#readings #cleancode #topublish

Philosophy of Software Design

Index:

Preface

The most fundamental problem in computer science is *problem decomposition*: how to take a complex problem and divide it into pieces that can be solved independently.

Summary of Design Principles

See also „Summary of Design Principles", P185.

1. Complexity is incremental, you have to sweat the small stuff;
2. Working code is not enough;
3. Make continual small investments to improve system design;
4. Modules should be deep;
5. Interfaces should be designed to make the most common usage as simple as possible;
6. It is more important for a module to have a simple interface than a simple implementation;

7. General-purpose modules are deeper;
8. Separate general-purpose and special-purpose code;
9. Different layers should have different abstractions;
10. Pull complexity downward;
11. Define errors (and special cases) out of existence;
12. Design it twice;
13. Comments should describe things that are not obvious from the code;
14. Software should be designed for the ease of reading, not ease of writing;
15. The increments of software development should be abstractions, not features;

Summary of Red Flags

See also „Summary of Red Flags", P186.

1. (Chapter 4) *Shallow modules*: one whose interface is complicated compared to the functionality it provides. They do not help against complexity, because the benefit they provide is negated by the cost of learning and using their interfaces.
2. (Chapter 5) *Information leakage*: when the same knowledge is used in multiple places, such as two different classes that both understand the format of a particular type of file.
3. (Chapter 5) *Temporal decomposition*: execution order is reflected in the code structure: operations that happen at different times are in different methods or classes. If the same knowledge is used at different points in execution, it gets encoded in multiple places resulting in information leakage.
4. (Chapter 5) *Overexposure*: If the API for a commonly used feature forces users to learn about other features that are rarely used, this increases the cognitive load on users who do not need the rarely used features.
5. (Chapter 7) *Pass-through Method*: A method which does nothing except pass its arguments to another method, usually with the same API as the pass-through method. *This typically indicates that there is not a clean division of responsibility between the classes.*
6. (Chapter 9): *Repetition*: If the same piece of code appears again and again, you have not found the right abstractions.
7. (Chapter 9): *Special-General Mixture*: A general-purpose mechanism contains code specialized for a particular use of that mechanism. This makes the (general) mechanism more complicated and creates information leakage between the mechanism and the particular use case. Modifications to the use-case require changes to the underlying mechanism.
8. (Chapter 9) *Conjoined Methods*: Each method should be (separately) understandable. If two pieces of code are physically separated, but can only be understood by looking at each other, that is a red flag.
9. (Chapter 13) *Comment Repeats Code*: If the information in a comment is already obvious from the code next to the comment, then the comment is not useful. One example of this is when the comment uses the same words that make up the name of the things that is describing.
10. (Chapter 13) *Implementation Documentation Contains Interface*: Interface documentation describes implementation details that are not needed

in order to use the thing being documented.
11. (Chapter 14): *Vague Name*: If a variable or method name is broad enough to refer to multiple things, it does not convey enough information to the developer and the underlying entity is more likely to be misused.
12. (Chapter 14): *Hard to pick name*: If it is hard to find a simple name for a variable or method that creates a clear image of the underlying object, that is a hint that the object might not have a clean design.
13. (Chapter 15): *Hard to describe*: The comment describing a method or variable should be simple yet complete. If such a comment is difficult to write, then there may be a problem with the design.
14. (Chapter 18): *Nonobvious code*: If the meaning and behavior of the code cannot be understood with a quick reading, it means there is an information gap.

1. Introduction: It is all about complexity

Because programming does not have considerable constraints, the greatest limitation in writing software is our ability to understand the systems we are creating.

Complexity increases inevitably over the life of any program. The larger the program, and the more people work on it, the more difficult it is to manage complexity.

There are two general approaches to fighting complexity:

1. Eliminate it by making code simpler and more obvious;
2. Encapsulate it (e.g., *modular design*), so that programmers can work on a system without being exposed to all of its complexity at once;

In modular design, a software system is divided up into modules, such as classes in an object-oriented language. The modules are designed to be relatively independent of each other, so that a programmer can work on one module without having to understand the details of other modules.

A quick comment on agile development

Because of the issues with waterfall development, most software development projects today use an incremental approach such as *agile development*, in which the **initial design focuses on a small subset of the overall functionality.** This subset is designed, implemented, and then evaluated. Problems with the original design are discovered and corrected, a few more features are designed, implemented and evaluated.

By spreading out the design in this way, problems with the initial design can be fixed while the system is still small; later features benefit from experience gained during the implementation of earlier features, so they have fewer problems.

Incremental development means that software design is never done. Design happens continuously over the life of a system: developers should always be thinking of design issues. Since reducing complexity is the most important thing of software design, developers should always be thinking about complexity.

