Concurrency Design Patterns, Software Quality Attributes and their Tactics

Jiang Zheng, K. Eric Harper
Industrial Software Systems, ABB Corporate Research
940 Main Campus Drive, Raleigh, NC, USA
{jiang.zheng, eric.e.harper}@us.abb.com

ABSTRACT
With the prevalent application of multi-core CPUs, software practitioners are facing the challenge of developing high quality multi-threaded programs. Applying concurrency design patterns is one of the best practices in multi-core software engineering. We comprehensively surveyed 28 concurrency design patterns, and provided a problem-oriented guide that navigates software developers towards the "right" pattern(s) with minimal search/reading effort. The guide also illustrates the relationship between concurrency design patterns and the quality attributes they address. Additionally, further investigation was conducted on how these concurrency design patterns implement quality attributes tactics. We present a mapping between surveyed concurrency design patterns and the tactics for two important quality attributes: performance and modifiability. The results of these studies provide an insight into concurrency design patterns for software developers who are seeking appropriate or improved solutions to multi-threaded software development issues.

Categories and Subject Descriptors
D.1.3 [**Programming Techniques**]: Concurrent Programming – *Parallel programming.*

D.2.2 [**Software Engineering**]: Design Tools and Techniques – *Object-oriented design methods.*

General Terms
Performance, Design.

Keywords
Multi-core, Concurrency, Parallelism, Design Pattern, Quality Attribute, Quality Attribute Tactic

1. INTRODUCTION
Nowadays modern multi-core CPUs are widely used in various computers. The trend has shown that multi-core CPUs are expected to be used in almost all servers and laptop PCs within two years [19]. In order to take advantage of multi-core computer architectures, more and more software practitioners are investigating parallel programming techniques. However, using multi-core CPUs creates more challenges for software developers to design and program high quality applications, although it is desirable to meet growing computing demands [19].

Applying design patterns is one of the best practices in software engineering. A *software design pattern* is a general reusable solution to a commonly occurring problem in software design [3]. Concurrency design patterns, also known as *multi-threading design patterns*, or *multi-threaded design patterns*, are design patterns that address multi-threaded programming issues. Typical multi-threaded programming issues include managing and coordinating threads, simplifying complex multi-threaded coding, preventing concurrency defects such as data racing and deadlock, with the goal of increasing performance.

However, most concurrency design patterns are complex and hard to be understood, which requires significant reading and investigation. Developers often have the questions such as when to use which pattern(s), which specific problem(s) a pattern deals with, and the pre-conditions necessary to apply a pattern.

In this paper, we concentrate on how to better transfer knowledge and best practices of concurrency design patterns to the software designers and developers. We comprehensively surveyed 28 concurrency design patterns, and provided a problem-oriented guide that navigates the software designers and developers towards the "right" pattern(s) with minimal search and/or reading effort. This problem-oriented guide is presented via frequent asked questions and a chart describing the relationship between concurrency design patterns and quality attributes they address. *Quality attributes* are a system's architecturally critical non-functional requirements, such as performance, maintainability, and reliability [2]. Additionally, further investigation was conducted on how these concurrency design patterns implement quality attributes tactics. A *quality attribute tactic* is a design decision that influences the control of a quality attribute response [2]. From the perspective of a software designer, each tactic is a design option. For example, in order to achieve high performance, the software designer may choose one or more options from increasing computation efficiency, reducing computational overhead, introducing concurrency, maintaining multiple copies, and/or other performance tactics which are listed in Section 4. This paper also presents a mapping between surveyed concurrency design patterns and the tactics for two important quality attributes: performance and modifiability.

Several different communities may benefit from these studies. Software developers who are facing multi-threaded programming

IWMSE'10, May 1, 2010, Cape Town, South Africa.

problems and want to have deeper understanding in concurrency design and implementation may benefit from the best practices of concurrency design patterns and related algorithms. Software architects and designers who are seeking appropriate or improved solutions to concurrency software design/coding problems for their systems may benefit from reading the problem-oriented guide, as well as the mapping between concurrency design patterns and quality attributes tactics.

