The Software Dragons

(ErrorLog:Documentation:Introduction).

(Version 1.2.4)
17th November 2004

El Sueño de la Razon Produce Monstros.
(The Sleep of Reason Produces Monsters.)

Francisco Goya

1 Introduction

This paper is the result of my experience in the area of the software development. Twelve years ago I had written my first program in Fortran. I was delighted with the possibility to transform a text written by a human in a sequence of rational actions executed by a computer. Right from the beginning I was interested in one other magic too. This was a process in the human's head that transformed unclear ideas into the strict text of a program. All this years I studied humans working with the computer. This time I have some rest. I'm currently not forced to write computer programs and can write about writing of computer programs. [sic][1]

The form of this article may be not usual for you. It is not scientific-like. I had read many wise books and know many scientific-sounding words, but I use simple terms. Here are no mathematical models, statistical results or quotations of "classical works". This paper is a fairy-tale for software developers.[2]

The subject of the article may look simple. It isn't. You should have some experience in real software projects to understand the concept. Maybe it isn't reasonable for you to read this, if you don't have any assumption what a Software Dragon may be. You can and shall interpret the things I write about in your own manner. There are a lot of examples in the article. They are added not only to explain the Software Dragons, but also to help you to remember examples from your own experience.

[1] I should apologise for my English. I write in English, but I don't think in English. As results there are some conflicts between grammar and contents. In all cases contents was preferred. Of course this aspect of the article is the subject of improvement. If you'll help me, please visit my site.

[2] Some people replied that I dislike science. This is not true. The situation is simpler. I hope I have what to say and I won't hide my toughs behind complex words.

This article is an introduction into the Software Dragons. It describes the problem in common. From this article you get to know how and for which purposes you can use Software Dragons in your own work.[3]

2 The Software Bugs

The Dragon is a very good term to describe many problems of the software industry. To introduce it I begin from other things. This is not the easiest way, but I have my own reasons to choose such course.

At first consider one more simple and ordinary creature. This is well known "program bug". There was many times explained that the term "bug" shouldn't be used. I think you have read some of these articles. We'll consider one more interesting question, why programmers *use* this word and why they *like to use* it?

Let's define "bug" as a virtual creature that had crept in the program, hides in the program code and disturbs the correct execution of the software. The "real" cause of the problem is usually an error of a programmer. As rule this is the programmer seeking the "bug". The will to hide own faults is one of reasons to speak about "bugs", but this is not the primary motive. The programmer itself acts in the manner he is catching in his program a small and crafty insect.

In the companies, where bugs are attribute of life, the work is usually boring. The time pressure, bad management and absence of rewards may make a programmer unhappy. As a normal human he's seeking things that can make his existence more valuable. Frequently solution is the virtual world of games. A man can take pride, if he have killed all monsters and got more points as yesterday. Many programmers I know are fans of computer games.

If you remember how looked computer games fifteen years ago, you can say, that the modern debugger is much more interesting to play with. I had spoken with many people about grounds of their bad programming style, absence of comments, bad documentation. One of common signs of these people is the love to a beautiful game. The game getting fun. The game whose results are important for other people. The game where the reward is money and better staff position. This game is a war against small software monsters. Most gamers call it the "debugging".[4]

One may say these are not monsters and "bug" is a technical term. Don't be mistaken. If a human is looking like he is playing, if he is feeling like he is playing, if he is speaking like he is playing, then he *is* playing.[5]

[3] You cannot find here prescriptions how to treat your problems, at least your current problems.

[4] Ruslan Shevchenko had noted that the "debug" game is interesting by itself and is not caused by bad management. This is partially true. The value of "debugging" may be extraordinary high for the people that like seek solutions of very complex problems. As rule these people are already scientists. I had asked one such scientist about his wonderful programming style. He replayed that he doesn't need formal things and even comments, while he is working in a RESEARCH organisation where the members ether have PhD or masters degrees.
I should note here that some other scientists have the best programming styles I had even seen.

[5] The psychologists have found many games in the humans' behaviour. The world of software is a would of many kinds of very interesting games. There are games for programmer and for user, for clerk and for manager, for beginner and for expert, for one participant and

If you have understood this point of view on software bugs, we can return to the primary subject of this article. The bugs are small software monsters. They haven't power and are enjoyable to play with. I won't describe them below. We'll consider the more dangerous and bigger kind of virtual software monsters. The monsters who bring not pleasure but terror.

To come to The Software Dragons we need firstly go into other hidden areas of the programming. Let's consider the nature of the software.