Goals of the book

1. Describe the nature of software complexity;
2. Present techniques one can use during software development to minimize complexity;

2. The Nature of Complexity

The ability to recognize complexity is a crucial design skill. It allows to:

1. identify problems before you invest a lot of effort in them;
2. make good choices among alternatives;

Complexity defined

Complexity is anything related to the structure of a software system that makes it hard to understand and modify the system.

Isolating complexity in a place where it will never be seen is almost as good as eliminating the complexity entirely.

Complexity is more apparent to readers than writers.

Symptoms of complexity

3 general ways:

1. *Change amplification*: A seemingly simple change requires code modifications in many different places.
2. *Cognitive load*: How much a developer needs to know about a task in order to complete it.
3. *Unknown unknowns*: It is not obvious which pieces of code must be modified to complete a task, or what information a developer must have to carry out the task successfully.

Of all 3, the latter is the worst. **An unknown unknown** means that there is something you need to know, but there is no way for you to find out what it is, or even if there is an issue.

Causes of complexity

Complexity is caused by two things: 1. dependencies , 2. obscurity.

1. Dependencies exist when a given piece of code **cannot** be understood and modified in isolation. One of the goals of software design is to reduce the amount of dependencies and to make sure that these remain as simple and obvious as possible.
2. Obscurity occurs when important information **is not** obvious.

Dependencies lead to change amplification and a high cognitive load. Obscurity creates unknown unknowns and also contributes to cognitive load.

3. Working code is not enough (Strategic vs. Tactical Programming)

One of the most important elements of a good software design is the *mindset* you adopt when you approach a programming task. Many organizations encourage a tactical mindset, focused on getting features working as quickly as possible. **However,** if you want a good design, **you must take the more strategic approach** where you invest time to produce clean designs and fix problems.

Tactical programming

The main goal is to get something working. However, tactical programming makes it nearly impossible to produce a good system design, because it is inherently short-sighted.

Strategic programming

The first step towards becoming a good software designer is to realize that **working code is not enough**. The most important thing is the long-term structure of the system.

You should not think of „working code" as your primary goal (though of course your code must work). Your primary goal must be to produce a great design, which also happens to work.

This approach requires an investment mindset. Rather than taking the fastest path to finish your current project, you must invest time to improve the design of the system. Some of this investment is proactive, such as finding a simple design for each new class: rather than implementing the first idea that comes to mind, try a couple of alternatives and pick the cleanest one. Some of this investment is reactive, by taking time to fix mistakes.

How much time to invest?

The best approach is to make lots of small investments on a continuous basis: 10 to 20% of your total development time.

Once you start delaying design improvements, it is easy for the delays to become permanent and for your culture to slip into the tactical approach.

4. Modules should be deep

Design systems so that developers only need to face a small fraction of the overall complexity at any given time. That is, *modular design* - minimize dependencies between modules.

The arguments for a method create a dependency between the method and any code that evokes the method. If the required arguments change, all invocations of the method must be modified to conform to the new signature.

To manage dependencies, we think of each module in two parts:

1. an *interface* - everything a developer working in a different module must know about the given module - *What it does, not how*;

2. an *implementation* - carries out the promises made by the interface;

The best modules are those whose interfaces are much simpler than their implementations. They have two advantages:

1. A simple interface minimizes the complexity that a module imposes on the rest of the system;
2. If a module is modified in a way that does not change the interface, *then no other module will be affected by the modification.*

What is an interface?

The interface to a module contains two kinds of information:

1. formal - specified in code. The signature of a method, the type of its return value, information about exceptions. The formal interface for a class consists of the signatures for all of its public methods, plus the names and types of any public variables;

2. informal - the high-level behavior. The informal aspects of an interface can only be described using comments, and the programming language cannot ensure that the description is complete or accurate.

Abstractions

An abstraction is a simplified view of an entity, which omits unimportant details.

In modular programming, each module provides an abstraction in form of its interface.

Deep modules

They allow a lot of functionality to be accessed through a simple interface. In contrast, a shallow module has a relatively complex interface but not that much functionality (thereby not hiding much of the complexity associated with it).

Classitis

The conventional wisdom in programming is that classes should be small, not deep. This, however, results in classes which are individually simple, but at the cost of increasing the complexity of the overall system.

Examples - How to handle the most common case

Providing choice is good, but interfaces should be designed to make the common case as simple as possible.

5. Information hiding (and leakage)

Chapter 4. argued that modules should be deep. This and the following chapters discuss techniques for creating deep modules.

Information hiding

Each module should encapsulate a few pieces of knowledge, representing design decisions. *The knowledge is embedded in the module's implementation but does not appear on its interface.*

This reduces complexity in two ways:

1. Simplifies the interface to a module;
2. Makes it easier to evolve the system. If a piece of information is hidden, there are no dependencies on that information outside the module containing the information, so a design change related to that information will affect only the one module;

Note: Hiding variables and methods in a class by declaring them private is not the same thing as information hiding, because information about these can still be exposed via public methods (like getter and setter).

