**Business Plan**

Productivity of software engineers is the factor that determines the profit of any software company. If a software engineer does in 2 days what it would take him 20 days (a man\*month) to accomplish without our tool, then in 2 days the company earns with our tool the profit which otherwise the company would need 20 days to earn.

**Software Comprehension Tools**

**Serge Rogatch**

[serge.rogatch@gmail.com](mailto:serge.rogatch@gmail.com) ; +375 29 5014930 ; Skype: sarge.rogatch

Table of Contents

[Executive Summary 3](#_Toc491537469)

[Business Description & Vision 4](#_Toc491537470)

[Definition of the Market 6](#_Toc491537471)

[Description of Products and Services 9](#_Toc491537472)

[Organization & Management 11](#_Toc491537473)

[Marketing and Sales Strategy 12](#_Toc491537474)

[Financial Management 12](#_Toc491537475)

[Appendices 14](#_Toc491537476)

# Executive Summary

Productivity of software engineers is the factor that determines the profit of any software company. If a software engineer does in 2 days what it would take him 20 working days (a man\*month) to accomplish without our tool, then in 2 days the company earns with our tool the profit which otherwise the company would need 20 days to earn.

The productivity of software engineering work depends on, among others, the ability and knowledge of software engineers, and on the complexity of software. The more complex a software system is, the more time is required for its comprehension by a software engineer who needs to design, implement or change something in the software. A software engineer spends time on software comprehension when he/she reads thousands of lines of source code (the program) in a file, browses thousands of such files without an opportunity to read the lines of source code in each file, reads thousands of pages of technical documentation for the program, runs the program and pauses it at a certain place to see whether the program is correct in this place by stepping through the source code line by line. Consider large software beyond development phase (i.e. in maintenance phase) and a software engineer who has not been on this project since the very beginning or has not been developing this particular piece of software. Then he/she spends only 5% of time on typing the lines of source code (i.e. coding the program); the other 95% of time he/she needs for comprehension because the software engineer needs to determine what lines of source code to type.

Our tool lets a software engineer to comprehend a program quicker, by dividing the scope for comprehension between a human and our tool – an automatic program. Our tool takes care of the details, analyzes huge amounts of data (dynamic or static call & dependency graph of the software) and presents to the user only the clear and concise conclusions aligned with the purpose. E.g. a program of 10 million lines of source code, which is usual-size software, contains as much text as a fiction book of 200 thousand pages, which would be an enormously large fiction book. If software were the fiction book, our tool would present the user with the gist of the book on 20-100 pages instead of the original 200 000 pages.

It takes a lot of man\*hours to design and develop a complex software system. Then each small change takes a lot of time because comprehension of a huge and complex program requires that. A small change in one place of a program usually causes defects in the other places of the program if that small change was done without understanding of the rest of the program, and the rest of the program is huge and the relations in the program are complex. Comprehension of the small place which the software engineer needs to change is usually a minor effort w.r.t. the efforts spent on comprehension of the rest of the program and comprehension of this change in the context of the rest of the program, so to ensure that the change is valid. The ratio between the former and the latter is 1 to 100 usually. Our tool will reduce it down to 1 to 10, which is a 10 times gain in productivity of software engineering work. We achieve this by narrowing down the search area for the potential side effects for each small change, because our tool visualizes the (indirectly) connected pieces of source code and categorizes them into loose and strong connections.

Therefore, the main purpose of our tool is reverse architecting from source code in major programming languages – Java, C/C++, C#/.NET languages, Python, PHP, JavaScript, etc. . However, they can also use it for reverse engineering from binary executable code, if they give the disassembly on input to our tool.

# Business Description & Vision

The purpose of the business is a major increase in productivity of software engineering work. The company will be creating tools which let software engineers work much more productively, e.g. do in 2 days with our tool what would otherwise (without our tool) take 20 working days. Our mission is development of programs, which would comprehend other programs automatically. So that our programs let a software engineer comprehend those other programs many times quicker and let a software engineer work productively with programs which are too complex and huge for a human to comprehend without an automatic tool taking care of tens of millions of small details.

