Software Performance AntiPatterns

Connie U. Smith
Performance Engineering Services
PO Box 2640
Santa Fe, New Mexico, 87504-2640
(505) 988-3811
http://www.perfeng.com/

Lloyd G. Williams
Software Engineering Research
264 Ridgeview Lane
Boulder, Colorado 80302
(303) 938-9847
boulderlgw@aol.com

ABSTRACT

A pattern is a common solution to a problem that occurs in many different contexts. Patterns capture expert knowledge about "best practices" in software design in a form that allows that knowledge to be reused and applied in the design of many different types of software. Antipatterns are conceptually similar to patterns in that they document recurring solutions to common design problems. They are known as *anti*patterns because their use (or misuse) produces negative consequences. Antipatterns document common mistakes made during software development as well as their solutions.

While both patterns and antipatterns can be found in the literature, they typically do not explicitly consider performance consequences. This paper explores antipatterns from a performance perspective. We discuss performance problems associated with one well-known design antipattern and show how to solve them. We also propose three new performance antipatterns that often occur in software systems.

1.0 INTRODUCTION

A pattern is a common solution to a problem that occurs in many different contexts [5]. It provides a general solution that may be specialized for a given context. Patterns capture expert knowledge about "best practices" in software design in a form that allows that knowledge to be reused and applied in the design of many different types of software.

Patterns address the problem of "reinventing the wheel." Over the years, software developers have solved essentially the same problem, albeit in different contexts, over and over again. Some of these solutions have stood the test of time while others have not. Patterns capture these proven solutions and package them in a way that allows software designers to look-up and reuse the solution in much the same fashion as engineers in other fields use design handbooks.

The use of patterns in software development has its roots in the work of Christopher Alexander, an architect. Alexander developed a pattern language for planning towns and designing the buildings within them [1]. A pattern language is a collection of patterns that may be combined to solve a range of problems within a given application domain, such as architecture or software development. Alexander's work codified much of what was, until then, implicit in the field of architecture and required years of experience to learn.

In addition to capturing design expertise and providing solutions to common design problems, patterns are valuable because they identify abstractions that are at a higher level than individual classes and objects. Now, instead of discussing software construction in terms of building blocks such as lines of code, or individual objects, we can talk about structuring software using patterns. For example, when we discuss using the Proxy pattern [5] to solve a problem, we are describing a building block that includes several classes as well as the interactions among them.

Patterns have been described for several different categories of software development problems and solutions, including software architecture, design, and the software development process itself.

Recently, software practitioners have also begun to document *antipatterns*. Antipatterns [2] are conceptually similar to patterns in that they document recurring solutions to common design problems. They are known as *anti*patterns because their use (or misuse) produces negative consequences. Antipatterns document common mistakes made during software development as well as their solutions. Thus, antipatterns tell you what to avoid and how to fix the problem when you find it.

Antipatterns are refactored (restructured or reorganized) to to overcome their negative consequences. A *refactoring* is a correctness-preserving transformation that improves the quality of the software. For example a set of classes might be refactored to improve reusability by moving common properties to an abstract superclass. The transformation does not alter the semantics of the application but it may improve overall reusability. Refactoring may be used to enhance many different quality attributes of software, including: reusability, maintainability, and, of course, performance. Refactoring is discussed in detail in [4].

Antipatterns address software architecture and design as well as the software development process itself. Our experience is that developers find antipatterns useful because they make it possible to iden-

Appears in Proceedings 2nd International Workshop on Software and Performance, Sept. 2000

tify a bad situation *and* provide a way to rectify the problem. This is particularly true for performance because good performance is the *absence* of problems. Thus, by illustrating performance problems and their causes, performance antipatterns help build performance intuition in developers. Patterns, which do not contain performance problems, may be less useful for building performance intuition, especially if their performance characteristics are not discussed (as is typically the case).

While both patterns and antipatterns can be found in the literature, they typically do not explicitly consider performance consequences. It is important to document both design patterns that lead to systems with good performance and to point out common performance mistakes and how to avoid them. This is a supplement to software performance engineering that will improve the architectures and designs of software developers.

This paper explores antipatterns from a performance perspective. We discuss performance problems associated with one well-known design antipattern and show how to solve them. We also propose three new performance antipatterns that often occur in software systems.

While their emphasis is different, both patterns and antipatterns address common software problems and their solutions. The emphasis in the patterns community, however, is on quality attributes, such as reusability or maintainability, other than performance. As the use of patterns and antipatterns becomes more widespread, it is vital to also identify those that are likely to have good performance characteristics. We propose antipatterns for performance problems that we encounter in many different contexts but have the same underlying pathology. Because we find them so often, it is important to document these antipatterns so developers will be able to recognize them before they occur, and select appropriate alternatives.

2.0 RELATED WORK

Antipatterns are derived from work on patterns. As noted in the introduction, this work is aimed at capturing expert software design knowledge. There is a large body of published work on patterns including [5], [3], and the proceedings of the Pattern Languages of Program Design (PLoP) conferences. While there is occasional mention of performance considerations in the work on patterns, the principal focus is on other quality attributes, such as modifiability and maintainability.

Meszaros [6] presents a set of patterns that address capacity and reliability in reactive systems such as telephony switches. Petriu and Somadder [7] extend these patterns for use in identifying and correcting performance problems in distributed layered client-server systems with multi-threaded servers.

Smith [10] presents a set of principles for constructing responsive software systems. While they were published before the work on software patterns began and presented with a different focus, they can be viewed as performance patterns.