3 The three results of a software project

Suppose. One company has finished a software project. The management values the results as very good. The firm sells many copies of the software. The customers are satisfied with the product. Everything seems to be OK. Of course the product has some problems. There are some errors in the software. The users will get some additional features. On the market come some new programs and the firm's program must be supplemented to support them. None of the problems seems to be dangerous.

The firm starts one new project to solve these small problems and produce the next version of the software. The second project goes not as well as the first, but nobody notices any danger. Only after missing of many deadlines and exceeding of budget management becomes the opinion that the second project goes not normal and simple changes are not simple. Sometimes it's too late. New version cannot be ready in the time it should be and the company loses its market share.

This situation repeats many times. It is described in many articles and books. Most of them discuss the errors and the problems of the second project. Only few of them consider the first "successful project" as the primary source of future failures.

The problem is that usually only two of three project outcomes are considered. Let's describe them from the point of view of a software firm. It'll be helpful for future use.

The first and most visible result of a software project is the software. It is the source code passed through compiler and linker. These are project documents being processed by computer for one other computer that executes the program code. There are some variations, but the idea is clear.

The second result is the documentation for the users. We consider all the information user can get from any sources in the way other than trying to work with the software. These are user manuals and tutorials, advertisement and support, books and articles, etc. These are project documents being processed by people for other people outside the project.

Usually the project is valued as successful, if these two outcomes are good. They are important for financial results of this project, for the customers, for the market position of the firm. In many cases they are only results to be taken into account.

In case the project were the last project of the firm, this opinion were right. Of course it may be last, but it would be strange, if the management predict that the next project won't start or will be full unsuccessful.

---

for big teams. The description of them is the subject of the other article(s).

The third outcome of the project is usually forgotten. It can be invisible for high management, for marketing or for support departments, even for the management of the project itself. Unfortunately it can be more important as two others. This is project information being processed by people and sometimes by computer for members of this project and for members of future projects.

There are many different software standards, but only a little part of them are made for the improvement of the third outcome. Most others I have seen are applied only to report to management about the presence of the development standards. Usually none of the project members accepts the third outcome as one of the project tasks. As result only a little part of the project documents is good enough to be used after end of this project. The other part is bad enough to rise a big number of problems in the following projects.

We approach to the world of Software Dragons. They are virtual software monsters. We cannot observe any material signs of them. We cannot measure them. Only one thing we can see are traces of them. These are missed deadlines, devoured resources, loss of quality and killed projects.

Software Dragons are virtual. To exist they need a material object to reflect them. The material mirror of the virtual Dragon is a project's staff. To define Dragons we must consider the developers before. Let's describe what they are doing in the manner similar to our description of developers being in the world of bugs. We'll consider the third outcome of the project.

4 The third outcome

The software can be described from different points of view. The best procedure for our tasks is to consider software as a result of the human's activity. To simplify our model we'll speak only about the mental activity. We won't investigate such things as the speed of typing or the number of errors by drawing a diagram.

In other words our primary subject is the process in the human's head. There are many models describing it too. Let's make or model maximal simple and maximal applicable to solve real problems. We get out all activities that can be made by a computer and all human's activities that aren't bound with the information. The rest is the human's ability to make decisions.

There is a principal difference between solution and decision. The solution can be found, if a person has a task to solve, all information needed and a method to apply. In case of decision a human may or may not have enough information, appropriate method and even the task itself. He can have even an information conflict, where facts he knows demand quite different decisions. To find a solution one can use formal or empirical methods. To make a decision a human or a group of people must use a mixture of knowledge, experience and intuition.

We won't consider the chemistry of the process of the decision here. Let's use the following simple black box model:

1. A human or a group of people gets information to make a decision.

2. Different factors are applied to this person or group when he or it is making a decision. These factors can be random or systematic, measurable or hidden, informational, psychological, social etc. For us is important only

the existence of them and the fact that we cannot take into account all the factors influenced the decision.

3. The decision is made and applied to the software. In other words the person or the group decides to do something or not to do something with the software product and/or the process of software development.

We consider a simplified model. In our model the software is the result of all decisions about software.[6] The word "decision" will be used below in meaning of the result of a decision, not of the process of making it.

In general the decisions about software are hierarchical and sequential. The first and main decision is the decision to start a software project. All other decisions are depended from it. In a real project some decisions are independent, some can be made simultaneously. We haven't any method to prove such facts. To simplify our model we don't consider this problem now.

For instance in our model the decision to code an operator in a function can be made only after one have decided which function must be coded and which programming language must be used. The decision to code a class member function can be made only as a result of the decision to make this class. The decision to use a class in one module cannot be made without the decision to make this module, etc.