We can start with a team of 18 programmers, 3 software architects, 3 team leads, 1 project manager and the research author being the CEO of the company. The first tool should be ready in 1 year since start because the author estimates it as 24 man\*years of work. In the next year we will be selling the tool as a service (SaaS - “Software as a Service”) and earn the revenue which exceeds the investment amount. In fact, to earn the revenue that exceeds the amount of investments it is enough that 7537 software engineers use our tool for a working year. Assuming a team is 6-8 software engineers (including architects/leads), that is 942 to 1257 teams. Or that is 38 companies of 200 software engineers in each.

Because our tool is for software engineers, selling it by or to non-technical people does not seem efficient. There should be direct communication between our software engineers, who sell, and software engineers of the other companies, who can understand the value of our tool and then persuade the management to buy it. To sell the tool, the software engineers of our company will be visiting the software development forums on the internet and answering the relevant questions which software engineers of the other companies ask, so to show the software engineers of the other companies how our tool can solve their current problems. This will let the knowledge about our tool propagate quickly and purposefully, consequently grow the company towards its potential. E.g. in the USA the potential is:

* There are more than 1 million of software engineers: <http://en.wikipedia.org/wiki/Software_engineering_demographics>
* At least 1 out of 10 software engineers need to run our tool every day (some of them do not realize this yet, but they will realize the value of the tool when the tool exists and our software engineers explain them how to use it).
  + Reviewers need to run our tool when someone posts code for review, so to analyze how this new code fits into the old architecture, and how it possibly changes the architecture.
  + Architects need architecture visualizations performed by our tool in nightly builds or after commits, so to analyze how the architecture evolves and keep their knowledge up to date.
  + Team Leads and upper management need quality metrics for their source code, while our tool can produce the architecture quality metrics and pinpoint the problematic places.
  + Usual software engineers need to run our tool to comprehend the software, especially newly hired software engineers to get first insights about the software and catch up quickly.
  + Quality Assurance engineers to identify the potential side effects (thus, the aspects to focus on when testing) of the recent changes/fixes/features by software engineers.
* We assume the price of our service is $1 for one run (one analysis of their software with our tool)
* The potential revenue is $1 \* (1M / 10) \* 250 = $25M per year, according to the above data and assumptions. Where 250 is the number of working days per year.

According to the above calculation, just in the USA the potential revenue is $25M per year at a minimum. There are many software engineers in the other countries: <http://programmers.stackexchange.com/questions/19720/where-can-i-find-statistics-on-worldwide-developers-and-software-companies/20300#20300> . In 3 years since start (which is 2 years since the development of the first tool finishes and sales start) we plan to reach the potential revenue, meaning that each competent software engineer who needs our tool gets it clear to him/her and starts using it.

To wrap up, the business goals for the first few years are:

1. Develop the first tool in 1 year since the team is ready
2. Earn enough revenue to cover all the expenses in 2 years since start
3. Reach the bar of $25M annual revenue from customers in the USA plus the proportional revenue from customers in the other countries, e.g. $6M in the UK, $7M in Canada, $19M in Japan, $3M in India, etc. We should be able to achieve this goal in 3 years since start.

In our vision the future of our company is in development of computer programs with comprehension capabilities increased in depth and in breadth. In depth, those will be the programs which better understand the meaning of the input data given to those programs. In breadth, we will start with programs for comprehension of other programs, and then proceed to programs for comprehension of e.g. fiction books, scientific articles, schemes of vehicles and buildings, etc. We will be improving our programs so that they can comprehend more, distinguish between the important and the unimportant, get it right, distinguish between true and false, make less mistakes. The better our programs do in the aforementioned activities, the more value our programs give to people, so that they get ready to use our programs more often and thus pay more.

The key company principal is Serge Rogatch (spelled as Sarge Rogatch in the passport and therefore many other documents, e.g. Master’s diploma). In 2010 he researched [automatic structure discovery for large source code](https://arxiv.org/ftp/arxiv/papers/1202/1202.3335.pdf), which will be the underlying technology for the first tool of the company.