Antipatterns extend the notion of patterns to capture common design errors and their solution. The most extensive work on this topic is by Brown, et. al. [2]. Their work, like the work on patterns however, focuses principally on quality attributes other than performance.

This paper extends the work on antipatterns to explicitly address the performance of software architectures and designs. It presents three common performance mistakes made in software architectures. The first antipattern, the “god” class, was proposed by other authors as a problem with software quality. We show how it also leads to poor performing software. We propose three additional antipatterns that also cause poor performance. They may also have other negative impacts on other quality attributes, but they are not addressed here. Additional performance antipatterns appear in a forthcoming book by the authors.

Each of the antipatterns is defined in the following sections using this standard template:

- Name: the section title
- Problem: What is the recurrent situation that causes negative consequences?
- Solution: How do we avoid, minimize or refactor the antipattern?

3.0 THE “GOD” CLASS

This antipattern is known by various names, including the “god” class [8] and the “blob” [2]. Both Reil and Brown, et. al. discuss the impact of this phenomenon on quality attributes such as modifiability and maintainability. The presence of a “god” class in a design also has a negative impact on performance.

3.1 Problem

A “god” class is one that performs most of the work of the system, relegating other classes to minor, supporting roles. A design containing a “god” class is usually easy to recognize. It typically has a single, complex controller class (often with a name containing `Controller` or `Manager`) that is surrounded by simple classes that serve only as data containers. These classes typically contain only accessor operations (operations to `get()` and `set()` the data) and perform little or no computation of their own. The “god” class obtains the information it needs using the `get()` operations belonging to these data classes, performs some computation, and then updates the data using their `set()` operations.

The following (very) simplified example illustrates the effects of a “god” class. Consider an industrial process control application in which it is necessary to control the status of a valve (open or closed). Figure 1 shows a fragment of the class diagram for a possible design for this application. The `Controller` class in Figure 1 behaves like a “god” class. The `Valve` class has no intelligence. It simply reports its `status` (`open` or `closed`) and responds to `open()` and `close()` operation invocations. The `Controller` does all of the work; it requests information from the `Valve`, makes decisions, and tells the `Valve` what to do.

The `Controller` class is tightly coupled to the `Valve` class and requires extra messages to perform an operation, as shown by the following code fragment.

Appears in Proceedings 2nd International Workshop on Software and Performance, Sept. 2000

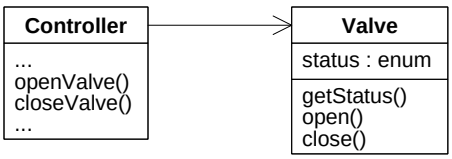

Figure 1: A "god" Class

```
void openValve() {
   status currentStatus;

   currentStatus = theValve->getStatus();
   if (currentStatus != open)
      theValve->open();
}
```

To open a valve, the controller must first request the valve's status, then check to see that it is not open and, finally, tell the valve to open. This operation requires two messages to open the valve, one to get the `status` and one to invoke the `open()` operation. Moreover, if the definition of `status` is changed in `Valve`, a corresponding change must be made in all of the applicable operations in `Controller`.

The `open()` and `close()` operations in `Valve` simply set the appropriate value of `status`.

The solution can be refactored to reduce both the coupling between `Controller` and `Valve` and the number of messages required to perform an operation by moving the status check to the `Valve` class. Figure 2 shows the refactored class diagram.

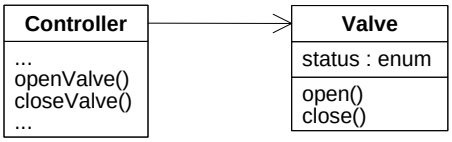

Figure 2: Refactored Solution

Now, the status check is in the `Valve` class (close to the data needed to perform the check). The `openValve()` operation in `Controller` is simply:

```
void openValve() {
   theValve->open();
}
```

and the `open()` operation in `Valve` becomes:

```
void open() {
   if (status != open)
      status = open;
}
```

There is also a variant of the "god" class that, rather than performing all of the work, contains all of the system's data [8]. The functions are then assigned to other classes. When one of the function classes needs data to perform an operation, it obtains it from the "god" class via a `get()` operation and, if data needs to be updated, the function class updates it using a `set()` operation. Even though the data is encapsulated by accessor functions, the data form of the "god" class is the moral equivalent of global data or the common block in FORTRAN.

Both forms of the "god" class are the result of poorly distributed system intelligence. A good rule of thumb when designing object-oriented systems is to keep related data and behavior in the same place. Both types of "god" class violate this heuristic by assigning behavior to one class and the data needed to provide that behavior to another.

A "god" class may creep into a design in several different ways. Behavioral "god" classes are often the result of a procedural design that masquerades as an object-oriented one. "God" classes are also often introduced while upgrading a legacy system to an object-oriented design. A behavioral "god" class may be created when developers attempt to capture the central control mechanism in the original, procedural design. On the other hand, if the original system contained a large, global data structure, it is likely to appear as a data "god" class in the new design.

From a performance perspective, a "god" class creates problems by causing excessive message traffic. In the behavioral form of the problem, the excessive traffic occurs as the "god" class requests and updates the data it needs to control the system from subordinate classes. In the data form, the problem is reversed as subordinates request and update data in the "god" class. In both cases, the number of messages required to perform a function is larger than it would be in a design that assigned related data and behavior to the same class.