Now we can define the third outcome of a software project. To make a decision one need the information and the experience. Let's say that third result of a software project is a distributed knowledge base about the software product and the process of development such kind of software. If we consider this project's outcome together with the project's staff, we can speak about a kind of distributed expert system. In most cases the knowledge base cannot be separated from this expert system because large part of it is hidden in the heads of people. Of course this part is subject to lose by time and staff changes.

The quality of this knowledge base and this expert system defines the quality of the current software development process and of the future projects. The second problem must be considered is the cost of them. DoD organisations make complex and expensive things to raise the quality of this knowledge base. However the projects in such organisations fall too. The firms doing ordinary software business are unable to carry analogous cost.

There exist a problem how to get this knowledge base maximal cheap without loss of quality. The second problem is how to make this expert system maximal effective and stabile. In other words there exist a problem how to force good decisions about software and how to prevent bad in the future.

The solution of this problem is not the primary subject of this paper. Let's return to the main path.[7] Now we are ready to speak about Software Dragons.

[6] I repeat again, we consider a model. The live software development is more complex. Don't miss the model with the reality.

[7] Some people, who had read first drafts, asked me to continue this theme. This is the subject of other parts of the project "ErrorLog".

5 Why the Dragons?

There are two possibilities to write an article about software. The first and most common of them is to describe a problem then to give a solution. This is a good way to introduce a new Method or a new Measurement System or a new Tool. Most of the articles I have read are written in such manner.

This article is written in a form rare to find today. I have described the problem. Now I'll explain why this problem is more complex as you mean. This is a good way to introduce Software Dragons.

I think you already have got at least two ideas about third outcome of a software project.[8] Suppose I'll write the first form article. To introduce the solution I'll expand the above black box model to make a model of the solution of a problem. I'll describe ways a project member or a project team gets the information. I'll describe some factors that influence the decision. I'll describe how the decision can be applied to the software product and to the software process. To make the model complete I'll say that other factors that may influence the decision are small or unimportant. The most common and simplest way is of course don't speak about them at all.

Such kind of solution is not correct. You can apply it only in the universe of the model. An attempt to use it in the real world for a real project can lead to disaster.[9]

What is wrong? Wrong is the presupposition that the factors you cannot describe, measure or observe and the factors you don't know about may be not considered. They cannot. Even invisible factors can be neither small nor unimportant.

This fact is negative. It cannot be used for any constructive purpose. To avoid this problem we need to introduce some positive virtual objects having the same effect in the real world.

The most important task for us is to get results at least not worse as we want. Firstly we need to separate negative and positive effects. The virtual object having the positive influence on the process of a decision is well known and has the name "good luck". I won't describe it below. We're interested about evil things. We'll describe the nature of "bad luck". Let's consider what may be wrong with a decision.

6 The bad decision and its role in a software project

Let's define a decision as bad, if one of the following decisions must be the opposite decision. This definition is pure theoretical. In most real situations we don't know what decision *must* be corrected. We cannot prove a decision. In case of a solution we may find a formal method to check it. There are no such methods to examine a decision. Usually there is a person who says that his own speculations are the correct proof. Unfortunately they aren't. The worst fact is that they cannot be.

[8] If you haven't, you may have not enough experience to understand this paper.

[9] Of course it may be successful too. In this case I can speak about "proved solution". All failures may be explained as the results of incorrect use. Well, a new kind of Silver Bullets is ready to be used.

For practical purposes we can use two following "soft" rules:

1. If one decision is the opposite of one previous, this previous was probably bad.

2. If a software project has fallen, some bad decisions had probably been made.

As you see we can form an estimate only for the decisions in the past. More interesting is that such estimation is a decision too. Consequently in the future we may detect that *this* decision was wrong.

It is wise to find a bad decision early. One decision implies some work to realise it and a number of additional decisions that imply additional work too. If the primary decision is bad, a part of this work was probably not needed to be done. This is waste of time and resources. We can define it as *the cost of the bad decision.* Of course if the bad decision was corrected before this unnecessary work was done, the cost of the bad decision is lower. Consequently we can speak about the *potential* and the *actual* cost of a bad decision.

One other interesting thing is the correction of a bad decision. Common practice is to make a big number of lower level decisions, where the right action is to come to an opposite decision of the same level. The economical reason may be the attempt to save actual cost of the bad decision. The political reason may be the "polite valuation" of the decisions being made by higher level management. In comparison to getting results in a right manner this leads to waste of time, money, product quality, etc. Let's define this difference as *the cost of the crooked way.* It is a part of the actual cost of the bad decision about the correction of one other bad decision.

