Code Simplicity - (Author: Max Kanat-Alexander)

(Literal summary)

Chapter 1.- Programming, and what's a program

Programming has to become the art of reducing complexity to simplicity. Otherwise, nobody could keep working on a program after it reached a certain level of complexity.

A "good programmer" should do everything in his power to make what he writes as possible to *other programmers*.

But, what's a program, really?

1.- A sequence of instructions given to the computer.

2.- The actions taken by a computer by the result of being given instructions.

The first definition is what programmers see when they are writing a program. The second definition is what users see when they are using a program. A computer program is both of these things; the instructions the programmer writes and the actions the computer takes.

Chapter 2.- Design: the missing science

Definition - "*Design*":

1.- To make a plan for a creation.

2.- A plan that has been made for a creation that hasn't been built yet.

3.- The plan that an existing creation follows.

When we are *designing software* (1), we are planning it out. Once we've done this, the result is a "*software design*" (2). Code that already exists also has "*a design*", which is the structure that it has or the plan that it seems to follow.

Every programmer is a designer (either for the overall architecture of the entire program or for concrete parts of it. A design process is even needed for writing a single line of code).

Chapter 3.- Goals / Purpose of software

There is a single **purpose** for all software: **to help people**.

Definition - "*Help*":

"to make it easier for (a person) to do something; aid; assist. Specifically... to do part of the work of; ease or share the labor of."

For example, a word processor exists to help people to write things, and a web browser exists to help people browse the Web. Even when you're writing libraries, you're writing to help programmers.

The purpose of software is not "*to make money*". That can be your personal purpose or the purpose of your organization, but not the purpose of the software itself.

The first thing we need to do is determine the purpose of our software, with a simple sentence in natural language. It's best to keep it simple.

Then, let's look at all our feature requests. For each one, we can ask ourselves if it helps on the program's purpose. If the answer is "it wouldn't", we should remove that feature from our list.

Then, for any of the remaining features, we should drive down the answer as a short sentence. Useful reasons to ask and answer this question:

1.- It helps resolve uncertainties about the feature's description or how it should be implemented.

2.- It helps the team come to an agreement about the value of a feature.

3.- It will make it obvious that some features are more important than others, helping project leaders to prioritize work.

4.- It can help us to determine which features to remove if the program has become bloated.

This procedure also works with lists of bugs.

Goals of the Software Design

1.- To allow us to write software that is as helpful as possible.

2.- To allow our software to continue to be as helpful as possible over time.

3.- To design systems that can be created and maintained as easily as possible by their programmers, so that they can be -and continue to be- as helpful as posible.

Chapter 4.- The future: making decisions about the software

"How do I make decisions about my software?"

The Equation of Software Design: $D = \frac{V}{E}$

where:

D = Desirability of the change.
V = Value of the change.
E = Effort involved in performing the change.

The desirability of any change is directly proportional to the value of the change and inversely proportional to the effort involved in making the change.

Even if your question if "*Should we stay and not change*"?, this equation tells you the answer (value and effort of staying the same vs value and effort of changing).

Value: The degree to which this change helps anybody anywhere. Precising a "numerical" value of any particular change is difficult, but you can *rank* changes by their value.

Value is actually composed of 2 factors: the *probability* of the value (how likely it is that this change will

help a user), and the *potential* value (how much this change will help a user during those times when it does help that person). Try to avoid features with low potential value and low probability of value. Also, features that have no users have no inmediate value.

Example: a feature that allows blind people to use your program has a high potential but low probability of value, but it's still a valuable change.

When considering value, you should also have to consider:

- How *many* users (percentage) will this change be valuable to?
- What is the *probability* that this feature will be valuable to a user?
- When it is valuable, *how much* valuable will it be?
- *Balance of harm*: Some changes my cause some harm in addition to the help they bring. Calculating a change's value includes considering how much harm it may do, and balancing that against the help it brings.

Effort: You can try to describe effort as "*a certain number of hours of work by a certain number of people*". It can be hard to calculate, because changes can have hidden costs than can be hard to predict, such as the time you will spend in the future fixing any bugs the changes introduce. Like in "*value*", you can still rank changes by how much effort they will *probably* require, event if you don't know the exact numbers for each.

