Object Oriented Design & Design Patterns Notes

Rev 2

Goals of Design

- Easy to maintain
- Easy to change when requirements change
- Easy to enhance / extend

Notice that there are many more of these types of statements, but almost all variations amount to “it must be easy to change”. After all, even if we built something almost entirely wrong, riddled with bugs, and with terrible performance, correcting this is really just a kind of change.

Also notice that exercises that verify this are very rare. Most formal training doesn’t have time or resources to have students actually experience this, and hypothetical learning rarely sticks. This is why this class should be taken after some years of real experience; you likely won’t learn new facts, but you’ll understand and be able to think clearly about, and use, the facts you probably learned in your formal training.

First Requirements for Successful Changes

- Must be able to find the piece of code that’s affected
  “Know where to look”
  “Keep together what belongs together”
  “Keep together what changes together”

- Must be able to identify the source of the bug
  “Know who to blame”
  “Find the bug, not consequences of the bug”

- Must be able to understand the code that must change
  “Readable”
  “Understandable”
  “Accurately self-documenting”
  Note that inaccurate documentation is usually worse than no documentation at all.

- Must be able to make changes without causing chaos
  “Minimize the consequences of change”
  “Principle of minimum knowledge”
  “Keep apart what belongs apart”
  “Keep apart what changes independently”

What Kind of Changes?

Changes in the real world come in many kinds. Programmers will be immediately familiar with most of these:

- Requirements:
  - Business rules
  - Laws
  - User interfaces
  - Algorithms
- Bug fixes
- Testing with mocks vs testing with real support
- Performance improvements
- Database schema / DBMS
- Deployment model
- Load capacity / scale requirements
- Supporting technologies
- Network infrastructure
- Security rules
- Deployment environment
- Supporting libraries

- Languages / language version

Of course many more things change beyond just technology. Importantly, in real projects the team members change too.

So, there is a huge variety of types of change, but successfully identifying the potential changes that might affect your system is key to planning for it, and planning is key to facilitating your response to it.

Some changes are more likely than others, and domain knowledge can be key to identifying which are which.

Some changes are hard to address, others are trivially easy.

Good design typically tries to at least embody preparations for changes that are both likely and hard to respond to. Unlikely changes might be prepared for if the preparation is trivially easy. However, it’s not a good choice to invest much effort into preparing for highly unlikely changes if the preparation is time consuming or makes the code harder to work with. Look for an appropriate balance.

Prioritizing Preparation for Change

Not all changes are equally likely, clearly it would be inappropriate to invest heavily in a change that's highly unlikely to happen and for which the preparation has a high cost. On the other hand, even if a change seems quite unlikely, if the cost of preparation is extremely low, it likely makes sense to prepare for the change anyway.

Preparations for changes that have virtually zero cost are sometimes thought of as "just good practice" or even as language idioms, and it's not uncommon to do these things without thinking, or in some cases even knowing why.

Use of factories and builders, in preference to "constructors" is one example.

Directionality of Change

Changes can sometimes be hard in one direction but easy in another. If no other pressure leads to one approach over the other, it's sensible to choose the option that will be easy to change away from rather than the option that would be harder to change.

Making class fields entirely private, and making classes final are good examples, since if access is later deemed necessary, no consequences arise beyond those of the change itself.

Names

- The first recourse when identifying where to look is the names of things:
    - Modules/packages/namespaces
    - Source files/classes
    - Functions/methods/variables

  If names are incorrect or misleading, the job becomes much harder!

- The amount of detail in, and typically the length of, a name, should be proportional to its scope. Local names with meanings obvious from context may be short. Global names should be long, descriptive, and precise

- Don’t delay fixing an ambiguous or misleading name.

Grouping Related Code

- Grouping software into related chunks helps with finding where to look.

- Relationships might be on a macro, or micro, scale; they’re likely hierarchical. For example:
  - Code related to the Accounting system
  - Code related to the business entity Customer
  - Code related to an item sold

- “Code” in this sense might not be executable as such. It includes all the material related to the “computational model”, such as the data representation, and might include HTML, CSS, or images used for UI presentation, or similar artifacts.

