More New Software Performance Antipatterns: Even More Ways to Shoot Yourself in the Foot

Performance antipatterns document common software performance problems as well as their solutions. These problems are often introduced during the architectural or design phases of software development, but not detected until later in testing or deployment. Solutions usually require software changes as opposed to system tuning changes. This paper presents three new performance antipatterns and gives examples to illustrate them. These antipatterns will help developers and performance engineers avoid common performance problems.

1.0 Introduction

Experienced performance specialists have seen the same performance problems time after time. Each time the problem is corrected but, when the next system is implemented, the same (or a similar) set of performance problems is found again.

This phenomenon is similar to one observed in the software engineering community: certain implementations which have negative consequences are found repeatedly as solutions to recurring design problems. For example, when trying to solve the problem of coordinating different subsystems within a single system, developers often implement a stovepipe architecture. This solution makes the system difficult to change and expensive to maintain.

Brown and co-authors [Brown, et al. 1998] introduced software antipatterns to document common software development mistakes and their solutions. These antipatterns describe what to avoid and how to fix the problem if you encounter it. The antipatterns presented by Brown and co-workers address both software architecture and design issues but do not specifically address performance.

Antipatterns are *refactored* (restructured or reorganized) to overcome their negative consequences. A *refactoring* is a correctness-preserving transformation that improves the quality of the software. For example, a data structure may be revised to improve the efficiency of the retrieval processing. The transformation does not alter the semantics of the application but it improves performance. Refactoring may also be used to enhance many other quality attributes such as reusability and maintainability in addition to performance. Refactoring is discussed in detail in [Fowler 1999].

Software antipatterns are an extension of the notion of software architectural and design *patterns*. Patterns document common solutions to frequently-occurring software development problems [Schmidt, et al. 2000], [Buschmann, et al. 1996], [Gamma, et al. 1995]. Architectural and design patterns capture expert knowledge about “best practices” in software design by documenting general solutions that may be customized for a particular context. They make it possible to reuse that knowledge in the development of many different types of software. These patterns focus on quality attributes such as reusability or modifiability but do not specifically address performance.

Recently, we introduced software *performance* patterns and antipatterns [Smith and Williams 2002]. Performance patterns describe “best practices” for developing responsive, scalable software. We extend the notion of patterns to explicitly include performance considerations.

Performance antipatterns document common performance mistakes found in software architectures and designs. The presence of these antipatterns may also have negative impacts on other quality attributes but they are not addressed here. Our experience is that developers find performance antipatterns to be especially useful because they illustrate how to identify a bad situation *and* provide a way to rectify the problem.

By illustrating common problems and their solutions, performance antipatterns help build performance intuition in developers.

In [Smith and Williams 2002], [Smith and Williams 2001], and [Smith and Williams 2000] we introduced five software performance antipatterns. Then, in [Smith and Williams 2002b], we documented four more. This paper is the next in the series. It describes three new software performance antipatterns that were proposed by CMG attendees and participants in SPE seminars. Each of the antipatterns is described 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?

This paper also includes a summary of currently known performance antipatterns as a reference.

2.0 Falling Dominoes

Remember the children's game where you line up a set of dominoes with just the right spacing then tip over the first domino and watch the rest fall, one after another? This antipattern is named after the children's game. It's fun as a game but, when it happens to your software, it can be disastrous.

2.1 Problem

Remember the AT&T telephone outage of January 1990? The problem occurred when a 4ESS switch in New York had a problem and initiated a recovery procedure [Neumann 1990]. During the recovery (a 4 to 6 second process), the switch could not accept calls, so it sent a "congestion" signal to all of the 4ESS switches to which it was linked indicating that it was not accepting new calls. That's when the problem began.

When the New York switch was ready to again accept new calls, it sent a call attempt message to another switch. That message caused the second switch, we'll call it B, to reset its internal logic to show that the New York switch was back in service. While switch B was resetting its logic, a second call attempt from the New York switch came in. The second message confused switch B's software causing it to initiate its own recovery procedure. Switch B then sent messages to its connected switches indicating that it wasn't accepting additional calls. Once switch B had reset, it send out new call requests and the problem repeated causing a chain reaction of switch failures.

