The Dragon of The Source Code.

(ErrorLog:Documentation:StyleGuide).

(Version 2.3.0) DRAFT
17th March 2002

> *...everybody knew about this in 1980, and even 1970 and 1960. What's interesting is that nobody did much about the Y2K problem back than. Even more interesting will be that we will have a 2010 problem, in some places.*
>
> *Gehrald M. Weinberg*

1 Introduction

The second part of "Dragons" is an example of effects described in the article "The Software Dragons." If you hadn't read the first part, you may not understand some used terms.

This article doesn't describe new methods, powerful tools, fashionable programming languages or others things that can decrease the first three outcomes of your project, I'll describe here how decrease a dangerous part of the fourth outcome, your own Dragon of The Source Code. There are many papers full of good advises. This one is quite different. It tells you why these advises don't work.

To understand the Dragon of The Source Code you need to have some experience in the maintaining of the code written by others. You need to know that the time necessary for "small changes"

- may be much shorter,
- may be equivalent
- and may be much longer

than the time necessary to write the program anew.

If you haven't got such experience, you may not accept things I'll speak about.

2 The evil nature of The Dragon of The Source Code

The Dragon of The Source Code is the fourth outcome of a software project stored in the text form. This is a bundle of all bad features of the source code and of the correspondent documentation.

The life cycle of The Dragon of The Source Code is very simple. If there is no source code present, this Dragon doesn't exist. The bigger the amount of source code is, the more place to live he has. If the source code won't be used in any way in the future, the Dragon die.

The Dragon of The Source Code creates problems by changes in the source code. He may make the correction of errors and the adding of new features very complex. Sometimes the simplest solution is to throw the old code in a trash can and to write the whole software anew.

The power of the Dragon of The Source Code depends not only on the size of the code. Much more it depends on the quantity and "quality" of implemented bad decisions. In our model bad decisions are forced by different Dragons. We may speak The Dragon of The Source Code is a darling of them. He is fed and supported by other Dragons. Well powered and underpinned Dragon of The Source Code can make the software unchangeable. His favourite music is the panicky cry "Don't touch the code! It works!" (Of course this decision is initiated by Dragons too and is definitely bad.) [1]

The Dragon of The Source Code may be hidden. The trivial situation is, when nobody works with the code. Let's say the Dragon is frozen.

More interesting is when some people work with the code, but don't touch the Dragon. They may know dangerous places and don't enter into. They may even become Dragon's servants. They cover up internal errors by "protection code" to hide visible effects of failures by program runs. The software is seemed to be good for use and the developers do not need to go into Dragon's house.

At a certain moment the amount of errors makes this game impossible. The developers are forced to change holy texts. With the whole power meets them the Dragon in the wilderness of ancient errors, patches, and patches of patches. The first step into forbidden zone may kill "working program".

The Dragon of The Source Code may wait his historic moment many years and decades. For instance until new millennium.

Each text of a program has his own Dragon. To get this Dragon in your project you need to get the source code. It's enough. As soon as you are forced to make something with the code, you meet the Dragon living in.

An experienced developer can feel the Dragon's size early. If the Dragon is big, the proved fighter begins "small changes" by rewriting of the source code he shall modify. If the programmer is young or self-assured, he obtains experience.

The software industry has made many kinds of equipment for wars against The Dragon of The Source Code. If you get advertising "for software developers",

[1] This is an example of the raising of cost of the crocked way. Unfortunately we can measure it only after the project comes on the right path. At this moment the concrete value is not important.

you have seen such things. We won't consider them here. We won't study how "to find the safe way in the chaos", how "to see what the function *really* does" or how "to make thousand pretty diagrams from thousand ill-designed classes".

Let's consider quite different problem. The Dragon of The Source Code is being born, when somebody begins to write code. This Dragon is growing together with the amount of LOC.[2] This Dragon becomes cunning, when somebody makes a silly error. This Dragon gets power, when somebody implements a bad decision.

If the Dragon of The Source Code cannot exist without the source code, we know where to seek his traces.

We can consider how we breed him. We can consider how we support him. We can consider how we take care of him. We can also consider how his presence affects us. Consequently we can understand how do *not* grow him and how to limit his potential.

Of course we cannot consider all aspects of the Dragon in this article. The number of methods to do bad things is unlimited. At least you may get some information and understand how apply it to your own Dragon.