- The first “grouping” of software in history was probably the notional “module”. Modules are typically conceptual, but mean have specific, possibly different meanings, in different languages.

- A class is, at one level, a kind of module and serves to keep all the material related to a single “conceptual model” in one place--ensuring you “know where to look”

- A library should keep together what changes together too. If the library is up-versioned, and it includes too many things that don’t change together, you’ll be updating many things that didn’t need to be

Encapsulation

- If any code can modify data, it’s very hard to know “who to blame”, and thereby very hard to know what to fix.

- Encapsulation; rigorously using private to limit code that can change a value, allows blame and isolates the source of particular categories of error.

- If data are properly encapsulated, responsibility for their correctness is identified as “in the class”. This makes it easier to pay attention at the right time, and avoids callers having to do this.

- Encapsulation also minimizes knowledge of the model required by users, so callers cannot make mistakes --this reduces bugs.

- Encapsulation mandates:
    - private fields
    - Construction/initialization only of “valid” objects
    - Prevention of invalid changes by all methods in the class

Error Reporting

- When something bad happens, this must not be ignored.

- Things that should be impossible by the design intent of the software (such as out of bounds array access)

should kill the process, or in the case of a server handling independent transactions, that transaction.
  - These correspond to RuntimeExceptions in Java, there's no expectation of recovery

- Things that fail in the environment should be addressed.
  - These correspond to CheckedExceptions in Java; that language attempts to "force" the programmer to do something.

- If a call attempts to do something untenable to a class (and the API cannot be designed to avoid this), an exception is appropriate. Avoid creating an object in an "illegal" state.
  - Consequences of illegal/nonsensical states can be very hard to debug. Consider an invoice that's due on 32nd of January. How will the system respond?

- However, exceptions in many languages (all JVM languages, and likely many others) are expensive, and should not be used for normal program flow control.

- Null pointers and other "sentinel values" are far too easy to ignore, and prone to creating consequential errors that are hard to trace to their source.

- A monadic type, such as Option/Optional and Either, is a preferable way to address "that didn't work" situations in many cases for modern software. Option types in particular are far less error prone than returning null pointers, and avoid the typical overhead of exceptions.

Documentation

- “Paper” documentation is typically almost useless for internal projects (“user documentation” is likely both useful and necessary for library type projects).
    - It’s often misleading or plain wrong, due to not being updated
    - It’s typically hard to read, since programming and documentation are very different skills
    - Documentation of gross behavior and interconnections (perhaps architecture) is more likely to be useful, in large part because aspects at this level of abstraction tend to change much more slowly, and also because these aspects are harder to glean from other sources; specifically one must examine a lot of code to work this out.
    - As a general principle it might be worth making a very deliberate decision about whether any particular document will deemed worth the effort of maintaining; if not, it might not be worth creating in the first place.

- Accurate names for variables, methods, classes, etc. are usually helpful, but must be kept accurate by hand. Policy and code review is usually necessary to ensure this.

- Code that is simple and short can be genuinely self-documenting. But for this, functions should typically be in the 3 to 5 line range, and carefully divided into sub-functions that accurately capture a single concept in each function.

- Side effects (including mutable objects!) make it much harder to understand what code does. It is likely beneficial for readability and correctness to migrate to working with immutable data and data structures whenever possible.
    - This approach also mitigates many problems in threaded code too, which will become increasingly important as CPUs gain more cores, but not much clock speed.

- Good tests should provide documentation. A test should have the form: in this situation - when you do this - this is the result
    - So, extensive unit tests are probably the best form of documentation, since they cannot be incorrect if they pass!

Separate Things That Change Independently

- If two behavioral parts of “one concept” change independently or if one part of the concept changes while the other part remains constant, the changing behavior should be separated out.

- This allows changing one behavior without risk to the other behavior’s code

- It also allows multiple variations of the changing part to be created

- If done with appropriate technique it can allow many independent behaviors to change, without creating a combinatorial explosion of permutations.

