Identification and Annotation of Concurrency Design Patterns in Java Source Code using Static Analysis

by

Martin Mwebesa

A thesis submitted to the
Faculty of Science
in conformity with the requirements for
the degree of Master of Science
in Computer Science

University of Ontario Institute of Technology
Oshawa, Ontario, Canada
December 2011

Abstract

Concurrent software is quickly becoming a very important facet in Software Engineering due to numerous advantages, one of which is increased processing speed. Despite it's importance, concurrent software is fraught with very difficult to detect bugs, for example deadlocks and data races. Concurrency design patterns were created to offer successfully tried and tested means to design and develop concurrent software to, amongst other things, minimize the occurrence of these hard to detect bugs. In this thesis we discuss our novel static analysis technique to detect these concurrency design patterns in Java source code and identify them using commented Java annotations. Using our technique the commented Java annotations are inserted above Java constructs that are not only part of the Java source code but also make up the various roles that comprise the concurrency design pattern. The identifying of the concurrency design patterns in the Java source code can aid in their maintenance later on, by matching the inserted Java annotations to the various Java constructs they are annotating. Maintaining the concurrency design patterns within the Java source code in effect aids in maintaining the Java source code error free.

Acknowledgments

I would like to thank the University of Ontario Institute of Technology (UOIT) for giving me the opportunity to pursue my Master of Science in Computer Science and funding most of this education for me.

I would like to thank my Examining Committee for taking the time to read my thesis, questioning me fairly on the material within and giving me valuable feedback.

I would like to thank Dr. Jeremy S. Bradbury, my advisor and supervisor, for introducing me to the wonderful and exciting world of scientific research, for challenging me, for mentoring me through my research and those challenges, for introducing me to the realm of concurrent software and for also introducing me to the TXL programming language - of which I am now proficient.

I would like to thank my colleagues in the SQR Group for being part of this journey with me, lending me their ears and also writing some test programs for me.

I would like to thank my very close friend, Dr. Cephas Masikini, for encouraging me to pursue Graduate Studies in my chosen field, Computer Science, and for also giving me encouragement when the going got rough.

I would finally, but by no means least, like to thank my family, Uncle Steven and Mama, for encouraging me along the way and giving me the foundation to pursue rigorous study.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Motivation

In Software Engineering a design pattern can be defined as follows:

"A design pattern systematically names, motivates, and explains a general design that addresses a recurring design problem in object-oriented systems. It describes the problem, the solution, when to apply the solution and its consequences. It also gives implementation hints and examples. The solution is a general arrangement of objects and classes that solve the problem. The solution is customized and implemented to solve the problem in a particular context." [GHJV95]

In other words design patterns can be defined as templates to resolve common software engineering problems. There are various advantages to using design patterns, including the following [GHJV95]:

- Developers benefiting from successfully tried and used designs and architecture.
- Enables reusability of a system by allowing the choice of design alternatives that avoid the comprising of reusability.
- Enables better documentation and maintenance of an existing system by providing

explicit specification of class and object interactions and their underlying intent.

Concurrent software is software where its components do not necessarily run, one after the other as is the case in traditional sequential software. Instead in concurrent software, its various components will run on separate threads hence causing them to complete their processing at different times, non-sequentially. Leveraged properly concurrent software is faster and more versatile than traditional sequential software especially now, in an age where we have multi-core processors that can run numerous threads at the same time.

With the possible increase in processing speed that can occur with the use of concurrency, comes a heavy price of "bugs" that are extremely hard to find. With traditional sequential software you were assured of processes within the software application working sequential, one after the other hence easing the process of finding errors. With concurrent software, it can be difficult to isolate a bug when one is not sure in what order the application threads are running.

Basically, despite the potential power and versatility possible with concurrent software there is difficulty in testing the software for errors as well as more potential to mistakenly inject a concurrency bug. Some common concurrency related bugs include data races and deadlocks which will be elaborated on later in Section 2.2.2. Concurrent software design patterns are design patterns specifically created to alleviate many of these issues starting with the design phase of the concurrent software development.