We need a name for cost of developing software without bad decisions. Let's define it as *the optimal cost.* It may be possible to reduce optimal cost with some magic, for instance due "good luck". This isn't the subject of our article. We'll consider the possibilities to reduce the cost of a real project to reach the optimal cost.

Usually the optimal cost is less than the cost of the crooked way. As result the correction of the bad decision is more profitable. In some situations the opposite may be true. Let's define this as the *stabilisation of the bad decision.* Principally the stabilisation can be endless. This is usual for low level decisions. The high the level of a bad decision is, the quickly the cost of the crooked way rises. If a high-level bad decision won't be corrected, the potential cost of it shall be paid sooner or lather.

The interaction of bad decisions may be complex. For our purposes we need remember that it is not lineal. We can estimate the potential cost of a bad decision as lower or equal to the potential cost of the bad decision of the higher level. If a result of one bad decision is one other bad decision, the actual cost of the first can rise to its potential cost. Consequently we can say that a bad decision may save actual cost of one previous. It isn't good to believe in this. The opposite result is more frequent.

I feel this theoretical speculation shall be explained. Let's consider some examples.

Our first example is quite extreme. Suppose management decides to begin a project to develop a software and there is no potential market for this software.

The primary decision to start the project is a bad decision. Even all lower level decisions and all other work is perfect, the project cannot be successful. The result of the project is useless. Theoretically the potential cost of this bad decision is infinite. In any real situation it is limited by project's resources. In our situation the potential cost of the bad decision is the upper limit of time and money the company can waste to develop this useless software. Of course we may add the money spent to marketing in empty market, the indirect losses and the cost of support too. These are results of other bad decisions. In our model we'll consider only loses within the budget of a software development project.

Actual cost of the bad decision about useless project is as rule lower than potential cost. Firstly management may save money by applying new progressive support tools and development methods. The staff experience is one other thing that may save budget. It is a positive effect of the project. Of course hardware and software that the company have bought for this project may be used for more reasonable purposes too.

As second example consider a decision to use a programming language that isn't appropriate for the design of the software product. The potential cost of this decision are all resources the company can spent for coding and testing. This extrapolation has the following reason. After the project reaches some definite size the problems with the language will rise the complexity very quickly and the development won't be done further. Of course the project's staff will adapt the language to its tasks. Many support tools and crafty tricks may be developed. The resources spent for them are the cost of the crooked way. This leads to the stabilisation of the bad decision about programming language. If a next task is to add some little function, it is not reasonable to change the language and write all the source code anew. Bigger tasks can be unreachable with the old programming language. If the cost of the crooked way exceeds the resources of the project, the project usually falls. Only the correction of the bad decision can help. Of course the situation is fatal, if there are no resources to do things in the right manner and the crooked way cannot reach the project's goals too.

I hope you have felt, what the bad decision is and I can close this theme.[10] Didn't you forgot that we'll speak about Software Dragons? Now you are ready to accept the following definition.

7 The definition of The Software Dragons

Software Dragons are virtual software monsters. They have evil nature and may grow to very big sizes. The power of one Software Dragon is usually more than the power of one project member. The Software Dragons devour resources of the software project. They force project staff to make bad decisions and to spend resources to materialise them. They hide bad decisions and prevent the correction. In case a bad situation in a project cannot be hidden, they force people to follow a crooked way. They do all possible to raise actual cost of a bad decision and the cost of a crooked way. The highest goal of the Software Dragons

[10] The first reaction about the cost of a bad decision was quite morbid. I want to warn you. It may be interesting to write an article, a book or a doctor work and find hundred and one method to convert my descriptions into numbers and formulas. I think this is not worth to do. Either you get the method less precise than a meaning of an ordinary expert or the cost of the measurement will be more then the cost you try to find.

is to devour all resources that company may spent for a software project. If they reach this goal, the project has no chances to be successful.

The Dragons live around and in software projects. They grow devouring resources. They force bad decisions to get their food. The more bad decisions the people have made, the more the amount and bigger the size of the Dragons is. We can say that a software project has the fourth outcome. This is an army of Software Dragons. Let's speak that a software project has three good and one bad results. If the power of the bad result is bigger than the power of three others, the following project has big chances to fall. If a Dragons' army exceeds the strength of the good outcomes before the end of a project, this project is going to be hopeless too.