It's important to take into account *all* the effort that might be involved, not just the time you're going to spend programming. Include research, communication with developers, planning... Every single piece of time connected with a change is part of the effort cost.

Maintenance: The *Equation of Software Design* is missing an important element : *time.* All changes require maintenance. We must also consider value both *value now* and *value in the future*. Realistically, effort also involves both the *effort of implementation* and the *effort of maintenance*.

$$E = E_i + E_m$$

$$V = V_n + V_f$$

Where:

E_i = Effort of implementation
E_f = Effort of maintenance
V_n = Value now
V_f = Value in the future

The Full Equation of Software Design: $$D = \frac{V_n + V_f}{E_i + E_m}$$

The desirability of a change is directly proportional to the value now plus the future value, and inversely proportional to the effort of implementation plus the effort of maintenance.

But, as time goes on, our equation reduces to: $$D = \frac{V_f}{E_m}$$

Nearly all decisions in software design reduce entirely to measuring the future value of a change versus its effort of maintenance. There are situations in which the present value and the implementation effort are large enough to be significant in a decision, but they are extremely rare.

Often, designing a system that will have decreasing maintenance effort requires a significantly larger effort of implementation - quite a bit more design work and planning are required. However, remember that the effort of implementation is nearly always an insignificant factor in making design decisions, and should mostly be ignored.

It is more important to reduce the effort of maintenance than it is to reduce the effort of implementation.

When in doubt, design your software like it's going to be used for a long, long time: keep it flexible and don't make any decisions you can't never change, and put a lot of attention in design. The future should be our primary focus.

However, *there are some things about the future that you don't know. The most common and distrous error that programmers make is predicting something about the future when in fact they cannot know*. Just make decisions based on information that we have now, for the purpose of making a better future. It will be a future, but it doesn't mean that you have to predict it. Keep your software simple and with a good design, you'll be able to adapt to that future with smaller changes.

Chapter 5.- Change

The Law Of Change: *The longer your program exists, the more probable it is that any piece of it will have to change.*

You don't have to predict *what* will change, you just need to know that things *will* change.

The Three Flaws: there are 3 broad mistakes that software designers make when facing the Law Of Change:

1.- Writing code that isn't needed ("You ain't gonna need it" - YAGNI):

This rule states that *you shouldn't write code before you actually need it*.

You *might* need the code in the future, but since you can't predict the future you don't know how the code needs to work yet. If you write it now, before you need it, you're going to have to redesign it for your real needs once you actually start using it.

Also, since the code never runs, it might slowly become out of sync with the rest of your system and thus develop bugs. Then, when you start to use it, you'll have to spend time debugging it.

So, *don't write code until you actually need it, and remove any code that isn't being used.* Yes, you should get rid of any code that is no longer needed.

2.- Not making the code easy to change ("Rigid Design"):

This happens when a programmer designs code in a way that is difficult to change. Two ways of doing rigid design:

- Make too many assumptions about the future.
- Write code without enough design.

"Code should be designed based on what you know now, not on what you think will happen in the future."

This isn't to say that planning is bad. A certain amount of planning is very valuable in software design. You'll be fine as long as your changes are always small and your code stays easily adaptable for the unknown

future.

3.- Being too generic ("OverEngineering"):

Some developers solve problems by designing a solution so generic that (they believe) it will accomodate every possible future situation.

Problems with *overengineering*:

- You can't predict the future, so no matter how generic your solution is, it will not be generic enough to satisfy the actual future requirements you will have.

- When your code is too generic, it often doesn't handle specific situations very well from the user's perspective.

- Being too generic involves writing a lot of code that isn't needed, which brings us back to our first flaw (don't write code that isn't needed).

Rule: *Be only as generic as you know you need to be right now.*

Incremental Development and Design:

There is a method of software development that avoids the three flaws by its very nature, called "*incremental development and design*". It involves designing and building a system piece by piece, in order (instead of the entire system).