# Definition of the Market

The market for the tool is every software company and every software engineer who needs to comprehend a program of more than 10`000 lines of source code. A program of less than 10000 lines of source code is a tiny project in software engineering industry. So every project in software engineering industry needs our tool, unless the project is tiny. For example, the operating system Windows XP is about 45`000`000 lines of source code, Windows Server 2003 is about 50`000`000 lines of source code, Mac OS X 10.4 is about 86`000`000 lines of source code, and Linux kernel 2.6.35 is about 13`500`000 lines of source code – according to this article: <http://en.wikipedia.org/wiki/Source_lines_of_code> . A tiny project in software engineering requires 13.5 man\*months, according to this article <http://www.codinghorror.com/blog/2006/07/diseconomies-of-scale-and-lines-of-code.html> and book “Software Estimation” by Steve McConnell. It follows that every project that requires more than a man\*year (specifically, 13.5 man\*months according to Steve McConnell) for its development, also needs our tool for comprehension of this project by software engineers. Our tool lets a customer to alleviate the diseconomies of scale because larger projects are harder to comprehend and our tool lets software engineers to comprehend such projects quicker.

For example, a software company needs our tool:

* When it buys or sells its software product entirely (i.e. with source code) to another software company – to estimate the price of the software product judging by the quality of the source code of that software product. The easier it is to comprehend the source code of a software product, the higher is the price for selling such software product entirely because it is cheaper to maintain and extend a high-quality software product than a low-quality one. Our tool can tell both the seller and the buyer company how easy it is to comprehend the source code of the software upon sale, therefore how high the quality of that software is. A company, which does not use our tool risks to sell its software to another company at a too low price or to buy software from another company at a too high price, therefore risks losing much more money than the expenses for our service.
* When a software engineer makes some changes to a program, which the company currently maintains or develops – to estimate how well the changes fit the design of the software. If the software engineer doesn’t use our tool and the changes do not fit the design, then as a consequence the expenses of the company start growing because the software becomes of lower quality (thus harder to maintain).
* When a software architect analyzes how the architecture (the structure) of the software evolves over time, e.g. day to day, or a team lead or project manager analyses how the quality (therefore, the maintenance cost) evolves, e.g. in nightly builds, in Agile iterations (weekly/biweekly) or releases (monthly/quarterly). If the company doesn’t use our tool, then more man\*hours of a software architect is required to maintain the design documentation up to date, and each small change takes a lot of time because it is hard to comprehend the software (i.e. how to make the change) and the side-effects of the small change. Without our tool it is harder and takes more time for a team lead or a project manager to understand the cause-effect relation between the implementations his/her software engineers provided or decisions he/she took (*the cause*), and the progress towards delivery of a piece of software functionality or completion of a milestone (*the effect*).
* When a quality assurance specialist searches and identifies the places, which could become defective due to the recent changes made to the program by the software engineers of the company. Our tool will illuminate the connected places for a given place in the source code of the software. Without such a possibility a software engineer would need to perform manually what our tool will do automatically, and a quality assurance engineer would need to guess or read the documentation which without our tool is hard to maintain up to date with the source code, and to spend time on querying the software engineer about the changes and the potential side-effects rather than launching our tool and seeing that in its output himself/herself.

As of 2010, the US business and government spending on the part of IT goods and services which mostly consists of the aforementioned activities is $368`000`000`000 = $83 billion for IT outsourcing + $95 billion for IT consulting services + $190 for Software according to Figure 1 below. Studies show that 67% of expenses in a Software Life-Cycle they spend on Maintenance, according to Figure 2 below.

