Principles of Software Design

This paper presents fundamental principles of effective software design. If consistently applied, they help us to achieve simple, understandable, and robust software designs. They provide criteria for choosing between various design alternatives. All design decisions should be evaluated against them to ensure their consistent application. These principles also provide a framework for evaluating the quality of software designs created by others.

Goals of Software Design

Software design is primarily about managing complexity. Software systems are often very complex and have many moving parts. Most systems must support dozens of features simultaneously. Each feature by itself might not seem very complicated. However, when faced with the task of creating one coherent structure that supports all of the required functionality at once, things become complicated very quickly. Human capacity to deal with complexity is quite limited; people become overwhelmed and confused relatively quickly. Perhaps the primary objective of software design is to make and keep software systems well organized, thus enhancing our ability to understand, explain, modify, and fix them.

Based on this view of software design, disorganization (or sloppiness) is the antithesis of good software design. As the laws of physics teach us, the universe tends to become more disorganized over time unless we take active steps to make and keep it organized. Software systems are very much the same way. If created or modified without careful forethought, software systems quickly become incomprehensible, tangled messes that don't work right and are impossible to fix. This is especially true for systems that remain in use over extended periods of time, and are periodically upgraded to support new features. Even if a system starts out with a good design, we must consistently strive to preserve the integrity of its design throughout its lifetime by carefully considering all changes we make to it.

Based on these principles, we can list several important goals of software design:

- Software that works
- Software that is easy to read and understand
- Software that is easy to debug and maintain
- Software that is easy to extend and holds up well under changes
- Software that is reusable in other projects

Design Is an Iterative Process

Software design is a complex undertaking. Therefore, you will rarely get a design right the first time. Implementing a design provides new insights into its deficiencies: things you didn't think about, better ways of doing things, etc. Such insights should feed back into your design to make it better. For this reason, design and implementation activities are usually interleaved in short iterations: Design, code, test, debug, Design, code, test, debug, …

The notion that a complex system can be completely designed in every detail before implementation begins is fallacious. Such an approach deprives designers of valuable knowledge and experience that come only from actually implementing the design. The opposite extreme is also dangerous, starting implementation having done little or no design at all. Those who start coding immediately and wing it as they go are even more prone to failure than those who try to design everything up front. The truth lies between these two extremes. You should do enough design to have a fairly detailed idea of how things will work, and then implement the design to discover its deficiencies. Then, go back and incorporate what you've learned into the design, and then implement some more. This process will eventually converge on a good design.

Abstraction

Abstraction is one of the software designer's primary tools for coping with complexity.

Most programming languages and their associated libraries are meant to be general purpose. They can be used to implement solutions to problems in any application domain (finance, retail, biology, communications, etc.). Due to their general purpose nature, these languages provide only low-level abstractions such as bit, byte, character, string, integer, float, array, file, etc. that model the machines on which the software will run rather than the application domain of the problem being solved. Programs written solely in terms of these low-level abstractions are extremely difficult to understand. Effective software design requires the creation of new, higher-level abstractions that map directly to the application domain rather than the underlying computer.

In object-oriented design, application-specific abstractions are represented as classes. Classes encapsulate the state (or data) and operations (or algorithms) associated with a particular higher-level application concept. For example, the design for a word processor would contain classes such as `Document`, `Font`, `Table`, `Figure`, and `Printer`. Similarly, the design for a web browser would contain classes such as `Favorites`, `URL`, `Viewer`, and `NetworkProtocol`. Software written in terms of such higher-level abstractions is far more understandable to the human reader because it is expressed in terms of the application domain rather than the underlying machine.

There may also be mid-level abstractions such as `ArrayList`, `ThreadPool`, and `ConnectionManager` that don't map directly the concepts of the application domain, but that still play an important role in the implementation of the system. Such mid-level abstractions are helpful in bridging the gap between high-level application concepts and low-level facilities provided by the programming language.