If the switches hadn't received a second call attempt message while resetting (i.e., the messages had occurred farther apart), the problem would not have occurred. The problem was solved by reducing the message load on the network, allowing the switches to reset themselves. The source of the problem was ultimately traced to a misplaced "break" statement in the C code.

This is an example of the Falling Dominoes software performance antipattern. Falling Dominoes occurs when one failure causes performance failures in other components.

Bob Gallo reported an example of this antipattern in a broadcast component that received input then broadcast it to many other components. When a communication channel failure caused one of the receiving components to repeatedly request re-transmission, it slowed down the entire system. Another instance of the antipattern occurred when one receiver failed, and it caused a feeder process to quit sending to all receivers.

Ralph Gifford reported an instance of this antipattern in an application that created 5 connections per user through DB2 Connect. One problem caused all to fail.

These are not only performance problems, they are also reliability and fault tolerance problems.

2.2 Solution

The solution is to make sure that broken pieces are isolated until they are repaired. The broadcast component could monitor re-transmission requests and when a threshold is reached, stop sending to a receiver until it is repaired.

Feeder processes should not stop when one receiver fails. The failed process should be isolated until it is repaired.

For the DB2 connections, an alternative that shares connections should be sought first. Again, failures should be isolated.

In the AT&T example, since failure of one node causes failure of others, the rate at which nodes fail is dependent on the number of nodes that have failed as well as the number that are active (i.e., available to fail). If the rate of failure is first-order (linear) in both active and failed nodes, we can write

$$R_F = kAF$$

where:

R_F = the rate at which nodes fail
k = rate constant
A = number of active nodes
F = number of failed nodes

The rate constant, k, is an indicator of how rapidly failures propagate.

If x is the number of failed nodes at time t, $A = A_0 - x$ and $F = F_0 + x$. Then:

$$\frac{d}{dt}x = k(A_0 - x)(F_0 + x)$$

Integrating this gives the following formula for the number of nodes that have failed as a function of time:

$$x = F_0\left(\frac{e^{at} - 1}{1 + be^{at}}\right)$$

where:

$a = (A_0 + F_0)k$

$b = F_0/A_0$

Figure 1 graphs the number of nodes that have failed versus at for various relative values of k.

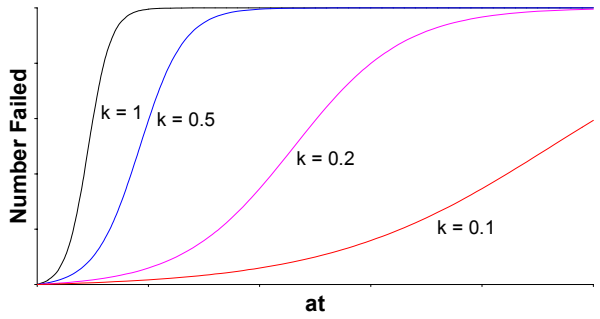

Figure 1: Percent of Nodes Failed

Initially, there are no failures. But, if a failure occurs for some external reason, the rate of failures is initially slow, then rapid as more and more failed nodes cause additional failures, then slow again as the number of active nodes becomes smaller and smaller.

The above analysis addresses the number of failures in cases such as the AT&T example. Figure 2 shows how the amount of useful work performed by the system is affected by the Falling Dominoes in the general case. In region 1, the system begins with normal processing (no failures), all of the processing is useful work. As failures occur, the system begins executing extra processing for errors (such as retransmitting messages). Because the system is less than 100% busy, it is possible to do error processing in addition to the useful work, so the total processing increases.

In region 2, error processing executes at the expense of useful work. Eventually the system reaches a state

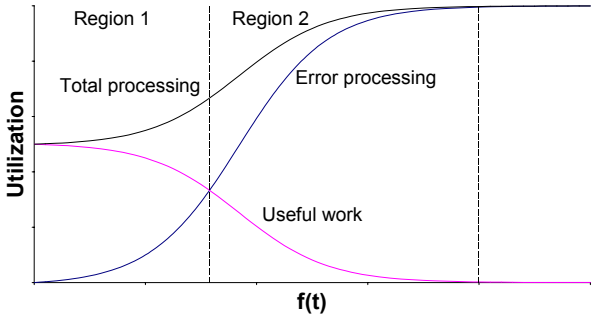

Figure 2: Effect of Error Processing on Useful Work

where error processing consumes the available resources and there is no useful work performed at all.

