Patterns for Exploring Design Alternatives

Charles Crowley
crowley@cs.unm.edu
Computer Science Department
University of New Mexico
Albuquerque NM 87131
505-277-5446

Abstract: Design often consists of considering a series of solutions that combines generating and analyzing potential solutions. One way to find a new solution is to look at a variant of the current solution. If the solution uses existing primitives we might consider adding a useful new primitive that makes the solution easy. Or we might consider a new primitive that solves the problem directly. Or we might consider changing out point of view and doing things in the opposite way. Or we might consider a solution with requires a special-case solution of the same problem. The patterns in this paper explore these alternatives.

Subject: Design ideas

Keywords: levels of implementation, reversal, opposite point of view, special case

Introduction

When designing an implementation for a new function in a system you need to come up with a good design solution. This involves two activities. First you have to come up with the design solution and second you have to know it is good. The first activity is a creative activity where you want to be open and think of any many alternatives as possible. The second activity is an analytical activity where you want to be critical and be sure that this is a good solution. It has been said that if you cannot think of two different ways to implement a specification that you don't really have a specification at all but a description of an implementation. So we will not be satisfied with one design idea but we will try to come up with some alternatives.

The abstract version of this process is in two distinct stages. First you come up with a list of alternatives and then you analyze each one. Like the waterfall model in software engineering, this model does not fit real practice. It is very hard to consider a design without doing some analysis of it. Generally this first analysis will determine whether the design works at all, that is, whether it solves the design problem. If it works the analysis will determine whether it works poorly and well. Then you go on and think of other ideas often using the present idea as context and thinking of a variation of it.

You want to consider as many alternative designs as possible (well, within reason). You might start with one design and use it as an inspiration to think of other designs. While your goal is to find a good design, you should not worry too early whether a new idea will turn out to be good enough. It is best to be creative and not too critical while looking for alternatives. For each solution you come up with you will analyze it and see how good it is. This involves various factors like how fast it would be to implement, how fast it would run, how much space it would take, how easy it would be to change later, etc. A particular design idea might not be good enough in itself but it might still be worth exploring because it could lead you to another related design idea that is very good. (See

Booch (1991, Chapter 6) and Stroustrup (1997, Chapters 23-25) for discussions of design and design iteration.)

Designers commonly move from idea to idea, analyzing each one enough to get a rough idea of how good it is and then often using the idea as a basis of or an inspiration of the next idea. It is rarely good enough to look at one potential solution. In this paper I will present some patterns that help you explore alternative designs. These patterns make no guarantees that the alternative designs will be good, just that they are possibilities to explore. Once you have an idea it is then possible to analyze it and see if it is a good one.

Note that this is different than typical design patterns. All design patterns suggest design ideas but most of them consider evaluation as well. They give a good solution in a context. The patterns in this paper do not make any evaluations of the design ideas generated. They just give some ways to generate design ideas.

As an example, let us assume that you are working on a word processor that already has a string search function. You are charged with adding a function that will search for a string taking formatting into account. For example, one could use this function to search for all instance of the work "*book*" in italics, or in the "emphasized" character style.

Idea 1: One thing you might think of it to implement the facility in the macro language of the word processor. It would be a simple loop that went through the text character by character, comparing against the search string. The basic sub-function would be a function that compared two strings for equality of characters and formatting. This solution builds on primitives you already have available and does not require any new functions to be added to the word processor.

Idea 2: In analyzing this solution you think that maybe it would be too slow because the macro language is interpreted and is not meant for heavy computing like comparing tens of thousands of characters. An approach to this problem is to separate the concepts of comparing the characters and comparing the formatting. You already have a fast primitve function that searches for a string in the text. We can use that and only check for formatting on strings where the characters match. We only need a function that compares fonts (and assumes the characters are equal). Thus we solve a special case of the problem (matching fonts) and use it, with other existing primitive functions, to solve the whole problem.

Idea 3: If we want a little more speed we can implement a formatted string compare function in C++ (assuming that is what the word processor is written in) and offer it as a primitive for macro writers. This way the two complex functions (string search and formatted matching) are done in native code and the macro language is used as it should be, to glue together calls to fast native functions.