The tricky part of using this method is deciding on the order of the implementation. In general, you should pick whatever is simplest to work on at each step. Sometimes, you may even need to take a single feature and break it down into many small, simple, logical steps so that it can be implemented easily.

Chapter 6.- Defects and Design

The chance of introducing a defect into your program is proportional to the size of the changes you make to it.

Making small changes is likely to lower maintenance effort.

Small changes = fewer defects = less maintenance.

This law seems to be in conflict with the *Law Of Change*. It's balancing these laws that requires your intelligence as a software designer.

The best design is the one that allows for the most change in the environment with the least change in the software.

About changes:

1.- **Never "fix" anything unless it's a problem,** and you have evidence showing that the problem really exists. Just get real evidence that a problem is *valid* before you address it.

The most famous error in this area is "*premature optimization*". That is, spend time optimizing code before you know that it's slow. The only parts of your program where you should be concerned about speed are the exact parts that you can show are causing a real performance problem for your users. For the rest of the code,

the primary concerns are flexibility and simplicity.

2.- **Don't repeat yourself**. The most well known rule in software design:

"In any particular system, any piece of information should, ideally, exist only once."

This also applies to blocks of code (not only strings or constants). You should not be copying and pasting blocks of code. Instead, use functions/includes/etc.

"*Law of Defect Probability*": if we can reuse old code, we don't have to write or change as much code when we add new features, so we introduce fewer defects.

Chapter 7.- Simplicity

The Law Of Simplicity: The ease of maintenance of any piece of software is proportional to the simplicity of its individual pieces.

The simpler the pieces are, the more easily you can change things in the future. The idea is to make the individual components of your code as simple as possible, and the make you sure they stay that way over time.

You'll often have to take what you've created and make it simpler. You have to redesign pieces of the system continously as new situations and requirements arise. Simplicity requires design: if your project lacks a good design, and it continues to grow, you'll eventually end in "complexity".

Simplicity and the Equation of Software Design: The most important thing we can do that will reduce the effort of maintenance in the Equation is make our code simpler. Just look to the code, see if it is complex, and make it less complex for ourselves right now. You spend a little time doing the simplification now to save a lot of time later.

Even documentation has to be simple and clear. The only thing worse than complex documentation is **no** documentation.

How simple do you have to be? *Stupid, dumb simple (*relative to our target audience, which is *other programmers*).

Be consistent: Consistency is a big part of simplicity. Code that isn't consistent is harder for a programmer to understand and read. This applies not only to variable or function names and their format; your program should behave in a consistent fashion internally. A programmer who is familiar with how to use one part of your code should be immediately familiar with how to use another part of your code.

As an example: similar objects should have a similar set of functions working similar. Don't force programmers to relearn the way the system works every time they look at a new piece of it.

Readability: Readability of code depends primarily on how space is occupied by letters and symbols. There is no rule exactly about how code should be spaced, except that it should be done in a consistent manner and the spaces should help the reader.

Naming Things: An important part of readability is giving good names to variables, functions, classes, etc. Ideally, *Names should be long enough to fully communicate what something is or does, without being so long that they become hard to read.*

Comments: You generally should not add comments about *what* a piece of code is doing. That should be obvious from reading the code (if it isn't obvious, the code should be made simpler). Only if you can't make code simpler should you have a comment explaining what it does. *The real purpose of comments is to*

explain why you did something, when the reason isn't obvious.

Chapter 8.- Complexity

The main source of all the problems is *complexity.*

Complexity builds on complexity - it's not just a linear thing, like "*we have 10 features, so adding 1 more will only add 10 percent more time*". The new feature will have to be coordinated with all 10 of your existing features. if it takes 10 hours of code implementing the feature itself, it may well take another 10 hours of coding time to make the 10 existing features all interact properly with the new feature. The more the features there are, the higher the cost of adding a feature gets.

Some projects start out with sych a complex set of requirements that they never get a first version out (if you're in this situation, you should just trim features: get out something that works and make it work better over time).