The increasing importance of concurrent software and the role of design patterns in solving many Software Engineering problems is a major motivating factor for my pursuit of this thesis. These same design patterns are not only beneficial in the design phase of concurrent software's life cycle but can actually be utilized in the maintenance and quality assurance phases. Unfortunately, the use of concurrency design patterns to aid in the maintenance of concurrent software has been in general under researched. As will be discussed in Section 2.3 there has been work on the role of design patterns in software

maintenance [Mef06, SP09] but this has been centered on the traditional object oriented patterns, specifically creational, structural and behavioral design patterns.

1.2 Hypothesis and Problem Definition

One major problem encountered in the use of design patterns during the implementation and maintenance of software is that they can be unintentionally and very easily broken. Sabo, et al. describe the reasons for this as follows:

"It often happens that design patterns can not be identified in source code because the conceptual entity of the pattern at the design level is scattered over different parts of an object or even multiple objects on the implementation level. Intentions of design pattern and system specific intentions are superimposed in implementation and without explicitly distinguishing between sections implementing each of them, it becomes very difficult to identify the constructs constituting the pattern later. Due to the inability to identify individual parts of a pattern, they may be modified inadvertently which may result in breaking the pattern and losing the benefits gained by its application in the system." [SP09]

Keeping this problem in mind, we propose a technique using static analysis with TXL, to identify concurrency design patterns in Java source code and annotate that source code with details regarding the design patterns identified. Java source code annotations are described in Section 2.3.1 of this thesis. These annotations can aid in the maintenance of the design patterns within the Java source code and hence resolve the problem described above. This can be achieved by matching these annotations to the source code they are annotating. If there is a match between the Java annotations and the source code being annotated then the design pattern is not broken, however if the Java annotation and Java source code no longer match then the design pattern has been broken. Our identification and annotation technique using TXL will be elaborated on in Chapters 3 and 4 of this thesis.

1.3 Thesis Organization

The rest of this thesis will be organized into the following 5 Chapters:

- **Chapter 2:** A background of the 8 concurrency design patterns we will be identifying and an overview of existing techniques used in detecting design patterns in general.
- **Chapter 3:** A discussion of our detection technique. This will involve a background into TXL, a pattern based source transformation language that we are using to detect the design patterns. We will also be discussing how we created a detection tool using TXL rules and our refinement of these rules based on a preliminary study.
- **Chapter 4:** A discussion of our Java annotation technique detailing how we implemented these annotations using TXL.
- **Chapter 5:** An overview of the evaluation methodology and evaluation results of our technique. Our empirical evaluation is used to verify that our identification and annotation approach can be the proven solution to Section 1.2
- **Chapter 6:** This will be the conclusion to the thesis where we will summarize the research contributions, limitations of our technique and possible future work overview.

Chapter 2

Background

2.1 Overview

In this chapter we will discuss concurrency design patterns, elaborating on the 8 that we will be detecting and identifying using our technique. We will then discuss Java annotations and give an insight into 4 techniques that use Java annotations to detect and identify design patterns within Java source code. These 4 techniques differ from our technique in that they target the creational, structural and behavioral design patterns whilst our technique targets concurrency design patterns.

2.2 Concurrency Design Patterns

In this section we will define concurrency design patterns and discuss the 8 concurrency design patterns we targeted in our research.

2.2.1 Overview of Design Patterns

In general concurrency design patterns address mainly 2 kinds of problems encountered in concurrency [Gra02]:

1. Accessing of shared resources: Ensuring that the accessing of shared resources occurs one at a time because in concurrent processes there is a possibility that threads can interfere with each other when accessing a resource at the same time. This could cause a deadlock or a data race, both of which will be elaborated on in Section 2.2.2.

2. Controlling the sequencing of operations: Determining in what order shared resources will be accessed.

In the following subsections we will be discussing the following 8 concurrency design patterns [Gra02]:

1. **Single Threaded Execution (also called Critical Section):** This pattern implements guarded methods, which is done in Java using the synchronized keyword.

2. **Lock Object:** This pattern enables a thread to have exclusive access to multiple objects.

