Chapter 10: Performance Patterns
Patterns

• 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
  – Allows knowledge to be reused
  – Applied in design of many different types of software

• Pattern address the problem of “reinventing the wheel”
History of Patterns

- The use of patterns in software development has its roots in the work of Christopher Alexander, an architect:

  *Each pattern describes a problem which occurs over and over again in our environments, and then describes the core of the solution to that problem, in such a way that you can use this solution in million times over, without ever doing it the same way twice.*
Design Patterns

• In the late 1980s, several people in the software development community began to apply Alexander’s ideas to software
  – *Design Patterns: Elements of Reusable Object-Oriented Software*, by Erich Gamma, Richard helm, Ralph Johnson, and John Vlissides (the Gang of Four)

• Design patterns identify abstractions that are at a higher level than individual classes and objects
  – Construct the software using patterns
    • Singleton Pattern, Proxy Pattern
Most popular book in Computer Science
Sold over one million copies in print
History of Eclipse

- 1997 – VisualAge for Java (implemented in small talk)
- 2001 – Eclipse (change name for marketing issue)
- 2003 — Eclipse.org
- 2005- Eclipse V3.1
- 2006- Eclipse V3.2
Architecture of Eclipse

• The eclipse plug-in architecture – increase modularity
• Everything is a plug-in
• Extension points
  – its component configuration points
Performance Patterns

- The performance patterns describe best practices for producing responsive, scalable software
- Performance patterns complement and extend the performance principles
- Seven performance patterns address performance and scalability
  - Fast Path
  - First Things First
  - Coupling
  - Batching
  - Alternate Routes
  - Flex time
  - Slender Cyclic Functions
Performance Patterns vs. Design Patterns

- Each performance pattern is a realization of one or more of the performance principles.
- The performance patterns are at a higher level of abstraction than design patterns.
  - A design pattern may provide an implementation of a performance pattern.

Level of abstraction:
- Performance Principles
- Performance Patterns
- Design Patterns
- Centering Principles
- Fast Path
- Proxy
Pattern Template

• Each pattern is defined in a standard template:
  – **Name**: The title of the subsection
  – **Problem**: What is motivating us to apply this pattern?
  – **Solution**: How do we solve the problem?
  – **Benefits**: What are the potential positive outcomes of applying this pattern?
  – **Consequences**: What are the potential shortcomings and consequences of applying this pattern?
Fast Path

• Concerned with improving response time by reducing the amount of processing required for dominant workloads
  – Example: menus in automated telephone system

• Problem: *dominant workload*

• Solution:
  – Create an express “train” that stops only at the most important stations along the route
  – Identify the data most frequently used together
  – Implemented by Proxy patterns
  – Based mainly on the *centering principle*
If (myImage == 0) {
  return extent;
} else{
  return(myImage->getExtension());
}

if (myImage == 0) {
  myImage = load(fileName);
}
myImage->draw();
Fast Path (Con’t)

• Benefits:
  – Reduces the response time for dominant workload functions by reducing the amount of processing required for the most frequent uses of the software
  – Reduces the overall load on the system by avoiding some resource consumption

• Consequences:
  – It is not enough to recognize the need for the Fast Path you must also ensure that it is likely to be used
  – Usage patterns change over time
  – Use the instrumenting principle to monitor usage patterns, and adapt your system to changing patterns
First Things First

• Focus on the important processing tasks to ensure that, if everything cannot be completed within the time available, then the least important tasks will be the ones omitted

• Problem:
  – Temporary overload may cause input data to be lost or response times to be unacceptably slow
  – Example: online-trading

• Solution:
  – Assign priorities to tasks and execute them so that the most important activities receive preference
  – Example: transaction of billions of dollars
  – Use the *Centering Principle* to focus attention on the most important work
First Things First (Con’t)

• Benefits
  – Focuses on the most important tasks and ensures that they complete
  – Maximizes the quality of service of the system and improves scalability

• Consequences
  – Only appropriate if the overload is temporary
  – If the overload is not temporary, reduce the amount of processing required by other means or upgrade the processing environment
Coupling

• Match the interface of an object with its most frequent uses
• Problem: Applications use fine-grained objects to request remote information  
  – The number of interactions is large  
  – Cost of remote calls is high in distributed systems  
  – Responsiveness is poor in multi-tier Web applications  
  – Using a class structure identical to the physical database schema can lead to performance problems
Coupling (Con’t)