The rest of this paper is organized as follows. Section 2 describes the survey of concurrency design patterns. Section 3 illustrates the problem-oriented guide. Section 4 presents the mapping between surveyed concurrency design patterns and tactics for achieving performance and modifiability qualities. Section 5 and Section 6 presents conclusions and future work, respectively.

2. SURVEY OF CONCURRENCY DESIGN PATTERNS

A software design pattern is a general reusable solution to a commonly occurring problem in software design, although a design pattern is not a finished design that can be transformed directly into code [3]. A design pattern is a description or template for how to address a problem that can be used in many different situations.

Concurrency design patterns deal with two types of problems that often occur in the multi-threaded programming paradigm. The most pervasive problem is s*hared resources*, i.e., when concurrent operations access shared resource, such as same data, operations may interfere with each other if they access the resource at the same time. The other important problem is *sequence of operations.* It is necessary to ensure that operations access the shared resource in a particular order, if they are constrained to access a shared resource one at a time. [4]

We surveyed and investigated a total of 28 concurrency design patterns [4-18]. Due to limited space in this paper, we only list the name, short description, and reference of each concurrency design pattern in Table 1. Detailed information may be found through reference books, papers, and articles. For better understanding, we also grouped these concurrency design patterns into five categories, as shown in Table 1: (1) *Structural Patterns*: design patterns that make the design easier by identifying a simple way to realize relationships between entities; (2) *Behavioral Patterns*: design patterns that identify and realize common communication patterns between objects, to increase flexibility in carrying out this communication; (3) *Architectural Patterns*: design patterns that offer well-established solutions to architectural problems; (4) *Enterprise Patterns*: design and architectural patterns for use in distributed and enterprise applications; and (5) *Other General-Purpose Patterns*: design patterns that are difficult to be classified into the previous four categories. Note that some of the general-purposed patterns are very basic and important concurrency design patterns.

3. PROBLEM-ORIENTED GUIDE

Multi-threaded programming introduces a significant number of concerns and problems, such as race conditions, deadlock, thread scheduling, programming complexity, code modifiability and maintainability. However, no single concurrency design pattern is capable of addressing all multi-threaded programming concerns and problems. People who are facing different concurrency programming problems seek specific and appropriate solutions.

We created a "problem-oriented" presentation of the concurrency design patterns that guides the developer towards the "right" pattern(s) with minimal search/reading effort. The problem-oriented guide includes frequent asked questions about concurrency design patterns and related multi-threaded programming. We list these frequent asked questions and their answers as follows:

Q1: *OK, I have no idea about any concurrency design patterns. Where can I start from?* (i.e., *What are the simplest and/or most fundamental concurrency design patterns?*)

A: You may start with the following concurrency design patterns: the *Single Threaded Execution Pattern* [4], the *Asynchronous Processing Pattern* [4], and the *Scoped Locking Idiom* [13]. Then you may want to go further to learn the *Thread Pool Pattern* [5] which is another classic and frequently used concurrency design pattern.

Q2: *I would like to design an event-driven software system. What concurrency design patterns can I utilize?* (i.e., *Which patterns are suitable for which types of applications?*)

A: The following concurrency design patterns are suitable for event-driven applications: the *Event-based Asynchronous Pattern* [9] implements an event-driven model to perform asynchronous tasks; the *Reactor Pattern* [12] is suitable for a server application in a distributed system that receives concurrent events from clients; and the *Leader/Followers Pattern* [18] is often used in an event-driven application where multiple service requests arrive from a set of event sources and must be processed by multiple threads that share the event sources.

Similarly, if you would like to develop a Web application, please refer to the *Ajax Design Patterns* [8] whose `XMLHttpRequest` objects enable asynchronous data exchange with web servers, and the *Session Object Pattern* [5] which is suitable for a server needs to know the state of each session and uses a stateless protocol to communicate with its clients. Additionally, all *Enterprise Patterns* (see Table 1) are suitable for distributed applications; the *Optimistic Concurrency Pattern* [5] is often used in database systems; the *Double Buffering Pattern* [4] is often utilized in graphical processing applications; and the *Half-Sync/Half-Async Pattern* [15] often appears in I/O intensive applications.