3. **Guarded Suspension:** This pattern allows for a process to wait until specific preconditions have been met before processing.

4. **Balking:** This pattern allows for a thread to stop processing completely if a condition has not been met.

5. **Scheduler:** This pattern allows for the ordering of threads in concurrent software.

6. **Read/Write Lock:** This pattern allows for concurrent reads and exclusive writes to operations in a concurrent software application.

7. **Producer-Consumer:** This pattern allows for coordinated sequential producing and consuming of objects in a concurrent software application.

8. **Two-Phase Termination:** This pattern allows for the orderly shutdown of a thread or process in a concurrent software application.

```
public synchronized void set (Datum d) { datum = d; }
```

Figure 2.1: Illustration of Single Threaded Execution pattern

Other concurrency design patterns exist, including: Double Buffering, Asynchronous Processing and Future, but we will be focusing on just the above patterns as they are sufficient in supporting our research hypothesis.

2.2.2 Single Threaded Execution (Critical Section)

Of the 8 concurrency design patterns we will be detecting, the Single Threaded Execution pattern is the most fundamental as it resolves issues related to shared resources by ensuring that only one thread accesses a resource at a time. This is the most common synchronization scenario and hence the importance placed on this design pattern. This pattern is used by most of the other concurrency design patterns.

The Single Threaded Execution design pattern helps prevent issues (incorrect results) occurring from multiple threads accessing a specific resource (object) at the same time through concurrent calls to a method. This issue is a data race [Gra02].

This prevention is done by implementing guarded methods. In Java this basically means declaring these methods that can be called concurrently but may lead to incorrect results, as synchronized. Figure 2.1 illustrates this. Introduction of deadlocks is a potential issue in implementing this pattern. A deadlock occurs when two threads each have exclusive access to a resource and each thread is waiting for the other to release the resource before continuing [Gra02]. They could potentially wait forever causing the operation to fail to complete.

2.2.3 Lock Object

This design pattern is a refinement of the "Single Threaded Execution" design pattern and it enables a single thread to have exclusive access to multiple objects. To avoid a thread having to obtain a lock on every single object it needs and thus consuming lots of overhead, the solution offered by this design pattern is to have threads acquire a synchronization lock on an object created for the sole purpose of being the subject of locks, before continuing with any operations. This object is referred to as a Lock Object hence the name of the pattern. Figure 2.2 illustrates the use of the Lock Object pattern.

```
import java.util.ArrayList;
public abstract class AbstractGameObject {
  private static final Object lockObject = new Object();
  //...
  //...
  private boolean glowing; //True if this object is glowing.
  //...
  //...
  public static final Object getLockObject (){
    return lockObject;
  }
  //...
  //...
  public boolean isGlowing() {
    return glowing;
  }

  public void setGlowing (boolean newValue){
    glowing = newValue;
  }
}

class GameCharacter extends AbstractGameObject {
  //...
  private ArrayList<E> myWeapons = new ArrayList();

  public void dropAllWeapons (){
    synchronized (getLockObject()){
      for (int i = myWeapons.size()-1; i>=0; i--){
        ((Weapons)myWeapons.get(i)).setGlowing(true);
      }
    }
  }
  //...
}
```

Figure 2.2: Illustration of Lock Object pattern [Gra02]

There are various ways that a Lock Object can be incorporated into a program. One common way this is implemented is by creating a static method in the class e.g. getLockObject() [Gra02] that returns a lock (the sole lock). Subclasses of this class call this method to get the Lock Object to synchronize operations in the program.

Using this pattern one can ensure that only one thread at a time is accessing a set of objects without too much performance overhead and code complexity. A shortcoming of this pattern is that an inefficient use of resources could occur when using the Lock Object. This occurs because using the Lock Object results in an operation getting exclusive access to all objects, some of which it may not actually be using but may be needed by other operations. These other operations could have otherwise executed concurrently but are now forced to wait to get access to the Lock Object before they proceed.