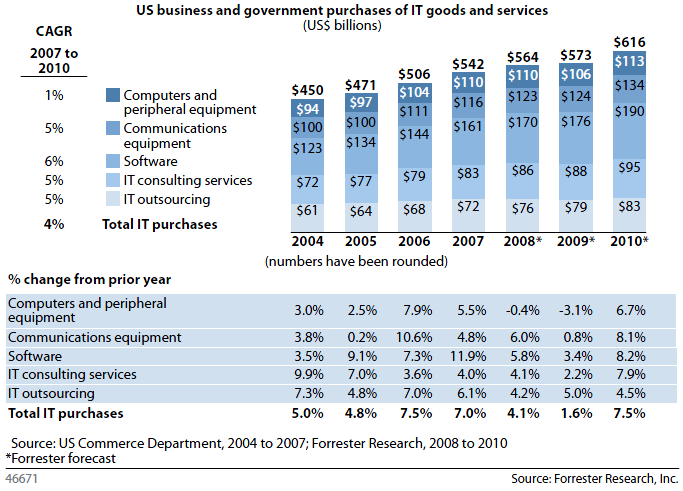


Figure 1 - source: http://www.zdnet.com/blog/btl/forrester-cuts-2009-it-spending-projections/10056

The conclusion is that in the USA 67% \* $368`000`000`000 = $246`560`000`000 is the current spending plus the added value which software companies charge from their customers for the activities which without our tool they do at 1/10 of the productivity which they could achieve using our tool. Assume that the added value which the companies charge is 100% on average. Then the expenses of the companies are 50% of the price they charge, i.e. the expenses are $ 123 billion. Then the ultimate version of our tool will save the companies 0.9\*$123 billion = $110.7 billion because it increases the productivity 10 times meaning that the same amount of work can be done in 1/10 of the expenses which the company would spend on software engineering activities without our tool.

Let the first version of our tool reach the bar of $25 million revenue as estimated in section “Business Description & Vision”, which is 1/443 of the potential of the ultimate version of our tool. Assume that we charge 10% of the saved expenses. This means that out of $110.7 billion saved for the other software engineering companies, we get $11.07 billion for our service and our tools.

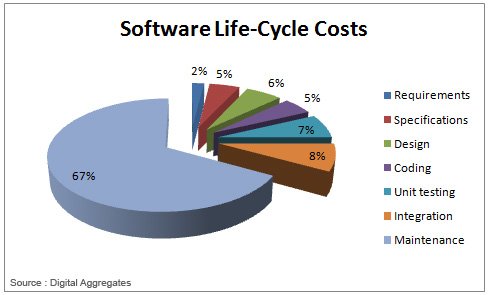


Figure 2 - source: <http://thibautvs.com/blog/?p=953> , also Agarwal R. (2004) “Software Development vs. Software Maintenance”

To the knowledge of the author, we have no direct competitors because the other companies do not provide tools for comprehension of software. Such a technology is simply not yet the state-of-practice – it is state-of-the-art, and we want to apply it in practice. There are many tools for estimation, planning and analysis by indirect metrics, e.g. <http://en.wikipedia.org/wiki/Comparison_of_development_estimation_software> . Although those tools belong to Computer-Aided Software Engineering (CASE) category too, they are applicable at the Requirements/Specifications/Design/Unit testing phase of Software Life-Cycle (Figure 2 above), or they belong to Maintenance phase but don’t directly increase the productivity because e.g. they only provide some quality metrics. Maintenance phase of Software Life-Cycle requires productive comprehension of the program, rather than design or estimation. The software comprehensibility tools of our company therefore aim at the largest piece of Software Life-Cycle costs – Maintenance – and the purpose of our tools is 10 times increase of productivity so that we save 90% of what currently constitutes 67% of Software Life-Cycle costs, meaning that our tools save 90%\*67% = 60% of costs in Software Engineering industry.

# Description of Products and Services

Our company will be producing the software tools which let other software companies work more productively. For example, the quicker a software engineer can comprehend a piece of program which he/she needs to extend or improve, the sooner this software engineer can deliver the extension or improvement which his/her employer can sell for profit. Speedup of software comprehension activities is the purpose of our first tool. It is a computer program which analyses other computer programs, with the goal to understand as many as possible details without human attention and therefore leave as little as possible details to the attention of human software engineer. The more details our tool understands, the more software engineer’s time it saves, because without the automation provided by our tool the software engineer would need to spend time on looking into most of those details himself/herself.