Idea 4: As long as we are adding C++ code we might consider implementing the entire function in C++. But this adds more code to the word processor and is unlikely to be that much faster. The problem is that once we decided to separate the text search from the formatting compare we will have to fetch the text twice, once for each purpose. If we rewrite the string search function itself we can avoid the double fetch on strings where the text matches.

Idea 5: This is also not much faster since we are only make the glue code faster not the code in the loop. But while we are thinking of rewriting the search function we can think of other ways to do it. Once we decided to separate the string search from the format compare we were locked into a fixed order: search for text then compare formats. If we decide to take the opposite point of view we should consider searching for the formatting first and then searching for the text. At first this seems to be a poor idea but it actually would be very fast for uncommon formatting. The reason is that our word processor represents the text as a tree structure. Each inner node has formats that apply to it and has sub-nodes with text or more formatting. Suppose you come to a node high in the tree that represents the next 10,000 characters and indicates that they are all in the normal roman font. If you are looking for bold text you can skip this entire node without looking at and sub-nodes or text at all. This would be faster than the fastest string matching algorithms. (See Gamma et al. (1995, pp. 201-205) for a discussion of this data structure for text.)

This example previews all of the patterns described in this paper: special case, three levels of implementation, and taking the opposite point of view. In the next sections we will examine each of these patterns and give other examples of their use.

Three Levels of Implementation

Context

You have a system consisting of a set of existing functions already implemented and you want to add a new function to the system. This situation might occur when you are adding a new function to a finished system. More commonly it will occur when you are designing the system and you are figuring out how to add each function one by one. You would like to generate as many ideas as possible about ways to implement the new function.

Problem

It is hard to come up with good design ideas. Guidelines can be helpful.

Solution

When you want to add a new function to a system you can use three approaches:

1. Build on existing mechanisms already in the system.
2. Add a new low-level mechanism and use it to build the function.
3. Add a new high-level mechanism that exactly provides the required function.

Each design idea will fall into one of these three categories and you should try to think of at least one idea in each category. In almost every case, solutions exist in each category. (See Crowley (1997, PP. 369-373) for another discussion of this pattern.)

Motivation

Consider the problem of waiting for a message from two queues at the same time. There are three solutions to the problem.

1. Use two sub-processes or threads: Each sub-process (or thread) waits on one of the two queues and forwards messages to a third queue. The main process waits on the third queue.
2. Use a non-blocking receive: The operating system provides a non-blocking receive. The process uses a loop to check for messages in one queue, check for messages in the other queue, and then delay a short time. (This is a busy waiting solution.) (See Crowley (1997, pp. 298-300) for a discussion of non-blocking receives.)
3. Use a special purpose receive: The operating system provides a form of receive that waits for two queues at the same time.

Each of these solutions has good and bad features. The two-sub-process solution builds on existing functions (the process or thread manager in the operating system) and does not require any new system calls or operating system services. It is fairly inefficient since processes are expensive to create and destroy. With threads this solution is less expensive.

The non-blocking-receive solution requires a new system call (or a change in the semantics of an existing system call). In addition, this solution requires busy waiting. Note that the non-blocking receive does not solve the problem directly but it provides the key building block needed to construct the solution.

The two-queue-receive solution also requires a new system call but the operating system can take over the waiting and allow the process to sleep while waiting. The operating system can recheck the queues when they change and so the waiting will not be busy. This solution is the only one that solves the problem directly.

The two-subprocess and non-blocking-receive solutions generalize easily to N queues with no additional functions required. The two-queue-receive solution can be generalized to pass in an array of semaphores on which to wait but this requires another new function. System V UNIX has such a primitive for arrays of semaphores.

Known Uses

These three categories of solution can be found in almost every design situation where you are adding a new function. Here are some examples.

Iterating through an abstract data type (data structures)

Suppose you are implementing a set as an abstract data type (ADT). One operation that users of the set ADT will want is to iterate through all the elements in the set. There are three ways we could implement this iteration:

1. Allocate enough memory to hold all the members in the set and return the allocated memory containing all the members of the set. Iterating through the set consists of iterating through this returned list. This solution uses the existing memory allocation function that can handle objects of variable size.