Thus, another solution to Falling Dominoes is to monitor the ratio of error processing to useful work, and, when an appropriate threshold has been reached, shut down failed components rather than continue to execute error processing.

3.0 Empty Semi Trucks

Imagine putting one bag of potato chips on a semi truck, sending it across the country to deliver the chips and back, then putting another bag of potato chips on it and repeating the process. Obviously this is an inefficient way of transporting goods.

3.1 Problem

This problem occurs in software systems where an excessive number of requests is required to perform a task, such as retrieval of information from a database. The problem may be due to inefficient use of available bandwidth, an inefficient interface, or both.

3.1.1 Inefficient Use of Bandwidth

This manifestation of the Empty Semi Trucks antipattern occurs in message-based systems when a very small amount of information is sent in each message. The amount of processing overhead is the same regardless of the size of the message. With smaller messages, this processing is required many more times than necessary.

Todd Merrill reported one example of sending a large spreadsheet of data with one cell in each packet. In another example offerred by Bart Domzy, the colors for a web page were defined in such a way that each one resulted in a separate download. To make matters worse, it was a web page with static content—the definitions were for future flexibility, but caused a very high performance penalty.

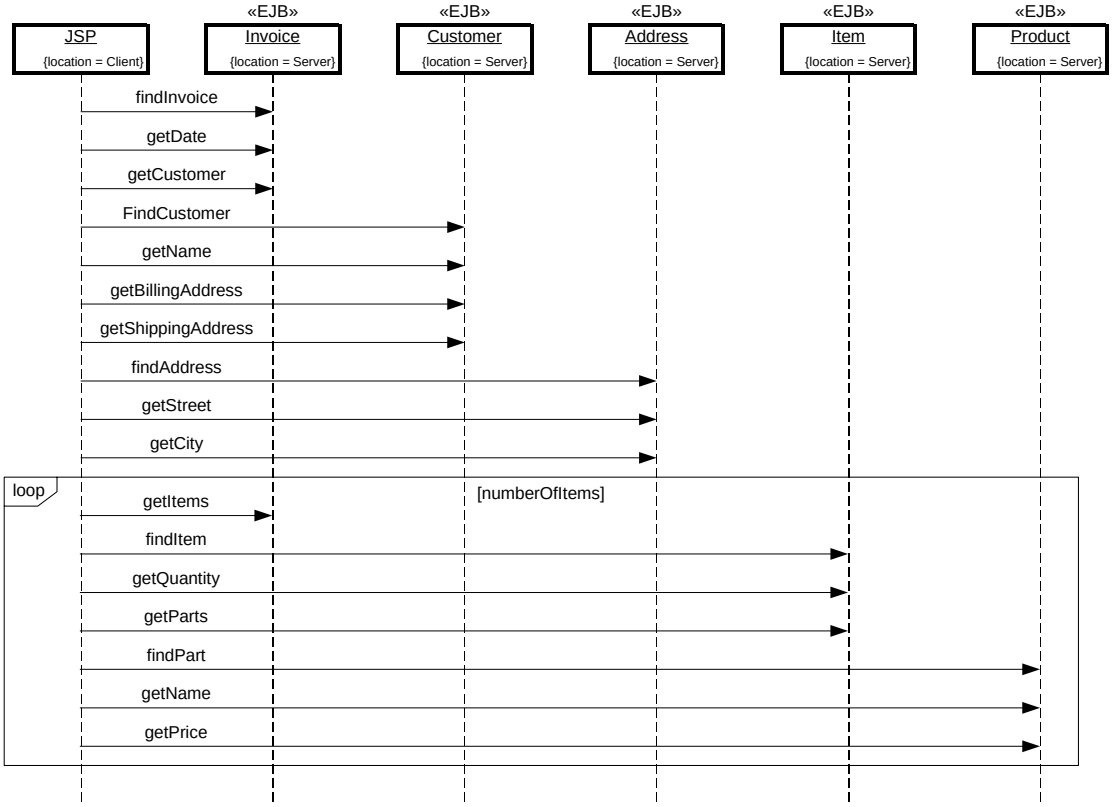

Figure 3: Tate's Example of Round-Tripping [Tate 2002]

Another example that we have seen and heard often involves MQ Series passing very small (e.g., 10 byte) messages, even though MQ had been set up with large containers (e.g., 400 bytes).