Our tool takes on input a computer program which a software engineer needs to understand, and produces a structural diagram, which lets the software engineer comprehend the program many times quicker than in the time necessary for reading the program. A structural diagram of a computer program is usually called “architectural diagram” because in programming the term “structure” is reserved for data structures of a program or a programming language. An architectural diagram of a computer program displays a piece of the architectural design of the computer program, aiming to let a software engineer understand a piece of the computer program quicker than by looking into the lines of text of the computer program. The architectural design of a computer program is the [lossily compressed](http://en.wikipedia.org/wiki/Lossy_compression) knowledge about the computer program which lets a software engineer understand, extend or alter (e.g. debug or improve) the program many times more productively than in the absence of such knowledge. Uncompressed knowledge says the value of each unknown, in the other words, for each Xi it says that Xi=Vi , where Xi is the unknown and Vi is its value. The uncompressed knowledge about a computer program is the computer program itself – a set of text files containing the source code of the program. Compressed knowledge doesn’t say directly what the value of some Xi is – instead, it says how to determine Xi , or how to determine how to determine Xi , or how to determine how to determine how to determine Xi , etc. . The architectural design of a computer program is the **lossily** compressed knowledge about the program because a software engineer cannot determine every fact about the computer program from the architectural design and needs to read the source code of the program to determine the facts, which he/she cannot determine from the architectural design. But the most important (for productive work) facts about the computer program the software engineer can determine from the architectural design.

An architectural diagram is a coarse-grained map for the source code of the program. The latter contains all the details. An analogy for the former is something between a table of contents, an index and a brief retelling, depending on the level of granularity the user requests. Our tool produces a nested decomposition of software system into subsystems from the source code of the program being analyzed, or from its dynamic call graph, or from a disassembled binary executable. The user can then drill down from top-level subsystems down to smaller nested subsystems, choosing the path that interests him/her.

To work productively, software engineers think over the architectural design before they start typing a computer program. They store the architectural design knowledge in documents and update these documents as the computer program evolves. The current medium-size software products contain hundreds of thousand lines of source code and hundreds of pages of documentation. The current large-size software products contain millions of lines of source code and thousands of pages of documentation. A software engineer cannot comprehend completely a software system consisting of so many details, even in lifetime. Neither can the complete knowledge about the software system be distributed across a team of software engineers efficiently. This entails abrupt increase of the expenses of software engineering companies when their computer programs grow larger – because the productivity of work drops due to lack of understanding of the computer programs by software engineers, meaning that each small change to a computer program requires more (w.r.t. what it was when the computer program was small) man\*hours to be spent by software engineers and paid by the companies. In the beginning, software engineers spend 40% of time for coding a computer program and 60% of time for design of the code to be typed and comprehension of the text connected to the text to be typed. However, when the computer program grows larger, software engineers start spending 20%, 10%, 5% or less time for coding and the rest of time for comprehension of the computer program. A software engineer, which types less than 144 lines of source code per working day (8 hours), spends more than 95% of time for comprehension of the computer program: 7 hours 36 minutes out of an 8 hours working day. A tool which lets a software engineer comprehend the computer program 10 times quicker, in such conditions will let the software engineer work 6.9 times[[1]](#footnote-1) more productively, meaning that a software engineer equipped with the tool types 993 lines of source code per working day and creates as much value as a team of 6 software engineers plus a team lead which are not equipped with the tool.

A few examples of the architectural diagrams, which our prototype can produce now, are in Figure 3 and Figure 4 in the Appendices section. The commercial tool should integrate well with visualizers, including search and navigation, and in later years, we should develop our own better visualizers of large trees and hierarchies. Of the available visualizers, also researched by the author, there are 2D and spherical 3D visualizers, treemaps, etc. You can see other different visualizations in the research document.

