And they are usually paid.

But from my personal point of view, none of these tasks cannot be used for filtering senior-level and architect-level candidates! Why? Because these types of tasks check only basic development skills. However, almost *all* developers on the market have *some* development skills. So, by giving such tasks you will be able only to remove from the pool the worst or completely non-motivated candidates. It will not allow you to find the best people. If you will increase the length of the test tasks from items #2 and #3 up to several days of work, it may give you better understanding of candidate's skills. But it takes too much time, and only candidate who urgently looks for *any* job will agree such conditions. That’s why by doing this you also will not be able to determine who is the best.

So, what type of test task to use? From my personal point of view, the best type of test task for finding the best candidates is a task of *code analysis*. As I told above, almost *all* developers on the market have *some* development skills. But very small part of them can perform high-quality analysis of code.

Here is the way to create such test tasks. You should take large piece (or several large pieces) of "dirty" code that took several days of work to be written, consists of several thousands lines, and contains many naturally occurring or artificially introduced problems and potential problems (up to several tens problems). The candidate should find in this code all or the most part of these problems. What skills this test checks that other tests don't?

1. Skill of reading "dirty" code written by other developers. Remember, the majority of developers in the world work with code written by somebody else. Only few work with completely new code written by themselves. So, this skill is extremely necessary for any developer.
2. Skill of clear understanding what is good and what is bad. Not only on programming language level or algorithm level but also on architecture level and business level. (Certainly, if appropriate architecture/business level problems were introduced into the code.) It will allow you to detect easily whether a candidate has only coding/technical skills or he really understands "the nature of things" and "the nature of business".
3. Knowledge of some rarely used features of technologies that cannot be checked by other test tasks without extremely increasing their length. (Certainly, such features and technologies should be present in the code provided for analysis.)

Other benefits of such type of test include the following:

- it takes the same time as tests from item #1, no more than 2-3 hrs for analysis of code that took 2-3 days to be written;
- it is completely transparent from legal point of view for the candidate, the candidate who could find more problems is the best, the candidate who found less problems is worse; only numbers, no opinions (however, it may be problematic for you, if the best candidate is not the most psychologically compatible candidate).

Particularities of filtering "developers-engineers"

But what about filtering "developers-engineers"? If you look at the particularities of this type of developers you will understand that they require completely different approach for filtering and testing in comparison to analysts and creators.

- Degree in Engineering is obligatory. Dealing with complex costly hardware requires skills that cannot be obtained at home through self-learning.

> **There are many exceptions.**
>
> ***Anonymous Open Source Developer***

- Good references and background check are obligatory. Because these people will have direct access to the core of infrastructure. Note, that in many cases it may be possible to isolate

development team and management team from production infrastructure completely, by limiting their direct access to test machines only. So, the only who will have direct access to production machines and production copy of code are "developers-engineers", system administrators, and company owners.

> **Good references and background check are obligatory for all future employees.**
>
> ***Anonymous Open Source Developer***

- Theoretical interview is obligatory. Due to the nature of their specialization they will often have too few time to search solutions of real-world problems online or in documentation; so, they should keep many details in heads. This interview should contain mainly questions about emergency situations.
- Code analysis task is just desired for this type of developers.

The first 3 items listed above are applicable to filtering system administrators too.

Why you need to hire only very experienced technical staff?

It is obvious that more experienced developers perform their work more quickly and with better quality. But I would like to highlight in this section several more reasons why you may need to hire such people.

1. **You need several persons for performing the role of manager-architect, as it was told above.** If hiring has been performed correctly then for the team of 5 developers, you will find that at least 1 of them is experienced enough to replace your current *official* manager-architect (you will see management/architecture-related experience in CV of that developer, or you will see appropriate skills in the process of daily work). For the team of 10 developers you should see 2-3 such architect-level developers. In other words, if your current *official* manager-architect performs his work responsibilities well, he/she becomes interchangeable too! However, many *official* managers/architects hire people with lower qualification because they are afraid to lose their positions. It is the life, you should understand this and avoid this.
2. **You need to handle unpredictable situations when the direction of project development is changed significantly.** Some people may say that if software architecture was planned in a very detailed way then it may be implemented by mid-level developers. But in startup environment everything created on the stage of planning is usually a preliminary variant, even if it is very detailed. In the process of implementation of a project that is larger then 4-6 calendar months (with any size of the team) all plans may be changed 1-3 times (for example, just because company owner decided so). And only experienced team will be able to handle situation correctly in such constantly changing environment.