3.1.2 Inefficient Interface

The effect of an inefficient interface is illustrated by the Round-Tripping antipattern [Tate 2002], a special case of the Empty Semi Trucks performance antipattern. Tate introduces the problem using an anecdote about his two year old daughter who picks up her toys one at a time and walks them to the toy box versus his five year old daughter who picks everything up and then makes one trip to the toy box. (He doesn't mention how he gets the children to pick up their toys—an interesting problem in its own right.)

Tate uses a J2EE implementation of a distributed application to illustrate how the problem appears in software. The application displays invoices using a Web interface. The problem is illustrated in Figure 3.

3.2 Solution

The solution to the Empty Semi Trucks Antipattern depends on its cause.

If the problem is inefficient use of bandwidth, application of the Batching performance pattern [Smith and Williams 2002] will help by combining items in a message, amortizing the overhead of initialization, transmission, and termination processing over several items instead of just one. For example, spreadsheet cells could be combined into one packet, perhaps by rows or columns depending on the size. Static Web pages can define colors within the page's HTML. MQ messages to the same process can be combined into a single message.

Figure 4 illustrates the benefit of combining information to form one larger message rather than sending individual messages.

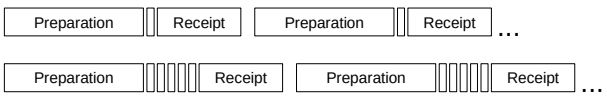

Figure 4: Effect of Batching

In this case, ten items can be sent in less time than three items without Batching. The optimum batch size can be determined using SPE models [Smith and Williams 2002].

If the problem is an inefficient interface, the Coupling performance pattern [Smith and Williams 2002] should