We plan SaaS model (Software as a Service), where customers upload their anonymized data (i.e. not the actual source code, but only a graph to be clustered, generated and stripped by the plugins we provide) to our servers. Then at our servers we do the heavy computations (including on GPUs which the clients may not have), add our fee to the charge for this client, and send the results back to the client. This way (doing main processing on our servers) we do not impose any restrictions on what hardware the clients can use, and we ensure that the processing time is acceptable for our clients (which may not be the case if we would do the processing on clients’ desktop computers without GPUs). In addition, this way we protect our Intellectual Property better: algorithms and the other pieces of our applications.

# Organization & Management

This is a startup company. There is currently only the author looking for a team or investments to hire a team. The goal of the initial team will be implementation of the first tool and its sale to the customers, involving explanation of the value which our first tool gives to a customer and software engineering best practices given the presence of our tool. That is the reason why the software engineers who implemented our tool are the best people to sell the tool: because our software engineers will be able to explain the advantages of our tool to the software engineers of the customer companies.

The initial team will consist of 3 departments:

1. Kernel group: to develop the server-side application
2. Java group: to develop client-side applications which let our tool comprehend the programs in [Java programming language](http://en.wikipedia.org/wiki/Java_(programming_language)) and let the clients use our tool conveniently from Java IDEs (Integrated Development Environments, e.g. [IDEA](https://www.jetbrains.com/idea/) and [Eclipse](http://www.eclipse.org/))
3. .NET group: to develop client-side applications which let our tool comprehend the programs in the programming languages of [Microsoft .NET Framework](http://en.wikipedia.org/wiki/.NET_Framework) (e.g. C#) and let our clients use our tool conveniently from [Microsoft Visual Studio](http://www.microsoft.com/visualstudio/eng) IDE as a plugin.

Each of the departments will consist of 6 programmers, 1 software architect and 1 team lead. The team leads will report to 1 project manager. The project manager and the software architects will report to the CEO.

We plan to work without Quality Assurance engineers (at least in the first year), because we prefer to cover our software with automatic tests so to minimize the need of manual testing. The software engineers would implement the automatic tests, so we should have a few engineers with some background in Quality Assurance Automation.

We plan to use a custom project management approach, taking the best from Agile and Waterfall. We will have weekly results & planning meetings and daily reports. However, we do not plan to follow all the ceremonies of Scrum/Kanban etc. rigidly. We plan to have some balance between coding and documenting, rather than completely abandoning software documentation, as some understand Agile. Starting after the first month or two, we plan to improve/extend our tool incrementally, in small iterations like in Agile. In addition, we will do mid- and long-term planning and design too, where possible and appropriate.

From the profits, the company will pay the deserved to every contributor, e.g. a researcher on whose research the basic research of Serge Rogatch is based, a teacher, an inventor, a consultant, an intermediary, etc. who facilitated creation of our tools. This covers the patents, the licenses, donations to open source projects used, and other royalties.

# Marketing and Sales Strategy

The demand for a computer program which comprehends other computer programs exists on the market because software engineers currently spend most of their time on comprehension of computer programs, rather than typing the computer programs. Some software engineering companies or even software engineers themselves might not realize this yet, and it might be hard to explain someone the value of the tool until the tool is created. When the first tool is created, we will be able to show them what they never saw before (rather than just theorizing about it so to build a distant copy of it in their imagination, as distant as the difference in perception between the teller and the listener), to let each of them use it in his/her specific situation and to teach them to comprehend their computer programs with a tool which automates part of the comprehension process which they formerly had to do manually (more precisely, with their eyes and heads).

The software engineers of our company will be answering the questions asked on the internet by software engineers of the potential customer companies, so to explain how their current problems can be solved with our tool and how to use the tool for productive work.

The pricing strategy is simple: we plan to charge $1 for one run of our tool. In one run our tool analyses one snapshot of a computer program. Because computer programs evolve over time (currently, not by themselves, but due to the changes made by software engineers), the customers need to analyze with our tool the same computer program many times – up to 1 run of our tool for each commit to source control (in the other words, snapshot) of the computer program, or even more often – several times per commit, while the software engineer tries several implementation options and looks how they fit in and influence the architecture. Approximately, if each software engineer of each customer company needs to run our tool once per day, then 1 000 000 software engineers in 250 working days per year give us $1 \* 1 000 000 \* 250 = $250 000 000 annual revenue.