Q3: *Which patterns can help avoid race conditions?*

A: Please refer to the following concurrency design patterns: the *Lock Object Pattern* [4] ensures that a set of objects is being accessed by only one thread at a time; the *Read/Write Lock Pattern* [4] ensures that calls to an object's set methods do not interfere with each other, or with calls to the object's get methods; the *Monitor Object Pattern* [14] ensures only one method runs within an object at a time by synchronizing method execution; the *Lock File Pattern* [5] ensures exclusive access to a resource by having each cooperating object first test whether a lock file has already existed; the *Double-Checked Locking Pattern* [16] ensures that the critical section is performed only once by the second check; and the *Single Threaded Execution Pattern* [4] ensures that operations are not performed concurrently if they cannot be performed concurrently correctly.

Table 1. Collection of 28 Concurrency Design Patterns

Type	Design Pattern Name	Description
Structural Patterns	**Double Buffering**	Avoids delays in the consumption of data by asynchronously producing data with the goal of the data being ready for consumption before the consumer needs it. [4]
	Lock Object	Arrange for threads to have exclusive access to a set of objects by having the threads acquire a synchronization lock on an object that exists only for this purpose. [4]
	Producer-Consumer	Coordinates the asynchronous production and consumption of information or objects. [4]
	Read/Write Lock	Allows concurrent read access to an object, but requires exclusive access for write operations. [4]
	Scheduler	Provides a mechanism for implementing a scheduling policy. [4]
	Thread-safe Decorator	Ensures intra-component method calls avoid self-deadlock and minimize locking overhead. [13]
Behavioral Patterns	**Active Object**	Decouples method execution from method invocation and simplifies synchronized access to a shared resource by methods invoked in different threads of control. [6]
	Balking	Useful if an operation must be done either immediately or not at all. [4]
	Event-based Asynchronous	Provides an event-driven programming model for performing asynchronous operations. [9, 10]
	Future	Describes how to keep classes that invoke an operation from having to know whether the operation is synchronous or asynchronous. [4]
	Guarded Suspension	Provides guidance on what to do when a thread has exclusive access to a resource and discovers that it cannot complete the operation on that resource because something is not yet ready. [4]
	Monitor Object	Synchronizes method execution to ensure only one method runs within an object at a time. [14]
	Reactor	Simplifies event-driven applications by integrating the synchronous demultiplexing of events and the dispatching of their corresponding event handlers. [12]
	Thread-Specific Storage	Improves performance and simplifies multi-threaded applications by allowing multiple threads to use one logical access point to retrieve thread local data without incurring locking overhead for each access. [17]
Architectural Patterns	**Asynchronous Processing**	Describes how to avoid waiting for the results of an operation when you don't need to know the result immediately. [4]
	Half-Sync/Half-Async	Decouples synchronous I/O from asynchronous I/O in a system to simplify concurrent programming effort without degrading execution efficiency. [15]
	Leader/Followers	Provides an efficient concurrency model where multiple threads take turns sharing a set of event sources in order to detect, demultiplex, dispatch, and process service requests that occur on these event sources. [18]
Enterprise Patterns	**AJAX**	Shows the best practices that can dramatically improve the web development projects. [8]
	Lock File	Checks for the existence of a lock file prior to accessing the resource. [5]
	Optimistic Concurrency	Be optimistic and perform the transaction logic on private copies of the records or objects involved. [5]
	Session Object	Uses a single object to contain all the state information needed during a session by the server, and makes that object accessible to all other objects that need state info. for the current session. [5]
	Static Locking Order	Ensures that all objects acquire locks on resources in the same order. [5]
	Thread Pool	Avoids the expense of creating a thread for each task by reusing threads. [5]
Other General-Purpose Patterns	**Double-Checked Locking** [1]	Allows atomic initialization, regardless of initialization order, and eliminates subsequent locking overhead. [16]
	Scoped Locking	Ensures that a lock is acquired when control enters a scope and the lock is released automatically when control leaves the scope. [13]
	Single Threaded Execution	Prevents concurrent calls to the method from resulting in concurrent executions of the method. [4]
	Strategized Locking	Strategizes a component's synchronization to increase its flexibility and reusability without degrading its performance or maintainability. [13]
	Two-Phase Termination	Provides for the orderly shutdown of a thread or process through the setting of a latch. [4]

[1] This pattern might not work reliably in some situations [1].