The effect of a "god" class on message traffic is shown clearly in a case study presented by Sharble and Cohen [9]. They present two designs for an "Object-Oriented Brewery." One (Design 1) was produced using a data-driven design technique and contains a behavioral "god" class. The other (Design 2) was produced using a responsibility-driven technique and corresponds to an appropriately refactored version of Design 1. The difference in the number of messages required by each design for various scenarios is shown in Figure 3.

Sharble and Cohen were concerned with design metrics, such as coupling and cohesion, and not performance. They used message counts as a complexity measure. Viewed from a performance perspective, however, their data dramatically illustrates the performance impact of a "god" class..

As Figure 3 shows, the number of messages required to accomplish a function is greater in every case for the design containing the "god" class – sometimes by a factor of two or more. This excessive message traffic can degrade performance and is especially problematic in distributed systems where an object of the "god" class executes on a different node than objects of its subordinate classes.

3.2 Solution

The solution to the "god" class problem is to refactor the design to distribute intelligence uniformly across the top-level classes in the

Appears in Proceedings 2nd International Workshop on Software and Performance, Sept. 2000

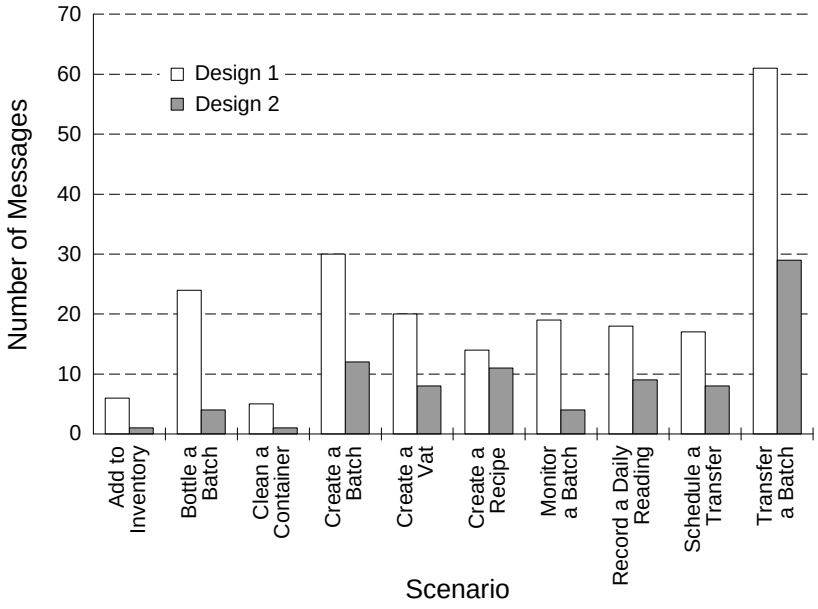

Figure 3: Message Counts for Scenarios (from [9])

application. It is important to keep related data and behavior together. An object should have most of the data that it needs to make a decision. Beware of either: 1) an object that must request lots of data from other objects and then update their states with the results, or 2) a group of objects that must access a common object to get and update the data that they deal with.

This solution to the "god" class problem embodies the locality principle [Smith, 1988; Smith, 1990] because an algorithm and the data that it requires are localized in the same object.

The performance gain for the refactored solution will be:

$$T_s = M_s \times O$$

where T_s is the processing time saved, M_s is the number of messages saved and O is the overhead per message. The amount of overhead for a message will depend on the type of call, for example a local call will have less overhead than a remote procedure call.

4.0 EXCESSIVE DYNAMIC ALLOCATION

With dynamic allocation, objects are created when they are first accessed (a sort of "just-in-time" approach) and then destroyed when they are no longer needed. This can often be a good approach to structuring a system, providing flexibility in highly dynamic situations. For example, in a graphics editor, creating an instance of a shape (such as a circle or rectangle) when it is drawn and destroying the instance when the shape is deleted may be a very useful approach. Excessive Dynamic Allocation, however, addresses frequent, unnecessary creation and destruction of objects of the same class.

4.1 Problem

Dynamic allocation is expensive. Reil [8] describes an object-oriented approach to designing a gas station in which, when your car needs gasoline, you pull over to the side of the road, buy a piece of land, build a gas station (which, in turn builds pumps, and so on), and fill the tank. When you're done, you destroy the gas station and return the land to its original state. Clearly, this approach only works for the wealthy (and patient!). You certainly do not want to use this approach if you need gas frequently.

The situation is similar in object-oriented software systems. When an object is created, the memory to contain it (and any objects that it contains) must be allocated from the heap and any initialization code for the object and the contained objects must be executed. When the object is no longer needed, necessary clean-up must be performed and the reclaimed memory must be returned to the heap to avoid "memory leaks." While the overhead for creating and destroying a single object may be small, when a large number of objects are frequently created and then destroyed, the performance impact may be significant.

The sequence diagram in Figure 4 illustrates Excessive Dynamic Allocation. This example is drawn from a call-processing application in which, when a customer lifts the telephone handset (an `offHook` event), the switch creates a `call` object to manage the call. When the call is completed (an `onHook` event), the `call` object is destroyed. (Details of the call processing are in the sequence diagram referenced by `handleCall`, which is not shown here.)