The best form of information hiding is when information is **totally hidden** within a module, so that it is irrelevant and invisible to users of the module.

Information leakage

Occurs when a design decision is reflected in multiple modules. This creates a dependency between modules: any change to that design decision will require changes to all of the involved modules.

```
If a piece of information is reflected in the interface for a
module, then by definition it has been leaked.
```

One of the best skills you can learn as a software designer is a high level of sensitivity to information leakage.

```
Ask yourself: how do I reorganize these classes such that this
particular piece of information affects a single class?
```

- If the affected classes are relatively small and closely tied to the leaked information, it may make sense to merge them into a single class;
- Another approach would be to pull the information out of all affected classes and create a new class that encapsulates just that information. *This approach, however, will only be effective if you can find a simple interface.*

Temporal decomposition

One common cause of information leakage, because you split the system into the time-order of the operations that will occur.

For example, splitting an application which reads, modifies and writes out a file. With temporal decomposition, one could end up splitting this into 3 classes. But, since the reader and writer actually interact with the file format, it might make sense to join this into a single class.

```
Order usually does matter, so it will be reflected somewhere
in the application. However, it should not be reflected in the
```

`module structure unless the structure is consistent with information hiding.`

When designing modules, focus on the knowledge that is needed to perform each task, not the order in which the tasks occur.

Too many classes

The most common mistake made by students is to divide their code into a large number of shallow classes, which lead to information leakage between the classes.

A general theme in software design: **information hiding can often be improved by making a class slightly larger.**

A second reason for increasing the size of a class is to raise the level of the interface: for example, rather than having separate methods for each of the 3 steps of a computation, have a single method that performs the entire computation, resulting on a simpler interface.

Defaults illustrate the principle that interfaces should be designed to make the common case as simple as possible. They are also an example of partial information hiding: in the normal case, the caller need not be aware of the existence of the defaulted item.

The best features are the ones you get without even knowing they exist.

Information hiding within a class

Try to design the private methods within a class so that each method encapsulates some information or capability and hides it from the rest of the class. *In addition, try to minimize the number of places where each instance variable is used.*

Think about the different pieces of knowledge that are needed to carry out the tasks of your application, and design each module to encapsulate one or a few of those pieces of knowledge.

This will produce a clean and simple design with deep modules.

6. General-Purpose Modules are deeper

General-purpose approach:

- Pros: By implementing a more general mechanism, you might save time in the future;
- Cons: You might implement something too general, which is not of use.

Make classes somewhat general-purpose

That is, *the module's functionality should reflect your current needs, but the interface should not* (e.g, the interface should be a bit more general-purpose).

See example of a more general purpose API in section 6.3.

Generality leads to better information hiding

One of the most important elements of software design is determining *who needs to know what and when.* When details are important, it is better to make them explicit and as obvious as possible.

Questions to ask yourself

(When you want to find the right balance between a general-purpose and a special-purpose interface)

What is the simplest interface that will cover all my current needs?

By reducing the number of methods in an API *without* reducing its overall capabilities, you are creating more general purpose methods. However, reducing the number of methods only makes sense as long the API for each individual method remains simple.

In how many situations will this method be used?

See if you can replace several special-purpose methods with a single general-purpose method.

Is this API easy to use for my current needs?

This question can help you to determine when you have gone too far in making an API simple and general-purpose. The example given there was when `insert` and `delete` operate on a single character, when it should instead operate on a range of characters.

7. Different layer, different abstraction

In a well-designed system, each layer provides a different abstraction from the layers above and below it; if you follow a single operation as it moves up and down through layers by invoking methods, the abstractions change with each method call.

If a system contains adjacent layers with similar abstractions, this is a red flag that suggests a problem with class decomposition.

Pass-through methods

When adjacent layers have similar abstractions, the problem often manifests itself in the form of *pass-through methods.*

This is a method which does little but evoke another method, whose signature is similar or identical to that of the calling method.

When you see these methods, consider the two classes and ask yourself: „exactly which features and abstractions is each of these classes responsible for?"

Three ways of removing pass-through methods:

- Expose the lower level class directly to the callers of the higher level class (7.1 b);
- Redistribute the functionality between the classes (7.1 c);

- Merge the classes (7.1 d);

When is interface duplication OK?

Having methods with the same signature is not always bad. *The important thing is that each new method should contribute significant functionality.*

One example where it is useful for a method to call another method with the same signature is a *dispatcher*. This is a method that uses its arguments to *select one of several other methods to invoke.* Then it passes most or all of its arguments to the chosen method.

```
def pizza_dispatcher(base, topping, sauce):
    if base == "thin" and topping == "mushrooms" and sauce == "marinara":
        return funghi(base, topping, sauce)
    if base == "thick" and topping == "pineapple" and sauce == "herbs":
        return american_style(base, topping, sauce)
```

Decorators

The decorator design pattern encourages API duplication across layers, as it takes an existing object and extends its functionality.