Q4: *Which patterns can help avoid deadlock?*

A: Please refer to the following concurrency design patterns: the *Thread-safe Decorator Pattern* [13] avoids self-deadlock due to intra-component method calls by acquiring/releasing locks and performing the synchronization checks in interface methods only; the *Guarded Suspension Pattern* [4] avoids the deadlock situation that can occur when a thread is about to execute an object's synchronized method, but the state of the object prevents the method from completing; and the *Static Locking Order Pattern* [5] avoid deadlocks by ensuring that all objects acquire locks in the same order.

Q5: *Besides the above-mentioned patterns in the last two FAQs, what other patterns can help improve the robustness/reliability of my multi-threaded application?*

A: Please refer to the following concurrency design patterns: the *Balking Pattern* [4] avoids a method in a thread being waited until another thread completes its task by returning the method without doing anything; the *Scoped Locking Idiom* [13] improves robustness by automatically acquire/release locks when the control enters/leaves critical sections defined by method and block scopes; and the *Two-Phase Termination Pattern* [4] ensures appropriate shutdown of threads.

Q6: *My single-threaded application runs very slowly. How can I improve responsiveness and throughput by utilizing concurrency design patterns so that my application becomes more efficient?* (i.e., *Which patterns can help improve software performance?*)

A: Please refer to the following concurrency design patterns: the *Double Buffering Pattern* [4] avoids delays in data consumption by asynchronously producing and putting data in a buffer before it is needed; the *Producer-Consumer Pattern* [4] produces objects to a queue without having to wait for object consumption; the *Read/Write Lock Pattern* [4] allows concurrent calls to an object's get methods; the *Scheduler Pattern* [4] provides a way to explicitly control when threads may execute which piece of code; the *Thread-safe Decorator Pattern* [13] ensures no unnecessary locks acquired or released; the *Active Object Pattern* [6] enhances concurrency by allowing client threads and asynchronous methods to run at the same time; the *Balking Pattern* [4] allows one thread to execute while ignoring the another; the *Event-based Asynchronous Pattern* [9, 10] runs time-consuming tasks "in the background" without interrupting the application; the *Thread-Specific Storage Pattern* [17] avoids locking overhead for each access by allowing multiple threads to use one logical access point to retrieve thread local data; the *Asynchronous Processing Pattern* [4] advocates that requests should be queued and processed asynchronously; the *Half-Sync/Half-Async Pattern* [15] decouples synchronous I/O from asynchronous I/O in a system; the *Leader/Followers Pattern* [18] allows multiple threads to take turns sharing a set of event sources; the *Ajax Design Patterns* [8] allows asynchronous data exchange with web servers; the *Optimistic Concurrency Pattern* [5] improves transaction processing throughputs by not waiting for locks that a transaction will need; and the *Thread Pool Pattern* [5] reduce the expense of threads creation by reusing threads.

Q7: *My muluti-threaded application is in a mess and hard to maintain. How can I improve maintainability and modifiability by using concurrency design patterns?*

A: In the *Producer-Consumer Pattern* [4], a queue acts as a buffer for produced objects. In the *Read/Write Lock Pattern* [4], the logic for coordinating read and write operations should be reusable. In the *Scheduler Pattern* [4], the scheduling policy is encapsulated and reusable. In the *Thread-safe Decorator Pattern* [13], only the interface methods take the responsibility of complex concurrency check operations. In the *Active Object Pattern* [6], a proxy acts as an intermediary that buffers changes to the method invocation. The *Event-based Asynchronous Pattern* [9] hides many of the complex issues inherent in multi-threaded design. The *Future Pattern* [4] uses an object to encapsulate the result of a computation in a way that hides from its clients whether the computation is synchronous or asynchronous. The *Reactor Pattern* [12] decouples application functionality into separate classes, which enables the reuse of the connection establishment class for different types of connection-oriented services (such as file transfer, remote login, and video-on-demand). The *Thread-Specific Storage Pattern* [17] allows multiple threads to use one logical access point. In the *Half-Sync/Half Async Pattern* [15], the Queueing layer coordinates the exchange of data between the Asynchronous and Synchronous task layers. The *Leader/Followers Pattern* [18] separates the concerns of event processing and detecting. The *Ajax Design Pattern* [8] allows programmers to separate methods and formatting for specific information delivery functions on the Web. In the *Session Object Pattern* [5], the number of objects used to maintain the state of sessions is minimized. Session objects provide a common interface for accessing values shared by different parts of a program. The *Thread Pool Pattern* [5] avoids the expense of creating a thread for each task by reusing threads. The *Strategized Locking Pattern* [13] coordinates a component's synchronization to increase its flexibility and reusability without degrading its performance or maintainability.