3. **Accidental adding of a "novice star" into mid-level team may cause large problems.** If you have decided to hire mid-level personnel only, you may meet the following situation. One of persons whom you hired may be not mid-level developer, but "novice star". What happens in this case? You will get a lot of problems! Just because you are not ready that a "star" is working in your team. If this person was hired as a first developer for the project he/she may start writing too complex code that is not clear for developers whom you will hire later. If this person was hired together with other mid-level developers he/she may gradually "take ownership" of the most part of core code, just because he/she is more proficient/productive than other developers. In both cases (because you are not ready for such situation) you and your project will become dependent on single developer and personal compatibility issues may arise between this person and other team members. So, the best approach is to hire only very experienced staff for mid-size and large projects.

Some people may argument that best developers may demonstrate non-adequate behavior in work environment. But this problem cannot arise in case if you hire people with noticeable (more than 2-3 years) experience who passed so-called "market validation". And it is not significant at all if you hire remote workers.

Note, that if you don't want to follow provided instructions for hiring the best of best, but instead of this you want to hire mid-level personnel for performing even development of the core of software and other complex tasks, it will significantly affect all other rules listed in this document. You may need a completely different approach for working with mid-level personnel.

WORKFLOW

Task management

Software development should be performed by iterations. Initially the length of each iteration may be 2-4 weeks, but in production phase the length of iterations may be decreased to 2-3 days.

Each task should pass the following phases marked by status in task management system: planning, to do, in progress, under review, testing, done, deploying, verifying.

Tasks that are "under review" are the tasks that are completed by developer, but need to be reviewed by architect before being merged to main development branch.

In startup environment you may avoid strict following of methodologies. But if you hire 5-10 developers from the very beginning you may desire to start using Agile methodology from the first day (with strict following all formal procedures required by the methodology, chosen development method, chosen process framework, and chosen practice). See the following article for details: https://en.wikipedia.org/wiki/Agile_software_development.

In any case, you should perform tracking of spent time for each developer every day for each task.

You should track that tasks are assigned in a way to make all developers interchangeable. In other words, you should avoid situation when, for example, one developer works on backend-related tasks only, and another developer works on frontend-related tasks only.

> **In case either backend or frontend is complicated, that is nearly impossible.**
>
> ***Anonymous Open Source Developer***

If you have not task management system that is built-in into repository hosting, you may want to use an external one. For example, https://www.atlassian.com/software/jira.

If you want to use an external time tracking software, you may pay attention to this service: https://toggl.com/.

Time estimates of tasks in the task management systems can be done initially by your architect, but later they should be corrected by the whole team together.

> **Estimate should be done by the person who does the task.**
>
> ***Anonymous Open Source Developer***

Try to avoid installing on developers' computers the software that performs screen capturing and key logging. It will give you feeling of total control of the development process. But in fact, it just creates psychological pressure for developers, and they will work less efficiently. So, all benefits are eliminated with drawbacks. Moreover, screen capturing and key logging may motivate developers to perform more non-thoughtful monotonous coding work instead of making regular pauses and relaxation periods to think for finding a better solution in each case.

The problem of logging false hours in task tracker can be resolved in a better way, through regular detailed multi-step code review performed by an experienced person (as explained further).

Communication

Communication in a geographically distributed team can be performed with use of https://slack.com/.

Try to avoid hierarchical relationships in the team, friendly horizontal connections (strengthened by appropriate salaries) are preferred. However, if you plan to pay minimal salaries, don't try "to play a democracy", it is ridiculous (because by doing it you generally force employees to be your "fiends"). In any case don't forget that you are not a friend of your workers anyhow, keep a distance.

Software development is one of professions where the rule *"I'm a boss, you are a fool"* does not work. So be prepared to manage people who are smarter than you are not only in the field of development but in the field of business too.

Conversion of specifications to tasks and documenting them

In a complex project the whole process of writing specifications and creating tasks is performed continuously, but not at the very beginning of the project only.

Specifications can be classified as functional (describing functionality from end-user's point of view) and technical (including technical details of implementation). Each functional specification may be broken into several technical specifications, and further each technical specification may be broken into several tasks in task management system. (Although opposite situation when several functional specifications are combined into single technical specification is possible too.)

Usually texts of all these functional/technical specifications have high business value and may take many pages, however tasks based on them and assigned to developers are not so significant. That's why it is better to store texts of functional/technical specifications in internal documentation. Inside tasks descriptions in task management system you should put only links to appropriate articles in internal documentation. If some things were changed in the process of implementation, these changes may be applied to original articles.