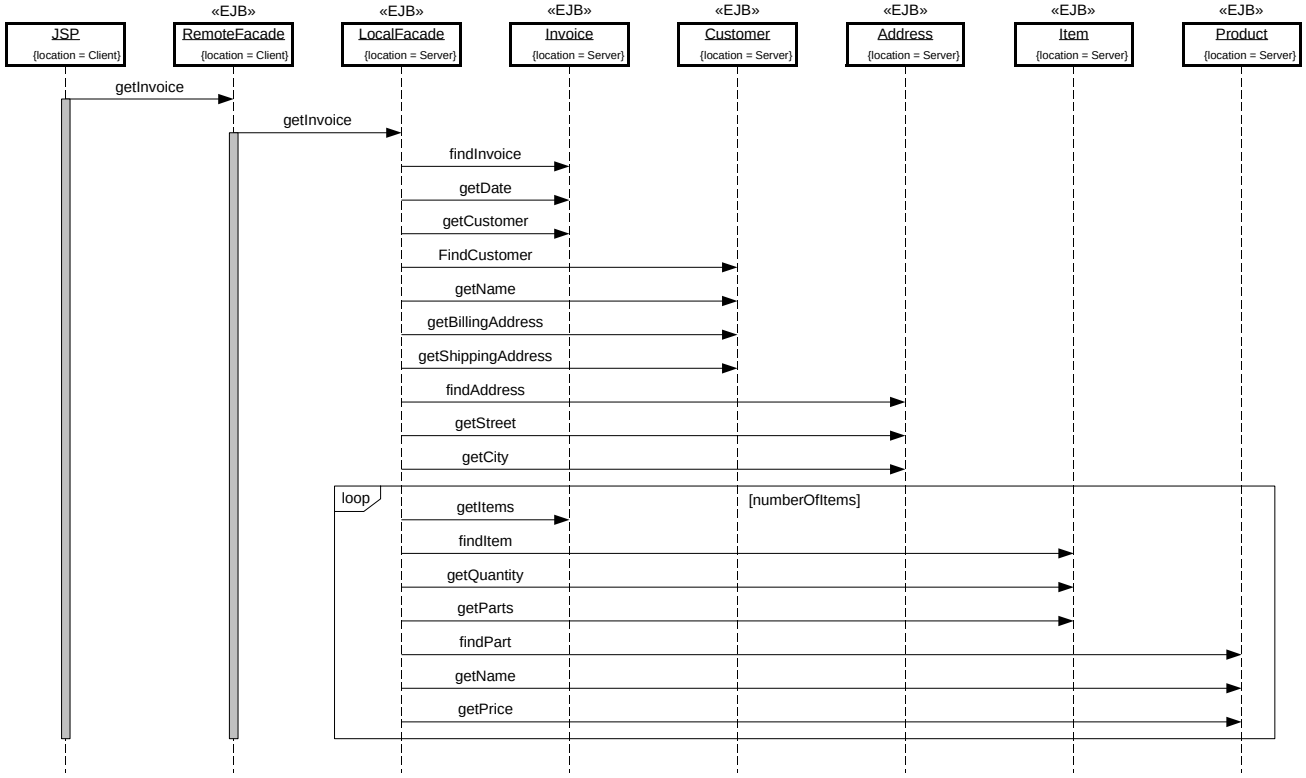

Figure 5: Tate's Solution to Round-Tripping [Tate 2002]

be used to provide a more efficient one. For the invoice application, use of the Session Facade design pattern [Sun 2001] would reduce the number of distributed requests required to display an invoice.

Figure 5 shows Tate's application of the Session Facade pattern to the invoice application with 20 items. Here, the 150 remote requests (10 for the invoice info plus 7 requests for each of the 20 items) have been reduced to one, and the time for the communication is reduced from 4.5 seconds to 30 ms.!

In general, the time savings, *T*, is:

$$T = (t_p + t_r) \times M$$

where:

- t_p is the time for preparation (processing time e.g., for acquiring message buffers as well as transmission overhead e.g., sending message headers),
- t_r is the time for the receipt (similar processing time and transmission overhead for acknowledgements, etc.),
- *M* is the number of messages eliminated.

The performance of this solution could be further improved by using the Aggregate Entity pattern [Larman 2000] to replace the EJBs that access the database with ordinary Java objects. This would reduce the overhead due to EJB-to-EJB communication.

4.0 Tower of Babel

The Tower of Babel failed because the builders did not understand one another's languages and thus were unable to communicate. The analogy we address here is a system of concurrent processes that must exchange information with one another, but internally use different formats to represent the information.

4.1 Problem

We see this problem most often when information is translated into an exchange format, such as XML, by the sending process then parsed and translated into an internal format by the receiving process. While this can be a good way to solve the problem, when the translation and parsing is excessive, the system spends most of its time doing this and relatively little doing real work.

One GIS (geographical information system) application experienced serious performance problems. Developers suspected that the problem was in the complex algorithms for accessing and rendering large, detailed maps. Upon analysis, however, the problem was actually found to be a Tower of Babel in which the request for the GIS data was translated into XML, it was then

parsed into an internal format for preprocessing, then the modified request was again translated into XML and passed downstream, and so on. After the data was retrieved, the larger amount of data was also repeatedly translated then parsed by multiple processes.

This antipattern often occurs when someone decides to adopt a new standard (such as XML) regardless of whether it is an appropriate solution. It also occurs when developers are overly cautious. For example, the https secure protocol is good for improving the integrity of interactions between a secure facility and an external user. Within the secure facility, however, it is usually not necessary to use the https protocol to pass information from one secure processor to another. It results in unnecessary translation and parsing, and may be a problem if it occurs frequently in a high-volume environment.

4.2 Solution

The Tower of Babel is often a problem on frequently used paths. Its solution uses the Fast Path performance pattern [Smith and Williams 2002] to determine which paths should be streamlined then minimizes the processing due to unnecessary translation and parsing on those paths. For those cases, select an internal data representation that is optimized for the patterns of use. When data arrives at the system boundary and/or is sent to another system, then parsing and translation is appropriate. The Coupling performance pattern may be useful for matching the data format to the usage patterns.

For the GIS application, eliminating unnecessary XML translation and parsing resulted in a ten-fold reduction in processing time!