Q8: *My multi-threaded programs are really complex and hard to understand. How can I simplify my design by using concurrency design patterns?*

A: Please refer to the following concurrency design patterns: the *Lock Object Pattern* [4] and the *Lock File Pattern* [5] provide ways to simplify coordination of access to multiple resources; the *Active Object Pattern* [6] invokes methods on different threads of control so that the synchronized access to a shared resource is simplified; the *Monitor Object Pattern* [14] simplifies synchronization of methods invoked concurrently on an object; the *Reactor Pattern* [12] simplifies event-driven applications through the integration of the synchronous events demultiplexing and the corresponding event handlers dispatching; the *Thread-Specific Storage Pattern* [17] allows developers to use thread-specific storage completely transparent at the source-code level via data abstraction or macros; and the *Half-Sync/Half-Async Pattern* [15] simplifies concurrent programming by decoupling synchronous I/O from asynchronous I/O in a system.

Q9: *I can understand that most of the concurrency design patterns address the problem of shared resources. But what about operation ordering?* (i.e., *Which patterns address the problem of "sequence of operations"?*)

A: The *Scheduler Pattern* [4] manages the scheduled threads order to execute single threaded code using an object that explicitly sequences waiting threads. The *Leader/Followers Pattern* [18] coordinates concurrency that multiple threads take

turns sharing a set of event sources. The *Static Locking Order Pattern* [5] ensures that multiple shared resources are always locked by all objects in the same relative order. The *Two-Phase Termination Pattern* [4] orderly shuts down a thread or process via the setting of a latch.

Q10: *I think most of the concurrency design patterns are tactical and low-level. Are there any architectural concurrency design patterns?* (i.e., *Which patterns are strategic and high-level patterns?*)

A: Please refer to all the *Architectural Patterns* and *Enterprise Patterns* (see Table 1), as well as the *Thread-safe Decorator Pattern* [13] in the Structural Patterns category.

The problem-oriented guide can as well partially be presented as a mapping between concurrency design patterns and the quality attributes they address, as shown in Table 2. Row headings in the table correspond to surveyed concurrency design patterns. Column headings in the table correspond to quality attributes. A "●" in a cell stands for strong relationship between the corresponding concurrency design pattern and that quality attribute. For example, the Thread Pool Pattern can help improve both modifiability and performance, so there are two "●"'s in the corresponding cells. The question mark for the Double-Checked Locking Pattern indicates that this pattern might not work reliably. This table may help software designers find appropriate concurrency design patterns according to the quality attribute(s) they would like to address.

4. QUALITY ATTRIBUTES TACTICS

The next research question following the problem-oriented guide is how these concurrency design patterns achieve the corresponding software quality attributes. Software architects and designers create a design by using fundamental design decisions, which are called *tactics* by the Software Engineering Institution (SEI), to achieve desired software qualities, such as performance, modifiability, and reliability [2]. Bass et al. captured architectural guidance for achievement of six categories of quality attributes including availability, modifiability, performance, security, testability, and usability [2]. Among these quality attributes, performance and modifiability have strongest relationship with concurrency design patterns. We examined how concurrency design patterns achieve performance and modifiability qualities. A number of concurrency design patterns deal with the quality of reliability as well, especially addressing how to prevent race conditions and deadlock. However, no explicit set of reliability tactics were found in the literature. Therefore, we leave the research on how concurrency design patterns implement reliability tactics as future work.