As stated above, part of effective abstraction is identifying a good set of classes that effectively model the application domain. Another part of effective abstraction is carefully defining the interfaces (i.e., operations) supported by those classes. While classes represent the nouns of the application domain, the operations supported by classes represent the verbs. The ability to execute a complex, domain-specific operation by

calling a single method on an object leads to concise, highly-readable code. For example, the following line of code might be used to print a document:

```
defaultPrinter.print(document);
```

While only one line of code is required to print a document, it obviously requires a lot of low-level work underneath to actually send the document's contents to the printer. This work must be carried out by the implementation of the `print` method. How does `print` actually print the document? I don't know, and I don't want to know. Unless I am actually implementing the `print` method, I prefer to ignore those details, thus freeing me to think about something else. The ability to hide all of this complexity behind a simple method call demonstrates the power of abstraction: a complex idea can be conveyed very concisely, thus shielding the reader from many low-level details that might otherwise cloud their thinking.

Naming

Abstraction involves taking something that is complicated, giving it a simple name, and then referring to it by its simple name. This way, complex ideas can be conveyed very concisely. With this in mind, one of the most important tools for achieving effective abstraction is the *identifier*. An identifier is a name that we assign to something. We choose names for classes, methods, variables, constants, source files, etc. While selecting a name might seem to be a relatively inconsequential thing, it is not. The names we choose for things go a long way toward determining how readable our code becomes. Even if I create the right class, if I name it poorly, much of the benefit to be gained from abstraction has been lost. For example, if I name the class that represents printers as `Thingy` instead of `Printer`, I have done significant harm to the readability of my design.

The name assigned to a class, variable, or method should clearly and accurately reflect the function performed by that class, variable, or method. The name `Printer` implies that a class represents a printer; the name `calculatePayrollTax` implies that a method calculates payroll taxes; the name `homeAddress` implies that a variable stores a home address. In contrast, the names `Thingy`, `doStuff`, and `info` would convey no information whatsoever to the reader. Name selection makes a huge difference.

In general, class names should be nouns, and method names should be verbs. There are occasional exceptions to this rule, but it applies in the vast majority of cases. One exception to this rule relates to methods that get/set object attribute values, such as `getName` and `setName`. Depending on the style you prefer, one or both of these methods could alternatively be named with a noun.

Cohesion

Abstractions (i.e., classes and operations) should be highly cohesive.

Each class should represent one well-defined concept, and should be given a name that clearly reflects the concept it represents (e.g., `URL`). Cohesive classes are almost always easy to name. In fact, the name they should be given is often obvious, because they represent only one concept. The operations on a class should all be highly-related to the concept represented by the class. For example, URL operations should all be highly related to storing and manipulating URLs. Operations like `getPath`, `getFileName`, and `resolveRelative` would be appropriate. Operations that are loosely related or unrelated to the concept represented by the class should be placed on some other class. For example, a `URL` class should not have a `display` method that renders the document referenced by the URL on the screen. The rendering function is only loosely related to the concept of a URL, and so should be placed on a different class (e.g., `FileViewer`).

Class operations should also be highly cohesive. Each operation should perform one well-defined task, and should be given a name that clearly reflects the task it performs (e.g., `rebootComputer`). Cohesive operations are almost always easy to name, because they do only one thing. If a method does a bunch of loosely related or unrelated things, it will either be hard to find a good name that describes what the operation does, leading to inferior names like `handleStuff`, or the method's name will become too long (e.g., `sweepFloorAndDoDishesAndPayBills`).

Abstracting All the Way

A typical design contains many classes, some larger and more complex, others relatively simple. Some abstractions are simple enough that they can be directly represented using one of the built-in data types provided by the programming language (e.g., integer, string, float, etc.). For example, concepts such as "title", "pay grade", or "credit card number" could be directly represented using strings or integers. The question is: Is it worth creating classes to represent relatively simple abstractions such as these? Should a designer create classes named `Title`, `PayGrade`, and `CreditCardNumber`, or just go ahead and use strings or integers directly to represent this kind of information? Of course, even if we create such classes, internally they will store integers or strings anyway. Does it help to create such classes, or is it OK to just use the built-in types directly?