While constructing a single `Call` object may not seem excessive, a `Call` is a complex object that contains several other objects that also must be created. In addition, a switch can receive hundreds of thousands of `offHook` events each hour. In a case like this, the

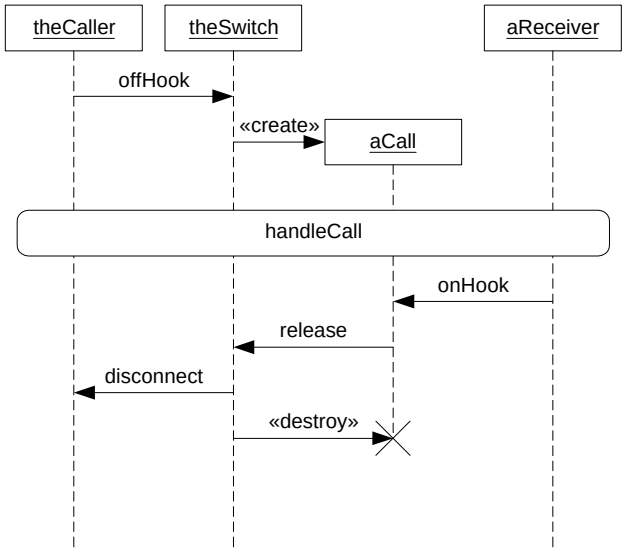

Figure 4: Excessive Dynamic Allocation

overhead for dynamically allocating call objects adds substantial delays to the time needed to complete a call.

The cost of dynamic allocation, *C*, is:

$$C = N \cdot \sum_{depth} (s_c + s_d)$$

where *N* is the number of calls, *depth* is the number of contained objects that must be created when the class is created, s_c and s_d are the service time to create the object and to destroy the object respectively, and *S* is $s_c + s_d$.

Figure 5 shows the cost of Excessive Dynamic Allocation for some typical values of depth and S, the sum of the creation and destruction time. The figure shows how the overhead for dynamic allocation increases as the number of calls increases. Note that the graph shows the total service time for dynamic allocation regardless of the number of processes handling these calls. Calls are multi-processed so the response time depends on the number of processes and on contention delays among them. Reducing the service time, however, also reduces the response time.

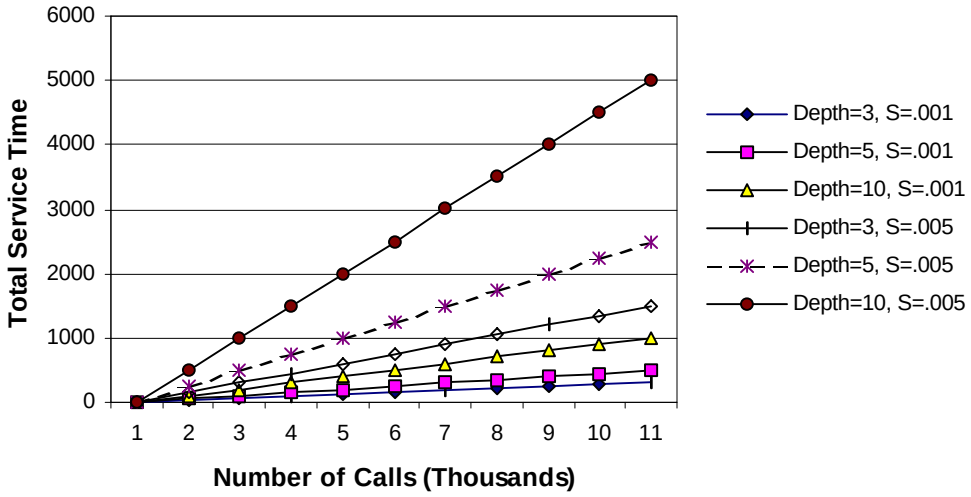

Figure 5 Cost of Excessive Dynamic Allocation

4.2 Solution

There are two possible solutions to problems introduced by Excessive Dynamic Allocation.

The first is to “recycle” objects rather than create new ones each time they are needed. This approach pre-allocates a “pool” of objects and stores them in a collection. New instances of the object are requested from the pool and unneeded instances are returned to it. This approach is useful for systems that continually need many short-lived objects (like the call processing application). You pay for pre-allocating the objects at system initialization but reduce the run-time overhead to simply passing a pointer to the pre-allocated object. This is an application of the processing versus frequency principle – we minimize the product of the amount of processing times the frequency that it is performed [Smith, 1988; Smith, 1990]. Returning unused objects to the pool eliminates garbage collection overhead and possible memory leaks.

The second approach uses sharing to eliminate the need to create new objects. An example of this is the use of the Flyweight pattern [5] to allow all clients to share a single instance of the object. An example of this application of the Flyweight pattern to alleviate Excessive Dynamic Allocation in the ICAD example (Section 5) appears in [12].

Appears in Proceedings 2nd International Workshop on Software and Performance, Sept. 2000

The first improvement approach affects the cost in Figure 5 by reducing the service time, S, to the time to allocate/return an object from the pool, and changing the depth to 1 because the pre-allocated objects already have created the subordinate objects. The improvement for the second approach is similar.

5.0 CIRCUITOUS TREASURE HUNT

Do you remember the child's treasure hunt game that starts with a clue which leads to a location where the next clue is hidden, and so on until the "treasure" is finally located? The antipattern analogy is typically found in database appl.ications. Software retrieves data from a first table, uses those results to search a second table, retrieves data from that table, and so on until the "ultimate results" are obtained.

5.1 Problem

The impact on performance is the large amount of database processing required each time the "ultimate results" are needed. It is especially problematic when the data is on a remote server and each access requires transmitting all the intermediate queries and their results via a network and perhaps through other servers in a multi-tier environment.

The computer-aided design case study originally described in [Williams, 1998] illustrates this antipattern. The ICAD application allows engineers to construct and view drawings that model structures, such as aircraft wings. A model is stored in a relational database and several versions of the model may exist within the database.