For instance consider the situation with a not appropriate programming language. If the project is "successful" finished, the company has not only a "working software". The satellite of this software is the Dragon of Previous Expenditures. It will prevent the change of the programming language and protect bad software design. As result a lot of money will be lost by maintaining. The more resources are spent the more power this Dragon has. The tangle of problems will grow continually until all resources are intended to feed the Dragons.

8 How Dragons can help you

Well. We know who Software Dragons are. They disturb your work. They are evil. They are dangerous. They are your enemy.

This is a weighty argument to study Software Dragons. If you know your enemy, you have possibility to make a defence. You can protect yourself from their evil business. There are many advantages to speak about Software Dragons.

8.1 You can call things with their names

The most important ground to introduce Dragons is the possibility to call things proper. Speaking about Dragons you describe our world. This description may be unusual. You cannot find a formal proof for it. It may sound "not serious". At the other hand this explanation is clear. It contains no "wise words" and "scientific definitions". Such things aren't good in themselves. Special terms may be used to clear the thoughts and to hide them.

For instance a mathematician uses special terms in his proof and an other mathematician gets the whole information from few words. A physician uses Latin terms to describe illness and a patient doesn't understand anything, but gets an impression about wise doctor. The second situation repeats in the software branch much more frequently. The Software Dragons can help you to avoid the smog of "scientific" words.

For instance a manager says that he'll buy a software system to introduce the new process of the software development. He boasts that this process saves a lot of money. It increases the productivity of the staff in many times. It uses best scientific proved methods. As result all projects shall be done in time and within budget. This is a well-known fact. The Method is simple and can be used by ordinary persons. The manager is going to fire unnecessary experts and

won't waste money for consultants.[11]

In case one believe in the words "scientific proved" and "best", he can make a big mistake. The software development is an activity of humans. This is a process of thinking. The science doing research of this process is called psychology. It doesn't know what is "best". It is very careful with "scientific proved". (Above all if a proof was be done on statistically incorrect group, what is the usually situation in SE.) Some facts in psychology are well-known, unfortunately the interpretations of them are very different.

Of course it is possible to make a psychological analyse of the manager from our example. We may mount a scientific base under the analyse of the decision about the introduction of "the new Method". This analyse should be done by professional psychologist and is complex. Fortunately we can estimate this decision much more simple, if we translate the manager's words in the terms of Dragons. It would sound like this:

> The Dragon of Lost Control is devouring our projects. To win the war against this Dragon I'll buy the newest Silver Bullets. This is The Magical Universal Process. This is a best way of treatment for our company. It promises rapid changes and has extraordinary magical power. I'll apply it, kill the Dragon and become The Lord of Software.

8.2 You can avoid the fear and other emotions

The second ground to introduce Dragons are emotions. Software development is a humans' activity. The problems and mistakes are humans' problems and mistakes. Consequently they are bound with strong emotions. Nobody can speak about personal errors without feelings. Even the phrase "I had made an error" is accepted different as "An error had been made by me."

Emotions disturb the analyse of problems. They hide and distort information. They make a point of view very subjective. As result each error analyse becomes the game of the searching of a scapegoat.

To avoid this problem we need use something that isn't bound with strong feelings. Virtual software monsters are good tool for this purpose. We have seen such situation with software bugs. It's simpler to speak about virtual monsters than abut personal mistakes.

The strongest of emotions in software projects is the fear. It is the cause of many mistakes. Fear paralyses the human's logic and force to make bad decisions. Very frequently people say, "I know how to make it proper, but I cannot act in such way." To abstract from the source of the human's fear we can consider a Dragon. The human may be afraid of a Dragon. The Dragon is big enough and harmful enough to force the human make the same mistakes.

The analyse of feelings in a group of people is a complex and expensive measure. Sometimes collective emotions are more powerful than one separate team member has. Evil Dragon's nature can help to describe the sources of errors without complex psychological analyse. It can help to explain collective errors. The errors which weren't be made by each separate person alone. As result it can help to find a possibility to avoid such errors in the future.

[11] And they do this. I seek experts and "best programmers" that left a company in a such situation. I have a number of questions to ask.

Let's consider one example with the Dragon of The Deadline. A group of programmers is developing a software module. Each knows that the module must be redesigned. Nobody begin to do this. The Dragon of The Deadline repeats consequently, "We haven't time for such big rework." As result the group is trying to solve problems with "small changes". Consequently the deadline is missed. The delay is much more than the time were needed to rewrite the module from the beginning.

You can analyse the cause of this error in terms of Dragons. You can ask group members what they mean about the Dragon of The Deadline. You can discuss with them, how this Dragon has forced the group to do wrong things. Consequently you can find together, how to protect the next project from the influence of this Dragon.