In general, the time savings, T, is:

$$T = (s_c + s_p + s_t) \times 2N$$

where:

- s_c is the service time to convert the internal format to the intermediate format (such as XML) to send the request or response,
- s_p is the service time to parse the intermediate format,
- s_t is the service time to translate the intermediate format into the internal format
- N is the number of processes in the end-to-end scenario that require the translation to/from the intermediate format.

The multiplier of $2N$ in the formula is because the entire process—convert, parse and translate—executes for both the input message and the reply.

Figure 6 shows the overhead for some typical values of s_c, s_p, *and* s_t with N ranging from 1 to 10.

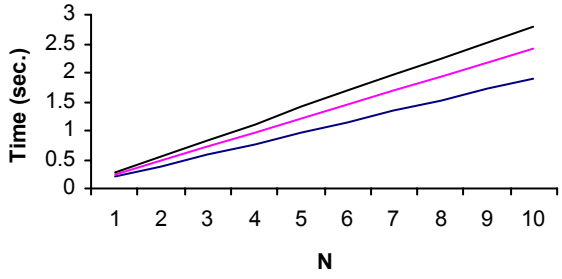

Figure 6: Overhead for Excessive Translation

5.0 Use of Performance Antipatterns

These software performance antipatterns have four primary uses:

- identifying potential problems in software architectures and designs
- focusing on the right level of abstraction by identifying the fundamental problem and its solution rather than a specific “fix” that might be outdated over time.
- effectively communicating what the problem is and why it is a problem, and building performance intuition
- prescribing solutions that embody sound, well-accepted performance principles [Smith and Williams 2002].

Because these performance antipatterns occur frequently, it is easy to find examples. When you do find them, it is still important to quantify execution characteristics, such as the arrival rate of requests or processing time requirements, to determine whether the presence of a performance antipattern limits scalability or if you are within scalability targets. For example, you may find Empty Semi Trucks in your application. But, if it only executes twice a day it won’t pose the same problem that it will if it executes on the Fast Path millions of times a day.

6.0 Summary and Conclusions

Performance antipatterns document common performance mistakes made in software architectures or designs. The use of software performance antipatterns has proven to be valuable in detecting and correcting performance problems as well as building performance intuition in developers.

This paper has presented three additional performance antipatterns. The table below summarizes all the currently documented performance antipatterns for reference.

Antipattern	Problem	Solution
Falling Dominoes	Occurs when one failure causes performance failures in other components.	Make sure that broken pieces are isolated until they are repaired.
Empty Semi Trucks	Occurs when an excessive number of requests is required to perform a task. It may be due to inefficient use of available bandwidth, an inefficient interface, or both.	The Batching performance pattern combines items into messages to make better use of available bandwidth. The Coupling performance pattern, Session Facade design pattern, and Aggregate Entity design pattern provide more efficient interfaces.
Roundtripping [Tate 2002]	Special case of Empty Semi Trucks. Occurs when many fields in a user interface must be retrieved from a remote system.	Buffer all the calls together and make them in one trip. The Facade design pattern and the distributed command bean accomplish this buffering.
Tower of Babel	Occurs when processes excessively convert, parse, and translate internal data into a common exchange format such as XML.	The Fast Path performance pattern identifies paths that should be streamlined. Minimize the conversion, parsing, and translation on those paths by using the Coupling performance pattern to match the data format to the usage patterns.
Unbalanced Processing [Smith and Williams 2002]	Occurs when processing cannot make use of available processors, the slowest filter in a “pipe and filter” architecture causes the system to have unacceptable throughput, or when extensive processing in general impedes overall response time.	1) Restructure software or change scheduling algorithms to enable concurrent execution. 2) Break large filters into more stages and combine very small ones to reduce overhead. 3) Move extensive processing so that it doesn’t impede high traffic or more important work.
Unnecessary Processing [Smith and Williams 2002b]	Occurs when processing is not needed or not needed at that time.	Delete the extra processing steps, reorder steps to detect unnecessary steps earlier, or restructure to delegate those steps to a background task.
The Ramp [Smith and Williams 2002b]	Occurs when processing time increases as the system is used.	Select algorithms or data structures based on maximum size or use algorithms that adapt to the size.
Sisyphus Database Retrieval Performance Antipattern [Dugan, et al. 2002]	Special case of The Ramp. Occurs when performing repeated queries that need only a subset of the results.	Use advanced search techniques that only return the needed subset.
More is Less [Rogers and Boyer]	Occurs when a system spends more time “thrashing” than accomplishing real work because there are too many processes relative to available resources.	Quantify the thresholds where thrashing occurs (using models or measurements) and determine if the architecture can meet its performance goals while staying below the thresholds.
“god” Class [Smith and Williams 2002]	Occurs when a single class either 1) performs all of the work of an application or 2) holds all of the application’s data. Either manifestation results in excessive message traffic that can degrade performance.	Refactor the design to distribute intelligence uniformly over the application’s top-level classes, and to keep related data and behavior together.