Branching model in repository

Each repository may contain multiple branches. Branch is a separate copy of software where development of some functionality is performed. After completion of a task the task-based branch is merged into main branch and may be deleted. In according to my experience in a small (up to 10 persons) team that requires active supervising the best branching model in Git is the following one:

- "master" – branch with production copy of software;
- "develop" – branch with main development copy of software;
- "alice" - main branch of some developer with name Alice;
- "alice--357--new-contact-form" - feature-branch of developer with name Alice where "357" is a number of task from task management system, and "new-contact-form" is a shortened title of that task.

What such branching model gives to you?

- At any moment you can login to repository and easily see what is done by every person in a team.
- You can easily correspond each branch to some task in task management system, and further to the appropriate pages in internal documentation.

- Code review (with subsequent fixes) for each task may be performed independently from other tasks until reaching the desired quality of work.

Note, that process of reviewing each task by architect may be delayed due to different reasons, and each task may require multiple cycles of reviews/fixes before being merged to "develop". It means that lifetime of some feature-branches may be stretched up to several weeks. So, for example, you may see in a repository about 50 active feature-branches from development team of 10 persons or so.

Also, you may want to review the following article: http://nvie.com/posts/a-successful-git-branching-model/.

> **"Develop" branch is not needed at all; having user-based branch is bad way of branching; only feature branches are required. Possibility to login to repository and seeing who does what gives you nothing in long-term perspective because each developer will perform updates from "master" branches eventually. So, there's no sense in having personal branches. After testing each feature branch, it should be immediately merged to "master" branch and deployed.**
>
> ***Anonymous Open Source Developer***

Expected workflow time expenses from manager-architect

Here is a breakdown of time expenses for a person holding the official position of manager-architect in the team consisting of 5 developers.

- You should expect that manager-architect will spend about 5 hrs/week for reviewing work of each developer, giving her/him tasks, and communicating with her/him. That's why for the team of 5 persons workflow expenses will be about 25 hours per week.
- For communication with CEO, clients, and business personnel manager-architect will spend about 5 hours per week more.
- And 5-10 more hours per week will be spent for writing new and updating existing specifications.

All this together will take about 40 hours. So, in the situation with more than 5 developers in the team, workflow tracking responsibilities may be distributed, more experienced developers will perform supervising of novice developers.

Code review

Why do we need to spend much time (up to several tens hours of manager's time per week) for code review? There are several reasons:

- it allows to determine whether developers log false hours in tracker (it happens sometimes) or just work too slowly (it happens more often than sometimes);

- it improves the final quality of code by eliminating some errors and checking conformance of the code to standards and specifications;
- it resolves communication problems, sometimes initial instructions in tasks are unclear for developers and they may do something incorrectly.

In other words, the procedure of regular code review is not a “silly bureaucratic formality” but one of the most important parts of workflow and project management process.

Note, that active usage of code review procedure may eliminate the necessity to arrange regular meetings for the whole team that are required by different workflow methodologies.

> **Do not eliminate meetings ever. In long-term perspective, it causes complicated psychological issues within the team.**
>
> ***Anonymous Open Source Developer***

Continuous integration

Repositories store code. But for seeing the code of web-application in work you need to have it installed on server. It is possible to configure auto-deployment of code from any branch in repository to server on each submission of code to repository. So, for example, you may create on development server branch-based subdomains like master.*, develop.*, alice.* etc. (Separate domains for feature-based branches may be sometimes required too.) And you will be able to see in your browser immediately what is done right now in the project and by every developer.

Ideally, after auto-deployment you should also configure automated running of tests in each deployed copy of software. But in the first phase startup such task may be skipped.

See other details at http://en.wikipedia.org/wiki/Continuous_integration and https://en.wikipedia.org/wiki/Continuous_delivery.

Quality assurance

I would like to look at the process of quality assurance not from developer's point of view but from manager's or client's point of view. Usually, it is strictly desired to determine in contracts or specifications a fixed and easy readable level of quality assurance that must be met at each phase of the project lifecycle. For reaching this purpose I would like to offer to your attention the following gradation of quality assurance levels, from the lowest to the highest. The gradation corresponds not only to software quality but also to the cost of performed work. Description of each level consists of actions performed to guarantee the desired quality level and results that you will obtain with a high probability.

1. Actions: manual testing of UI/UX performed by external (and usually cheap foreign) personnel with minimal technical and business qualification, abnormal conditions are tested in minimal amount. Results: software may contain a lot of hidden problems that may arise in abnormal conditions; however, it may function properly in the most part of circumstances.

Costs: easily controllable and easily manageable. Conclusion: this level of QA is enough for the software with minimal requirements for performance and security.