3 In the dark of the source code

The source code of a software is separated into modules, libraries, files, packages, classes, functions, blocks, etc. We can consider development of each such unit as a separated micro-project. Let's describe its results in terms of tree project outcomes.

3.1 The outcome for a computer

The first outcome is the code for a computer. Usually a program called "compiler" converts human readable text into computer instructions. The result of a compilation is repeatable. It is determined by the source code and options used by compilation.

Compiler doesn't make decisions. If it "understands" the source code, it produces relevant output. If a source code unit can be interpreted in some different ways, compiler doesn't guess what to chose. It simply reports that this unit is incorrect and gives to the human the possibility to make a decision oneself.

Let's define a source code unit as clear for a computer, if it is passed through compilation.[3] Until the source code is computer-clear, the result of a project can be anything but not the software. We see that the production of the output of first type is the most visible goal of each micro-project.

3.2 The outcome for an user

Each source code unit can be considered as a separate component. In this role it participate by usage in different parts of a software project. Let's define

[2] Of course: Lines Of Code.

[3] You may know that sometimes a compiler produces unexpected and unrepeatable output. It is the result not of some intellectual decisions but of the errors in the compiler program itself.

a programmer as the user of a source code unit, if he apples it in any way as whole and don't need to know its internal structure. The developer of a software component may participate in user's role too.

The information for an user helps to decide how to apply a software component in the user's code.. This is the outcome of the second type of a micro-project..

User-oriented information has different levels. The simplest describes how to tell to the computer to use a component. This knowledge is easy to obtain. There exist many development tools that produce needed data automatically in clear and comfortable form.

This information is easy to check. In case the component is used in an incorrect form the compiler reports an error. For instance if a programmer calls a function, he needs to know the name of the function, the type of the return value, the types and the number of formal parameters. If he forgets to write one of the parameters, he gets a compilation error.

The next level of user-oriented information is about purposes of a source code unit. As rule there exist some source to receive it. Each component has formal, informal or intuitive rules of use. Sometimes they are documented. In this case an user can find the information in the certain place in the standard form.

The second origin are the names used. If a programer write a source code unit, he usually have an idea about purposes. He chooses meaningful names and adds comments in the headers. In this case an user can find the information looking in the source code.

If none of above sources contains necessary information, the user may ask the developer of the source code unit, or other users. In the worst case the user shall study the source code to make some assumption about its purposes.

The advantages of user-oriented information are well known. In some companies management introduces corresponding standards and forces the developers to follow them.

Unfortunately there don't exist tools that can proof how good the user has understand instructions. The user-level documentation cannot be verified automatically too. The common practice is to consider such documentation as not very important and don't give resources to make it good enough. The results are lacks, ambiguity and errors.

If an user cannot get precise information, he makes assumptions, those may be correct and may be wrong. For instance a programmer sees the function "GetMaxWindow". This function gets no parameters and returns long integer value. The user may decide that this function finds the descriptor for the window of maximal size or the maximal size of a window or maximal possible number of windows. If he has more than one ideas, he can check each and choose the most probable. If he has only one assumption, he'll use the software component according with it. If this decision is wrong, the software gets a new error.

There exist other levels of user-orientedd information. It is usually called know-how. If we speak polite, this is the experience of real use. If we speak precise, this is the information, that in a real situation a component will do not what it shall do. The storage of such knowledge are as rule the heads of

people called "gurus". Using a component in differfent situations they obtain the information about unexpected effects.

For instance a programmer uses home-made linked list. He doesn't know that component was not tested. In some situations the insertion of an element causes a software failure. The programmer should invest a lot of time to understand the cause. After many attempts he finds, that due an internal error he must allways call function ClearAll() after creation of an empty list. Generally seakig this situation is not valid, but as rule it won't be corrected. The error becommes "feature of use" and the programmer becomes the guru.

One other example are assumptions about future use. For instance a developer writes a function with two parametrs of type short integer. He decides that variable of type short integer is big enough to hold some internal result of multiplication of these parametrs. In some real situations the parameters may be big enough to make the result of multiplication greater as permitted. The computer cannot represent this result and truncates it. The software gets a rare failure the cause of that is be very complex to find.

As rule an user cannot easily get the information about unexpected effects. The information about internal assumptions is usually hidden too. Only one way to obtain the information is to get a software failure and to find the cause. This is analogous with passage thought a dark room. Let's define it as user's dark of the source code.