2.2.4 Guarded Suspension

The Guarded Suspension design pattern is used in a situation where a pre-condition exists that prevents a method from doing what it is supposed to do. This pattern allows for the execution of the method to be suspended until those conditions have been met.

A good illustration of when the Guarded Suspension design pattern could be useful is in the push and pull method of a queue (see Figure 2.3) [Gra02]. The push() method adds objects to the queue while the pull() method removes objects from the queue.

Both methods are synchronized using the synchronized construct so that multiple threads can safely make concurrent calls to them. Because both methods are synchronized a problem could occur when the queue is empty and a call is made to the pull() method. The pull() method waits for the push() method to provide it with an object to pull, but because they are synchronized, the push() method cannot occur until the pull() method returns. The pull() method will never return until the push() method executes. This condition is an example of a deadlock as described in Section 2.2.2, one of the most common concurrency related bugs. The solution here would be to add an IsEmpty() precondition which when true would

```
class Queue{
   public synchronized pull(){
            while (isEmpty()){ //the precondition
                  wait();
            }
            //...
            //...
  }
 public synchronized void push(){
        //...
        notify();
 }
}
```

Figure 2.3: Illustration of Guarded Suspension design pattern [Gra02]

cause the execution of the pull() method to be suspended as long as the queue is empty. This solution is the classical implementation of the Guarded Suspension design pattern.

The use of wait() and notify() methods which all classes in Java inherit from the Object class are used in the implementation of this pattern. The use of notifyAll() notifies all waiting threads unlike notify() which just selects one waiting thread to notify.

The wait() method when called causes a thread to release the synchronization lock it holds on the object in which it is called. The thread that calls the wait() method is suspended or put on hold until the thread is notified that it can continue through a notify() or notifyAll() method call. At that point the thread attempts to recapture the lock and when it does the wait() returns, hence allowing the method from which it was called to proceed.

```
public class Flusher {
 private Boolean flushInProgress = false;
 public void flush(){
      synchronized(this){
    // ensures only 1 call will proceed normally and
    // concurrent calls will balk.
          if (flushInProgress)
               return;
          flushInProgress = true;
      }
      //...//rest of the code to start the flush goes here.
 }
 void flushCompleted(){
      flushInProgress = false;
 }
}
```

Figure 2.4: Illustration of Balking pattern [Gra02]

The Guarded Suspension design pattern is related to the Balking and Two Phase Termination design patterns discussed further below.

2.2.5 Balking

The Balking design pattern allows for an object's method or methods to return without completing if the object is not in an appropriate state to execute the method.

A good example of the design pattern is an automatic toilet flusher [Gra02]. This kind of flusher usually has 2 ways of flushing, one being a light sensor and the other being a manual flusher. If both calls happen concurrently there would be various courses of action:

1. Start a new flush immediately.

2. Wait until the current flush completes and then start a new flush.

3. Do nothing.

The third choice listed above is referred to as "Balking" and occurs when a method handles a situation by returning without performing its normal function. Figure 2.4 illustrates the Balking pattern using the flusher example discussed above. The Balking design pattern is related to the following design patterns we have discussed previously:

- Single Threaded Execution design pattern, which implements synchronization on the object by simply using the Java synchronized construct.
- Guarded Suspension design pattern, which offers an alternative course of action - waiting, when an object is in an inappropriate state to execute a method.

2.2.6 Scheduler

The Scheduler design pattern allows for the controlling of the order in which threads are scheduled to execute single threaded code. The pattern achieves this by using an object that explicitly sequences the waiting threads. Basically, this pattern provides a mechanism to implement a scheduling policy independent of any specific scheduling policy provided by the operating system. The scheduling policy is encapsulated in its own class making it easily reusable.