The motivation for decorators is to separate special-purpose extensions of a class from a more generic one. **However, these tend to be shallow and introduce a lot of boilerplate for a small amount of new functionality.**

Before creating a decorator class, consider the following alternatives:

- Could you add the new functionality directly to the underlying class? This would make sense if the new functionality 1) is relatively general-purpose, or 2) is logically related to the underlying class or 3) will be used by the other users of the underlying class;
- If the new functionality is specialized for a particular use case, would it make sense to merge it with the use case, rather than creating a separate class?
- Could you combine new functionality with an existing decorator?
- Does the new functionality work as a standalone class?

Interface vs implementation

The interface of a class should normally be different from its implementation. The representations used internally should be different from the abstractions that appear in the interface. If the two have similar abstractions, then the class is probably not very deep.

See the example of section 7.4 about implementing a text class around lines (where line is the smallest unit) vs around characters. The latter encapsulates the complexity of line splitting and joining inside it, making it deeper and simplifying the higher level code which uses it.

Pass-through variables

A variable that is passed down through a long chain of methods. These variables add complexity because they force all of the intermediate methods to be aware of their existence, even though the methods themselves do not have any use for the variables.

Eliminating pass-through variables can be challenging. One approach is to see if there is already an object shared between the *topmost* and *bottommost* methods.

The solution (the author most often uses) is to introduce a context object. This stores all of the applications global state (anything that would otherwise be a pass-through or global variable).

Because the context will be needed in many places, it can potentially become a pass-through variable. To reduce the number of methods that must be aware of it, a reference to the context can be saved in most of the system's major objects.

In order for an element to provide a net gain against complexity, it must eliminate some complexity that would be present in the absence of the design element.

If different layers have the same abstraction, then there is a good chance that they have not provided enough benefit to compensate for the additional infrastructure they represent.

8. Pull Complexity Downwards

Another way of thinking about how to create deeper classes.

Which one is better:

- Let users of the module deal with the complexity?
- Handle the complexity internally?

If the complexity is related to the functionality provided by the module, then the second answer is usually the right one. This is because most modules have more users than developers, meaning that the latter should strive to create as simple a interface as possible.

In other words, *it is more important for a module to have a simple interface than to have a simple implementation.*

Taking it too far

Use discretion when doing this as it can easily be overdone. Pulling complexity down makes the most sense if:

1. the complexity being pulled down is closely related to the class's existing functionality';
2. doing so results in many simplifications elsewhere in the application;
3. it simplifies the class's interface;

Be aware that pulling complexity down might lead to information leakage between classes, see backspace example of chapter 6.

9. Better together or better apart

One of the most fundamental questions in software design:

`Given two pieces of functionality, should they be implemented together in the same place or separately?`

This question applies at all levels in a system:

- functions;
- methods;
- classes;
- services;

When deciding whether to combine or separate, the goal is to reduce the complexity of the system as a whole and improve its modularity.

It might appear that the best way to achieve this is to divide the system into a large number of small components. *However, the act of subdividing components creates additional complexity that was not present before the subdivision.*

- Some complexity comes just from the number of components. The more components, the harder it is to keep track of all of them.
- Subdivision can result in additional code to manage the components;
- Subdivision creates separation. For example, methods that were together in a single class before may be in different classes (possibly in different files). Therefore separation makes it harder for developers to see the components at the same time.
  - If the components are truly independent, then separation is good as it allows the developer to focus on one component at a time;
  - If there are dependencies between the components, then the separation is bad because the developer needs to switch context back and forth;
- Subdivision might result in code duplication;

Here are a few indications that two pieces of code are related:

- They share information. For example, input variables;
- They are used together and their relation is bidirectional;
- They overlap conceptually: there is a higher-level category that includes both pieces of code;
- It is hard to understand one of the pieces of code without looking at the other;

Bring together if information is shared

For example, the two methods of Section 5.4 for 1) reading the text of an incoming request and place it in a string object, 2) parse the string to extract various components of the request.

Although the first method was only trying to read the request, *not parse it*, it could not identify the end of the request without doing most of the work of parsing it.

Because of this shared information, it is better to both read and parse the request in the same place.

Bring together if it will simplify the interface

This often happens when the original modules each implement part of the solution to a problem.

Bring together to eliminate duplication

If you find duplicate code, you can:

- refactor it into a separate method and replace the repeated code snippets with calls to the method. Most effective if the repeated code is long and the replacement method has a simple signature.
- refactor the code so that the snippet in question only needs to be executed in one place. One example would be to execute the snippet as part of an attribute during instantiation.

Separate general-purpose and special-purpose code

If a module contains a mechanism that can be used for several different purposes, then it should provide just that one general-purpose mechanism. *It should not include code that specializes the mechanism for a particular use*, nor should it contain other general-purpose mechanisms.

The way to separate special-purpose code from general-purpose code is to pull the special-purpose code upwards, into the higher layers, leaving the lower layers general-purpose.