This kind of darkness is a good home for the Dragon of The Source Code. The user's dark is the orfigin of many greal loses. The most known are Y2K problem and Arian 5.

We'll consider this darkness more detailed lather. Let's describe the third way to use source code.

3.3 A developer as a computer

However some projects contain the holy code, which nobody is permitted to change, such forever-true texts are rare to meet. As rule each source code unit is subject to check or to change. Let's use the term "developer" to denote a person who is forced to do this. The creation of a new component is simply the case where the code to modify does not exist.

In contrast to the user a developer implements his decisions in the same unit of the source code. To do appropriate decision he needs the user-oriented information and much more. There exist many different levels of developer-oriented information. We'll consider most important of them.

The first and simplest is the compiler level. If a developer needs some information about result of the compilation, he may process the source code as a compiler. It is simple in some languages and difficult in other. In special cases he must find details of the language conventions, compiler characteristics or particular hardware. Of course he may have difficulties, but in common cases such information exist and the sources of it are well known.

For instance a progframmer needs to check the greatest possible value for a numerical variable. He must find its definition and get its type. Then he can read in the documentation of the compiler about limits for variables of this type.

In other cases the search of compiler-level information is is more complex. For us is important that it is allways possible and the results are strict defined.

At least a human can compile the code manually. This is usually not simple, but not more complex than to write a program in assembler. No darkness is present here.

The second is the computer-level. This is the information how a computer execute the results of the compilation.

For instance a programmer must check why a function returns the negative value. He may read the source code and "execute" it similar to the computer. He may use debugger and consider the results of execution of each code statement step by step. He may make a prototype to simulate the behaviour of the program component. He may insert into code the output of information he need and read the log file.

The computer-level information can be received in different ways. It may be simple or complex to get. This process may be error prone and expensive. Only the clearness of it is important for us yet.

If the process of information search is correct, the result is determined. We may say that the computer-level is as clear as the compiler-level.

3.4 The source of requirements

In many software projects only the first two levels of the developer-oriented information are taken into account. We'll briefly consider some others and describe their importance for a developer.

The next is the level of requirements. An user needs to know for which purposes he can apply a software component. The developer needs to know for which use he must make a software component. These are two views on the same information, two quite different views.

Let's say the coding of a component is the implementation of a tactical plan. The design is the preparation of this plan. The requirements are the goals of this plan. The design and coding are the answers on the question *how*. The requirements describe *what*. Abowe requirements is the level of strategy. It gives the answer on the question *why*.

By designing, creating or changing of a component the developer must decisions about requirements. This decisions determine the user-oriented, the computer-oriented and two lower levels of developer-oriented information. In many cases the requirements for a component are documented, but origins of them aren't. Consequently the more expensive strategical errors are much more complex to find.

For instance one developer was being written a class, that should hold a number of certain objects. He had decided to implement a double linked list. To optimise the speed of search he had applyed a complex algorithm. He had spent a lot of time for coding and testing. He had wrote good comments and explained the tricks he had used.

After some time the users of the class had found aa numbeffr of faults. To this moment the developer was left the project. One other developer had wasted a lot of time to understand and to correct the source code.

Accidentaly one team member asked him to simplify the interface of this class. This forced the developer to check how the class was used. He found only two cases of use and the maximal possible number of objects in the list was four.

He had rewrote the class in a simple manner. The result was more comfortable and more reliable.

Of course a bad decision about requirements for a small component is not such dramatic as error in requirements of a big project. The problem is in the quantity. If many small software components are developed for wrong purposes, the tangle of small problems can become a very big problem.

There are many levels of requirements. We can consider a code unit as a hierarchy of smaller units and as a part of the bigger hierarchy. Sometimes a logical unit can be separated in different physical units. For instance one header file, two files with implementation code and one document in the project's database. We can make different logical and physical hierarchies for different purposes.

The biggest "unit" of one project consist of all documentation and source code files. It may be a part of some other hierarchy of related projects, but this isn't interesting for us. We can say, that highest level of user-oriented information are the requirements for the software project. Consequently the highest level of developer-oriented information are the reasons of these requirements.