2. Actions: coverage of UI/UX with automated business logic tests developed by QA developers, abnormal conditions are tested in minimal amount. Results: slightly better than in item #1 above. Conclusion: the same as in item #1 above.
3. Actions: manual code review with simultaneous basic manual UI/UX testing performed by the most qualified developers in the team, abnormal conditions are tested in minimal amount. Results: significantly better than in items #1 and #2, however some unpleasant problems may persist in different parts of UI/UX. Conclusion: this level of QA is required for the software that may be regularly used in abnormal circumstances and performance/security issues are important, but not critical.
4. Actions: coverage of all software internal business logic and all software UI/UX with automated tests developed together by QA developers and all other developers in the team, abnormal conditions are tested in minimal amount. Results: slightly better than in item #3 above. Conclusion: the same as in item #3 above.
5. Actions: full range of quality assurance techniques is applied. Results and conclusion of this QA level are obvious, as I think. Costs: extremely high and comparable to the development of a separate project of the same size.

So, I suppose, that for many large and complex business products you will need to distribute the QA budget in the following way:

- 50% of resources for item #3 (manual code review performed by top-level developers);
- about 10-15% of resources for each remaining item from the aforementioned list applied to different parts of software (in other words, automated stress tests may be developed for one part of software and non-qualified manual testing of UI/UX may be applied to another).

It will allow you to reach the maximal quality level for the minimal cost. However, you should understand that provided distribution is not based on any scientifically collected data and reflects my personal vision of the process solely. So, you should apply it to your project very carefully.

Also, it is recommended to perform the following actions to guarantee the quality of software in the project and minimize QA budget:

- attach to the project 1-2 external part-time consultants who will review work performed by the team and give recommendations; they will spend 5-10 hrs per week each, so it is cheap, but easy way to guarantee quality of results (it is important even if initially you have only 1-2 developers in the team, and it is extremely important if you have high-performance requirements and want to minimize expenses for automated QA at the beginning of the project);
- in startup projects development of automated tests may be skipped in initial phase, because the higher priority is to create a basically working product (and don’t forget that significant

part of possible problems is avoided through creation of plugin-based architecture, hiring only the best developers on the market, and careful regular multistep review of all developed code);

- in the most part of cases you don't need ideal quality of code/architecture, they should be just "good enough" - however, every person has his/her own understanding of "good enough", and that's why you should agree about correct meaning of "good enough" at the very beginning of the project;
- you may need to have in your internal documents a basic instruction for manual QA to test "killer features"; it may include about several tens steps that need to be performed by developers after major changes in the code and that cannot be automated at the current step of development due to many reasons (for example, because these parts of software are under active development and changed daily);
- profiling should be enabled in copies of software used by developers to catch immediately all errors related to performance.

Having international geographically distributed team may facilitate for you some QA procedures, because they will be performed on the fly exactly in the process of work by developers. For example, the following errors may be detected very easily by such team:

- timezone-related errors causing incorrect time calculation or temporary unavailability of software;
- all errors related to text encoding, localization, and internationalization;
- all errors related to client-side platforms from different manufacturers;
- unavailability of software or delays in servicing requests from end-users in some geographical regions;
- problems caused by narrow Internet bandwidth or excessive size of code/graphics used in browsers.

In addition to all listed above, please, check the following articles:

- https://en.wikipedia.org/wiki/Acceptance_testing
- https://en.wikipedia.org/wiki/Unit_testing
- https://en.wikipedia.org/wiki/Software_performance_testing
- https://en.wikipedia.org/wiki/Usability_testing

Also, please, note, that current business trend is to use QA developers as integral part of development team, but not as a separate team.

HOW GOOD CODE SHOULD LOOK?

You may find online a lot of articles written by geeks that describe how good code should look. However, as you could guess, business requirements may dictate different practical standards depending on situation. If you develop software that does not control human safety and cannot cause multi-million financial losses then the following *minimal* standards would be completely enough for your business:

> **The rules listed below are quite lousy and not really important. How would they guarantee that team will be able to implement changes in the project that will not require time expenses increasing exponentially?**
>
> ***Anonymous Open Source Developer***