Example: insertion cursor and selection

The insertion cursor is where the user's cursor stands as they are writing in a file. The selection cursor is where the user's cursor stands as they are selecting text in a file. The former is always visible, the latter only when there is text selected.

The selection and insertion cursor are related:

- the cursor is always positioned at one end of the selection;
- cursor and selection tend to be manipulated together;

Therefore it might seem logical to use a single object to manage both the selection and the cursor.

However, the combined code provided no benefit as the higher-level code still needed to be aware of the selection and cursor as distinct entities and manipulated them separately.

Example: separate class for logging errors

Separating the error logging added complexity with no benefit (see P76) as the logging methods were shallow, called in a single place and highly dependent on their invocation context.

Example: editor undo mechanism

The core of undo/redo consists of a general-purpose mechanism for managing a list of actions that have been executed and stepping through them during undo and redo operations.

Rather than placing this into the text class (which then generates information leakage and opens the class to modification in the future), one can extract the general-purpose core of the undo/redo mechanism and place it in a separate `History` class (see P78).

The `History` class manages a collection of objects that implement the interface `History.Action` (with `.redo` and `.undo` methods). Each `History.Action` describes a single operation (like a text insertion of a change in the cursor location), providing methods that can undo or redo the operation.

The `History` class knows nothing about the information stored in the actions or how they implement their `undo` and `redo` methods. It maintains a list describing all the actions executed over the lifetime of an application and provides the two methods to walk backwards or forwards to the list, in response to user-requested undos and redos.

`History.Actions` are *special-purpose* objects: each one understands a particular kind of undoable operation. They are implemented outside of the `History` class, in modules that understand particular kinds of undoable actions.

The text class might implement `UndoableInsert` and `UndoableDelete` objects to describe text insertions and deletions. Whenever it inserts text, the text class creates a new `UndoableInsert` object describing the insertion and invokes `History.addAction` to add it to the history list.

The `History` class also allows actions to be grouped so that (for example) a single undo request from the user can restore deleted text, re-select the deleted text and reposition the insertion cursor. One way to do this is by defining *fences*.

This approach divides the functionality of undo into 3 categories, each of which is implemented in a different place:

- A general-purpose mechanism for managing and grouping actions and invoking undo/redo (implemented by the `History` class);
- The specifics of particular actions (implemented by a variety of classes, each of which understands a small number of action types);
- The policy for grouping actions (implemented by the high-level user interface code to provide the right overall application behavior);

Each of these categories can be implemented without any understanding of the other categories.

Often, it makes sense to combine special-purpose code for one mechanism with the general-purpose code for another. The text class is an example of this: it implements a general-purpose mechanism for managing text, but it includes special-purpose code related to undoing.

Splitting and joining methods

The issue of when to subdivide applies not just to classes but also to methods.

In general, developers tend to break methods too much. Splitting up a method:

- introduces additional interfaces, which add to complexity.
- separates the pieces of the original code, which makes the code harder to read if the pieces are actually related.

Methods containing hundreds of lines of code are fine if they have a simple signature and are easy to read. *The methods are deep (lots of functionality, simple interface), which is good.*

`When designing methods, the most important goal is to provide clean and simple abstractions.`

Each method should do one thing and do it completely.

If a method has a clean and simple interface and it is deep (interface much simpler than implementation), then it probably does not matter whether it is long or not.

Splitting a method only makes sense if it results in cleaner abstractions overall. Two ways:

1. Factoring a sub-task into a child method. This makes sense if a sub-task is clearly separable from the rest of the original method (e.g, the child method is general-purpose);
2. Split a method into two separate methods, each visible to callers of the original method. It is a good sign if the new methods are more general-purpose than the original method.

10. Define errors out of existence

Exception handling is one of the worst sources of complexity in software systems.

Goals:

- Show why do exceptions contribute disproportionately to complexity;
- How to simplify exception handling;

Key learning: Reduce the number of places where exceptions have to be handled. In many cases, the semantics of operations can be modified so that the normal behavior handles all situations and there is no exception condition to report.

Why exceptions add complexity

Two ways of handling exceptions:

1. Move forward and complete in spite of the exception;
2. Abort the operation in progress and report the exception upwards;

When exception handling code fails, it is difficult to debug the problem, since it occurs so infrequently.

Too many exceptions

Programmers exacerbate the problems related to exception handling by defining unnecessary exceptions.

Classes with lots of exceptions have complex interfaces, and they are shallower than classes with fewer exceptions.

An exception is a particularly complex element of an interface. It can propagate up through several stack levels before being caught, so it affects not just the method's caller but potentially higher-level callers.

The best way to reduce the complexity damage caused by exception handling is to reduce the number of places where exceptions have to be handled.

Define errors out of existence

`unset` (in TCL) deletes an existing variables. Therefore, if the variable does not exist, an exception is raised.