Figure 6 shows a portion of the ICAD class diagram with the relevant classes. A model consists of elements which may be: beams, which connect two nodes; triangles, which connect three nodes; or plates, which connect four or more nodes. A node is defined by its position in three-dimensional space (x, y, z). Additional data is associated with each type of element to allow solution of the engineer's model.

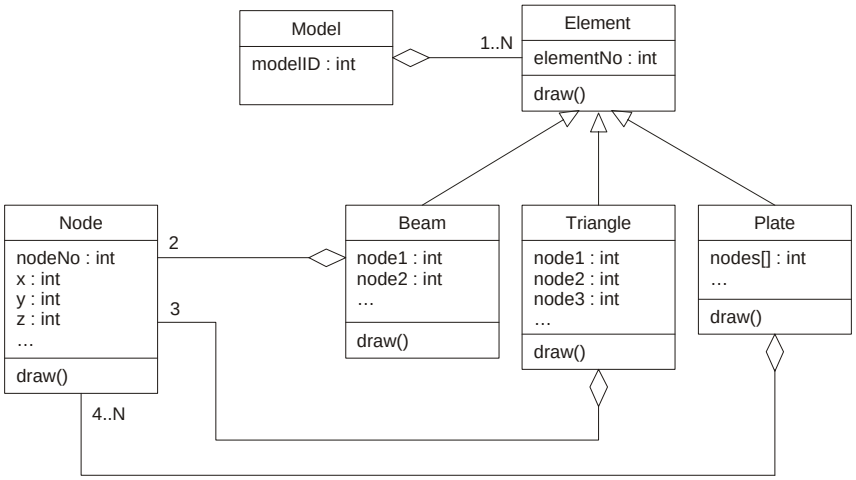

Figure 6: ICAD Classes and Associations

This example focuses on the DrawMod scenario in which a model is retrieved from the database and drawn on the screen. Figure 7 shows a sequence diagram for this scenario. A typical model consists of 2000 beams and 1500 nodes (a single node may be connected to up to 4 beams). The software first finds the model id, then uses it to find the beams, and repeats the sequence of steps to: retrieve each beam row, use the node number from the beam row to find then retrieve the node row which contains the "ultimate results" – the node coordinates. This information is then used to draw the model. For a typical DrawMod scenario there are 6001 database calls: 1 for the model, 2000 for beams, and 4000 for the nodes.

A large number of database calls causes the most serious performance problems in systems with remote database accesses; because of the cost of the remote access, the processing of the query, and the network transfer of all the intermediate results.

Another instance of the antipattern is also found in object-oriented systems where operations have large "response sets." In this case, one object invokes an operation in another object, that object then invokes an operation in another object, and so on until the "ultimate result" is achieved. Then each operation returns, one by one, to the object that made the original call.

The performance impact is the extra processing required to identify the final operation to be called and invoking it, especially in distributed object systems where objects may reside in other processes and on other processors. When the invocation causes the intermediate objects to be created and destroyed the performance impact is even greater. This behavior also has poor memory locality because each context switch may cause the working set of the called object to be loaded. The working sets of intermediate objects may need to be re-loaded later when the return executes.

The class diagram in Figure 6 shows a simple example. Suppose that the model data has been retrieved from the database and is now

Appears in Proceedings 2nd International Workshop on Software and Performance, Sept. 2000

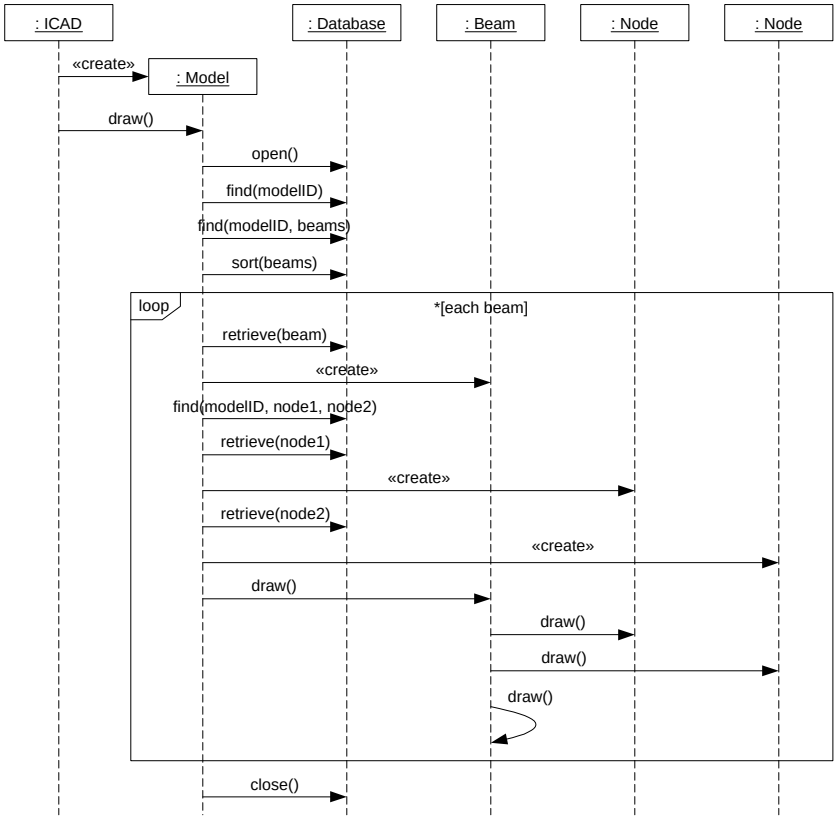

Figure 7: `DrawMod` Example of Circuitous Treasure Hunt

contained within each object. Then the model object must determine (from the association to `Beam`, probably a table of pointers) each Beam object to call, and each Beam must determine (from the association to Nodes) which Node objects to call. The Model calls the first Beam operation, then the Beam calls 2 Node operations, and so on.