An illustration of when the Scheduler design pattern would be useful is in building security software [Gra02]. Consider the scenario where all people accessing a building have a badge. When a person scans the badge at any security checkpoint, the acceptance (allowing entry) or rejection of the badge is printed on a hard copy log in a central area. A concurrency problem could potentially occur if people go through three or more checkpoints at the same or about the same time. To elaborate, as the first log is printing the other two calls to the printer must wait and when that first print job completes there is no guarantee in what order the other two logs will print. The use of a Scheduler would help alleviate this issue. Now that we have provided a motivating example we will discuss the different class objects required by this design pattern:

- **Scheduler Object:** Instances of the Scheduler class, schedule Request objects for processing by a Processor object. For reusability, the Scheduler object is encapsulated and is unaware of the Request class it schedules. The Scheduler object accesses Request objects through the ScheduleOrdering interface. The Request classes implement this ScheduleOrdering interface. The Scheduler object is responsible for deciding when the next Request will run but not the order in which the Requests will occur. The order in which the Requests will occur is determined by the ScheduleOrdering object. A Scheduler class example is illustrated in Figure 2.5.

- **Request Object:** The Request object implements the Schedule Ordering interface and encapsulates a request for a Processor object to compute. An example of a Request object is given in Figure 2.6.

- **Schedule Ordering Object:** An example of the Schedule Ordering class is given in Figure 2.7. As mentioned previously, the Request objects implement this interface for two primary reasons:

  - Because processor objects refer to this interface they avoid a dependency on the Request class.
  - Reusability is further increased in that the Scheduler objects call methods defined in this interface that make the decisions on which Request objects will be processed next.

- **Processor Object:** Instances of the Processor class perform a computation described by a Request object, as defined above. More than one Request object may be presented to the processor to process at a time - concurrency. The Processor object delegates the scheduling of these request objects' processing to the Scheduler to occur one at a time. An example of the Processor object is illustrated in Figure 2.8.

```
public class Scheduler {
  private Thread runningThread;
  private ArrayList waitingRequests = new ArrayList();
  private ArrayList waitingThreads = new ArrayList();
  //enter method is called before the thread starts using
  //a managed resource and does not return until the
  //managed resource is not busy.
  public void enter (ScheduleOrdering s) throws InterrupedException {
       Thread thisThread = Thread.currentThread();
       synchronized (this) {
            if (runningThread == null) {
                 runningThread = thisThread;
                 return;
            }
            waitingThreads.add(thisThread);
            waitingRequests.add(s);
       }
       synchronized (thisThread) {
            while (thisThread != runningThread) {
                 thisThread.wait();
            }
       }
       synchronized (this) {
            int i = waitingThreads.indexOf(thisThread);
            waitingThreads.remove(i);
            waitingRequests.remove(i);
       }
  }
 }
 //Call to the done method indicates current thread is
 //finished with the resource
 synchronized public void done () {
       if (runningThread != Thread.currentThread())
           throw new IllegalStateException (Wrong Thread)
       //...
       //...
       //...
      runningThread = (Thread) waitingThreads.get(next);
      synchronized (runningThread) {
           runningThread.notifyAll ();
      }
 }
}
```

Figure 2.5: Illustration of the Scheduler object of the Scheduler pattern [Gra02]

```
public class JournalEntry implements ScheduleOrdering {
  //...
  private Date time = new Date();
  //...
  //Returns time this journalEntry was created.
  public Date getTime() { return time; }
  //...
  //Returns true if given request should be scheduled before this one.
  private boolean scheduleBefore (ScheduleOrdering s) {
       if  (s instanceof JournalEntry)
            return getTime().before(((JournalEntry)s).getTime());
       return false;
 }
```

Figure 2.6: Illustration of the Request object of the Scheduler pattern [Gra02]

```
public interface ScheduleOrdering {
    public Boolean scheduleBefore (ScheduleOrdering s);
}
```

Figure 2.7: Illustration of the Schedule Ordering interface of the Scheduler pattern [Gra02]

```
class Printer {
  private Scheduler Scheduler = new Scheduler();
  public void print (JournalEntry j) {
       try {
            Scheduler.enter(j);
            try {
                 //...
            } finally {
                 Scheduler.done();
            }
       } catch (InterruptedException e) {
       }
  }
}
```

Figure 2.8: Illustration of the Processor object of the Scheduler pattern [Gra02]

2.2.7 Read/Write Lock

The Read/Write Lock design pattern allows for concurrent read access to an object but exclusive access for write operations. An example where this would be useful is in an online auction system where multiple users can read the current bid but only one user can update the bid at a time [Gra02]. In other words, concurrent reads of data are allowed but only single threaded access to data is allowed when updates are to be made. There are two main classes in the Read/Write Lock design pattern:

1. **A data class:** For example a public class Bid () that has get() and set() methods to read and write bids respectively.