Change the definition of `unset` such that it returns without doing anything. In this case, no exception needs to be raised.

Check another example of changing a method's definition to remove an exception in P89-90.

Mask exceptions

An exceptional condition is detected and handled at a low level in the system, so that higher levels of software need not be aware of the condition.

It results in deeper classes, since it reduces the class's interface (as there are fewer exceptions for the users to be aware of) and adds functionality in the form of the code that masks the exception.

Exception aggregation

Instead of catching the exceptions in the individual service methods, let them propagate up to the top-level dispatch method. A single handler in this method can catch all of the exceptions and generate an appropriate error response for missing parameters.

Exception aggregation works best if an exception propagates several levels up the stack before it is handled: this allows more exceptions from more methods to be handled in the same place.

Design special cases out of existence

For the same reason that it makes sense to define errors out of existence, it also makes sense to define other special cases out of existence.

The best way to do this is by designing the normal case in a way that automatically handles the special cases without any extra code.

Taking it too far

Defining away exceptions, or masking them inside a module, only makes sense *if the exception information is not needed outside the module.*

11. Design it twice

Designing software is hard, so it is unlikely that your first thoughts about how to structure a module or system will produce the best design.

You will end up with a better result if you consider multiple options for each design decision.

Try to pick approaches that are radically different from each other, as you will learn more.

After you have a rough design for the alternatives, consider the pros and cons of each one.

- How easy is to use the interface from the perspective of a higher-level user?
- Does one alternative have a simpler interface?
- Is one interface more general-purpose than another?
- Does one interface enable a more efficient implementation than another?

The design-it-twice principle can be applied at many levels in a system.

1. Picking the interface;
2. Designing the implementation - the criteria here are simplicity and performance;

12. Why write comments? The Four Excuses

Without comments, you cannot hide complexity. *The process of writing comments, if done correctly, will improve a system's design.*

When developers do not write comments, they usually justify their behavior with one of the following excuses:

- Good code is self-documenting;
- No time to write comments;
- Comments get out of date and become misleading;
- Comments are worthless;

Good code is self-documenting

There is a significant amount of design information that cannot be represented in code. For example, the informal aspects of an interface - such as a high-level description of what each method does or the meaning of its result - can only be described in comments.

Other examples are the rationale for a particular design decision or the conditions under which it makes sense to call a particular method.

If users must read the code of a method in order to use it, then there is no abstraction.

No time to write comments

If you want a clean software structure, which will allow you to work efficiently over the long-term, then you must take some extra time up from in order to create that structure.

Many of the most important comments are those related to abstractions (such as top-level documentation for classes and methods). These should be written as part of the design process, as the act of writing documentation serves as an important design tool that improves the overall design (see chapter 15).

Comments get out of date and become misleading

Keeping documentation up-to-date does not require an enormous effort. Large changes to the documentation are only required if there have been large changes to the code, and the code changes will take more time than the documentation changes.

Code review provide a great mechanism for detecting and fixing stale comments.

Benefits of well-written comments

The overall idea behind comments is to capture information that was in the mind of the designer but could not be represented in code.

This ranges:

- from low-level details involving a particularly tricky piece of code;
- to high-level concepts such as the rationale for a class;

When other developers come along later to make modifications, the comments will allow them to work more quickly and accurately.

Good documentation helps with two of the ways in which complexity manifests itself in software: *cognitive load* (by providing adequate information to make changes) and *unknown unknowns* (by clarifying the structure of the system) (see details in chapter 2).

13. Comments should describe things that are not obvious from the code

One of the most important reasons for comments is abstractions, which include a lot of information that is not obvious from the code.

Developers should be able to understand the abstraction provided by a module without reading any code other than its externally visible declarations. The only way to do this is by supplementing the declarations with comments.

Comments augment the code by providing information at a different level of detail.

Some comments provide information at a lower, more detailed, level than the code: these comments add *precision* by clarifying the exact meaning of the code.

Other comments provide information at a higher, more abstract, level than the code. These comments offer *intuition*, such as the reasoning behind the code, or a simpler and more abstract way of thinking about the code.

Do not repeat the code

After you have written a comment, ask yourself the following question: could someone who has never seen the code write the comment just by looking at the code next to the comment? If the answer is yes, then the comment does not make the code any easier to understand.

A first step towards writing good comments is to **use different words in the comment from those in the name of the entity being described.**

Example:

```
# the amount of blank space to leave
# on the left and right sides of each line
# of text, in pixels

text_horizontal_padding = 4
```

Lower-level comments add precision

Precision is most useful when commenting variable declarations such as class instance variables, method arguments and return values. The name and type in a variable declaration are typically not very precise.

Comments can fill in missing details such as:

- What are the units for this variable?
- Are the boundary conditions inclusive or exclusive?
- If a null value is permitted, what does it imply?
- If a variable refers to a resource that must eventually be freed or closed, who is responsible for freeing or closing it?
- Are there certain properties that are always true for the variable (e.g, invariants) such as „this list always contains at least one entry”?