Of course you may ask them the direct questions too. For instance are they afraid to say management about unrealistic time planing? In this case the most likely answer will be "Not". Above all in case a manager is participating in the discussion.

The most interesting thing is that people don't lie. They are afraid, but they think that they aren't. Feelings are complex to analyse. At first if they are the own personal feelings. A human older 6 years can excellent lie himself.

8.3 You can compose a life model

The Dragons can seem antiscientific. They aren't. Let's say that the Software Engineering may have two directions of research. The first is science and describes how the world is. The second is "science" and describes how the world should be without consideration of the reality.

Each science operates with models. These models are logical constructions based on axioms. A model should mirror the real world. Unfortunately it can produce a wrong image. At first the construction of the model my be not correct. Much more terrible is the fact that axioms may not match the real world. A model can be proved logical or mathematical. An axiom cannot. Only one possibility to proof axioms is to predict something and then to measure the results of an experiment. If experimental results don't conform to predictions, the axioms are probably wrong. Of course there exist experimental errors too.

The human is a very complex object to measure. A group of humans is much more complex.[12] Even you measure ten parameters there exist 100 others that you don't consider. Maybe there exist 1000 additional you don't know about.

Much worse for measurement is the fact that humans have his own will. They aren't computers. If you measure them, they know that they are measured. They have their own targets and change results of your experiments according with them. Consequently you can always get results predicted by a theory, even the theory is completely wrong. You get what you will, not the true.[13]

Using Dragons you apply your intuition. This tool is created by millions years of human evolution. It is tuned for deal in complex situations. You observe facts from real world and synthesise an intuitive model for practical

[12] Are you bored from my infinite repeats of this "is more complex"? This is a clear sign that you don't understand *how* complex the human is.

[13] There are precise psychological experiments. Unfortunately they are very expensive and complex to implement. For instance could you apply electroshock to get pure true unconscious reactions?

purposes. It may be not-scientific, but gives better results as the model based on a "good theory". You create an other point of view. You have no artificial laws you should to follow. If your model don't conform to the reality, you have no "scientific-based" obstacle to change it. Even in case the Dragon-based model is wrong, you may find the way to overcome your problems. You have no "scientific" limits and can use your creativity.

8.4 You can stimulate your own creativeness

The Dragons are one other view on the problem of the software complexity. If you want overcome Dragons, you must concentrate not on the measurements. You must find ways to reduce the complexity and prevent the addition of new problems.

Most problems of software development are psychological problems. Only few experienced professionals are able observe and describe them. Most people feel that something is wrong, but cannot understand what. Dragons are a good psychological method to activate hidden knowledge. One can describe with metaphors the things he cannot formulate any other way. Dragons are partially a fantasy, partially a game. Even the others cannot understand the proper interpretation, they can feel it.

One other good thing is the personification of problems. Abstract dangers don't impress. The people know what "should be done", but don't do this. A Dragon is a good ground to make things right. Even a virtual Dragon is more real for a human than hundred tables, diagrams and guidelines.

For instance a developer doesn't write "useless comments". He knows the purposes of comments, but won't waste time for such things. He thinks that nobody will understand the source code wrong. If the developer knows about the Dragon of The Source Code, he knows the face of enemy and has one other position. He understands that he *must* explain what the code does, because the Dragon has evil nature. If he don't write about possible problems, the Dragon of The Source Code want to use them to introduce errors in the program. The programmer now *protects* the code from enemy. This is much more effective than to follow abstract guidelines.

9 The Dragon Wars

We consider Software Dragons to find a procedure to overcome them. Let's describe firstly the usual counteraction. This is a war against Dragons.

Consider a typical software project. Even the people don't speak about Dragons, they act as they fight against big and powerful monsters. Unfortunately such wars begin when Dragons have a lot of power.

A little worm may months or years live in the shadow of a big project. Nobody knows about him. Nobody is afraid of him. Even nobody see how the Dragon grows in the dark.

But the time comes and the people understand that the project is in the shadow of a big Dragon of The Project's Complexity. All of the deadlines burn in the Dragon's fire. The excellent project structure breaks down from one hurt of the Dragon's tail. All resources are devoured by Dragon.

The people begin to make war on the Dragon of The Project's Complexity. They don't attempt to understand the Dragon, they don't speak about Dragon, but they seek Silver Bullets to kill the Dragon. They arm with CASE tools, modern methods, later generation languages and a heap of software. They battle against The Dragon of The Project's Complexity and the victory comes near and near. The Dragon is smaller and smaller.