1. Use official standards of languages/frameworks. However, some minor deviations are acceptable, if you really need them.
2. The code should be very very very boring. No exotic algorithms, no rare language constructions etc. All should be very simple and very readable.
3. Naming standards should allow only readable names in English for everything. Even if these names may look rather long sometimes. In other words, such method name as "*SendEmailToRecipient()*" is fine, but "*SndMail2Rcpt()*" is not.
4. The code should be "moderately" commented in according to "good enough" approach. (For example, such methods as *getName()* or *isValid()* do not require commenting as you understand.) See items #2 and #3 above to understand how readability should be achieved without excessive commenting. (Yes, it is my personal habit, as you could guess. Many other developers will tell you that there is no such thing as "excessive commenting".) However, if developer thinks that some block of code is not obvious for another experienced developer, he/she should write appropriate explanatory comments there immediately.
5. It is okay, if code contains some dirty places in amount that is allowed by "good enough" approach used by your team. Just agree together in the beginning of the project about allowed amount of "good enough". And note, that really "too dirty" places are eliminated from code in the process of code review performed by architect (and subsequent fixes performed by developers) as described above.
6. However, don't forget to add a "*todo:*" comment in each "dirty" place with explanation why such place has been created and how and when it should be fixed. This approach allows to have hundreds of "dirty" places in the code without degrading the whole quality of the software.

> **TODOs are rarely cleaned up. Dirty code with TODO comments isn't better than dirty code without TODO comments. It is still dirty code; so, the quality of software is degrading.**
>
> ***Anonymous Open Source Developer***

7. Keep your code very modular. Don't allow to create components larger than a thousand lines, or so. However, if you will see in a project with a size of several hundreds components that 1-5 of these components contain several thousands lines, it is not a big problem. Just check that they contain a lot of explanatory comments to allow easy refactoring of these components within the following 3-6 months.
8. Use named constants instead of "magic" numbers/strings.

It is possible to configure your version control system to lock submission of code that does not follow some preconfigured standards.

Please, read about broken windows theory at https://en.wikipedia.org/wiki/Broken_windows_theory and about technical debt at https://en.wikipedia.org/wiki/Technical_debt. You need to fix minor problems in code regularly to avoid bigger problems. Also read online about ideology of "s*oftware gardening*" that describes similarities in the process of gardening and software development in comparison to standard industrial engineering. (If you want more analogies between software development and other production processes, I would offer you to look at writing scripts for TV series. Maybe you will find something useful there too.)

If you need something more than aforementioned short list of criteria, you may find that many companies publish online their internal development standards. Like this Javascript Style Guide from Airbnb: https://github.com/airbnb/javascript. You may use them freely in any project. Note, that many such standards are based on artificial limitation of usage of features built-in into language/technology.

You should understand that following aforementioned standards (and any other standards) does not mean that your code is really perfect (or vice versa, not perfect). Definition of "perfection" varies depending on circumstances. You should rely on common opinion of all your most experienced developers and consultants to be sure in each particular case.

SYSTEM ADMINISTRATION

If your company owns/rents server(s) either in office or remotely, you may need to hire separate system administrator or assign one of developers to this role. You should understand that primary functions of system administrator consist of the following:

- writing “papers” (yes, writing “papers”!), including descriptions of system configurations, protocols of performed actions, protocols of actions in case of emergency, protocols of performed fixes, protocols of backup/restore configuration;
- performing regular upgrades and securing your environment;
- configuring automated offline/offsite backups and regular checking their validity for a case of emergency;
- resolving urgent system problems.

Check the following in your environment:

- your domains are prepaid for several years in advance;
- separate physical machines are used for development server and production server;
- only reputable hosting companies are used;
- if you use GoDaddy hosting (or similar one) check that your service plan allows immediate restoration of data from backup, and all purchased domains are really in your full ownership.

I'm not very proficient in this topic, so I cannot add more useful information. However, you should also know that currently the term *Development Operations* is used for all tasks related to system administration, although they do not overlap completely. The general idea of this change in terminology is improving communication between system administrators and development team. Please, search details online by yourself.

TECHNOLOGIES AND TOOLS

It is hardly possible to provide generic advice in this field, but here you may read some notes concerning web-based and cross-platform mobile software projects.

Tools for server-side part of web-based software projects (only programming languages are reviewed, because usually choice of programming language determines choice of other tools).

- Python, the ideal choice for large heavy complex highload projects (this opinion is based on description of projects at https://www.python.org/jobs/ and forum talks), https://en.wikipedia.org/wiki/Python_(programming_language).
- Ruby, very widespread and very fashionable in startup community, I suppose that you may consider it as a main replacement for PHP in mid-size projects, https://en.wikipedia.org/wiki/Ruby_(programming_language).