When documenting a variable, think *nouns* not *verbs*. In other words, focus on what the variable represents, not how it is being manipulated.

Higher-level comments enhance intuition

Comments which are written at a higher level than the code can provide intuition, by omitting details and helping the reader understand the overall intent and structure of the code.

This approach is commonly used for comments inside methods and for interface comments.

Example of a bad comment and its improvement:

```
# if a task state is not running
# and the number of used slots is smaller than 10,
# take a slot and mark the state as running
```

```
for i in range(1,5):
    if task.state != "running" or number_of_used_slots < 10:
        task.state = "running"
        number_of_used_slots += 1

# Try to set a task state to
# running if there are slots available
```

The comment does not contain any details; instead, it describes the code's overall function at a higher level.

Ask yourself (before writing a high-level comment):

- what is this code trying to do?
- what is the simplest thing you can say that explains everything in the code?
- what is the most important thing about this code?

Great software designers can also step back from the details and think about a system at a higher level. *This means deciding which aspects of the system are most important, and being able to ignore the low-level details and think about the system only in terms of its most fundamental characteristics.*

When documenting a method, it can be very helpful to describe the conditions under which the method is most likely to be invoked, especially if the method is only invoked in unusual situations.

Interface documentation

Code is not suitable for describing abstractions because it is too low-level and it includes implementation details that should not be visible in the abstraction.

If you want code that presents good abstractions, you must document those abstractions with comments.

The first step in documenting abstractions is to separate *interface comments* from *implementation comments*.

- *Interface comments* provide information that someone needs to know in order to use a class or method. These define the abstraction;
- *Implementation comments* describe how a class or method works internally in order to implement the abstraction;

The *interface comment* for a method includes both higher-level information for abstraction and lower-level details for precision.

Implementation comments: what and why, not how

Once readers know what the code is trying to do, it is usually easy to understand how the code works.

Longer methods have several blocks of code that do different things as part of a method's overall task. Add a comment before each of the major blocks to provide a high-level (more abstract) description of what the block does.

For loops, it is helpful to have a comment before the loop that describes what happens in each iteration, when these are complex.

In addition to describing *what* the code is doing, implementation comments are also useful at explaining the *why*.

14. Choosing Names

One of the most underrated aspects of software design. However, good names are a form of documentation because 1) they make code easier to understand, 2) reduce the need for other documentation and 3) avoid bugs.

Take a bit of extra time to choose great (not good) names, which are precise, unambiguous and intuitive.

Names are a form of abstraction: they provide a simplified way of thinking about a more complex underlying entity.

Names should be precise

Good names have two properties: precision and consistency.

As a general rule, names of boolean variables should always be predicates.

If you find it difficult to come up with a name for a particular variable that is precise, intuitive, and not too long, that is a red flag. The variable might not have a clear definition or purpose and you might be better off using multiple variables.

Use names consistently

Consistent naming reduces cognitive load. This has 3 requirements:

1. always use the common name for the given purpose;
2. never use the common name for anything other than that purpose;
3. make sure that the purpose is narrow enough that all variables with the name have the same behavior;

15. Write the comments first (i.e, Use the comments as part of the design process)

The best way to write comments is at the beginning of the process, as you write the code. This makes documentation a part of the design process.

Write the comments first

- For a new class, start by writing the class interface comment;
- Next, write interface comments and signatures for the most important public methods, while leaving the method bodies empty;
- Iterate over these comments;
- Write declarations and comments for the most important class instance variables in the class;
- Fill in the bodies of the methods, adding implementation comments as needed;

- While writing method bodies, one usually discovers the need for additional methods and instance variables. For each new method, write the interface comment before the body of the method. For instance variables, fill in the comment at the same time the variable declaration is written.

Comments are a design tool

Comments provide the only way to fully capture abstractions, and good abstractions are fundamental to good system design.

To write a good comment, you must identify the essence of a variable or a piece of code: what are the most important aspects of this thing?

Comments serve as a canary in the coal mine of complexity. If a method or variable requires a long comment, it is a red flag that you do not have a good abstraction.

16. Modifying Existing Code

How to keep complexity from creeping in as the system evolves.

Stay strategic

If you want to maintain a clean design for a system, you must take a strategic approach when modifying existing code.

Ideally, when you have finished with each change, the system will have the structure it would have had if you had designed it from the start with that change in mind.

Even if your particular change does not require refactoring, you should still be on the lookout for design imperfections that you can fix while you are still in the code.

Comments belong in the code, not the commit log

When writing a commit message, ask yourself whether developers will need to use that information in the code. If so, document this information in the code.

17. Consistency

If a system is consistent, it means that similar things are done in similar ways. This creates cognitive leverage, as once you have learned how something is done in one place, you can use this knowledge to understand other parts which follow the same approach.