To maximize the productivity of our customers, our company will provide them with advice and best practices on when our tool can be run to facilitate comprehension of their computer programs by their computer programmers, to let their computer programmers avoid making new mistakes and fix the mistakes which they made to their computer programs before they started using our tool, to let them take better decisions about the architectural design and implementation of extensions and improvements to their computer programs, to let them estimate how the quality (the potential productivity) of their computer programs evolves over time, to see which mistakes in the past are adversely affecting the present productivity, to prioritize the changes which the software engineers can do to their computer programs to increase the productivity ASAP, etc. .

# Financial Management

The startup cost is the cost of 1 year for 26 workers plus the expenses for office rental, computers and the infrastructure. The workers are 18 software engineers, 3 software architects, 3 team leads, 1 project manager and 1 CEO. According to the salaries known to the author and taxes/fees in Belarus, the cost of work in the Belarus is $56K \* 26 = $1456K for the year of development until we create the first tool therefore become able to get revenue. The company will also need an accountant, with salary plus taxes/fees about 37K per year. According to the office rental prices known to the author, the cost of office rental for the first year is 26 \* 12 \* $134 = $42K, which is 26 people times 12 square meters per person times $134 per square meter per year. The cost of computers is 30 \* $2.5K = $75K (we also need a computer for the accountant, and some servers shared across the engineering team). There are some more expenses for furniture and other hardware, which the author currently estimates as $13K. As the author understands [Microsoft BizSpark](https://www.microsoft.com/bizspark/default.aspx) program, as a startup we can get Microsoft Visual Studio, Windows and Office for free for the initial few years. The cost of licenses for usage of the other necessary software is 26 \* 12 \* $49 = $15.3K for the [agile project management system Rally](http://www.rallydev.com/platform-products/rally-editions) (Unlimited edition).

The total cost excluding the overhead unforeseen by the author is $1 638 300 for implementing the project in Belarus according to the costs known to the author. Assuming that the unforeseen expenses are 15%, the total start-up cost is $1 884 045. The amount can be split over the months of the initial development year, $154K per month to cover the monthly expenses. We can also change the schedule to start with a smaller team initially and then grow the team according to our capability to parallelize the work.

# Appendices

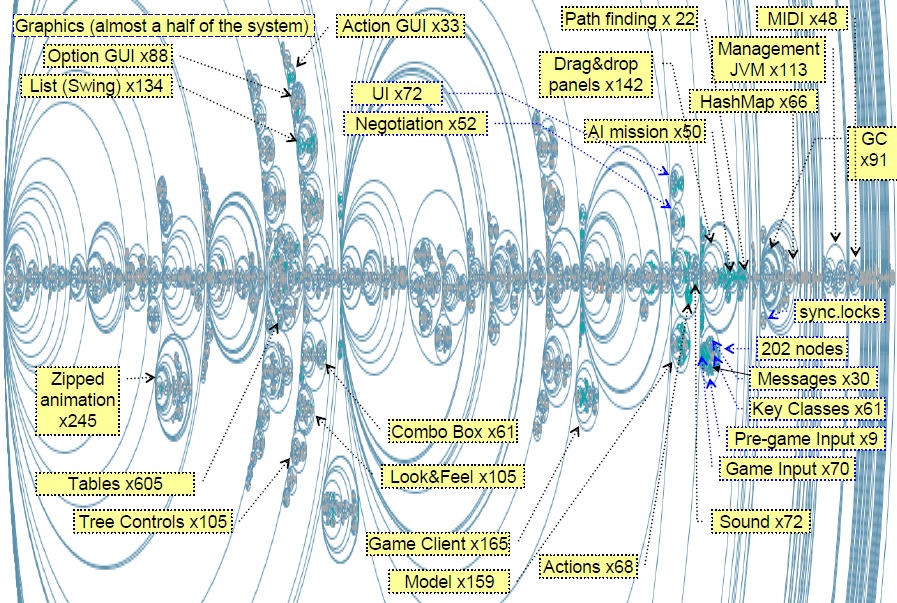


Figure 3: Example output of our tool - a structural diagram of FreeCol computer game