2. **A corresponding ReadWriteLock class:** For example a public class ReadWriteLock() that has readLock() and writeLock() methods which respectively set read and write locks on the method's calling thread. Figures 2.9 and 2.10 illustrate both of these methods. The ReadWriteLock class also has a done() method that when called will relinquish the read or write lock held by the thread. This method is also illustrated in Figure 2.11.

Within the data class there will be an instance of the ReadWriteLock object e.g. private ReadWriteLock lockManager = new ReadWriteLock(); When the data class's get() method to read a bid, is called the first thing it will do is call the readLock() method of the instance of the ReadWriteLock object e.g. lockManager.readLock(); and will not proceed until the lock is obtained. Obtaining this lock ensures that it is safe to get data from the object. The mechanics within the ReadWriteLock object ensure that while any read locks are outstanding i.e. they have not been relinquished; no write locks will be issued. Also, if any write locks were still outstanding the call to obtain the read lock would not return until all write locks were relinquished. This ensures the data being read has the latest update.

Likewise, the set() method will first call the writeLock() method before it proceeds with its execution. This ensures that it is safe to update/store the data. The mechanics within

```
synchronized public void readLock() throws InterruptedException {
  if (writeLockedThread != null) {
    waitingForReadLock++;
    while (writeLockedThread != null){
      wait();
    }//while
    waitingForReadLock--;
    //waitingForReadLock++;
  }//if
  outstandingReadLocks++;
}//readLock()
```

Figure 2.9: Illustration of the ReadLock() method of the Read/Write Lock pattern [Gra02]

the ReadWriteLock object ensures that no write lock can be obtained until all read and write locks have been relinquished.

When the get() and set() methods are completed with their processing the last thing they do before returning is relinquish their respective read and write locks by calling the Done() method of the instance of the ReadWriteLock object e.g. lockManager.Done(). The Read/Write Lock design pattern is most related to the following patterns:

- The Scheduler design pattern, as the Read/Write Lock design pattern is simply a specialized Scheduler.
- The Single Thread Execution design pattern, as the Single Threaded Execution pattern is just a simpler alternative that can be used if most access to the data are writes not reads.

```
public void writeLock() throws InterruptedException {
  Thread thisThread;
  synchronized (this) {
    if (writeLockedThread==null && outstandingReadLocks==0){
      writeLockedThread = Thread.currentThread();
      return;
    }//if
    thisThread = Thread.currentThread();
    waitingForWriteLock.add(thisThread);
  }//synchronized(this)
  synchronized(thisThread){
    while (thisThread != writeLockedThread){
      thisThread.wait();
    }//while
  }//synchronized (thisThread)
  synchronized(this){
    waitingForWriteLock.remove(thisThread);
  }//synchronized (this)
}//writeLock
```

Figure 2.10: Illustration of the Writelock() method of the Read/Write Lock pattern [Gra02]

```
synchronized public void done(){
    if(outstandingReadLocks > 0)
    {
      outstandingReadLocks--;
      if(outstandingReadLocks == 0 && waitingForWriteLock.size() > 0)
      {
        writeLockedThread = (Thread)waitingForWriteLock.get(0);
        writeLockedThread.notifyAll();
      }//if
    }
    else if (Thread.currentThread() == writeLockedThread)
    {
      if(outstandingReadLocks == 0 && waitingForWriteLock.size()>0){
        writeLockedThread = (Thread)waitingForWriteLock.get(0);
        writeLockedThread.notifyAll();
      }
      else
      {
        writeLockedThread = null;
        if(waitingForReadLock > 0)
          notifyAll();
      }//if
    }
    else
    {
      String msg = "Thread does not have lock";
      throw new IllegalStateException(msg);
    }//if
}//done()
```

