Principles of Software Design

This article reviews a number of principles that represent the essence of effective software design, and should be constantly remembered and applied when designing a software system.

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.

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 very difficult 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 the changes we make to it.

Based on this perspective, we can enumerate 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

The following sections discuss specific principles that, if applied, will help achieve these goals.

Simplicity

"Everything should be made as simple as possible, but not simpler" -- Albert Einstein

The design of a software system should be kept as simple as possible while still implementing the required functionality. We should avoid adding extra design features (classes, methods, parameters, levels of inheritance, etc.) that are currently unnecessary, and only exist to support

future functionality that might never actually be implemented. While it is good to keep an eye to the future, predictions about what functionality will be needed in the future often turn out to be inaccurate. Therefore, we should not encumber the current design of a system with features that are currently unnecessary. Always keeping a system's design as simple as possible does mean that when new functionality is added, it might be necessary to refactor and improve the design in order to support the new functionality. This is alright, because this work is not speculative - it is only done when the functionality is actually being added to the system, and we don't waste time supporting functionality that may never exist. There is a famous acronym that captures the idea well: YAGNI (you aren't going to need it).

Avoid Code Duplication

Code duplication should be strenuously avoided. Programs almost always 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 of code duplication are fairly obvious:

- If duplicated code needs to be modified, we must remember to change all N copies, and do so correctly.
- If duplicated code contains a bug, the bug will the replicated N times.
- Duplication makes the program longer, thus decreasing its maintainability.

At a deeper level, if a section of code is duplicated multiple times, this code probably represents an abstraction (a class or method) that is missing from our design. By finding a way to remove the duplication, we usually end up with a more elegant design.

The obvious solution to duplicated code is to isolate the 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 another (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 implementation language provides generic types (e.g., C++ templates or Java generics), a generic type or method may be a good implementation choice for the shared code.

The effort to remove duplication from our code will result in better designs that are easier to maintain. A famous acronym captures the idea well: DRY (don't repeat yourself).

Orthogonality

(This material on orthogonality is quoted from "The Pragmatic Programmer". [1])

"Orthogonality" is a term borrowed from geometry. Two lines are orthogonal if they meet at right angles, such as the axes on a graph. In vector terms, the two lines are independent. Move along one of the lines, and your position projected onto the other doesn't change.
In computing, the term has come to signify a kind of independence or decoupling. Two or more things are orthogonal if changes in one do not affect any of the others. In a well-designed system, the database code will be orthogonal to the user interface: you can change the interface without affecting the database, and swap databases without changing the interface.
Before we look at the benefits of orthogonal systems, let's first look at a system that isn't orthogonal.

A Non-orthogonal System
You're on a helicopter tour of the Grand Canyon when the pilot, who made the obvious mistake of eating fish for lunch, suddenly groans and faints. Fortunately, he left you hovering 100 feet above the ground. You rationalize that the collective pitch lever[2] controls overall lift, so lowering it slightly will start a gentle descent to the ground. However, when you try it, you discover that life isn't that simple. The helicopter's nose drops, and you start to spiral down to the left. Suddenly you discover that you're flying a system where every control input has secondary effects. Lower the left-hand lever and you need to add compensating backward movement to the right-hand stick and push the right pedal. But then each of these changes affects all of the other controls again. Suddenly you're juggling an unbelievably complex system, where every change impacts all the other inputs. Your workload is phenomenal: your hands and feet are constantly moving, trying to balance all the interacting forces.
Helicopter controls are decidedly not orthogonal.

Benefits of Orthogonality
As the helicopter example illustrates, non-orthogonal systems are inherently more complex to change and control. When components of any system are highly interdependent, there is no such thing as a local fix. To avoid this problem in software, aspects of a system that are unrelated to each other should be implemented in a way that allows them to be modified and evolved without affecting each other (i.e., it should be possible to change A without affecting B, and vice versa). In other words, we should eliminate effects between unrelated things.