A source code statement may be considered as smallest unit in our hierarchy. It has his own purposes and reasons. In most cases they are simple to understand by reading the code. They may be hidden too. For instance a developer can use special order of operations by the numerical calculation to get maximal accuracy. Sometimes empty cycles are used to delay the execution. Many developers use "magical" numbers without any explanation. In this situation other developer should guess is 15 a casual value, four lover bits being set to 1, sixteen minus one or something else. These are the cases when a small "right" change of one code line causes an unexpected failure of the program.

Hidden reasons and purposes of some code unit add their darkness for the developer. Usually the higher level of the software are better documented as lover. The opposite situation is rare in commercial companies. If the people make excellent documented parts of unknown whole, the project is as rule unsuccessful.

3.5 The problem of selection and other puzzles

We dig closely to roots. There is one other kind of information about reasons. All decisions are chooses between many possibilityes. This is a tactic of strategy. Such information describes *how* the question *why* was answered. The next level describes which possibilities were considered and which were unknown. The next tells *why* certain possibilities were presented and other werent. To climb from the origins to the given results we need answer the questions:

- *Why* the given information was presented?
- *What* was considered?
- *How* the given reasons were selected?
- *Why* the given requirements are reasonable?
- *What* was decided to do?

- *How* this was done?

Unfortuanately it is not possible to answer these questions in the opposite direction. If we have only an unit of source code, we know what it makes. We can only guess what it should do. We cannot determine requirements examining design. We cannot decide are reguirements correct or not, if we don't know the problem domain. If we know only a number of problems problem, we cannot decide how important each of them. If in documentation stands only "The problem X is important", we don't know why and for whom. The deeper we dig the more complex to answer the questions. This process is definitely dark.

Of course the root of decision is not necessary, if this decision is correct. However in many cases the decision is bad and this information is needed to understand this fact.

If a developer must correct an error, he needs to know:

- Which variants were considered?
- Why the implemented variant was preferred?
- Was it the first being found?
- Was it coosen after correct comparison with others?
- etc.

If a developer hasn't got answers on such questions, he must make assumptions. The worst problem is that he need not only find other possible variants, but also to guess which from them the previous developer knew.

Suppose a developer has found a function, which uses bubble sort. This developer knows that quick sort is better. The autor of the function may not knew about other algorithms. This seems strange. The developer must waste time to check other presuppositions, even the first is right.

If the situation is more complex, many results of a correction are possible. For instance:

- The first developer had made an error and the second corrects it.
- The first had made a right decision and the second finds the better one.
- The first had made a right decision and the second mistakes.
- The first had made an error and the second adds a new one.

The second developer may be not the last. Many people can change a code unit. Not always the reasons of changes are clear. In such cases some parts of the code getting his own life. Nobody knows what they are developed for. Nobody knows why they are developed in such way. Nobody can predict the results of changes. Let's call this nice phenomena "source code phantoms".

Frequently the origin of phantoms is the conversion from an other language. Each language has his own tricks. If a such trick was implemented in an other language, a good puzzle is waiting for the future developers. We can mention automatically codegeneration. In many cases generated code is a good gift for real puzzle fans.

There are more other methods to add dark for the developer. The widely used are ambiguous names, wrong comments and partially implemented units. Sometimes copy-paste operation produces very interesting effects.

The lost information give the developer a good training for his brain. We'll consider many sources of the dark later. Now return to The Dragon.

4 Errors are born in the dark

The Dragon of The Source Code is virtual. He lives in the dark. He is invisible. He cannot be measured. However the results of his business are material. For instance if a program frequently crashes, the presence of errors is a clear fact.

Virtual Dragon cannot affect the material source code directly. To introduce errors in a program he needs an *agent.* Of course the agent of the Dragon of the Source Code is the developer. As an agent of a chemical reaction, the developer transforms the virtual presence of Dragon into bad decisions. As secret agent he destroys the quality of the software and prepares the intervention of the Dragon of The Source Code.

To study the Dragon we need consider the place when virtual becomes material. This is the developer's head.

Of course this is not the simplest place for measurements. It is complex to examine how bad decisions are born in a real software project. We'll begin from the model we can manage. We need a device to measure too.

I suggest you will be this device. The best way to explain how a human make a decision is to demonstrate human being made a decision. This is a text and I haven't possibility to show you someone else except you.

We'll model the process of decision on different simple examples. Our measurement will be the time you need to solve the task. You don't need to find a solution of each exercises. Your estimate of the time you need to do this is good too. We'll consider principal problems and compare different variants. The precise amount of seconds, minutes or hours is not interesting for us now.