Figure 2.11: Illustration of the done() method of the Read/Write Lock pattern [Gra02]

```
//Producer class that produces the troubletickets.
public class Client implements Runnable {
      private Queue myQueue;
      //...
      public Client(Queue myQueue) {
            This.myQueue = myQueue;
            //...
      }
      //...
      public void run() {
            TroubleTicket tkt = null;
            //...
            myQueue.push(tkt);
      }
}
```

Figure 2.12: Illustration of the Producer object of the Producer-Consumer pattern [Gra02]

2.2.8 Producer-Consumer

The Producer-Consumer design pattern allows for objects or information to be produced or consumed sequentially in a coordinated manner. A good example of when this is useful is a ticketing system where numerous tickets are being submitted through a client system (the producer) and on the other end there is a dispatcher system (the consumer) that analyzes the tickets and sends them to the appropriate destinations for a resolution [Gra02].
The Producer-Consumer design pattern requires three main objects:

- **Producer Object:** A Producer class which supplies (produces) the objects to be consumed by the consumer class. There will be cases where there are no instances of the consumer class to consume the instance of the producer so, the producer objects are always placed in a queue. Figure 2.12 illustrates the Producer object.

- **Queue Object:** A Queue class which serves as the buffer between the producer and consumer classes. The producer objects are placed in a queue object and remain there until a consumer object pulls them out. Figure 2.13 illustrates this Queue object.

- **Consumer Object:** The Consumer class uses i.e. consumes, objects produced by the producer objects. As described above they pull these objects from the queue. If

```
//the Queue class buffering the consumer and producer class instances.
private class Queue {
     private ArrayList data = new ArrayList();
     //Putting objects in the queue (will be used the producer)
     synchronized public void push (TroubleTicket tkt) {
          Data.add(tkt);
          Notify();
     }
     //Pulling objects from the queue (will be used by the consumer)
     synchronized public TroubleTicket pull() {
          While (data.size() == 0) {
               Try {
                    Wait ();
               } catch (InterrupteException e) {
               }
          }
          TroubleTicket tkt = (TroubleTicket)data.get(0);
          Data.remove(0);
          Return tkt;
     }
     public int size() {
          Return data.size();
     }
}
```

Figure 2.13: Illustration of the Queue object of the Producer-Consumer [Gra02]

```
//Consumer class that consumes the troubletickets.
public class Dispatcher implements Runnable {/
     private Queue myQueue;
     //...
     public Dispatcher (Queue myQueue) {
          This.myQueue = myQueue;
     }
     //...
     public void run() {
          TroubleTicket tkt = myQueue.pull();
          //...
     }
}
```

Figure 2.14: Illustration of the Consumer object of the Producer-Consumer pattern [Gra02]

the queue is empty the consumer object must wait, i.e. it will not return, until the producer object puts an object in the queue. The consumer object is illustrated in Figure 2.14.

The Producer-Consumer pattern is related to the following design patterns:

- The Guarded Suspension design pattern which is used to handle the situation where the consumer objects want to get a producer object from an empty queue and according to the rules of the Guarded Suspension design pattern will wait. Also, there are situations where a max-size is implemented on the queue. In such cases when the max-size is reached the producer objects will wait for available space before being added to the queue, a pre-condition.

- The Pipe [Gra02] design pattern which is a specialized Producer-Consumer involving one producer object that is usually referred to as the data source and one consumer object usually referred to as the data sink.

- The Producer-Consumer design pattern can be viewed as a specialized Scheduler design pattern in that it has a scheduling policy. This policy is based on resource availability and this Scheduler does not need to regain control of the resource to reassign it to another thread when the current thread is done.

2.2.9 Two-Phase Termination

The Two-Phase Termination design pattern provides functionality to shutdown a thread or process in an orderly manner. It allows for various cleanup processes that are required to occur before a system actually shuts down. This is achieved by using a latch as a flag at specific points in the execution of the thread or process.