Bass et al. organized modifiability tactics into three categories according to their goals [2]:

1) *Localize modifications* to reduce the number of modules directly affected by a change. This category consists of five detailed tactics: a) *Maintain semantic coherence* that ensures all responsibilities in a module work together without excessive reliance on other modules; b) *Anticipate expected changes* when evaluating assignment of responsibilities; c) Instead of modifying a module, *generalize the module* so that request changes can be made by adjusting the input; d) *Limit possible options* especially works in design of product lines; and e) *Abstract common services* to reduce amount of modification (e.g., the use of application frameworks and middleware software).
2) *Prevent the ripple effect* to limit modifications to the localized modules. Detailed tactics include: a) *Hide information* that decomposes responsibilities for an entity into smaller pieces and assigning private/public; b) *Maintain existing interfaces* by adding multiple interfaces, adding adaptor, and providing method stubs; c) *Restrict communication paths* so that both the number of modules that consume data produced by the given module, and the number of modules that produce data consumed by it would be reduced; and d) *Use an intermediary* to convert the syntax form of a service.
3) *Defer binding time* to control deployment time and cost. Detailed approaches include: a) *Runtime registration* supports plug-and-play operation; b) *Configuration files* can set parameters at startup; c) *Polymorphism* binds method calls later; d) *Component replacement* implements load time binding; and e) *Adherence to defined protocols* supports runtime binding of independent processes.

Similarly, performance tactics are grouped from three different perspectives [2]:

1) *Resource Demand*: including: a) *Increase computation efficiency* by improving algorithms used in critical areas and resource tradeoffs; b) *Reduce computational overhead* to save time and resource costs; c) *Manage event rate* by reducing the arriving event sampling frequency; d) *Control frequency of sampling* for queued requests; e) *Bound execution times*, including limits on how much execution time and/or how many iterations is/are used to respond to an event; and f) *Bound queue sizes* by controlling the maximum number of queued arrivals.
2) *Resource Management*: manage various resources via a) *Introduce concurrency* so that blocked time can be reduced; b) *Maintain multiple copies* for either data or computations; and c) *Increase available resources* by using higher performance resources such as faster processors, additional memory, and/or faster networks.
3) *Resource Arbitration*: implement a priority assignment and dispatching, i.e., *scheduling policy*.

The hierarchy of modifiability and performance tactics is illustrated in Table 3.

Each concurrency design pattern may implement one or more tactics of multiple qualities. Bass et al. illustrated as an example that the Active Object Pattern implements three modifiability tactics (*Hide Information*, *Use an Intermediary*, and *Runtime Registration*) and two performance tactics (*Introduce Concurrency* and *Scheduling Policy*), and left other researchers to explore the relationship between other design patterns and quality attributes tactics under the framework [2].

Table 2. Concurrency Design Patterns and Related Quality Attributes

Quality Attributes vs. Concurrency Design Patterns		Configurability	Flexibility	Interoperability	Modifiability/Maintainability	Modularity	Performance / Efficiency	Portability	Reliability / Robustness (Race Conditions)	Reliability / Robustness (Deadlock)	Reliability / Robustness (Others)	Simplicity	Standards Compliance	Sustainability	Usability
Structural	Double Buffering						●								
	Lock Object								●			●			
	Producer-Consumer		●		●		●								
	Read/Write Lock				●		●		●						
	Scheduler	●			●		●								
	Thread-safe Decorator				●		●			●					
Behavioral	Active Object		●		●		●					●			
	Balking						●				●				
	Event-based Asynchronous				●		●								
	Future				●										
	Guarded Suspension									●					
	Monitor Object								●			●			
	Reactor	●			●	●		●				●			
	Thread-Specific Storage				●		●					●			
Architectural	Asynchronous Processing						●								
	Half-Sync/Half-Async		●		●	●	●					●			
	Leader/Followers				●		●								
Enterprise	AJAX			●	●		●						●		●
	Lock File								●			●		●	
	Optimistic Concurrency						●								
	Session Object				●									●	
	Static Locking Order									●					
	Thread Pool				●		●								
Other General-Purpose	Double-Checked Locking								?						
	Scoped Locking										●				
	Single Threaded Execution								●						
	Strategized Locking		●		●										
	Two-Phase Termination										●				

We illustrate the mapping between each surveyed concurrency design pattern and the related modifiability and performance tactics in Table 3. Row headings in the table correspond to surveyed concurrency design patterns. Column headings in the table correspond to modifiability and performance tactics. A "●" in a cell means the corresponding concurrency design pattern implements those tactics. For example, the Balking Pattern [4] implements the performance tactic of *bound execution times* because when multiple requests arrive at the same time, it allows one to execute while ignoring the other requests by returning without doing anything. Also this pattern avoids thread waiting and resource waste, which implements the *reduce computational overhead* performance tactic. So there are two "●"'s in the corresponding cells.