2. Provide two new system calls: one that will return the first element in a set and one that will return the next element in a set. These calls can be used to build a loop that goes through all the elements in the set.

3. Provide a system call that will iterate over all the elements in a set and call a function passed by the process for each element. This solves the problem directly.

See Gamma et al. (1995, pp. 260ff) for a discussion of internal and external iterators (alternatives 2 and 3 above). See Booch (1987, PP. 158-161) for a discussion of active and passive iterators (alternatives 2 and 3 above) in Ada.

Implementing threads (operating systems)

Threads can be added to an existing operating system in several ways.

1. Threads can be implemented at the user level with no help from the operating system by implementing a dispatcher in the user process. This solution is efficient but one thread can block all the others by making a blocking system call. (See

http://www.mit.edu:8001/people/proven/pthreads_man.html for an example interface to user level threads.)

2. What would a low-level primitive solution to this problem be? The problem with user threads is that they might block the entire process. A solution to that would be for the operating system not to block a process but to call a specially designated scheduler thread when it would have blocked and call it again when it would have released the block. This allows the scheduler in the user process to block the thread, run other threads and unblock the thread when the operating system operation is completed. (See Draves et al. (1991), Massalin and Pu (1989), and Marsh et al. (1991) for descriptions of variations on this approach.)

3. Threads can be implemented at the operating system level. (Almost all modern operating systems do this.)

The most interesting aspect of this example is now non-obvious the low-level solution is. This technique for thread implementation was not discovered until 1991.

Adding new widgets to the Tk toolkit (data structures)

Tk is a window system toolkit that was developed for use with the Tcl language but has proved so popular that it has been ported for use in several other languages (Scheme, Perl, even Prolog). The Tk toolkit evolved over several years and a number of new widgets were added along the way. A specific example of a very useful widget that was not available until recently in Tk was a file selection dialog box. Virtually every application uses a file selection box. So how does one implement a file selection box?

1. The most popular solution was to build a file selection box out of existing Tk widgets. This was quite easy to do and there were several available from Tk distribution sites. The problem was that they did not act like the built in widgets. You could not specify widget attributes in the same way for example.

2. Another solution was to build a *mega-widget* facility, that is, a set of low-level functions that allowed you to build new widgets. Then you could build a file selection box using the mega-widget facility. Incr Tcl (See http://www.tcltk.com/itcl/index.html for information.) was probably the most well known example of this but there were several others as well. Of course, a mega-widget facility allows the creation of any number of other widgets. Jaeger (1995) is a good review of mega-widget work in Tcl/Tk.

3. The high-level solution was for the developers to Tk to add a file selection box widget to the Tk toolkit. This was done in a recent release of Tk (http://sunscript.sun.com/TclTkCore/8.0.html).

An important issue in Tk was that many people wanted to use only the standard Tcl/Tk distribution with no extensions because you never knew who had which extensions. This makes the mega-widget solution less popular.

Adding an object system to a language (language implementation)

In the past 10 or 15 years object-oriented programming has become very popular. This created a demand for languages that support the object-oriented paradigm. There are several ways this can be done.

1. It is possible to add object-oriented facilities to most languages. For conventional languages like C, this must be done by coding conventions. The X library toolkit used this approach. It was written in C but it used coding conventions to make it look somewhat object-oriented.

   More dynamic languages can implement objects using existing language facilities. For example, it is quite easy to implement an object system in Common Lisp because Common Lisp implements closures which provide the basic hook you need.

   Tcl is a middle case. You can implement a pretty good object system in Tcl with no changes to the language. However, the system will be quite slow.

2. It is much easier to implement an object system in a language which has namespaces. A *namespace* is pretty much what it says, a collection of (unique) names collected together and only visible when the name of the namespace itself is used along with the name in the namespace. An example is the names in a C struct.

   Namespaces are the center of an object system because an object is basically a namespace. You start with namespaces and add public and private names, inheritance, dynamic binding, etc. and you have an object system.

   Namespaces are the hook you need to easily and efficiently implement an object system. In addition, namespaces are useful for other purposes. C++ recently added a namespace facility even though in also has a full object system.

3. The high-level solution is to implement a new language that is an object-oriented version of the old language. This is the most common solution because it is most efficient and can be used with any languages. Examples of this are: Object Pascal, Objective C, C++, Incr Tcl., and Perl 5.