Orthogonality is achieved by applying two other design principles, which are described next:

1. The Single Responsibility Principle
2. Minimize Dependencies

Single Responsibility Principle

Related Terms: Cohesion, Separation of Responsibility

Every class should have responsibility over a single part of the functionality provided by the software, and that responsibility should be entirely encapsulated by the class. All its services should be narrowly aligned with that responsibility. Robert C. Martin expresses the principle as,

"A class should have only one reason to change". [Source: Wikipedia, Single Responsibility Principle]

Every class should represent one well-defined concept. All operations on the class should be highly-related to the class' concept. Code that will change at different times for different reasons should be placed in different classes.

Every method or function should perform one well-defined task. Unrelated or loosely-related tasks should be performed in different functions.

The Single Responsibility Principle requires us to keep unrelated things separate in the code, which is necessary for achieving orthogonality.

Minimize Dependencies

Related Terms: 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. Class A depends on class B if:

- Class A invokes a method on class B
- Class A accesses the internal state of class B
- Class A inherits from class B
- Class A has a method parameter of class B
- Class A and Class B both access the same global data structure or file
- Etc.

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

- 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)
- 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).
- 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.

Thus far, we have focused on minimizing dependencies between classes. More generally, the same principle applies to any kind of software "module", including classes, functions and methods, packages, etc. The fewer dependencies a module has on other modules, the better off we will be. The following example from "The Pragmatic Programmer" demonstrates the idea [1].

"Suppose you are writing a class that generates a graph of scientific recorder data. You have data recorders spread around the world; each recorder object contains a location object giving its position and time zone. You want to let your users select a recorder and plot its data, labeled with the correct time zone. You might write:

```
public void plotDate(Date aDate, Selection aSelection) {
   TimeZone tz = aSelection.getRecorder().getLocation().getTimeZone();
   ...
}
```

But now the plotting routine is unnecessarily coupled to three classes: Selection, Recorder, and Location. This style of coding dramatically increases the number of classes on which our class depends. Why is this a bad thing? It increases the risk that an unrelated change somewhere else in the system will affect your code. For instance, if Fred makes a change to Location such that it no longer directly contains a TimeZone, you have to change your code as well.
Rather than digging through a hierarchy yourself, just ask for what you need directly:

```
public void plotDate(Date aDate, TimeZone aTz) {
   ...
}
```

```
plotDate(someDate, someSelection.getTimeZone());
```

We added a method to Selection to get the time zone on our behalf: the plotting routine doesn't care whether the time zone comes from the Recorder directly, from some contained object within Recorder, or whether Selection makes up a different time zone entirely. The selection routine, in turn, should probably just ask the recorder for its time zone, leaving it up to the recorder to get it from its contained Location object. Traversing relationships between objects directly can quickly lead to a combinatorial explosion of dependency relationships."

Minimizing dependencies is sometimes referred to as "writing shy code", meaning that a software module should know about or talk to as few other modules as possible.

The principle of orthogonality says that it should be possible to change one aspect of the system without affecting other unrelated aspects of the system. Minimizing dependencies is essential to achieving orthogonality, because unnecessary dependencies result in unnecessary ripple effects when changes are made.

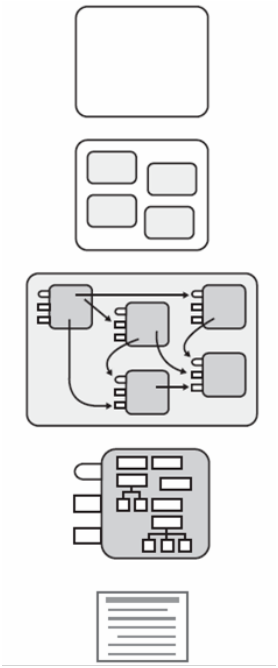
Decomposition

A fundamental technique for dealing with complexity is taking a problem and dividing it into several smaller sub-problems. The sub-problems are smaller and less complex than the original problem, thus making them more approachable. After solving each sub-problem individually, the solutions to the sub-problems are combined to create a solution to the original problem. This approach is sometimes 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 pieces 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 solve complex problems in bite-size chunks.