Table 3. Mapping between Modifiability and Performance Tactics and Concurrency Design Patterns

Quality Attributes Tactics vs. Concurrency Design Patterns	Modifiability Tactics														Performance Tactics									
	Localize Changes					Prevention of Ripple Effect				Defer Binding Time					Resource Demand						Resource Mgmt.			R. A.
	Maintain Semantic Coherence	Anticipate Expected Changes	Generalize Module	Limit Possible Options	Abstract Common Services	Hide Information	Maintain Existing Interface	Restrict Communication Paths	Use an Intermediary	Runtime Registration	Configuration Files	Polymorphism	Component Replacement	Adherence to Defined Protocols	Increase Computation Efficiency	Reduce Computational Overhead	Manage Event Rate	Control Frequency of Sampling	Bound Execution Times	Bound Queue Sizes	Introduce Concurrency	Maintain Multiple Copies	Increase Available Resources	Scheduling Policy
Double Buffering																					•	•		
Lock Object																								
Producer-Consumer	•					•			•						•						•			
Read/Write Lock					•																•			
Scheduler	•					•																		•
Thread-safe Decorator	•	•				•										•								
Active Object						•			•	•											•			•
Balking																•			•					
Event-based Asynchronous						•															•			
Future	•	•				•																		
Guarded Suspension																								
Monitor Object																								
Reactor	•				•	•			•			•												
Thread-Specific Storage						•		•	•							•								
Asynchronous Processing																					•			
Half-Sync/Half-Async								•	•												•			
Leader/Followers	•				•	•						•			•	•				•	•	•		
AJAX	•															•					•			
Lock File																								
Optimistic Concurrency															•						•	•		
Session Object	•				•																			
Static Locking Order																								
Thread Pool		•														•				•	•	•		
Double-Checked Locking																								
Scoped Locking																								
Single Threaded Execution																								
Strategized Locking										•														
Two-Phase Termination																								

Another good example is the Thread Pool Pattern [5]. The fundamental design decision of this pattern is to improve performance by *introducing concurrency*. A thread pool keeps a number of threads in the pool, which implements the m*aintain multiple copies* performance tactic. This pattern *reduces computational overhead* because it reduces the expense of creating a thread for each task by allowing threads to be reused. Also, this pattern provides a mechanism that ensures the total

number of threads that are running at one time never exceeds a pre-determined maximum. This design reflects a typical *bound queue sizes* performance tactic. Regarding the modifiability tactic, this pattern *anticipates the expected change* that the maximum size of a thread pool may be changed according to environment changes such as amount of memory, so that particular responsibilities may be assigned to manage this expected change.

Regarding the concurrency design patterns that mainly address reliability issues, neither modifiability nor performance is their fundamental design decision. Although they are strongly related to concurrency programming, their purposes are not to improve software performance by introducing concurrency. These concurrency design patterns include the Lock Object Pattern, the Guarded Suspension Pattern, the Monitor Object Pattern, the Lock File Pattern, the Static Locking Order Pattern, Double-Checked Locking Pattern, the Scoped Locking Pattern, the Single Threaded Execution Pattern, and the Two-Phase Termination Pattern.

5. CONCLUSIONS

One of the best practices in multi-core software engineering is applying concurrency design patterns. We comprehensively surveyed 28 concurrency design patterns. A problem-oriented guide, including frequent asked questions and a chart illustrating the relationship between surveyed concurrency design patterns and software quality attributes, was created to navigate the developer to the "right" pattern(s) with minimal search/reading effort. Also, we examined the performance and modifiability tactics, and showed the mapping between these tactics and surveyed concurrency design patterns. These results provide an insight into concurrency design patterns for software developers who are seeking appropriate or improved solutions to multi-threaded software design/coding issues.