Applicability

This pattern provides guidance whenever you are implementing a new function. You will have to do it in one of these three ways. What the pattern says is that you should look at all three ways and see which one is the best in your situation.

This pattern is a guide to give you ideas on implementation. It does not say that all three methods are reasonable in every situation, just that they are possible. Examine the three approaches and pick the best one.

Consequences

- *Features of using existing functions*:
  - no changes to the system are required
  - no new concepts to learn
  - can be inefficient if the existing functions are not appropriate for the problem
- *Features of using low-level mechanisms*:
  - easier to implement than high-level mechanisms
  - can be used for other purposes, encourages exploration of new solutions

    - flexible
    - harder to use, you must build the solutions yourself
- *Features of using high-level mechanisms*:
    - can be very efficient
    - easier to use, the solution is ready-made for you
    - not as flexible as low-level mechanisms because it can only be used for this particular problem
    - can be made more secure because you know exactly what the user is doing and do not have to provide for general cases

Implementation Issues and Variations

Two levels of implementation

There is an implied assumption throughout this section that there are two levels of implementation. It is assumed that the upper level of implementation will be in some kind of programming language, possibly a macro language. This language is based on a set of primitives implemented at the lower level. One issue is whether new primitives are provided by the lower level or the solution is accomplished completely at the upper level.

There are some boundary cases of this assumption. In some cases the upper level of programming might be a user executing commands interactively at the interface. If this pattern is used during system design the lower level is the set of primitives designed so far and the upper level is the code for the function we are currently working on.

Security Issues

Adding a low-level mechanism that can be used to solve the original problem allows the user to use the low-level mechanism in a variety of ways, and, no doubt, some will not have been anticipated when the low-level mechanism was designed. This is an advantage in terms of flexibility, but a disadvantage in terms of security. Some of these innovative ways to use the low-level mechanism may allow someone to break the security of the system. A high-level mechanism can not be used as many ways and so is less likely to present a security threat.

An example of the security problems caused by interacting low-level functions is the TENEX password bug. TENEX used eight character passwords for protecting users accounts and files. Guessing a password is unfeasible since there are $128^8 = 2^{56}$ or about 10^{17} possible passwords. TENEX checked the proposed password against the saved password character by character. TENEX also allowed users to handle their own page faults. The interaction of these two functions created a security flaw. A user could place a proposed eight character password so that one character was on one page and the next seven characters were on the next page. Then the user would make sure the first page was paged in and the next page was paged out. A password check would either return an invalid password, if the first character was not correct, or it would return a page fault, if the first character was correct. This meant that one could guess the first character in at most 128 guesses. Once you had the first character you could move the password down one character and use the same procedure to guess the second character of the password in at most 128 guesses. Thus the entire password could be guessed in at most 8*128 or 1024 guesses. (See Tanenbaum (1992, pp. 183-184) for more discussion on this security problem.)

It was very easy to fix this security flaw once they discovered it. They simply moved the proposed password to a system buffer and checked it there. The point is not that it is not easy to fix these security problems, the point is that adding low-level functions allows for many interactions and it is difficult to anticipate them all.

Related Patterns

The low-level mechanism solution is related to *solving a problem by solving a special case*. The low-level solution is the key thing you need. You can use it to solve the original problem.

The low-level mechanism solution will probably be part of a *two-level solution* to the problem.

The problem with the high-level mechanism solution is that the users cannot customize it as easily to fit their particular needs. This situation can be improved by adding *hooks* that allow the user to get control at certain times and extend the functionality of the system.

The difference between adding a high-level primitive that solves the problem directly and a low-level primitive that can be used to solve the problem is related to the idea of *separation of* concepts. The low-level primitive is better when it can be used in other ways to solve other problems. If it cannot be used in any other ways you might as well package it up in a high-level solution and save the user some work.

Opposite Point of View

Context

You already have considered some ideas for solving a design problem but none of them are good enough to pick so you would like to generate more ideas.

Problem

It is hard to come up with good design ideas. Guidelines can be helpful.