> **Python is not the ideal choice. Ruby was trendy about 10 years ago, now it isn't.**
>
> ***Anonymous Open Source Developer***

- Javascript/Node.js, the ideal choice if you want to perform all server-side and client-side development in your company with Javascript only; it may be comfortable if amount of client-side code is comparable to amount of server-side code; note, that environments similar to Node.js exist for Python and Ruby too; https://en.wikipedia.org/wiki/Node.js;
- Modern PHP evolves very quickly and gradually comes to the same level where Python and Java are used; it is not "a tool for amateurs" anymore. In case if PHP has been chosen, you should also pay attention to choosing an appropriate framework, because there are more options than in other languages; I prefer to use Yii framework, http://www.yiiframework.com/. Note, that experienced PHP developers with background in mid-size and large projects cost the same money as developers working with other languages: the cost depends on experience, but not on a used tool, so don't try to save money by choosing PHP, it will not give you anything.
- Using Java-based stack of technologies may be a good choice too (especially, if you plan to target a market of corporate clients with intranet software), but you should understand that top-level Java developers prefer to work in corporate environment with receiving appropriate benefits and job security, so you may meet obstacles in hiring people, as I suppose.
- For really high-performance projects you may need to write part of functionality in C/C++ (or other languages depending on circumstances and skillset of your team), and call these modules from modules written with use of aforementioned languages.

Tools for client-side part of web-based software projects:

- jQuery, the most popular Javascript library, https://en.wikipedia.org/wiki/JQuery;
- Angular, https://en.wikipedia.org/wiki/Angular_(web_framework);
- React, https://en.wikipedia.org/wiki/React_(JavaScript_library);
- Bootstrap, https://en.wikipedia.org/wiki/Bootstrap_(front-end_framework);
- and also you may look at this list of charting frameworks – https://en.wikipedia.org/wiki/Comparison_of_JavaScript_charting_frameworks.

Tools for cross-platform mobile software projects (note, that these tools do not resolve all problems, and in some cases you may need to use native development tools for each mobile platform either for the whole project or for some plugins):

- Apache Cordova, https://en.wikipedia.org/wiki/Apache_Cordova;

- Unity game engine, https://en.wikipedia.org/wiki/Unity_(game_engine);
- Codename One, https://en.wikipedia.org/wiki/Codename_One

Tools for data management:

- https://en.wikipedia.org/wiki/MySQL
- https://en.wikipedia.org/wiki/PostgreSQL
- https://en.wikipedia.org/wiki/MongoDB
- https://en.wikipedia.org/wiki/SQLite
- https://en.wikipedia.org/wiki/Graph_database
- https://en.wikipedia.org/wiki/Apache_Hadoop
- https://en.wikipedia.org/wiki/Elasticsearch
- https://en.wikipedia.org/wiki/Tarantool
- https://en.wikipedia.org/wiki/Apache_Spark
- https://en.wikipedia.org/wiki/Apache_Cassandra

Other tools that may be useful for you depending on circumstances:

- https://en.wikipedia.org/wiki/Load_Impact
- https://en.wikipedia.org/wiki/Selenium_(software)
- https://en.wikipedia.org/wiki/Sauce_Labs
- https://en.wikipedia.org/wiki/Amazon_Web_Services
- https://en.wikipedia.org/wiki/DigitalOcean
- https://en.wikipedia.org/wiki/List_of_build_automation_software

If all listed above is not enough for you and you want “to run on the edge”, then you may check GitHub for finding the most popular open source projects in the fields of your interests. But be careful! By choosing a wrong tool you may kill your budget and your project. Here are some lists that you may check:

- https://github.com/showcases
- https://github.com/trending
- http://gitmostwanted.com/
- https://github.com/sindresorhus/awesome
- https://github.com/bayandin/awesome-awesomeness

- https://github.com/Kickball/awesome-selfhosted

And finally, I would like to provide a short review of relatively exotic programming languages used for narrow specialized tasks in server-side part of web-based software projects. You should understand that I'm not very proficient in this topic, so some mistakes could be made. Only the most popular of the exotic programming languages are reviewed further and only main field of usage for each language is described:

- Clojure, for high-speed processing of large amounts of data, https://en.wikipedia.org/wiki/Clojure;
- Scala, for the same purposes as Clojure, but it is more comfortable for mainstream developers, https://en.wikipedia.org/wiki/Scala_(programming_language);
- Go, for network communication in companies with large network infrastructure consisting of multiple services, https://en.wikipedia.org/wiki/Go_(programming_language);
- Rust, more safe alternative of C++, https://en.wikipedia.org/wiki/Rust_(programming_language).

For data science tasks you may need to use R programming language, https://en.wikipedia.org/wiki/R_(programming_language).