Creating classes to represent relatively simple abstractions is often the better choice. Following are some criteria to help make the decision:

1) Domain Checking – Programs need to validate input values that come from end users, files, or other input sources. This is done by parsing or otherwise inspecting the input values to ensure they are valid and lie within acceptable ranges. For example, phone numbers might enter a program as string values, but most strings are not valid phone numbers. Rather than using strings to store phone numbers, it would be better to create a `PhoneNumber` class to store phone numbers. The `PhoneNumber` class would contain the code necessary to validate phone number inputs, probably in a constructor. Input strings containing phone numbers would be passed to the constructor, which would parse the string. If the

string contained a valid phone number, the constructor would store it in the object for later use. If the string was not a valid phone number, an exception would be thrown. Domain checking is an excellent reason to create classes to represent data values that could otherwise be stored directly as built-in data types.

2) Additional Operations – Creating classes to represent simple data values provides a place to put operations that operate on those data types. For example, URLs could be stored directly as strings, but if we do so there will be no place to locate URL-related algorithms that may be needed as the program evolves (parsing URLs into their component parts, resolving relative URLs, etc.). Creating a `URL` class, however, would provide an excellent place to put such URL-related operations.

3) Code Readability – Creating classes for simple abstractions can enhance a program's readability. For example, if you see a variable of type `String`, you don't know much about what the variable represents. If you see a variable of type `URL`, you know a lot about what it represents (i.e., a URL). Creating classes for simple data types enhances readability because variable, parameter, and return types are much more descriptive about what kind of data they represent. Of course, giving good names to variables and parameters will go a long way toward telling the reader what kind of data they represent. Return values, however, don't have names (at least not directly).

Decomposition

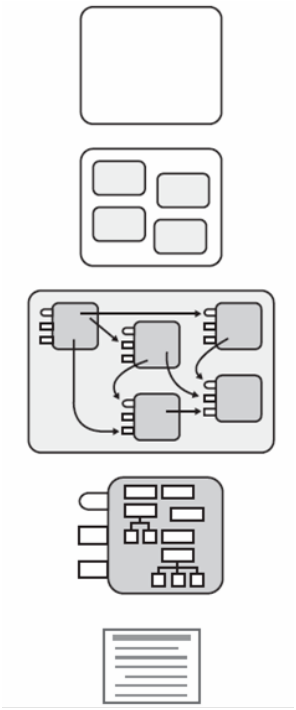
In addition to abstraction, another fundamental technique for dealing with complexity is taking the original problem and dividing it into several smaller sub-problems. The sub-problems are smaller and hence less complex than the original, thus making them more approachable. After solving each sub-problem individually, the solutions to the sub-problems can be combined to create a solution to the original, larger problem. This approach is frequently called "divide and conquer".

After breaking the original problem into sub-problems, we may find that the sub-problems themselves are still too complex to solve directly. In this case, we decompose the sub-problems yet again to create second-level sub-problems that are even simpler. Sub-problems are divided into smaller and smaller parts until the smallest sub-problems are simple enough to solve directly, and thus require no further subdivision. In effect, we create a tree of problems, where the original problem is at the root, and each successive level of subdivision adds another level of nodes to the tree. The solution to each sub-problem makes use of the solutions to the sub-problems below it. This approach allows us to cope with the inherent complexity of the original problem in bite-size chunks.

Decomposition is strongly related to abstraction. The solution to each sub-problem is abstracted as a class or method. The solution to the larger problem invokes the abstractions which encapsulate the sub-problem solutions. This results in a concise solution to the original problem, and allows the details of the sub-problem solutions to be temporarily ignored, thus reducing the cognitive burden of solving the original problem.

It is through the decomposition process that many of the necessary abstractions are discovered (or invented).

Levels of Design



Decomposition is inherently a top-down process. At the topmost level we have the entire *system*. The first level of decomposition divides the system into *subsystems*, each of which represents a major but somewhat independent chunk of the system's functionality. For example, the subsystems for a web browser might be `Network Protocols`, `File Viewers`, `History`, `Favorites`, `Printing`, etc.

At the next level of decomposition, each subsystem is further subdivided into *packages*. Each package is responsible for implementing a part of the subsystem's functionality. For example, a web browser's `File Viewers` subsystem might contain a separate package for each different file format that the browser can display (HTML, PDF, XML, etc.). The package corresponding to a particular format would contain the code that implements the file viewer for that format.