But the time comes and the people understand that the project is in the shadow of an army of Dragons. Different kinds of Software Dragons were coming from black shadows of CASE tools, of modern methods, of lather generation languages and of the heap of software. Deadlines burn again. Project structure breaks. Resources are devoured.

It may seem strange, but the most frequent conclusion from Dragon Wars is the decision to find Silver Bullets with more magical power. Silver Bullets casting industry is a very profitable business today. Silver Bullets have now "scientific" names and wise explanations of magical effects. Unfortunately such things cannot help against Software Dragons. The "scientific" smog is created by a Dragon too. This Dragon is one of most damaging Software Dragons. His name is The Dragon of Silver Bullets. This Dragon makes fog of "scientific" words and tells "proved" tales to hide other Dragons.

If somebody is under power of the Dragon of Silver Bullets, he cannot overcome other Dragons. He doesn't see the real problems. He believes in the existence of a simple and cheap solution. He doesn't solve problems, but seeks *The Right Silver Bullet.*

Maybe Silver Bullets are good. However we know that wizards are rare to meet today. People in wizards' clothes are more frequent, but they cannot give to the bullets enough magical power. Even they use wise words and "scientific proved" methods.

The people being under control of the Dragon of Silver Bullets are charmed with buzzwords. They open their purses and hope in the power of magic. If you know about Dragons, you know that to kill them you need not only bullets made from silver. You can check the real magical power of a wizard who sells such bullets. For instance you may ask about his magic wand.

The second problem of Dragon Wars is the fact that Software Dragons aren't absolute. One may describe problems of the same project in his own manner. For instance project members can see The Dragon of Abuse Management. In the contrary management can consider The Dragon of Stupid Employers as the ground of project's difficulties. Unfortunately the strategy of the Dragon War is dependent on the enemy Dragon. In our example the management will attack the employers as servants of the Dragon. The developers will protect themselves from the Dragon of management. As result the development of software will be forgotten. The project becoming the war of Dragons. All power will be spent on intrigues.

The problem of Dragon Wars is the power of the Dragons. Such wars begin where the tangle of problems is too big. Some difficulties may be removed, where the complexity of other grows. As in fairy tales the hero killed dragon become the new dragon. No magical transformations are reasonable to be planed. Any change can bring only about 10 per cent improvement.[14]

[14] This rule may have the following explanation. The progress may be much more than 10

It is complex, expensive and improbable to win a war against big and powerful Dragons. Only possible method is to avoid the necessity of such wars.

10 How to overcome Dragons

Dragons cannot be destroyed quickly. On the other side they become big and powerful not immediately too. This gives a possibility to overcome them.

Small Dragons are difficult to find. The host of small ongoing problems disturbs to see a system behind them. Dragons hide themselves before they have enough power and size. To see Dragons you should attentive seek them. You must study the nature of Dragons and their behaviour.

The big aid can be the previous experience. The projects devoured by Dragons can show the source and the conditions of the Dragons' growth. Of course the best way is to learn on errors of others. The same mistakes repeat again and again in different projects. It is good to know how things can be done proper, the better knowledge is how they may go wrong. You should become aware of problems in your project. Even of small problems. If you know about a possible mistake, you can avoid it. If you know some examples and recognise a Dragon that generates such errors, you can make systematic protection. It helps you to avoid a class of errors, even you cannot predict the possibility of some errors from this type.

Dragons have evil nature. In case you don't protect your project from them, they will use week places to attack. Your defence should be optimal. It is not good to spend extra resources to make unnecessary protection. It is worse to spend much more resources to rescue your project from Dragons that are already breach bad protection. You shouldn't forget that if an error is possible, you must prevent it. In other case you get this error sooner or lather only because Dragons have evil nature. Don't feed your Dragons.

For instance each programmer knows (or at least should know) that memory allocation can fall. This is a rare event, but it is possible. An enormous amount of errors in different programs exist only while programmers hadn't insert a small code unit that checks the situation where no memory is returned.

Other similar example is the rule to assign an initial value to each variable. The unexpected value can influence the behaviour of program in rare cases and in an unpredictable manner. As in previous example it is simple to code an initialisation for each variable. In case these protection steps are ignored, the cause of program failure is difficult to find. The search of only one not initialised variable in a big program may (and many times had) waste many man months.