18. Code should be obvious

Things that make code more obvious

Choosing good names and consistency are two things that have already been discussed.

Other techniques:

- Judicious use of white space;
- Document accordingly;

Things that make code less obvious

- Event-driven programming (where an application responds to external occurrences). Hard to follow the flow of control. To compensate, use the interface comment for each handler function to indicate when is this invoked.
- Code that violates reader expectations;

Conclusion

To make code obvious, readers must always have the information to understand it. This can be done in 3 ways:

1. Reduce the amount of information needed, using abstractions and eliminating special cases;
2. Take advantage of the information the reader gained in other contexts;
3. Present the important information in code, using good names and strategic comments;

19. Software trends

Object-oriented programming and inheritance

One of the key elements of OOP is inheritance.

One form of inheritance is *interface* inheritance, in which a parent class defines the signatures for one or more methods, but does not implement the methods. This is a good way to fight complexity because one can reuse the same interface for multiple purposes.

In order for an interface to have many implementations, it must capture the essential features of all the underlying implementations while steering clear of the details that differ between the implementation.

The second from is *implementation* inheritance, in which case the parent class defines not only signatures but also default implementations. Subclasses then choose to inherit the parent's implementation or override it. This might prevent the same method to be implemented in several of the subclasses.

However, note that it creates a dependency between the parent class and each of the subclasses, which means that a developer changing the parent class might have to check the child classes to understand if they are compatible with it.

Therefore, implementation inheritance should be used with caution, instead considering if an approach based on *composition* can provide the same benefits (see also Head First Design Patterns).

If implementation inheritance has to be used, try to separate the state managed by the parent class from that managed by the subclasses. One way to do so is for certain instance variables to be managed solely by methods of the parent class,

with subclasses having only read access to them (or through methods defined in the parent class).

Agile development

One of the most important elements is that development should be incremental and iterative. Each iteration includes design, test and customer input.

One of the risks of agile development is that it can lead to tactical programming, because it tends to focus developers on features rather than abstractions. This also encourages developers to put off design decisions in order to produce working software as soon as possible.

Test-driven development

Write unit tests before writing the code.

The problem with test-driven development is that it focuses attention on getting specific features working, rather than finding the best design.

One place where it makes sense to write the tests first is when fixing bugs. Before fixing a bug, write a unit test that fails because of the bug.

20. Designing for performance

How should performance configurations affect the design process? This chapter discusses how to achieve high performance without sacrificing clean design.

How to think about performance

How much should you worry about performance during the normal development process?

The key to develop performant code from scratch (such that one does not have to come back and improve it) is to be aware which operations are fundamentally expensive. A few examples:

- Network communication: even within a datacenter, a round-trip message exchange can take 10-50 microseonds, which is tens of thousands of instruction times. Wide-are round-trips can take 10-100 milliseconds;
- I/O to secondary storage: disk I/O operations typically take 5-10 ms, which is millions of instruction times. Flash storage takes 10-100 microseconds;
- Dynamic memory allocations;
- Cache misses: fetching data from DRAM into an on-chip processor cache takes a few hundred instruction times: in many programs, overall performance is determined as much by cache misses as by computational costs.

The best way to learn which things are expensive is to run micro-benchmarks, which target one operation.

Once you have a general sense for what is expensive and cheap, you can use that information to choose cheap operations whenever possible.

If the only way to improve efficiency is by adding complexity, then the choice is more difficult. If the faster design adds a lot of implementation complexity, or if it results in complicated interfaces, it may be better to start with the simpler approach and optimize later if performance turns out to be a problem.

Measure before modifying

If you start making changes on intuition (before measuring the system's existing performance), you might waste time on this that actually do not improve performance and make the system more complicated in the process (as you deviated from the design).

This serves two purposes:

1. Identify the places where performance tuning has the biggest impact (it is not enough to identify high-level performance);
2. Provide a performance baseline, so that you can test whether your changes improved the behavior;

Design around the critical path

Suppose you have identified a piece of code that is slow enough to affect the overall system performance. *The best way to improve its performance is with a fundamental change*, like introducing a cache or using a different algorithmic approach.

If there is no fundamental fix, you might have to redesign an existing piece of code such that it runs faster. *The key idea is to design the code around the critical path.*

1. Ask yourself: „What is the smallest amount of code that must be executed to carry out the desired task in the common case?" while disregarding any existing code. Consider only the data needed for the critical path.
2. Assume that you could redesign the system in order to minimize the code that must be executed for the critical path. This is your ideal.
3. Look for a design which can come close as possible to the ideal while still having a clean structure. *You should try to keep the ideal code mostly intact.*

When redesigning for performance, try to minimize the number of special cases that you must check. Ideally, there will be a single `if` statement in the beginning, which detects all special cases with one test. This means that the critical path (which happens when a special case is **NOT** found) can be executed with no additional tests. If a special case is found, the code can branch off to a separate place, the performance of which is not particularly important (as it is not in the critical path).