Solution

Design solutions usually have some kind of “direction.” This might be who calls whom, what gets looked up and what gets found, the order in which you do the steps, etc. Often this is a seemingly “natural” direction for things. A fruitful source of design alternatives is to change the direction in a design solution or to change your point of view.

Motivation

In a computer system with virtual memory, a running program generates virtual addresses and they are translated (by the hardware) to a physical address by looking up a page table entry in a page table. Normal page tables map from virtual addresses to physical addresses and so have a table entry for each possible virtual page. Actually, in real paging systems, most page references do not use the page table but are translated using a cache of the page table called the *translation look-aside buffer*. (See Crowley (1997, pp. 438-441) and Tanenbaum (1992, pp. 101-105) for discussions of translation look-aside buffers.)

This organization is fine for small virtual address spaces with small page tables but it starts breaking down as the virtual address space gets larger. If a program uses a full 32 bit address space (the maximum on many current hardware systems and about 4 billion bytes) and 4 Kbytes pages, it will require 2^{20} pages and a page table size of about 4 megabytes. Newer computer systems are moving to 64 bit words and potential address space of up to 64 bits. This will make the problem even worse. These large virtual address spaces tend to be very sparse, that is, not many pages are actually mapped but the ones that are mapped are spread out all through the virtual address space.

One design solution to this problem is to use of the principle of *Use a good idea over again*. This leads to two-level paging where the page table are themselves paged. This helps some but as address space get bigger you need to move to three-level paging. There are even processors that implement four-level paging. Page table lookup starts getting very slow in such systems. (See Crowley (1997, pp. 509-510) and Tanenbaum (1992, pp. 98-101) for discussions of two-level paging.)

Instead we can change our point of view and keep a table of physical pages (page frames) rather than virtual pages. There are far fewer physical pages and so the page table, now called an *inverted page table*, is much smaller. The problem is that mapping from virtual to physical pages now requires a search of the inverted page table that can be very slow. The solution to the problem is to use a fast searching technique, like a hash table. Since few translations require an actual page table lookup this solution is fast enough. . (See Crowley (1997, pp. 512-517) and Tanenbaum (1992, pp. 105-107) for discussions of inverted page tables.)

Inverted page tables work because the address space is sparsely used, that is, most parts are unused. A similar technique is used to implement sparse matrixes. A sparse matrix is one in which almost all of the entries are 0. The usual implementation of a matrix is a large area of memory with a memory cell for each entry in the matrix. A reversal of this is to have a memory cell only for the non-zero entries in the matrix. This requires lots of interlocked linked lists to keep track of which cells are non-zero but overall it can save huge amounts of storage.

Known Uses

Hooks versus polling (event processing)

It is common to have processes that must respond to events. There are two ways of finding out when an event has occurred. With *polling*, the process inquires repeatedly if the event has occurred. With a *hook*, the process requests to be notified when the event has occurred. A hook is a form of software interrupt. Hooks are more efficient than polling because they do not involve busy waiting. With polling the process calls the event source. Hooks reverse the direction of the call by asking the event source to call the process. Instead of making a call to ask if the event has occurred, the process receives a call when the does occur.

Hooks relate to another design technique called *registration*. A process registers with some agent to be called when some event occurs. A software purchaser registers with the vendor so he or she can be notified of new versions. A GUI factory registers the fact that it can provide a Motif interface. Hooks require a registration mechanism but registration can be used in other contexts as well.

Sometimes a program has a bug where some variable is being incorrectly set. The bug manifests itself some time later when the incorrect value is used but by then it is not clear how it was incorrectly set or by what code. What we need is a hook to tell us when the variable is set. Normal hardware does not provide this but many debuggers allow you to set a “watch” on a memory cell and stop execution when it is set. The debugger provides the hook mechanism (usually by interpreting the code instead of executing it directly). A similar facility exists in the Tcl programming language allows you to set a “trace” on a variable and be notified when it is read or written.

With a hook, instead of you calling them, they call you. We might call this the “Hollywood” principle but in this case they really do call back. This brings up a point of about polling. Polling does not depend on another process calling you at the right time. Using a hook assumes that there is some agent that will accept your registration and call your process at the correct time. Hooks require more cooperation and trust between processes than polling does.