Other ways to add complexity than just adding features are:

1.- Expanding the initial purpose of the software: If you start to add features that fulfill some *other* purpose, things get very complex very quickly. So, just, stick to your software purpose (that should match the *user's purpose*).

2.- Adding programmers to a working project: Adding an extra programmer means spending extra time (groove in that one programmer, groove in the existing programmers to the new person, etc). You're more likely to be successful with a small group of expert programmers than a large group of inexpert programmers.

3.- Changing things that don't need to be changed: Any time you change something, you're adding complexity, and introducing the possibility of bugs. This also requires time: time to device upon the change, time to implement the change, time to validate the change and how it works with other pieces of the software, etc. And the change will increase complexity, so the more you change, the more time is going to take each future change. It's still importante to make certain changes, but you should be making informed decisions about them.

4.- Being locked into bad technologies: Avoid picking the wrong technology to use in your system. There are 3 factors you can look at to determine if a technology is "bad" before even you start using it:

4.1.- *Survival Potential*: A technology's survival potential is the likelihood that it will continue to be maintained. "Popularity" can be a factor that determines if a technology will be maintained, always trying also to avoid being stuck in the "*one vendor*" case (popular technology with poor survival potential).

4.2.- *Interoperability*: is the measure of how easy it is to switch away from a technology if you have to. "*Can we interact with this technology in some standard way, so it would be easy to switch to another system that follows the same standard?* (ex: databases and standard SQL)".

4.3.- *Attention to Quality*: See if the product has been getting better in each release.

5.- Poor design or no design: Things are going to change, and design is required to maintain simplicity while the project grows: each new feature multiplies the complexity of the code instead of just adding a little bit to it.

6.- Reinventing the wheel: Don't invent a protocol / function when a perfectly good one exists. Do it only if nothing doesn't exists yet, or existing technologies are "bad technologies", or existing technologies does not cover your needs or are not properly maintained.

Complexity and complex problems:

Any time there is an "unsolvable complexity" in your program, it's because there's something fundamentally wrong with the design.

You also have to ask "*What problem are you trying to solve?*". So, when things get complex, back up and take a look at the problem you're trying to solve. Take a really big step back, not just "*How do I solve this problem using my current code?*". Ask yourself "*How should this sort of problem be solved?*", and see how your code needs to be reworked.

If you're having trouble with a complex problem, write it down on paper in plain language, or draw it out as a diagram. *Most difficult design problems can be solved by simply drawing or writing them out on paper*.

Handling complexity:

As a programmer, you will run into complexity. If some part of your system is too complex, there is a specific way to fix it - redesign the individual pieces, in small steps. Each fix should be as small as you can safely make it without introducing further complexity. Just split one complex piece into multiple simple pieces. This can take a bit of work, so you must be patient.

However, you cannot stop writing features and spend a long time just redesigning. You have to balance these two needs. One of the best ways is to do your redesigning purely with the goal of making some specific feature easier to implement, and then implementing that feature.

1.- Making one piece simpler: Ask you the key question: "How could this be easier to deal with or more understandable?".

2.- Unfixable Complexity: If you run into an unfixable complexity (ex: underlying hardware), your goal is *to hide* the complexity. Put a wrapper around it so that is simple to other programmers to use and understand.

3.- Rewriting: Some designers, when faced with a very complex system, throw it out and start over again.

Rewriting a system from the ground up is essentially an admission of failure as a designer.

You should **only** rewrite a system if **all** of the followings are true:

- Rewriting the system will be a more efficient use of time than redesigning the existing system.
- You have a tremendous amount of time to spend.
- You are a better designer than the original designer of the system or, if you are the original designer, your design skills have improved drastically since you designed it.
- You have the resources available to both maintain the existing system and design a new system at the same time. Never stop maintaining the current system so that the programmers can rewrite it.

Chapter 9.- Testing:

Law of Testing:

The degree to which you know how your software behaves is the degree to which you have accurately tested it. Unless you've tried it, you don't know that it works.

Apply this law by creating automated tests for every piece of code that has been written. You can run the tests just right after a change is made, and check that every single piece of code still works after each individual change.