The solution to each sub-problem is abstracted as a class or method. The solution to the larger problem invokes the abstractions that solve the sub-problems. This results in a concise solution to the original problem, and allows the details of the sub-problem solutions to be temporarily ignored, reducing the cognitive burden of solving the original problem. It is through the decomposition process that many of the necessary abstractions (classes, methods, etc.) 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 methods that 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 Single Responsibility 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, lower-level classes that 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". This is subjective, since everyone has a 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.
In addition to common sense, three principles for helping decide when we have decomposed things far enough are:

1. The Single Responsibility Principle
2. Size/length metrics
3. Complexity metrics

<u>Single Responsibility Principle</u>
If a class implements multiple responsibilities, you should decompose it into multiple classes, one for each responsibility. Similarly, if a method implements multiple responsibilities, you should decompose it into multiple sub-methods, one for each responsibility.

<u>Size/Length Metrics</u>
If a class or method is too big or too long, you should decompose it further. Classes that are very long (e.g., 2000 lines of code), that have a lot of methods (e.g., 50), or have a lot of

variables (e.g., 25) are probably doing the work of several classes, and probably violate the Single Responsibility Principle. Such classes should be further decomposed.

Similarly, a method that is 500 lines long has probably not been decomposed far enough. Methods that have been sufficiently decomposed are often less than 50 lines of code, and in many cases 50 lines is still too long. While there is no "right" method length, the basic principle is that when an algorithm has been decomposed sufficiently, the methods that implement the algorithm tend to be short – frequently, very short. The number of parameters on a method is also something to look at. A method that requires a lot of parameters (e.g., 10) might be doing too much and require further decomposition.

Complexity Metrics

Sometimes size/length metrics don't tell the whole story. It's possible for two methods with the same length measured in lines of code 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 almost 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 sub-methods that each perform part of the original method's work.

High-Quality Abstraction

As previously described, the decomposition process leads us to discover or invent appropriate abstractions (classes, methods, etc.) for implementing the software. Additionally, high-quality abstractions should be sufficiently small, simple, and obey the Single Responsibility Principle. Two more principles that help us create high-quality abstractions are "good naming" and "abstracting all the way".

Good Naming

Abstraction involves taking something that is complicated, giving it a short, descriptive name, and then referring to it by its name. This allows complex ideas can be expressed very concisely in code. 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, packages, 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 is. 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` convey no information to the reader. Name selection makes a huge difference.

In general, class names should be nouns, and method names should be verbs. There are 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.

Abstract All the Way
Related Terms: Avoid Primitive Obsession

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., string, integer, 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 strings or integers 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 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 values. For example, URLs could be stored directly as strings, but if we do so there will be no place to put URL-related operations that may be needed as the program evolves (parsing URLs into their component parts, resolving relative URLs, etc.). Creating a `URL` class 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).

Information Hiding

Related Terms: Data Hiding

In order to minimize dependencies, modules should hide their internal implementations so that other modules cannot depend on those internal details. For example, as much as possible, a class should make its variables and methods "private" rather than "public" or "protected" so that other classes cannot see, and therefore cannot depend on, the details of how the class is implemented. This preserves our ability to change how the class is works internally without affecting (i.e., breaking) other classes.

We now describe four specific ways to achieve Information Hiding.

Visibility
Limit the visibility of classes, methods, and variables as much as possible using the language's public, private, and protected visibility modifiers.

Make class variables and methods private whenever possible. Only make methods public or protected when necessary. Rarely make variables public or protected, opting to have data accessed through appropriate method calls.

When possible, nest classes and interfaces within other classes and limit their visibility as much as possible.

Naming
Choose names that do not unnecessarily reveal internal implementation details. 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 the details of how students are 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 implementations. 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.

Separate Interface from Implementation

Many languages have features that let you separate a class' public interface from its implementation. In C++, the interface of a class is declared in a header (.h) file, while its implementation details are put in a separate implementation (.cpp) file. Java supports abstract interfaces, which can be used to physically separate the public interface of a class from its implementation. Such language features enhance our ability to hide a class' internal implementation details.