A package is further decomposed into a collection of one or more *classes* that together implement that package's functionality. For example, the web browser's HTML viewer might consist of a dozen different classes.

The functionality of each class is further decomposed into routines which implement the operations (or algorithms) of the class. Significant algorithms are typically decomposed

further into multiple levels of subroutines. Decomposition continues until the leaf-level subroutines are simple enough to implement directly.

Hypo- and Hyper- Decomposition

Many software designers, especially beginners, tend to not decompose things far enough. This might be referred to as *hypo-decomposition* (hypo means deficient). An extreme example of this would be implementing an entire program in a single class. The one and only class would implement all of the functionality for the entire program. Such a class would be an egregious violation of the cohesion principle discussed earlier, which states that a class should "do one thing, and do it well". One could argue that a one-class application is very cohesive because the class does only one thing – it implements the entire application! While there is nothing wrong (and often much right) with having a class that represents the entire application (e.g., a `WebBrowser` class), it is wholly inappropriate to actually implement all of the application's functionality on that one class. Instead, the main class should delegate to other, smaller classes which implement various subsets of the program's functionality. The main class, then, is primarily a delegator (or "driver"), and performs little or no actual work itself other than driving the other classes. In general, if a class represents a large or complex concept, its functionality should be decomposed into one or more smaller classes that perform the actual work. Often these second-level classes will also need to be decomposed further into even smaller classes. This decomposition should be repeated until the resulting classes are too simple to decompose further.

At the other extreme are those who decompose things too far, which might be called *hyper-decomposition* (hyper means excessive). This mistake is harder to make and far more rare than hypo-decomposition. When decomposing a system, one must have a sense of when they have decomposed far enough. In general, we have said that a system has been decomposed sufficiently when its sub-parts are simple enough to "implement directly". Everyone has a slightly different sense of when that point has been reached.

Although rare, it is possible to decompose too far. For example, a `CreditCardNumber` class might be created to represent the concept of a credit card number. This seems like a good design choice. But, how should a `CreditCardNumber` object store the actual credit card number internally? A `String` seems like a natural representation for a credit card number (after it has been validated by the `CreditCardNumber` constructor, of course). Alternatively, it would also be possible to store a credit card number as an array of `Digit` objects. Most people would say that creating a `Digit` class to store individual digits in a credit card number is overkill, and an example of hyper-decomposition.

Size and Length Metrics

How shall we know when we have decomposed far enough? Length metrics, often measured in lines of code (LOC), can be helpful in making this determination. A method that contains a single LOC has been decomposed far enough, of course. A method that contains 500 LOC almost certainly has not been decomposed far enough. Methods that have been sufficiently decomposed are usually less than 50 LOC, and in many cases 50

LOC is still too long. Maybe 20 LOC would be a better goal. While there is no "right" method length, the basic principle is that when an algorithm has been decomposed sufficiently, the resulting subroutines which implement the algorithm tend to be short – frequently, very short.

Size metrics such as the number of parameters can also be used to judge how well a method has been decomposed. A method that requires 10 parameters is too complex. The problem might be one of insufficient decomposition.

Class size can also provide a clue as to whether or not a class has been decomposed far enough. A class with 50 methods has probably not been decomposed far enough. Such a class is probably doing the work of several classes. As a result, the class is also not cohesive, and should be further subdivided.

Classes with a lot of internal variables are also usually insufficiently decomposed. If a class contains a lot of variables (e.g., 20), there are probably smaller subgroups of those variables that are closely related to each other. These clusters of related variables will often suggest new classes to be created, thus further decomposing the original class. The original class would then become a client of the newly discovered classes.

Similarly, a class that contains 5,000 LOC has almost certainly not been decomposed far enough. A class containing 2,000 LOC often requires further decomposition, but not always. Well-designed classes often contain less than 500 LOC, frequently much less (but, not always). There is no "right" class length, but, in general, classes that have been decomposed sufficiently tend to be shorter rather than longer.