5.2 Solution

If you find the database access problem early in development, you may have the option of selecting a different data organization. For example, the DrawMod database could store the node coordinates (x, y, z) in the beam table. The sequence diagram for the alternative database design is in Figure 8. With the node coordinates in the beam row, the database call to find and retrieve nodes is unnecessary and is omitted. For a typical DrawMod scenario with 2000 beams, there will be 4000 fewer database calls.

In general the number of calls saved will be:

$$c_s = \prod_{j \in rootpath} a_j$$

where c_s is the total number of calls saved, a_j is number of associated objects in the level below for each object in this level, for every object *j* between the object originally containing the "ultimate result" (the *leaf* class), and the object containing the "first clue" (the *root* class). For example, for the leaf class (`node`) a_j is 2 nodes per beam, and for the intermediate class (`beam`) a_j is 2000 beams per model, so c_s is 4000.

There are some potential disadvantages to reorganizing the `DrawMod` data in this way. Optimizing the data organization for one scenario may de-optimize it for other scenarios. To determine the appropriateness of each alternative, the performance engineer will need to analyze the performance impact on other scenarios that use the database. It is unwise to optimize the database organization for a single scenario if it has a detrimental affect on all other scenarios; you want the "globally optimal" solution for the key performance scenarios. You evaluate the overall performance by revising each scenario that is affected by the change and comparing the model solutions. Details are beyond the scope of this discussion.

For distributed systems, if you cannot change the database organization, you can reduce the number of remote database calls by using the Adapter pattern [5] to provide a more reasonable interface for remote calls. The Adapter would then make all the other (local) database calls required to retrieve the "ultimate result" return only those results to the remote caller. This reduces the number of remote calls and the amount of data transferred, but does not reduce the database processing.

For designs with large response sets, an alternative is to create a new association that leads directly to the "ultimate result." For example, in Figure 6 we would add an association between the Model class and the Node class. In the DrawMod scenario, this would reduce the number of operations called from 6000 (2000

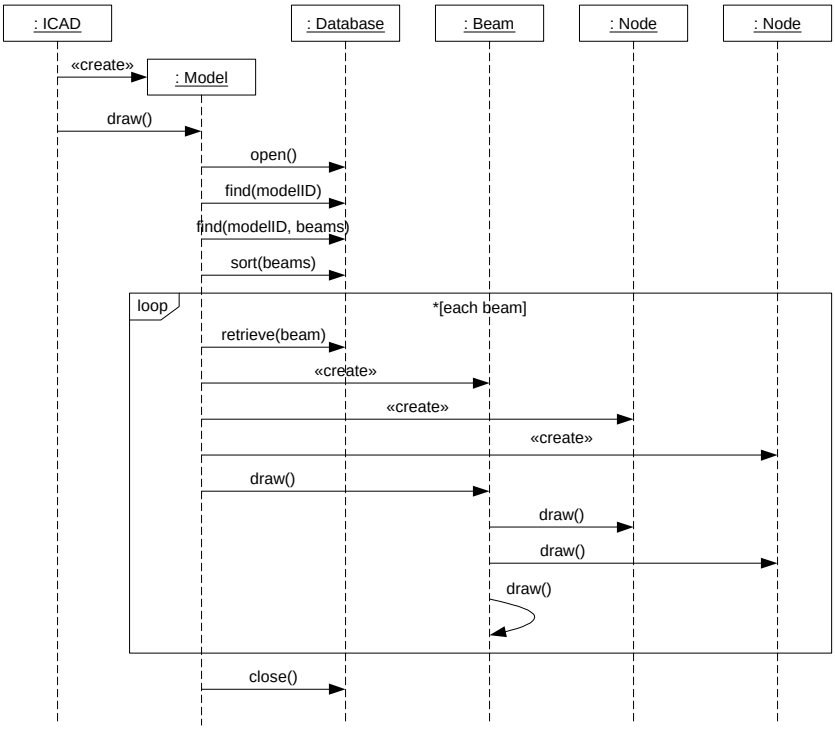

Figure 8: Refactored `DrawMod` Scenario

Beam calls plus 4000 Node calls) to 1500 (the number of Nodes per Model). The performance impact is substantial if these are remote calls that are made via middleware such as Corba or DCOM.

6.0 THE ONE LANE BRIDGE

On the south island of New Zealand there is a highway that has a one lane bridge that is even shared with a train. It isn't a problem there because there is light traffic in that part of the country. It would be a problem though if it were in downtown Los Angeles.

6.1 Problem

The problem with a One Lane Bridge is that traffic may only travel in one direction at a time, and if there are multiple lanes of traffic all moving in parallel, they must merge and proceed across the bridge, one vehicle at a time. This increases the time required to get a given number of vehicles across the bridge and can also cause long back-ups.

The software analogy to the One Lane Bridge is a point in the execution where one, or only a few, processes may continue to execute concurrently. All other processes must wait. It frequently occurs in applications that access a database. Here, a lock ensures that only one process may update the associated portion of the database at a time. It may also occur when a set of processes make a synchronous call to another process that is not multi-threaded; all of the processes making synchronous calls must take turns "crossing the bridge."