Data Hiding

A class' internal data structures should be private, and the data should be accessible only through the class' public method interface. This data encapsulation provides the following benefits:

1. Freedom to change data structure choices without affecting client code
2. Gives clients a nice method interface for conveniently manipulating the data
3. Protects data integrity by preventing clients from changing data in illegal ways
4. Allows for internal optimizations to make sure data access is fast and efficient (e.g., create internal indexes to make lookups fast)

Depend on Abstractions, Not on Concretions

As explained previously, dependencies between modules (classes, methods, etc.) should be minimized and avoided when possible. However, some dependencies are necessary and unavoidable. In this case, it is best to depend on abstract data types (i.e., interfaces and abstract classes) rather than concrete classes. For example, in Java it is better to declare a

variable or parameter with type `List` instead of the concrete type `ArrayList`. The reason for this is simple: a dependency on a concrete type is inflexible because it is hard-coded to reference a specific class. There is no flexibility to substitute different types of objects for the dependency object. For example, if a method parameter is declared to be of type `ArrayList`, it is impossible to pass a `LinkedList` object to the method. This is alright if the method really requires an `ArrayList`, but most of the time the method doesn't really care what kind of list it is given, and should use `List` instead. Rather than declaring variables and parameters with concrete types, it is often better to declare them with abstract types so different types of objects can be substituted in those places. This is what polymorphism is for, and writing code this way provides tremendous flexibility for when testing or modifying the program.

Other examples of concrete dependencies are:

1. Calling "new" to create an object requires that we specify the concrete type of the object. For this reason, classes should avoid calling new to create objects, and delegate object creation to factory classes or use dependency injection (we will learn about factories and dependency injection later in this course).
2. Calling a static method on another class is a concrete dependency. This is so because static methods must be called on a specific class, and this represents an inflexible dependency that cannot be redirected to a different class.

Depending on abstractions is especially important for writing testable code. When unit testing a class, we need to isolate the class under test from the objects it depends on. If we have written the class to depend on abstractions, we can easily pass it fake (or "mock") dependency objects that implement whatever testing behavior we desire. Alternatively, if we have written the class to depend on concrete types, we will have no such flexibility, and it will be more difficult to properly test the class.

Isolated Change Principle

Related Terms: Avoid Shotgun Surgery

The Single Responsibility Principle says that a class should implement just one responsibility. The Isolated Change Responsibility goes in the opposite direction. It says that a responsibility should be implemented by just one class. Or, more realistically, the implementation of a responsibility should have the smallest possible footprint in the source code. That is, the code that implements a responsibility should be isolated (i.e., centralized) in one place - one class, one method, or one package - as much as possible, so that if we change our mind about how that responsibility should be implemented, we won't have to change lots of code that is distributed all of the program to make the change. Design choices will change over time, and we need to make those changes as easy to make as possible.

For example, suppose our software needs to serialize objects so they can be passed over the Internet, and that in our original design we chose to use XML as the data format for serializing

objects. But then, months or years later, we need to change from using XML to using JSON to serialize objects. If we have obeyed the Isolated Change Principle, we will be able to make this change by modifying only one or perhaps a few classes in the system, while the bulk of the system has no idea what data format we are using. Unfortunately, in many systems, this design choice will have seeped into many areas of the code, and will be very difficult to change.

As another example, all of the database code in a system should be centralized in a single package of data access classes. No other code in the system should know what database we are using. This way, if we need to move to a different database, very little of the code in the system will be affected by the change.

In general, when you make a design choice, you should consider how likely it is that the choice might need to change in the future. For choices that are even somewhat likely to change, you should create a design that isolates the implementation of the choice so it will be relatively easy to change. And, remember, the future is very difficult to predict.

Error Handling

Errors will occur during the execution of even perfectly-written programs. When an error occurs, the method in which the error is detected should do one of three things:

1. Recover from the error and continue execution
2. Return the error to its caller
3. Log the error

A method should never ignore an error by doing none of the things listed above. For example, a catch block that catches an exception but does nothing with it is unacceptable, because if that error ever occurs, there will be no way to diagnose what's going on.

Additionally, a method that might return errors to its caller should document what those errors are so that anyone calling the method can handle the potential errors appropriately.

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.

References

[1] “The Pragmatic Programmer” by Andrew Hunt and David Thomas.
[2] Wikipedia article: “Single Responsibility Principle”.