- The separation must allow bringing the parts together in an appropriate way when needed.

- Several design patterns exemplify this

Patterns and Idioms For Separating Things That Change Independently

- Using variables for behavior

  To vary something we typically need a variable that refers to that thing, rather than hard-coding it. Object oriented languages typically do not treat functions as first class elements of the language so variables cannot refer simply to behavior.

  In languages that do permit this (either directly, or using a “pointer to function” idea) things can be fairly obvious.

  Most OO languages tend to use an object (which is a container for state and behavior) for the behavior it contains. Then a variable can refer to that behavior. Whether the object also contains meaningful state is independent of this question. This is how languages like Java, Scala, and even JavaScript actually handle variable behavior. (Scala and JavaScript have “syntactic sugar” that makes it look like an object can be “executed” directly. However, in reality, these are still objects)

The idea of using a pointer/reference to an object as a pointer/reference to behavior is central to many of the Gang of Four design patterns.

- The Command pattern describes a method call where the argument to that method is an object that's passed primarily for the behavior it contains, rather than as a "data thing".

  The command pattern brings the changing part (the argument to the method, also known as the command object) and the non-changing part (the method being called) together at the instant of the method's invocation. The caller gets to choose the variable behavior, and the called method implements the rest of it using that argument for assistance.

  Example use: sorting a collection. The basic "grab two items, compare and conditionally swap them" behavior is fixed, but the implementation of "compare" is client centric, and becomes the variable part, passed as a command object at the moment the request to sort is made.

- The Strategy pattern brings a basic behavior together with a variable part using a stored variable for the variable part. How and when the variable is initialized and/or changed is not part of the pattern. However, the behavioral object is available to one or more operations when those operations determine they need it.

  A strategy is generally preferable to implementation inheritance. A single implementation inheritance model does not work well when multiple independently changing aspects must be modeled, but by using multiple strategies in a single object, this is easy.

Further, behaviors defined by inheritance are fixed at the instance of instantiation of an object. With the strategy pattern, these behaviors can be changed at runtime whenever necessary.

Calculation of the fees and interest on a bank account would be a good example use of this pattern. Notice that the bank manager (or bank policy) might change the fees and / or interest calculation independently. Two strategy objects would be appropriate in this case. The strategies are available whenever needed by day to day bank operations, independently of whether the strategies are ever changed.

- The State pattern can be viewed as a variation of the strategy, such that operations are performed based on input “events”. The processing behavior of one event depends on the event type, but also the “state” of the system. That state is literally defined by the event processing object (a strategy-like object), and the result of each event processing is a new state, which is stored after the event processing completes.

- The Visitor pattern is similar to a command pattern but has the command passed into a data structure to be applied to each element of that data structure. The structure is responsible for navigating itself (so changes to the structure do not impact the caller). The behavior to be applied to each element is passed in using the method parameter approach of the command, and so is in the control of the caller.

Another feature of the Visitor pattern is that traditionally, the visitor (command-like) object implements several methods that accept different argument types. The data structure might contain those different types, and in this way, for example, a tree structure that contains leaf and branch nodes can have those leaves and branches processed differently. This “double dispatch” mechanism is necessary in languages (like C++, Java, and others) that do not support dynamic binding based on argument types.

- The Decorator pattern allows a “pipeline” of processing elements to be chained together in front of a basic service providing element. Each processing element implements the same interface as the basic processing element, and connects, or “plugs into”, that same interface. In this way, any permutation or combination of the processing elements may be assembled at runtime, and passed to any code that was written to interact with the original interface.

  Importantly, this pattern avoids a combinatorial explosion of classes that would be necessary if an inheritance approach were taken for this. It compares favorably even with Scala’s trait model of inheritance, which still cannot support dynamic changes of the pipeline elements after initial creation.

- The Chain of Responsibility allows a number of service providers to be collected together (potentially in a priority order). The managing infrastructure can then poll each element in turn to see if it’s able to respond to a request that was given to the managing infrastructure. Traditionally, the first service provider to answer in the affirmative causes the polling to be terminated.