I hope you have understand this article is not usual. Our model is be not ordinary too. Before introduce it I should explain the background.

I expect you wait "programer's tasks" or listings of code. These were things I had used before. My experience shows they are bad because two grounds.

Firstly a listing is written in a programming language. (Sometimes authors use simplified imitation of it, but this is not the matter.) There exist hundreds of different languages. The problem is that not all readers know the used for examples. They cannot understand the illustrations of thoughts from the text. As result they cannot understand the text completely.

The second problem exist because other readers know the programming language very well. In the worst case they work with it. If they show the source code, they automatically switch their brain into the "working" mode. As result they are completely involved in the process of search of the solution. The paper becoming only a context of this task. As result they cannot understand the text completely.

I'll use in this article many real examples I had collected. They are written in different programming languages and by different people. To make the examples

understandable I'll throw the languages off. Only the idea is important and should be clear. In all cases I'll try to make it as clear as possible. This means I'll choose the best form to represent the ideas. You can translate them into programming languages you know and into the examples you had seen.

To show this process I'll sometimes use C++. You needn't know this language. It will be used only to show you something like you see on your screen when you're writing software.

Our first model will be one exercise frequently used for IQ tests. You'll got a sequence of numbers. Your task is to decide what number is hidden under question mark. To find the solution you need to know a bit of mathematics and have not very low IQ.

4.1 The simple code

Let's begin. Our first sequence is trivial.

$$1, 2, 3, 4, 5, 6, ? \tag{1.a}$$

The answer is of course 7. To find it you need only to read this sequence. The solution comes "automatically" by reading.

The second example is a bit more complex:

$$1, 8, 27, 64, ? \tag{2.a}$$

This is the sequence of cubes. The answer is 125, 5 cube. I mean you have found the answer no such quick as the first. The complexity is not in calculation, but in searching the rule. Let's write the *rules* of our examples:

$$X_i = \quad X_{i-1} + 1 \tag{1.b}$$
$$X_i = \quad (1+i)^3 \tag{2.b}$$

Here and below "i" is a natural number $(0, 1, 2, 3, 4, ...)$ X_i is the i-th element. The 0th element is the first number of the sequence. It may and may be calculated by the formula. x^y is the y-th power of x.

4.2 The first portion of darkness

We'll see how the Dragon can force you to make bad decisions. Let's introduce some darkness into our model. Please solve the third example:

$$729, 512, 343, 216, ?(3) \tag{3.a}$$

The answer is 125 as 2.a and the rule is:

$$X_i = (9-i)^3 \tag{3.b}$$

Maybe a programmer can quick remember that 512 is the cube of 8, but it is simple for you to see in 343 the cube of 7? You rare use this fact in your work and it is dark for you.

The third rule is as simple as the second. The exercise is more complex. To find the rule you need to find the order of facts. This facts are harder to find out.

We can see this on the next example. The rule is very simple, if you know it.

$$4, 9, 10, 19, 24, ? \tag{4.a}$$

How many time do you need to find a solution?

Here is a swindle. The numbers are in the hexadecimal format. Let's rewrite the sequence in the more usual form:

$$4, 9, 16, 25, 36, ? \tag{4.b}$$

Now you can recognise the squares of 2, 3, 4, 5 and 6 much better. The answer is 31 what equals 49 in decimal format.

Suppose you don't know about the trick with the hexadecimal format. If you think this is a sequence of ordinary numbers, you may find some solution. Probably it won't be simple and your answer won't be 31.

The difference between hexadecimal and decimal numbers is the source of many misterious errors. I think each programmer had made them. Above all on the start of his career.

Let's consider the problem of the difference on other examples.

4.3 Different ways are different

Our fifth example is:

$$3, 4, 6, 10, 18, ? \tag{5.a}$$

Let's rewrite it in the following form:

$$3, (3+1), (4+2), (6+4), (10+8), ? \tag{5.b}$$

The next is 18 + 16. This would be 34. Now we can reconstruct the rule

$$X_i = X_{i-1} + 2^i \tag{5.c}$$

The answer is right and the rule is right, but there exist a more simple form:

$$X_i = X_{i-1} * 2 - 2 \tag{5.d}$$

We see that the idea coming first may be not simplest solution. It my be wrong one too. Let's consider the sixth sequence

$$3, 4, 6, ? \tag{6.a}$$

We can use the solution we already have and write 10. This variant is not the one possible. Other solution is 9:

$$3, (3+1), (4+2), (6+3) \tag{6.b}$$

If there exist one solution, one other may exist too. In many cases the consequences may be quite different. This is the problem of a selection.