See Crowley (1997, pp. 468-489) for more discussion of hooks. See Gamma et al. (1995, pp. 293-303) for a discussion of the Observer pattern which uses the idea of registration and the hook concept.

Visitor design pattern (design patterns)

In the Visitor design pattern it is necessary for a process to visit every node in a hierarchy and execute one of a set of procedures with the visited node as an argument. The procedure called differs depending on the type of the visitor and the type of the node. One solution is to test the type of the node at run time. The code would be roughly like this:

```
// This code visits each node
foreach node in hierarchy {
  switch( TypeOfProcessingRequired ) {
    case Proccessing1: process1(node); break;
    case Proccessing2: process2(node); break;
    ...
  }
}
void process1( NodeType node) {
  switch( RunTimeTypeCheck(node) ) {
    case type1: ...
    case type2: ...
    ...
  }
}
void process2( NodeType node) {
...
```

We can reverse this and get the node to call the visitor:

```
// This code visits each node
foreach node in hierarchy {
  node->visit(processor);
}
// Each node has a visit function
void visit( ProcessorType processor) {
  processor->process( self );
}
```

The first call (to `visit`) will go to the visit procedure for the node in question. This C++ dispatch will be based on the subclass of `node`. Then the node will call the processor. This C++ dispatch will be based on the subclass of `processor`. By reversing the direction of the call (and calling twice) we avoided the switch statement and any run time type identification.

What we are really doing with the visitor design pattern is to implement double dispatch (dynamic dispatching where the method chosen depends on the type of the first two argument to the method, not just the first argument as in typical object-oriented languages) in languages that do not implement it in the language itself.

See Gamma et al. (1995, pp. 331-344) for a discussion of the Visitor design pattern,

Reverse execution (debuggers)

Some recent debuggers allow you to execute programs in reverse. Why? Suppose you are running at full speed and get an error. Reversing the execution allows you to see what happened just before the error. The idea is that the real error occurs some time before it shows up in the output of the program. You can get to the error by executing from the beginning of the program to the point of the error or from the point the error shows up back to the error. The latter is likely to be a much shorter path. See Lieberman and Fry (1995) for a discussion of a reversible debugger.

Dataflow and demand-driven architectures (computer architecture)

In the dataflow architectural model, an operation is performed as soon as its inputs are available. This leads to the maximum possible parallel execution. A reversal of this is the demand-driven architectural model where an operation is not performed until its output is needed by another operation. This prevents unnecessary operations from being performed.

A similar reversal occurs in lazy functional languages. In a lazy language, a result is not computed until it is actually needed in another computation. In the meantime the result is represented by a placeholder, sometimes called a "promise." See Kamin (1990, pp. 153-204) for a discussion of lazy functional languages.

Optimistic parallel simulation (simulation)

Simulations often take huge amounts of computer time. One solution is to run them on massively parallel processors. A problem with this solution is that the parts of a simulation require a lot of communication and synchronization. An optimistic parallel simulation assumes it knows what the other parts of the simulation will do and continues on that basis without synchronization. When it gets a message to the contrary it backs up to where it made the optimistic assumption.

Applicability

Reversal is a technique for generating design ideas and it applies just about anywhere. Think about what you are doing and see if you can somehow reverse it and do it the opposite direction as you have been doing it. A reversal is not always better but it is an idea that you will want to consider.

Consequences

Usually the reversal allows you to optimize a different factor than the original design solution. Inverted page tables save space over conventional page tables but take for processing time. Hocks optimize processing time but require a hook registration and calling facility which creates extra software complexity. Optimistic algorithms allow extra parallelism but require a complex rollback mechanism.

Implementation Issues and Variations

Reversing procedure calls

Many reversals involve reversing the direction of a procedure call. Both the visitor pattern and hooks did this.

In most object-oriented designs, the methods in the subclasses call superclass methods for functions than have not been changed by subclassing. These calls go up the inheritance tree. In the template method design pattern, the basic algorithm is in the superclass and it calls the subclasses to do actions that can be specialized by subclassing. These calls go down the inheritance tree.

Frameworks use a similar reversal. In traditional component reuse the main functions are written for each application and the sub-functions are often reused functions from the library. In a framework the main functions are reused and the sub-functions are written for each application. Here the direction is not reversed. Instead which functions are reused is reversed.