Antipattern	Problem	Solution
Excessive Dynamic Allocation [Smith and Williams 2002]	Occurs when an application unnecessarily creates and destroys large numbers of objects during its execution. The overhead required to create and destroy these objects has a negative impact on performance.	1) "Recycle" objects (via an object "pool") rather than creating new ones each time they are needed. 2) Use the Flyweight pattern to eliminate the need to create new objects.
Circuitous Treasure Hunt [Smith and Williams 2002]	Occurs when an object must look in several places to find the information that it needs. If a large amount of processing is required for each "look," performance will suffer.	Refactor the design to provide alternative access paths that do not require a Circuitous Treasure Hunt (or to reduce the cost of each "look").
One-Lane Bridge [Smith and Williams 2002]	Occurs at a point in execution where only one, or a few, processes may continue to execute concurrently (e.g., when accessing a database). Other processes are delayed while they wait for their turn.	To alleviate the congestion, use the Shared Resources Principle to minimize conflicts.
Traffic Jam [Smith and Williams 2002]	Occurs when one problem causes a backlog of jobs that produces wide variability in response time which persists long after the problem has disappeared.	Begin by eliminating the original cause of the backlog. If this is not possible, provide sufficient processing power to handle the worst-case load.

7.0 REFERENCES

[Brown, et al. 1998] 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.

[Buschmann, et al. 1996] 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.

[Dugan, et al. 2002] R. F. Dugan Jr., E. P. Glinert, A. Shokoufandeh, "The Sisyphus Database Retrieval Performance Antipattern," Proceedings of the Workshop on Software and Performance (WOSP 2002), Rome, July 2002.

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

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

[Larman 2000] C. Larman, "Aggregate Entity Pattern," Software Development, vol. 8, no. 4, pp. 46-52, 2000.

[Neumann 1990] P. G. Neumann, "Cause of AT&T Network Failure," Risks Digest, vol. 9, no. 62, February 26, 1990.

[Rogers and Boyer] G. Rogers and R. Boyer, "The More is Less Antipattern" private communication.

[Schmidt, et al. 2000] D. Schmidt, M. Stal, H. Ronert, and F. Buschmann, Pattern-Oriented Software Architecture Volume 2: Patterns for Concurrent and Networked Objects, Chichester, England, John Wiley and Sons, 2000.

[Smith and Williams 2002] C. U. Smith and L. G. Williams, Performance Solutions: A Practical Guide to Creating Responsive, Scalable Software, Boston, MA, Addison-Wesley, 2002.

[Smith and Williams 2002b] C. U. Smith and L. G. Williams, "New Software Performance Antipatterns: More Ways to Shoot Yourself in the Foot," Proc. CMG, Reno, December, 2002.

[Smith and Williams 2001] C. U. Smith and L. G. Williams, "Software Performance Antipatterns: Common Performance Problems and Their Solutions," Proc. CMG, Anaheim, December, 2001.

[Smith and Williams 2000] C. U. Smith and L. G. Williams, "Software Performance Antipatterns," Proceedings of the Second International Workshop

on Software and Performance (WOSP2000), Ottawa, Canada, September, 2000, pp. 127-136.

[Sun 2001] Sun Microsystems, “Session Facade,” 2001, http://developer.java.sun.com/developer/restricted/patterns/SessionFacade.html.

[Tate 2002] B. A. Tate, “A Taste of 'Bitter Java:' The Round-Tripping Antipattern,” 2002, http://www-106.ibm.com/developerworks/java/library/j-bitterj-java/bjsidebar1.html.