- The Proxy pattern is structurally similar to the Decorator in that interposes an object between a client and a service, and that object implements the same interface as the service. This again allows the proxy to be inserted, changed, or removed without affecting either the client or server in the relationship.

  Generally, the proxy implements behavior that does not affect the function seen by the client (this differs from the decorator). However, the proxy might reject a request, log it, propagate it over a network, or other behavior that is non-functional in respect of the original service.

- The Flyweight pattern addresses a situation that might be described as a “dynamic enumerated type”. Generally enum types provide type safety for types that have a small number of valid values that is fixed at compile time. Examples would be Hearts, Diamonds, Clubs, Spade, representing suits of a deck of cards, or Monday, Tuesday, etc.

  Clearly, creating a new object to represent Monday every time it’s needed is wasteful. It is more efficient to have one Monday object, one Tuesday object etc. However, if the pool of valid values might vary at runtime, an enum is not suitable, since adding a new value requires modifying the source code, recompiling, and redeploying the program.

  The flyweight pattern addresses this by defining a pool of values, such that new values can be created at runtime (optionally with controls requiring privilege to do so) but that whenever an existing value is requested, the already-created object is returned.

This behavior is similar to that of the String constant pool in Java (only fully enforced, rather than being half-and-half).

The original description of the flyweight pattern by the gang of four also described objects composed of these pooled values. The specific example they cited was of characters on a WYSIWYG text editor screen. In that way, the memory intensive definition of the bit pattern for the graphical rendering of a letter (‘A’ for example) could be shared among all the occurrences of that letter, while the x,y coordinates of each letter might be different).

- The Iterator pattern groups the responsibility for iterating over a data structure into the data structure itself while associating the “progress” of the iteration with the client. In this way, two clients can iterate the same structure and each sees all the data items correctly. At the same time, if the type of data structure is changed, the client does not notice, because knowledge of the structure is embedded in the iterator object, not in the client.

- The Observer pattern separates the response to a stimulus from the originator of that stimulus. The best known example is probably the event handling on GUI components. A button knows that it has been “pressed” but the behavior that results must be easily, and likely dynamically, configured at runtime. A command-like object is passed into the button for later invocation, creating a situation sometimes referred to as "call-back".

It's also common for the observer pattern to be implemented in such a way as to permit multiple observers to be added to the "observable" so that each is invoked without the others being involved. Removal of observers at runtime is also typically supported.

- The Factory pattern is not an original gang of four pattern, but is widely used. The name "factory" is commonly used to describe use of a method or function (often, but not necessarily static) for the specific purpose of creating and initializing an object.

  Constructors have several limitations that are avoided by the factory approach. In particular, when invoking a constructor, the result will be either a brand new object of exactly the named type, or an exception. However, using a factory, these are all options:

  - A new object or an existing one, which can be a singleton, or from a pool,
  - An object of the factory's return type, or anything assignment compatible with that type (even private classes that implement an interface defined as the factory return are possible)
  - An exception

  This means that anything a constructor can do, a factory can do, but the factory can do much more. In particular, the factory can hide changes. So, if code uses a constructor, but it becomes clear that shared objects from a pool (see Flyweight) or a singleton would be appropriate, that change will have far reaching consequences in the code. However, such a change would be invisible had a factory been used from the start.

Factories have another benefit; they are regular methods with names. This name means that multiple factories can have identical argument types, but be distinguished by their names. By contrast, if multiple constructors are desired, they must be valid overloads; that is, their argument types must differ.

A further, albeit small benefit is that the name of a factory can be descriptive of what it does in a way that’s more helpful than with a constructor where the name is fixed. This name can, for example, shed light on the arguments, partially compensating for languages like Java that only pass arguments by position, and not by name. Imagine a constructor for a (Calendar)Date that takes three ints. Are these day, month, year, or perhaps year, month, day, or yet, month, day, year. A factory can be named ofYearMonthDay, and remove the ambiguity.

Group Things That Change Together

- Where several things change together, it’s helpful if they can all be found in “the same place”. This “same place” might be a package, a library, or in some cases even a class, for example.