Changing with Technology

Software page table lookups are a good example of how the design of operating systems has changed with the technology to support it. The original paging systems had small TLBs whose hit rate was not very high. For these systems, handling TLB misses in software would be too expensive and therefore the hardware had to handle the mechanics of page tables. Newer TLBs have high (>99%) hit rates, so we can afford to handle the TLB misses in software. This simplifies the paging hardware (since it does not have to do page table lookups) and frees up space on the chip for other uses (like bigger caches). It also allows more experimentation with page table structures and more sophisticated structures (like inverted page tables).

Optimism and pessimism

Some of these reversals depend on whether you are optimistic or pessimistic about an event occurring. With optimistic simulation and mutual exclusion you assume that you know what will happen and proceed on that assumption. With deadlock recovery you assume that deadlock is unlikely. With double check locking you assume you won't get the lock and so do the weaker read. With a hook you assume a poll will indicate the event has not yet occurred and avoid the overhead of repeatedly asking.

There are two issues: the cost of the action and the cost of being wrong. In a reversal we try to reduce the overall cost by avoiding too many actions or avoiding costly actions.

What is changed?

In inverted page tables we reverse the direction of the page lookup function (page frame to page instead of the traditional page to page frame). In the visitor design pattern and in hooks we reverse the direction of the procedure call. In optimistic algorithms we reverse the action with the check to see if the action is called for.

Special Case

Context

We have tried one or more simple solution to a design problem but they are not adequate. We need to try a more complex solution.

Problem

A potential solution involves a special case of the general problem we are trying to solve.

Solution

Use one of the previously rejected simple solutions to solve the simple case. This completes the more complex solution which solves the general case.

Motivation

Sometimes we try to solve a problem and run into a recursive instance of the same problem. We call this a recursive design problem. This happens in the implementation of messages on a multiprocessor. Messages can be used to implement mutual exclusion but to implement messages we have to solve the mutual exclusion problem with respect to access to the message queue data structures. Now we have a special case of the mutual exclusion problem. The general case involves any two processes and the time a process may be in a critical section has no limit (it could be seconds, minutes, even hours). In the case of access to the message queue data structures the critical section is only a few machine instruction executions long. We can use a busy waiting solution in this case because the time in the critical section is so short.

Known Uses

Name servers (operating systems)

A problem in networks is knowing the addresses of servers. Often we know the name of the server we want but not its address. If we can get the address of the name server then it can translate server names to network addresses.

Help systems (user interfaces)

Requiring people to remember how to execute all the commands in a system is too much of a memory load but we can require that they remember how to execute one command, the help command, and this gives them assistance on all other commands. This is an instance of the same principle — reducing a problem to a special case and then using a simple method for solving the general case.

Unique global names (operating systems)

The problem of assigning unique names in a distributed system can be reduced to the problem of assigning unique names at each level of the system. Since each level is managed centrally the unique name problem is easily solved.

Text editor data structures (algorithms)

A text editor must keep a long sequence of text and make frequent insertions and deletions to the sequence. A simple array of characters would require too much moving around of characters since a sequence of text may have tens or hundreds of thousands of characters. If you deleted the first character of a 50,000-character file you would have to move 49,999 character down one position in the array. To solve this problem suppose we decide to keep each line of text together and have a sequence of pointers to the lines. If we edit within a line we put the edited line in a new area and change the line pointer to it. The `vi` text editor uses this data structure. On the other hand keeping the characters in a linked list, one node per character would be terribly inefficient in space even though insertion and deletion would be easy but we could keep line pointers in a linked list because there are many fewer lines than characters.[1]

Applicability

It is common in a design problem to think of simple solutions that are not quite adequate for some reason. They might be too slow or take too much space. In developing more complex solutions we sometimes encounter another instance of the same design problem we are trying to solve. Usually this recursive design problem is different and, in most cases, simpler than the original one. This means that the simple solutions that would not work in the general case might work in the simplified case.

Consequences

This technique is of general use. Often a problem has no good solution but it can be reduced to a special case and a solution that did not work in the general case does work in the special case.