> **I haven't heard that Clojure gives any advantages in the field. Scala is just Java on steroids. Definition of Go usage field is not correct; however, it gives a good advantage in data-processing.**
>
> ***Anonymous Open Source Developer***

BUILDING IN-OFFICE WORK ENVIRONMENT

If you have decided to place people into office, it is recommended to implement the following requirements:

- separate cubicle for every developer with a size of about 9m^2 or more (it is recommended to validate the desired size of cubicle in professional literature);
- each developer should have access to window where garden/park, ocean/river coast, or open sky is visible;
- large displays (2 per person) and top level internal hardware should be available;
- common space with coaches, internal fountain or aquarium, and minimal lights should be available for relaxation and communication;
- laptops for working outside of cubicle (in common space or internal garden) should be available;

- do not mix sales/support/system teams and development team in the single office; these teams will create too much noise that will prevent efficient work of developers.
- pay special attention to purchasing configurable chairs and desks that allow your developers to work in flexible number of positions and to save their health (note, that in according to recent researches sitting is considered as second smoking and standing workplaces are not considered as healthy anymore);
- if you are from US and your team is not, you may be surprised that in many countries having a car is not considered as necessity; so, if you arrange a private minibus line with several trips per day for delivering your workers from homes to office and back, it may be very appreciated;
- if you plan to hire top-level team and junior-level team with providing different work conditions for both teams, it may be a good idea to place these teams into different buildings to avoid unhealthy communication between people from both teams.

In other words, environment should provide maximal comfort and minimize stresses that may cause conflicts between people. Certainly, some employers reach this goal in other ways: they just hire very communicative people having minimal level of aggression (the term "aggression" is used in its biological meaning but not in a trivial one) and minimal requirements for personal space (and generally, quality of life). However, such people may have lower professional level, as you can understand, because professional level often correlates with a level of sublimated aggression.

If you came from standard office business you may be surprised with all these requirements. But don't forget that working and living standards raise from generation to generation simultaneously with a technological progress. You are not surprised when your video message comes from one part of the planet to the opposite part for less than a second. So, why you should be surprised by aforementioned work conditions?

Many years ago Winston Churchill told something like this (the citation is given by memory), *"Your thoughts are formed by your homes. Your future is formed by your thoughts."* These words are completely applicable to office-based software development too: your thoughts are formed by your offices, your software is formed by your thoughts.

In any case you should always remember that humans are animals. (Or that human nature is very similar to the nature of animals, if your religious views do not allow you to agree the previous statement.) So, learning ethology is a good idea: https://en.wikipedia.org/wiki/Ethology.

SOCIOLOGICAL SIDE OF SOFTWARE DEVELOPMENT BUSINESS

In this section I would like to highlight several details that may affect your plans significantly if you are a newcomer in the software development business. All these details concern communication between people and sociological particularities on the software development business.

- Software development business involves dealing with “big money”. It means that you may meet there regularly all kinds of people who typically have some relation (or want to have some relation) to “big money”: people from government security agencies, people with semi-criminal background, people with artistic background and people coming from show business, people with non-standard psychological particularities (up to top levels of psychotic spectrum disorders and autism spectrum disorders), people from different rich diasporas, recent immigrants building new life in new country.
- Work in software outsourcing business is considered prestigious in the countries of 2nd and 3rd worlds. Profession of software developer gives some emigration opportunities. Hardware and regular Internet access may be costly sometimes. And in some places raising a kid with appropriate style of thinking requires high level of safety combined with a noticeable educational background in parents. What does it mean? It means that if you hire a top-level developer from 2nd or 3rd world country, there is a high probability that hired developer is not “a random guy”. Usually, he is coming from local “elites” or at least has some connections with them.
- Sometimes, if you open an office in 2nd or 3rd world country you should be ready to obtain “protection” from local government security agencies. Without such “protection” you may see at some fine day a lot of armed guys breaking doors in your office for performing “check of correctness of taxation documents”. The “protection” may be obtained either for a bribe or for assigning one or several “right persons” at managerial or top-level technical positions in your office.
- As you can understand, the developers is the main asset (or one of the main assets) of any software development company. In some countries of 2nd and 3rd worlds where formal laws and formal legal procedures do not matter much, such situation may create “vassalage” relationships or even “master-slave” relationships between company owners and developers. An attempt to change a job may cause criminal threats. Cartel agreements between companies preventing free flow of workforce are considered as a norm. Moreover, such style of relationships is some or another form is practiced even in the companies located in the 1st world, if company owners and the most part of workforce are recent immigrants. What does it mean for you, if you live in the 1st world? Knowledge of these facts may assist you in communication between you and your remote personnel.
- Globalization transforms all very quickly. Economical, cultural, demographic, and political situation has changed significantly everywhere since the mid of 2000's. Starting from this period massive economical growth with all appropriate consequences took place in 2nd and 3rd worlds. At the same time, migration flows have changed the society of the 1st world too. Currently, the gap between worlds is minimal. (The only region that is really far behind is Africa.) You should consider this fact if you plan to hire remote personnel.
- The difference between “developers-analysts”, “developers-creators”, and “developers-engineers” is not only psychological and technological. It is also social.