The sequence diagram in Figure 9 illustrates the database variant of the One Lane Bridge antipattern. Each order requires a database update for each item ordered. The structure selected for the database assigns a new order-item number to each item and inserts all items at the end of the table. If every new update must go to the same physical location, and all new items are "inserted," then the update behaves like a One Lane Bridge because only one insert may proceed at a time; all others must wait. There is also a second problem in that these inserts are costly because they must update a database index for each key on the table.

Similar problems occur when the database key is a date/time stamp for an entity, or any key that increases monotonically. We have also seen this problem for periodic archives where processing must halt while state information is transferred to long term storage.

6.2 Solution

With vehicular traffic, you alleviate the congestion caused by a One Lane Bridge by constructing multiple lanes, constructing additional bridges (or other alternatives), or re-routing traffic.

The analogous solutions in the database update example above would be:

- use an algorithm for assigning new database keys that results in a "random" location for inserts,
- use multiple tables that may be consolidated later, or
- use another alternative such as pre-loading "empty" database rows and selecting a location to update that minimizes conflicts.

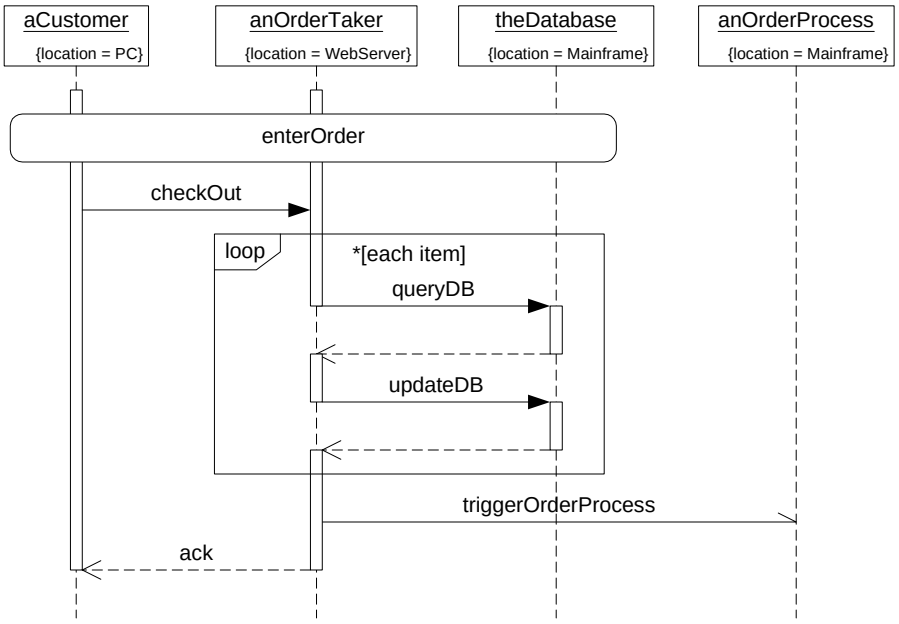

Figure 9: Database Contention Example

For example, an alternative for assigning date-time keys is to use multiple "buckets" for inserts and use a hashing algorithm to assign new inserts to the "buckets."

Reducing the amount of time required to cross the bridge also helps relieve congestion. One way to do this is to find alternatives for performing the update. For example, if the update must change multiple tables, it would be better to select a different data organization in which the update could be processed in a single table.

For the database example above, the magnitude of the improvement depends on the intensity of new item orders and the service time for performing updates. The relationship is:

$$RT = \frac{S}{1 - XS}$$

where *RT* is the residence time (elapsed time for performing the update), *S* is the service time for performing updates, and *X* is the arrival rate.

Figure 10 shows a comparison of the residence time for various arrival rates for two different service times. The first curve assumes the service time for the update is 10 milliseconds (thus the arrival rate of update requests must be less than 100 requests per second), and shows how the residence time increases as the arrival rate approaches the maximum. The second curve shows the improvement if the update service time is reduced by 1 millisecond! The figure illustrates the improvement achievable by reducing the service time (the time required to cross the bridge). If you change the structure of the database so that you update in multiple locations so fewer processes wait for each update, this is equivalent to reducing the arrival rate and the figure also shows the relative benefit of this alternative.

Figure 10 also illustrates the relative importance of the One Lane Bridge antipattern: if the intensity of requests for the service is high, it may be a significant barrier to achieving responsiveness and scalability requirements.

This solution to the One Lane Bridge problem embodies the shared resources principle [Smith, 1988; Smith, 1990] because responsiveness improves when we minimize the scheduling time plus the holding time. Holding time is reduced by reducing the service time for the One Lane Bridge and by re-routing the work.

7.0 SUMMARY AND CONCLUSIONS

This paper has explored antipatterns from a performance perspective. We show the performance consequences of a recognized antipattern, introduce three new antipatterns with negative performance consequences, and quantify their impact on performance.

The value of both antipatterns and their predecessor, patterns, is that they capture expert software design knowledge. This value has been amply demonstrated by their acceptance within the development community. One serious shortcoming of both patterns and antipatterns has been their lack of focus on performance issues. While some authors focused on performance [Meszaros, 1996; Petriu and Somadder, 1997], most have considered it as an afterthought, if at all. Demonstrating the performance characteristics of patterns and antipatterns is vital so that developers using them in designing software can select alternatives that will meet their performance goals.