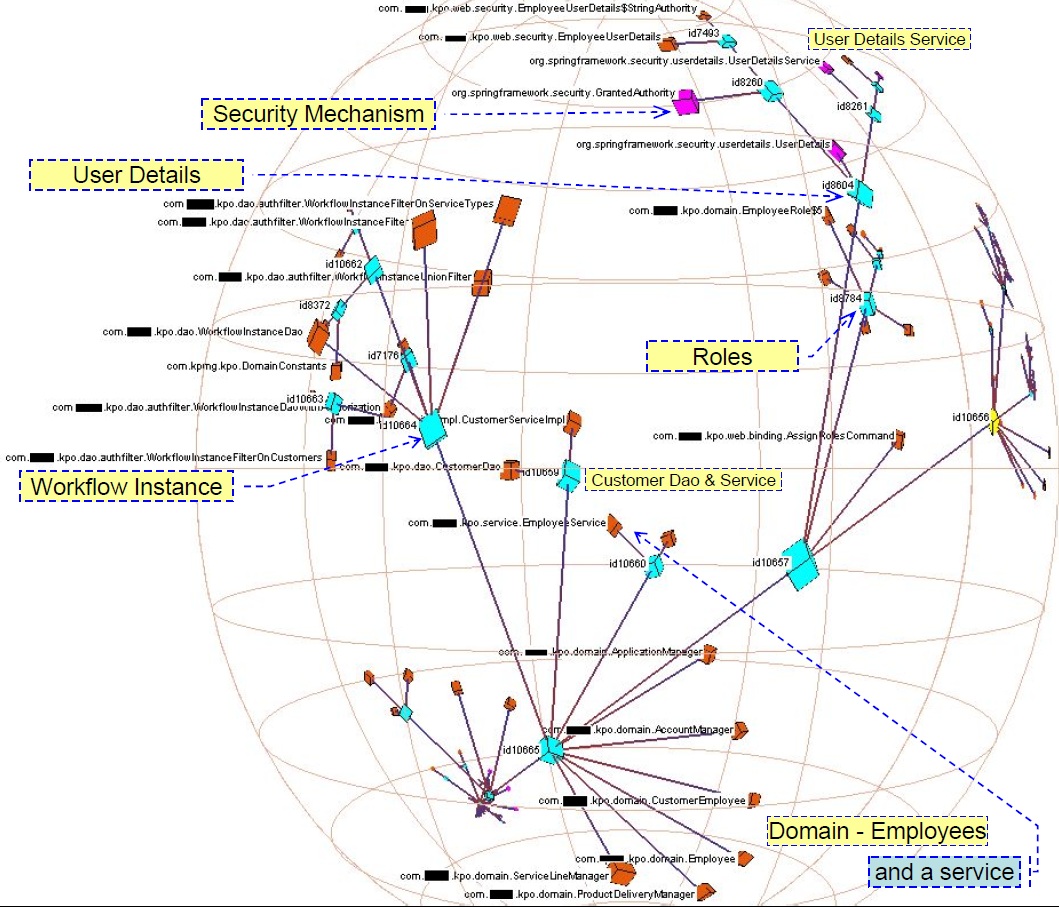


Figure 4: Example output of our tool for a software system - the tree of its subsystems is presented in Hyperbolic 3D space

1. 7 hours 36 minutes = 456 minutes spent on comprehension by a software engineer for each 24 minutes of coding at the speed of 6 lines per minute. If the software engineer equipped with our tool can comprehend the computer program 10 times quicker, then he/she spends 456/10 = 45.6 minutes on comprehension for each 24 minutes of coding. Then in a working day the software engineer can spend 480 / (45.6 + 24) = 6.9 times more time on coding: 24\*6.9 = 165.5 minutes of coding per day. [↑](#footnote-ref-1)