The protection from Dragons and prevention of Dragons' growth has one disadvantage. It cannot solve current problems. To overcome today's Dragon you must begin yesterday. The knowledge about Dragons cannot bring you a big help in a trouble of Dragon War. To save resources in the future you must

per cent, but adequate plan cannot be based on such hope. In case the company has a good process for software development, the improvements of such process cannot be big. In case there are possible the improvements of productivity about 50 or 100 per cent, the company has a very bad process yet. In other words it doesn't know how to develop software. Of course it is naive to wait from such company a correct use of the promising method. Consequently the bigger the changes may be the less probably they are.

spend resources today. This may demoralise somebody who hope in a quick improvement. Unfortunately good luck is not such frequent as bad luck. Rapid changes are possible. Most probably these will be changes from bad to worse.

In case you don't believe in a magical help, you must prepare yourself to long work. You'll go forward step by step. You reduce the complexity of your future work. The improvement is systematic. Your target is the absence of bad decisions. You cannot measure how many bad decisions you haven't made. You won't get impressionable results. You have nothing to show for the people that believe in Silver Bullets. Of course nothing except the results of your work.

The software developers are paid not for code writing, not for drawing diagrams and not for sitting for monitor. They are paid for making decisions. All other activities are assigned to support this process. If the developers make bad decisions, the money and time are wasted and the effectiveness of other things is not important. The Dragons disturb the root process of software development. This is a significant ground to pay attention to these virtual software monsters.

11 Conclusion

You have read this paper. I guess you feel that it is incomplete, muddled or controversial. It is such while it must be. As I had written I know how to make a "right" article. This paper is not an article. This is the first part of the documentation of the project "ErrorLog". Of curse this is not a "right" documentation. "ErrorLog" isn't a "right" project too. I'd like to tell you about the history of it.

The goal of this project is to find methods rising effectiveness of the software development in small companies. The first studies showed that the major grounds of ineffectiveness are bad decisions.[15]

The logical conclusion from such situation was to find methods that can prevent bad decisions. Unfortunately the attempts to discuss such methods with software developers, team leaders and project managers pointed the light on a very interesting problem. For the technical specialists the software management is a look over the back of the developer's head at the computer monitor.[16]

The "ErrorLog" demand quite other point of view. It was started as tool that simple logs the errors. Step by step the human as the source of these errors became more important. Now "ErrorLog" is a look at the face of software developer.

The first attempts to write a "right" documentation for "ErrorLog" failed. The people weren't ready to consider a human, but were seeking for a software tool that can free they from such necessity. On the seek for right form of the documentation I turned the log of errors into The Path of Dragons. Let's see which results can give this trick.

"The Dragons" are a documentation. It helps you change your mind and move your point of view.[17]

[15] The big companies are worse on this field, but they can live in such conditions, at first due good marketing. A major part of decisions in a big company is based on the internal politic not on the economic. A small company may die due few bad decisions and is much more interesting to be analysed.

[16] The results of these are many funny things. The Seek for Right Silver Bullets is not the worst of them. They will be described lather in "The Dragon of Silver Bullets".

[17] The good documentation always changes the user. The simplest situation is, where it

For several years I used to apply analogous methods. These were things far away from the software. The results of those experiments were good. Let's see what results will be this time.

You have made the first step into the world of Dragons. A step into black areas of human's mind. The black areas where monsters live.

There are many kinds of Software Dragons. All of them have his own effect on the project, on the project team, on the team member. Each Dragon has his own character. Most of them use not only intellectual but also emotional levers to steer a human or a team by the process of making decisions. The world of Software Dragons is very interesting but very complex. They have a good defence. Usually different kinds of Dragons fight together. The facility to hide is excellent too. It's impossible to study simultaneously all Dragons leaving together in one software project. Each of them is a subject for a separate study.

In the next part I'll describe the simplest kind of the Dragons. He has pure informational nature and cannot disturb the emotional sphere of the human. This is The Dragon of The Source Code.

12 Acknowledgements

At first I'd like to thank Gerald M. Weinberg for his excellent books, for his useful comments and for the suggestion to start the changes from myself.

I would like to thank Andrey V Khavryutchenko, Ruslan Shevchenko, Michael V. Tokarev and George Seriakov for valuable help, comments and criticism. I would like to thank Vladimir Koen-Tsedek for his explanations of language problems; Brian Henderson-Sellers and Paul Herbert for editing of first versions of this paper; Alexander V.Didytch, Alexander Drougov Anton A. Mints, Bohdan Sheptunov and Bernhard Treutwein for useful comments; Simon Cant for the model of the cognitive complexity of software; the members of RCSE group for very interesting discussions and experiments.

I would also like to thank all those involved in strange discussions about the nature of software and answered my stupid questions.

Of course big thanks for all people which thought me to understand the human.

The text is written in LaTeX.

adds knowledge. The better documentation teaches the user to live with new knowledge.