Patterns Grouping Things That Change Together

- The Abstract Factory pattern identifies that a group of related objects (the typical example is of GUI components) must all have a related look and feel. To provide for this, while allowing them all to change as a group, the abstract factory provides a centralized point for obtaining components. The abstract factory can be configured to create components from one of several sets of components, such that at any given time, all the components it creates have the appropriate look and feel.

Patterns Minimizing Knowledge And Limiting The Consequences Of Change

- The less a user of a service has to know about how to interact with that service, the better. If an implementation detail changes but the caller doesn't know or rely on that detail, the caller is unaffected.

- Encapsulation is the first line of defense in this respect.

- Interfaces, traits, and similar features of those languages that provide them also provide a barrier against seeing potentially volatile implementation details, and thereby against the consequences of many types of change.

- Also, good class API design avoids the caller having to understand things about "how to use" the objects whenever possible. In general objects that exhibit "stateful" APIs where some methods must follow others, or certain method calls or arguments are only legal if the object is in a particular state should be avoided.

Consider for example a (Calendar)Date object that provides a setDayOfMonth(int) behavior. Calling this with day number 31 is legal only if the month currently represented has 31 days.

Instead, this API could be served by a method such as setLastOfMonth. This would set 28, 29, 30, or 31 depending on the current month and year. This method cannot be mis-used. In this situation, more methods would be needed to fill out all the legitimate reasons for setting a particular day (adding a number of days, or weeks, and similar). However, there should be no need for the dangerous method setDayOfMonth.

Generally an API that requires less knowledge on the part of the caller is more robust and less error prone.

- The Adapter and Bridge patterns allow the interface/API used to access an object to vary independently of the native API for that object. They can be particularly helpful when integrating systems that were not created by the same group or company.

Identifying Unrelated Things

- A common failing in designs occurs when behavior is lumped into the nearest available class by mistake, simply because there is not better location yet defined.
- This behavior is often “business logic”, for example, how a company processes a customer order. It’s common to see this behavior lumped in the Customer entity class, because it feels like “Customer stuff”. However, this is a poor choice for at least two reasons:

- The processing of an order varies independently of the customer, so should almost certainly be separated for that reason anyway.
- Further, this processing is not intrinsic to the essential nature of a Customer, rather it's an artifact of the way the company chooses to do business. Business logic of this type is actually a legitimate "concern" (in the "separation of concerns" sense) in its own right. It should have its own class.

Patterns Identifying Unrelated Things

- Two patterns in particular identify this "unrelated concerns" situation. They are the Facade and the Mediator. In each case, business logic is separated from lower level service providers (and from the user of those services).

- In the case of the mediator, the business logic describes interactions between the elements of a system, and there is no particularly strong sense of "direction" from request to response. Instead, the interaction might start at any particular participant, and propagate around others under the control of the mediator.

  This pattern commonly occurs in GUIs, where the business logic resulting from, for example, pressing a button should not be embedded in that button, nor any of the other components that are updated after the button is pressed.

  The pattern also captures very well the nature of a request broker in a classical Service Oriented

Architecture system. Each “service” has its own responsibilities and they relate specifically and solely to that end point (e.g. “accounting” or “inventory control”) it would be very fragile to embed the larger business processes into any of these services.

- In the facade pattern, there is typically a sense of “direction”, such that a client makes requests of service elements. Some such elements already exist, but the goal is to build higher level services from these components. It’s not appropriate to embed compound logic into the existing services, nor into the client, so instead a new object embodies the aggregating logic needed to make requests of several of the smaller, pre-existing, services, and return a single result from that.

Afterword: Correspondence with SOLID principles

A key goal is to understand the problems that must be solved, the techniquest that can address them, and how those techniques work. Consequently we've discussed a number of approaches, for example, separating things that change independently. Note that these suggestions are in effect simply more practice-oriented rephrasing of the well known SOLID principles:

- Single responsibility
- Open/closed principle
- Liskov substitution principle
- Interface segregation principle
- Dependency inversion principle