The work presented here goes beyond merely describing the characteristics of architectural or design antipatterns, however. The Excessive Dynamic Allocation, Circuitous Treasure Hunt, and One Lane Bridge antipatterns document common performance mistakes and provide solutions for them. While these antipatterns may mani-

Appears in Proceedings 2nd International Workshop on Software and Performance, Sept. 2000

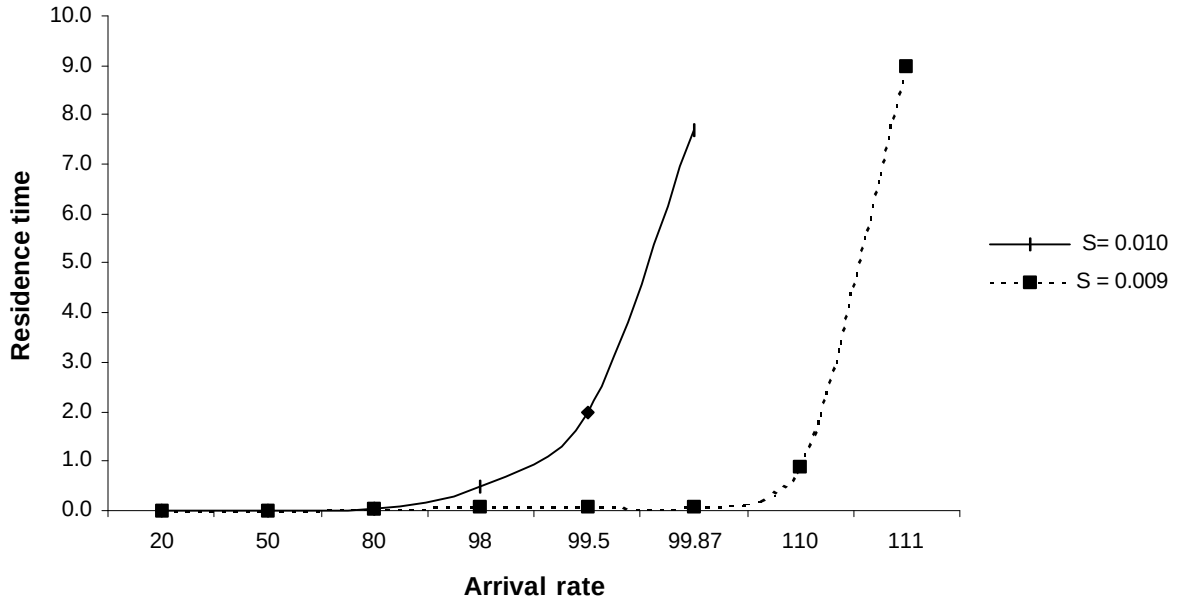

Figure 10: Performance Impact of the One Lane Bridge

fest themselves in a variety of ways (for example the One Lane Bridge problem may be caused by either database collisions or synchronization delays) the manifestations have a common underlying cause.

The solutions to these antipatterns embody sound, well-accepted performance principles [Smith, 1988; Smith, 1990]. These principles are similar to patterns in that they provide guidelines for creating responsive software. The antipatterns presented here provide a complement to the performance principles by illustrating what not to do and how to fix a problem when you find it. A simple analogy from electrical engineering would be using examples of series and parallel circuits (i.e., patterns) to illustrate how to build proper circuits and an example of a short circuit (i.e., an antipattern) to show what to avoid. Feedback from students in our classes indicates that both types of example are needed to instill performance intuition.

More work is needed on both patterns and antipatterns that includes their impact on performance as well as other quality attributes. We are continuing to identify other performance-related patterns and antipatterns. Many of these are described in a forthcoming book.

8.0 REFERENCES

[1] C. Alexander, S. Ishikawa, M. Silverstein, M. Jacobson, I. Fiksdahl-King, and S. Angel, *A Pattern Language*, New York, NY, Oxford University Press, 1977.

[2] W. J. Brown, R. C. Malveau, H. W. McCormick III, and T. J. Mowbray, *AntiPatterns: Refactoring Software, Architectures, and Projects in Crisis*, New York, John Wiley and Sons, Inc., 1998.

[3] F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, and M. Stal, *Pattern-Oriented Software Architecture: A System of Patterns*, Chichester, England, John Wiley and Sons, 1996.

[4] M. Fowler, *Refactoring: Improving the Design of Existing Code*, Reading, MA, Addison-Wesley Longman, 1999.

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

[6] G. Meszaros, "A Pattern Language for Improving the Capacity of Reactive Systems," in *Pattern Languages of Program Design 2*, J. M. Vlissides, J. O. Coplein and N. L. Kerth, ed., Reading, MA, Addison-Wesley, 1996, pp. 575-591.

[7] D. Petriu and G. Somadder, "A Pattern Language For Improving the Capacity of Layered Client/Server Systems with Multi-Threaded Servers," *Proceedings of EuroPLoP'97*, Kloster Irsee, Germany, July, 1997.

[8] A. J. Riel, *Object-Oriented Design Heuristics*, Reading, MA, Addison-Wesley, 1996.

[9] R. C. Sharble and S. S. Cohen, "The Object-Oriented Brewery: A Comparison of Two Object-Oriented Development Methods," *Software Engineering Notes*, vol. 18, no. 2, pp. 60-73, 1993.

[10] C. U. Smith, "Applying Synthesis Principles to Create Responsive Software Systems," *IEEE Transactions on Software Engineering*, vol. 14, no. 10, pp. 1394-1408, 1988.

[11] C. U. Smith, *Performance Engineering of Software Systems*, Reading, MA, Addison-Wesley, 1990.

[12] L. G. Williams, C. U. Smith, “Performance Engineering of Software Architectures,” *Proceedings Workshop on Software and Performance*, Santa Fe, NM, Oct. 1998.