4.4 The smile of the Dragon

Let's model a real situation. Suppose one developer has the rule:

$$X_i = X_{i-1} * 2 - i \tag{7.a}$$

The 0th element is 2. The developer writes the sequence:

$$2, 3, 4, 5, 6, 7 \tag{7.b}$$

OK! The sequence is very good. It is simple. Of course the developer has no time to describe the complex rule he had used. He checks the rule "add one" and is sure that even the 100th element can be calculated in the simple way. He makes a bad decision and doesn't notice the real rule proper. The Dragon's resource is created.

After any time the requirements change. The 0th element should be 3 instead 2. One other developer sees the sequence of numbers. The Dragon activate. Even the first developer had noticed somewhere (in a bad place) the real rule, the second very quick understands the rule "add one". He has no time to read "useless comments". (The Dragon use overconfident.) As result the new sequence is:

$$3, 4, 5, 6, 7, 8 \tag{7.c}$$

In case the proper rule were used, the sequence were a bit different:

$$3, 5, 8, 13, 22, 39 \tag{7.d}$$

If the proper rule is somewhere noticed, the fault can be corrected. If it isn't, it is very hard to restore. The second developer wrote the sequence quite wrong and the first variant of the sequence is erased. The Dragon laugh. [4]

4.5 How errors work

Let's add an other human factor to our model. Our next sequence is:

$$3, 4, 6, ???, 18, ?24? \tag{8.a}$$

We know that on the place of "???" is an error. The previous developer doubted is 24 correct or not. He had noticed this by "?". The rule is "previous element plus something". Could you find it?

A good candidate is the rule of the fifth example:

$$X_i = X_{i-1} * 2 - 2 \tag{8.b}$$

In this case instead "???" should be "10". "24" is not correct and should be replaced by "34". It seems to be right! You compile and begin to test. You get immediately a failure. After fifteen minutes you establish the fact that the cause is "34". You try 24 and it works. Could you find the rule?

Well. The rule is:

$$X_i = X_{i-1} + i \tag{8.c}$$

[4] Ruslan Shevchenko had commented the early versions of this article. This situation is based on an exaple from his experience.

On the place of "???" is an error. Two numbers 9 and 13 are left.

Do you think here are two errors? Maybe. The lose of two numbers can be the result of one bad decision or of two different.

The Dragon has used here an other bad human's property, the optimism. If you know that an error exist, you don't know how big it is until you correct it. If a program doesn't work proper, this is not the result of one error. This is a result of *at least* one error.

4.6 The "real" code

I mean you have felt how Dragon can attack you in the dark and can notice such situations in the future. However some readers may say that this model doesn't reflect the "real" source code. Of course it doesn't. It is too simple. We had considered trivial examples in the simple context.

Let's show the example for "working programmers". Suppose you have the next code block:

```
// Calculate sequence of numbers
#define BAK_DOZ      13  // constant used for calculation
int X[ BAK_DOZ ],        // sequence
    k = 100, MulDoz= 12;
long MaxP = 4;
X[0] = 9 ;
X[1] = MulDoz ;

for( int i = 0; i < MaxP; i++)
    X[i+2] = (7-i) * 3 * (i+1) ;
if( k / 28 + 11 < BAK_DOZ  )
 {
    X[4] = X[3] + MulDoz , X[5] = X[3] + MulDoz * 2 ;
  }
else
  {
    X[4] = X[4] + MulDoz , X[5] = X[5] + MulDoz * 2 ;
  }
X[6] = ( MulDoz + 1 ) * (MulDoz - 3);
X[7] = MulDoz * BAK_DOZ ;
X[8] = 212 ;
```

X[8] is incorrect. You have no information about presence of other errors. On the clock is alfrfeady 15:55, but your manager demand "the working version" before 11:15 a.m. tomorrow. Seven others "one-line-errors" are waiting for you. I know that rule for 8th element is very simple, but I cannot remember it. Only one I can tell you is that the dozen was important in the previous version. New rule is other, but I had used the existing code to save time. I really wish to help you, but my part should be ready 10:30 a.m. tomorrow and the manager is blaming me too.

Good luck!

5 The heavy context load

To be continued...