"Developers-analysts" are more close to academic circles and business circles, "developers-creators" are more close to artistic circles, and "developers-engineers" are more close to working class. In multiethnic and/or "traditionally colonial" societies it means that people from all these 3 professional groups may belong to different ethnic and cultural groups with rather different history and different traditional level of income. So, placing people from all these 3 groups together inside single office may cause tensions between them. If you will do it, then try to keep identical work conditions for all. However, even in mono-ethnic society all these 3 groups of developers may have completely different educational background with appropriate differences in behavior and life style.

USEFUL ONLINE RESOURCES AND BOOKS

- https://en.wikipedia.org/wiki/Lean_startup
- Internet trends since 2001st year till the current year: http://www.kpcb.com/internet-trends.
- Technological trends since 2010th till the current year: https://www.thoughtworks.com/radar (scroll down to see links to reports in PDF).
- Key small business statistics for Canada (probably for US all will be the same): http://www.ic.gc.ca/eic/site/061.nsf/eng/h_02689.html.
- Official statistics for all professional specializations in US: http://www.onetonline.org/ It allows to see salaries and projected growth/decline in different specializations. All data may be extrapolated on global scale and for the whole markets.
- "Facts and Fallacies of Software Engineering" by Robert L. Glass: http://www.amazon.com/Facts-Fallacies-Software-Engineering-Robert-ebook/dp/B001TKD4RG.
- Joel Spolski's test at http://www.joelonsoftware.com/articles/fog0000000043.html and other his articles at http://www.joelonsoftware.com/.
- More than 400 awesome free tools to build your business: http://growth.supply/free/.
- https://habr.com/en/ − a lot of articles on IT topics, both technical and business-related.
- https://www.eff.org/ − news about protecting civil liberties in the digital world.

APPENDIX A − RECOMMENDATIONS FOR NOVICE SOFTWARE DEVELOPERS

From the very beginning, this document has been targeted primarily at investors and business people. But after completion, it was obvious that the most part of material may be useful for novice

software developers too. That's why it has been decided to add here some notes and recommendations that are useful primarily for novice software developers.

- If you have just started learning software development and got stuck at some level, then you may be interested in learning that there are side ways to extend capacities of your brain to climb up professionally. For example, you may spend your free time for reading serious scientific literature/journals, reading mathematical textbooks, playing mathematical games, practicing construction of physical objects with your hands (like building houses, sculpting etc), and practicing construction of digital objects with 3D graphics tools.
- Knowledge of English is obligatory for finding good job in software development business. Knowledge of German will extend your career opportunities for 15%-30%, as I suppose. But in addition to it, knowledge of Russian would be extremely beneficial to improve informal relationships with co-workers, even if you plan to live and work in North America and EU only. With the same purpose, learning some Indian languages (there are about 10 the most popular) may be very beneficial too. Moreover, learning many relatively exotic foreign languages will extend your brain capacities in a similar way as described above.
- If you just started learning software development, then it is recommended to learn popular and emerging technologies in the ratio 1:3 or even 1:5 (with a preference towards emerging technologies). Just because your real career will start in 1-3 years only, and it may turn out that all currently popular technologies will disappear from the mass market by that time. Obviously, not all emerging technologies will reach the top, but one of these 3-5 will definitely do, and it may become your primary work tool for the next 10 years of career.
- Also, if you just started, then put a special attention to choosing frameworks/libraries that you will use in your daily work. These tools will form your style of thinking and your approach to development. So, learning software development through learning of great framework/library means shortening your road to success twice!
- If you have decided to work in office of large corporation in the city, then arrange in your time schedule 30-60 minutes for walking in nature every day. Or 30-60 minutes of good positive sex. Also, you should have some personal space for staying alone and relaxing at least for 60 minutes per day. Or you may become exhausted very quickly.
- Having contacts of good lawyer may be very beneficial for unpredictable circumstances. (Nothing personal, just business.) If you plan to earn some money with self-employment, then having contacts of good accountant is great idea too. And it is even more great idea to learn basics of law and accounting by yourself.
- If you plan to build your career as a regular employee, then in addition to practical software development skills, you may need to learn the following for passing job interviews : some popular algorithms and design patterns, writing code on paper and blackboard, history of profession.
- To keep your career and income balanced, you need to have two specializations in your CV.