• Solution:
  – Use more *coarse-grained* objects to eliminate frequent requests for small amount of information
  – The best way of constructing the aggregation will depend on the access patterns for the data
  – Data that is frequently accessed at the same time should be grouped into an aggregation
  – Use the *Centering Principle* to identify interfaces
  – Use the *Locality Principle* to combine information
  – Use the *Processing vs. Frequency Principle* to minimize the total processing required for the interface
Coupling (Con’t)

• Benefits:
  – Match the business tasks to the processing required to accomplish them
  – Reduce the total resource requirements of the system

• Consequences:
  – Start by identifying information that is stable, and use those objects to reduce the amount of communication overhead required to obtain data
Batching

• Combines frequent requests for services to save the overhead of initialization, transmission, and termination processing for the request

• Problem:
  – Requested tasks require considerable overhead processing for initialization, termination, and in distributed systems, for transmitting data and requests
  – For very frequent tasks, the amount of time spent in overhead processing may exceed the amount of real processing on the system
  – Example
    • Insert new rows
    • Send secured messages
Batching (Con’t)

• Solution:
  – Combine the requests into batches so the overhead processing is executed once for the entire batch instead of for each individual item
    • Sender-side batching (e.g., insert new rows)
    • Receiver-side batching (e.g., transfer secured messages over links)
  – Using the *Processing vs. Frequency Principle* to minimize the product of the processing times the frequency of requests
Batching (Con’t)

• Benefits:
  – Reduce the total amount of processing required for all tasks
  – Improve responsiveness by reducing the contention delay
  – Improve scalability by freeing up resources

• Consequences:
  – Batching is appropriate for frequent tasks that require a large amount of overhead processing
  – Batching is most effective when the amount of overhead and the frequency of requests are both high
Alternate Routes

• Spread the demand for high-usage objects spatially to different objects or locations
• Reduce contention delays for the objects
• Problems:
  – Occurs frequently in database systems when many processes need exclusive access to the same physical location, usually to execute an update
  – Happens when several processes must coordinate with a single concurrent process
  – When a single dispatching process receives inbound requests and determines which subsequent process is to handle the request
Alternate Routes (Con’t)

• Solution:
  – Find an alternate route for the processing
    • In database access situation, find a way for the access
to go to different physical locations
    • For the process coordination problems, find a way to
route requests to different processes
    • For the one-inbound dispatcher problem, use multiple
instances of the dsipatcher
  – Use the Spread-the-Load Principle
Alternate Routes (Con’t)

• Benefits:
  – Reduces delays due to serialization
  – Improves responsiveness and scalability
  – Reduces the variability in performance

• Consequences:
  – Make sure that your alternate route effectively spreads the load spatially
Flex Time

• Spread the demand for high-usage objects temporally to a different period of time
• Reduce contention delays for the objects
• Problems:
  – Processing is required at a particular frequency, or at a particular time of day
  – Users are allowed to select the time of day when they want the reports, but are all given the same choices for time of day
Flex Time (Con’t)

• Solution:
  – Identify the functions that execute repeatedly at regular, specific time intervals, and modify the time of their processing
  – Solution to the time-of-day problem is to move the processing to a different time of day
  – Solution to the processing-time-choice problem is to generate a random number for the selection choices
  – Solution to the periodic processing problem is to do less work more often
  – Apply the Spread-the-Load Principles
Flex Time (con’t)

• Benefits:
  – Spread the load temporally to reduce the congestion
  – Reduces the amount of time that processes are blocked and cannot proceed
  – Reduces the resource demand so that concurrent process encounter fewer queueing delays for computer resources

• Consequences:
  – Some of the Flex Time solutions require more processing
  – The net effect is to reduce the time that processes wait in queues
  – The Flex Time has the same potential problem as Alternate Routes
    • if everyone chooses the same alternate time, you have a new bottleneck
Slender Cyclic Functions

• Concerned with processing that must execute at regular intervals

• Problem:
  – A cyclic or periodic function is characterized by its:
    • Period: the amount of time between successive executions
    • Execution time: the amount of time required for the function to execute
    • Slack time: the amount of time between the completion of execution and the end of the period
Slender Cyclic Functions (Con’t)

• Solution:
  – Identify the functions that execute repeatedly at regular, specific time intervals, and minimize their processing requirements
  – Use both the Centering Principle and the Shared Resources Principles

• Benefits:
  – Reduce the processing requirements so that we have more resources available to share and thus reduce queueing delays
Slender Cyclic Functions (Con’t)

• Consequences:
  – Operating conditions may change over time
  – The cycle frequency may need to change, or the amount of processing per cycle may change
  – Instrument systems and monitor their performance over time for early warning of potential problems