Complexity Metrics

Sometimes length metrics based on LOC measurements don't tell the whole story. It is possible for two methods with the same length measured in LOC to have radically different complexity levels. For example, imagine two 100-line methods, the first containing only straight-line output statements (e.g., `println`), and the second containing complex logic with deeply nested loops and lots of branching. While these methods have the same length, their complexity levels are not even close. Straight-line output statements are readily understandable, while complex logic is far more difficult to understand. Both methods might benefit from further decomposition, but the second one demands it.

In general, methods containing complex arithmetic expressions, deeply nested structures, and lots of branching should be simplified by breaking up the complex routine into simpler subroutines that each perform part of the original routine's work. The original routine then becomes a driver routine that delegates much of the actual work to its subordinates.

Many routines naturally contain multiple sections (or paragraphs) of related statements that can be easily factored out into a separate subroutines. Moving a paragraph of related statements to a separate subroutine, giving the new subroutine a good name, and

replacing the original statements with a call to the new subroutine will do much to simplify the original routine. Consistently applying this technique of algorithm decomposition will have a significant positive impact on the quality of your code.

Algorithm & Data Structure Selection

A major part of software design is selecting appropriate algorithms and data structures for the problem at hand. Using an algorithm that is $O(n^3)$ on data sets that become very large will almost certainly be far too slow, regardless of how well we have decomposed and abstracted the problem. Similarly, storing data values as unsorted, linear lists will be far too slow if the data set is large and needs to be searched frequently. Selecting (or inventing) algorithms and data structures with good performance characteristics (including running time and memory consumption) for the intended application is a fundamental design skill. No amount of decomposition or abstraction will hide a fundamentally flawed selection of algorithm or data structure.

Minimize Dependencies (or, Low Coupling)

Large systems contain many classes. As a system is decomposed into its constituent classes, it is important to keep each class as independent as possible from the other classes in the system. Classes A and B depend on each other if:

1) Class A invokes a method on class B
2) Class A accesses the internal state of class B
3) Class A inherits from class B
4) Class A has a method parameter of class B
5) Class A and Class B both access the same global data structure or file
6) Etc.

Minimizing the number of communication channels and interactions between different classes has several benefits:

1) A class with few dependencies on other classes is generally easier to understand than a class with many dependencies on other classes (i.e., dependencies increase a class's complexity)
2) A class with few dependencies on other classes is less prone to ripple effects caused by changes or defects in other classes (i.e., dependencies make a system harder to modify and debug).
3) A class with few dependencies on other classes is easier to reuse in a different program than a class with many dependencies (i.e., dependencies discourage reuse).

Imagine a system in which every class depends on every other class. Every time any class is changed, we must consider the potential impact on all other classes (very confusing, indeed). Similarly, when a class has a defect, the defect will potentially impact the behavior of all other classes, thus making it difficult to track down where the defect actually resides (again, very confusing).

At the other extreme, imagine a system where there are no dependencies between classes (i.e., each class is an island unto itself). In this case, the software doesn't do anything. Making a program perform useful functions requires a certain level of communication (and therefore dependency) between the classes in the system. The goal is not to remove all dependencies, but rather to minimize the number and strength of dependencies.

When two classes must interact, it is desirable to keep the interaction as simple and straightforward as possible. The ideal form of interaction between two classes is through simple method calls. A method call is simple if it has a good name and the data passed through the parameter list and return value is easy to understand. Simple method calls have the advantage of being direct and obvious in the code. Other more indirect forms of communication between classes, such as accessing the same global data structure, make the dependency less explicit and harder to detect and comprehend. To the extent possible, interactions between classes should be through explicit, well-defined method interfaces.

Separation of Interface and Implementation

One important technique for minimizing dependencies between classes is maintaining a strict separation between a class's public interface and its internal implementation. A class's public interface consists of the operations (or methods) through which clients can access its services. In order to use a class, a client needs only to understand the class's public interface. The details of how the public interface is implemented internally are incidental to the client, and should not be accessed or relied upon by the client in any way. The code that implements the public interface, including all variables and subroutines that support that code, should not be accessed by clients. By relying only on the details of the public interface, a class's internal implementation can be changed without affecting (i.e., breaking) its clients. Only changes to the public interface itself affect the clients. The strict separation of interface and implementation goes a long way toward minimizing dependencies between classes.