Implementation Issues and Variations

Changing design problems

A design problem includes a lot of information beyond the simple statement of the problem. Information like how often the problem occurs, how big the data structures will be, etc. When we

[1] There are better data structures for text editors than these. The most popular is the ``gap'' method which keeps the characters in a large array with a gap in the middle, at the insertion point.

encounter a recursive design problem the general statement of the problem is the same but this related information is different. These differences allow a different set of solutions.

Relation to other patterns

Consider the design scenario depicted in the introduction. As you work on the design you come up with a series of ideas as you go along, each one in more or less detail. Some ideas will be too simple and fail because they are too slow, take too much storage, or do not completely solve the design problem. Sometimes as you explore a potential solution you come up with another instance of the same design problem. In the special case pattern we solve this second instance with one of previously rejected solutions. Thus we solve the problem twice, one for the special case and once for the general case.

This technique depends on having one or more previously examined solutions and these are generated using the first two patterns.

If we look at some of the known uses of the previous patterns, several of them could be considered as a special case solution.

- Inverted page tables are a special case of page table lookup where we can depend on the TLB (the page table cache) to have a high hit rate.
- The “use existing functions” case of *Three levels of implementation* often is for the special case where we are not allowed to add new primitives to the system.
- Hooks are a special case solution to the polling problem where there is some active entity willing to accept our registration and inform us when the event occurs.
- Optimistic algorithms are the special case where we can guess with high accuracy and where we can reverse actions when we guess wrongly.

Related Design Techniques

Two-level design is sometimes an example of reducing a problem to a special case. The lower level implements the special case and the upper level implements the general problem. A *little language* is a specific example of this.

Conclusions

We have characterized the design process as the investigation of a series of potential solutions. Each solution is analyzed to some extent to see how good it is. If it is not completely acceptable then we move on to another solution. This paper gives three patterns that are useful for exploring design alternatives during this process.

The first pattern, *Three levels of implementation*, gives three varieties of solution in almost any design context. If you have one solution this pattern will suggest two other potential solutions to try. The first variety of solution is to solve the problem using existing primitives in the system. The second variety is to add a new low-level primitive that provides a building block to help you build the solution. The third variety is the add a new primitive that exactly solves the problem. The second pattern, *Opposite point of view*, directs you to examine your assumptions about the solution and think about doing some things in the opposite way. Both of these patterns give you guidance in thinking of a new solution based on the current one.

The third pattern, *Special case*, is used in the design process when you encounter a special case of the problem while exploring a solution to the general problem. This pattern allows you to use one of your previously rejected solutions to solve the special case and hence solve the general problem.

References

Booch, G. *Object Oriented Design with Applications*. Benjamin/Cummings, Redwood City CA, 1992.

Booch, G. *Software Components with Ada*. Benjamin/Cummings, Redwood City CA, 1987.

Crowley, C. *Operating Systems: A Design-Oriented Approach*. Irwin, Chicago, 1997.

Draves, R., B. Bershad, R. Rashid, and R. Dean. Using continuations to implement thread management and communication in operating systems. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pp. 122-36, ACM SigOps, October 1991.

Gamma, E., R. Helm, R. Johnson, and J. Vlissides *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading MA, 1995.

Jaeger, S. Mega-widgets in Tcl/Tk: Evaluation and Analysis. In Proceedings of the Tcl/Tk Workshop, pp. 43-52, Toronto, Canada, July, 1995.

Kamin, S. Programming Languages, Addison-Wesley, Reading MA, 1990.

Lieberman, H, and C. Fry. Bridging the Gulf Between Code and Behavior in programming. In *CHI'95 Conference Proceedings*, Denver CO, May 1995.

Marsh, B., M. Scott, T LeBlanc, and E. Markatos. First-class user-level threads. In *Proceedings of the 13th ACM Symposium on Operating Systems Principles*, pp. 110-21, ACM SigOps, October 1991.

Massalin, H. and C. Pu Threads and input/output in the synthesis kernel. In *Proceedings of the 12th ACM Symposium on Operating Systems Principles*, pp. 191-201, Litchfield Park, AZ, USA, 1989.

Stroustrup, B. *The C++ Programming Language. Third edition*. Addison-Wesley, Reading MA, 1997.

Tanenbaum, A. *Modern Operating Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1992.