6. FUTURE WORK

We would like to investigate the mapping between concurrency design patterns and other software quality attribute tactics, especially reliability, because a significant number of concurrency design patterns deal with how to avoid concurrency defects, such as data racing and deadlock. Additionally, we would like to expand the frequent asked questions in the problem-oriented guide according to deeper communication with more software practitioners.

7. REFERENCES

[1] Bacon et al. 2001. "The 'Double-Checked Locking is Broken' Declaration", http://www.cs.umd.edu/~pugh/java/memoryModel/DoubleCheckedLocking.html.

[2] Bass, L., Clements, P., and Kazman, R. 2007. Software Architecture in Practice. Second Edition, Addison-Wesley, New York (November 2007).

[3] Gamma, E., Helm, R., Johnson, R., and Vlissides, J. 1995. Design Patterns. Addison-Wesley (1995).

[4] Grand, M. 2002. Patterns in Java, Volume 1: A Catalog of Reusable Design Patterns Illustrated with UML. Second Edition, John Wiley & Sons (2002).

[5] Grand, M. 2002. Patterns in Java, Volume 3: Java Enterprise Design Patterns. Second Edition, John Wiley & Sons (2002).

[6] Lavender, G. and Schmidt, D. C. 1995. Active Object - An Object Behavioral Pattern for Concurrent Programming. In Proceedings of the Second Pattern Languages of Programs conference (Monticello, IL, September 6-8, 1995), 483-499.

[7] Lea D. 1999. Concurrent Programming in Java: Design Principles and Patterns. Second Edition, Addison-Wesley (November 1999).

[8] Mahemoff, M. 2006. Ajax Design Patterns: Creating Web 2.0 Sites with Programming and Usability Patterns. O'Reilly. (June 2006).

[9] MSDN. Multi-threaded Programming with the Event-based Asynchronous Pattern. http://msdn.microsoft.com/en-us/library/hkasytyf.aspx

[10] MSDN. PictureBox Class. http://msdn.microsoft.com/en-us/library/system.windows.forms.picturebox.aspx

[11] Sandén, Bo. I. 1997. Concurrent design patterns for resource sharing, In Proceedings of the conference on TRI-Ada (St. Louis, Missouri, USA, 1997), 173-183.

[12] Schmidt, D. C. 1995. Reactor: An Object Behavioral Pattern for Concurrent Event Demultiplexing and Event Handler Dispatching. In Pattern Languages of Program Design, Reading, MA: Addison-Wesley (1995), 529-545.

[13] Schmidt, D. C. 1999. Strategized Locking, Thread-safe Decorator, and Scoped Locking: Patterns and Idioms for Simplifying Multi-threaded C++ Components. C++ Report, SIGS. 11, 9 (September 1999).

[14] Schmidt, D. C., 2000. Monitor Object - an Object Behavior Pattern for Concurrent Programming. C++ Report, SIGS, (October 2000).

[15] Schmidt, D. C. and Cranor, C. 1995. Half-Sync/Half-Async - An Architectural Pattern for Efficient and Well-structured Concurrent I/O. In Proceedings of the Second Pattern Languages of Programs Conference. (Monticello, Illinois, September 6-8, 1995), 437-459.

[16] Schmidt, D. C. and Harrison, T. 1996. Double-Checked Locking - A Optimization Pattern for Efficiently Initializing and Accessing Thread-safe Objects. In Proceedings of the Third Annual Pattern Languages of Programming Conference (Allerton Park, IL, USA, September 4-6, 1996).

[17] Schmidt, D. C., Harrison, T., and Pryce, N. 1997. Thread-Specific Storage - An Object Behavioral Pattern for Accessing per-Thread State Efficiently. In Proceedings of the Fourth Annual Pattern Languages of Programming Conference (Allerton Park, IL, USA, September 2-5, 1997).

[18] Schmidt, D. C., O'Ryan, C., Pyarali, I., Kircher, M., and Buschmann, F. 2000. Leader/Followers: A Design Pattern for Efficient Multi-threaded Event Demultiplexing and Dispatching. In Proceedings of the 7th Pattern Languages of Programs Conference (Allerton Park, IL, USA, Aug. 2000).

[19] Shah, A. 2008. Intel: Programmers Face Multi-Core Challenge. (April 2008) http://www.pcadvisor.co.uk/news/index.cfm?newsid=12594