Information Hiding

Because the separation of interface and implementation is so central to good software design, programming languages often provide features to help enforce this separation. Some languages physically separate a class's public interface and internal implementation into separate source files. Other languages require the designer to declare all class features (variables and methods) as "public", "private", or "protected", thus preventing clients from accessing private details. Such language features encourage designers to hide as much information as possible from clients, thus reducing opportunities for dependency between classes.

A class's public interface should be as small (or "thin") as possible, ideally including only a small number of methods. Each public method's parameters should be as simple as possible. All internal variables should be hidden, and only methods that are directly invoked by clients should be made public.

This advice applies even to inheritance relationships. By making variables "protected", it is easy for a superclass to directly expose its internal variables to its subclasses. This makes the subclasses highly dependent on the internal details of the superclass. As always, this makes it difficult to change the superclass implementation without breaking the subclasses. A designer may choose to make superclasses and subclasses highly coupled in this manner, but the downsides of doing so should be considered when making this decision. Another approach would be to define the superclass/subclass interface in terms of "protected" methods only (i.e., no "protected" variables), thus reducing the level of dependency between superclass and subclass.

Designers should also be careful to ensure that internal implementation details do not "leak out" of a class. For example, a method that performs a search algorithm might be named `binarySearch`. Unfortunately, the name `binarySearch` reveals the method's internal implementation. This choice of name forever binds the method to use the binary search algorithm as its implementation. Alternatively, naming the method `search` would preserve the designer's freedom to vary the internal algorithm without violating the client contract.

Similarly, a grade-keeping program might represent the notion of a class roll with a class named `StudentLinkedList`. However, doing so betrays the fact that the class uses a linked list as the internal data structure for storing a sequence of students. A better choice would be to name the class `ClassRoll`, thus hiding all details of how students are actually stored internally, and preserving freedom to change that representation at will.

There are times, however, when a class or method is inherently tied to a particular implementation. In such cases, it is appropriate to name classes or methods in terms of their internal details. For example, a class whose sole purpose is to implement a hash table could appropriately be named `HashTable` because its implementation is an inherent part of its existence. A hash table will always be a hash table, and that will never change. However, clients of the `HashTable` class should not reveal their internal use of `HashTable` unless that choice is inherent and will never change.

Avoid Code Duplication

Another core principle of good software design is that code duplication should be strenuously avoided. Frequently, programs will contain duplicated sections of code, or sections of code that are very similar. For example, searching an array for a particular value is a common operation, and this code could easily be duplicated many times throughout a program. Similarly, formatting of date/time values for end-user display is a common operation that is often be duplicated throughout a program.

The disadvantages associated with duplication are fairly obvious:

1) If the duplicated code needs to be modified, we must remember to change all N copies, and do so correctly.
2) If the duplicated code contains a bug, the bug will the replicated N times.

3) Duplication makes the program longer, thus decreasing its maintainability.

If the same or similar code appears in N places, the obvious solution is to isolate the duplicated code in one place, and then have all N clients invoke the shared copy. If all N copies are in the same class, the duplicated code can be factored out into a private method on that class. If the N copies are in different classes, the shared copy could be placed on one of the client classes, or placed on some other (possibly new) class that provides a logical home for the shared code. Another solution would be to place the shared code in a superclass, and then make each client class a subclass of the superclass.

If the duplicated code is similar but not identical, it might be possible to create a generic version that will serve the needs of all clients. If the implementation language provides generic types (e.g., C++ templates), a generic type or subroutine will often be a good implementation choice for the shared code.

Design Principles Summary

1. Abstraction
    a. Naming
    b. Cohesion
    c. Abstracting All the Way
2. Decomposition
    a. Levels of Design (System, Subsystem, Package, Class, Routine)
    b. Hypo- and Hyper- Decomposition
    c. Size and Length Metrics
    d. Complexity Metrics
3. Algorithm & Data Structure Selection
4. Minimize Dependencies (or, Low Coupling)
    a. Separation of Interface and Implementation
    b. Information Hiding
5. Avoid Code Duplication