A good example of when this pattern is useful is in a stock trading client-server system [Gra02]. The server is responsible for sending stock information to any client that connects to the server and indicates that they are interested in certain stocks. When the stock information changes the server updates the interested clients about this change. The server propagates this information via a thread it creates for each individual client.

The server is also responsible for administrative tasks like shutting down the entire server, disconnecting a client by shutting down the thread servicing the client and releasing all related resources used by the thread.

Figure 2.15 illustrates how the Two-Phase Termination design pattern works. First, the Session object's run() method is called. This run() method calls the session object's initialize() method and then repeatedly calls the Portfolio object's sendTransactionToClient() method within a loop that first checks the thread's isInterrupted flag. This isInterrupted flag is the latch and will always return false until the session object's interrupt() method is called. The interrupt() method sets the thread's interrupted flag to true. The session object's run() method is often handled by a different thread from the one that will do the shutdown. The shutdown thread calls the session's interrupt() method using myThread.interrupt(); and in so doing sets the latch to true. The threads in use here do not have to be synchronized as setting the flag is idempotent [Gra02], i.e. irrespective of which thread sets it the flag (latch) will still get set to true.

```
//Session class - performs servers stock info
//transmission and thread termination.
public class Session implements Runnable {
    //Thread to communicate with specific client.
     private Thread myThread;

     //Object containing stock information.
     private Portfolio portfolio;
     private Socket mySocket;
     //...
     public Session (Socket s) {
          myThread = new Thread (this);
          mySocket = s;
          //...
     }
     public void run () {
         initialize ();

         //checking the value of the latch
         while (!myThread.interrupted()) {
            //constant updates to client
             Portfolio.sendTransactionsToClient(mySocket);
          }
     }

     //this method sets the latch to true
     public void interrupt(){
        //setting the latch to true
          myThread.interrupt();
     }
     private void initialize() { //... }
     private void shutdown() { //... }
}
```

Figure 2.15: Illustration of the Two-Phase Termination pattern [Gra02]

2.3 Existing Design Pattern Detection Techniques using Java Annotations

A number of design pattern detection techniques exist but only a couple use Java annotations in their approach. In this section we will start by discussing what Java annotations are, then we will discuss 4 techniques that exist in detecting design patterns using Java annotations. As mentioned in Section 2.1, these 4 detection techniques differ from our technique in that they target the creational, structural and behavioral design patterns whilst our technique targets concurrency design patterns. Creational design patterns are design patterns that abstract away the instantiation process making a system independent of how its objects are created, composed and represented [GHJV95]. Structural design patterns are concerned with with how classes and objects are composed to form larger structures [GHJV95]. Section 2.2 gives a detailed description of concurrency design patterns.

2.3.1 Java Source Code Annotations

"Annotations are metadata or data about data. Annotations are said to annotate a Java element. An annotation indicates that the declared element should be processed in some special way by a compiler, development tool, deployment tool, or during runtime." [Jam05] Java annotations were introduced in the Java 2 Platform Standard Edition 5.0, also known as "Tiger" as part of JSR175 [Mic06].

As described above because Java annotations allow for the definition of actual Java elements they can be very helpful in the identification of various Java elements. For our purposes we would want these Java annotations to identify Java elements that comprise the various roles of specific concurrency design patterns.

A powerful aspect of Java annotations are custom annotations which provide a facility to define and implement one's own annotation. Custom annotations and their usage will

be elaborated on in Section 2.3.2 below. We also create custom annotations in our technique. Our annotation specifications are elaborated on in Chapter 4, Section 4.2 and our implementation of them using TXL is in Section 4.3.

Java annotations can only be used at the class declaration, field declaration and method declaration levels. Unfortunately because Java annotations can currently not be placed at the statement level they would not completely identify the design pattern intent within the Java source code. We have therefore opted to comment our Java annotations to allow us to place them anywhere in the Java source code including the statement level above Java constructs like loops, if-statements, return statements, wait statements, notify statements, to mention but a few.

As per JSR 308 [Ern10] there is a possibility that functionality to support Java annotations at the statement level will come into being.

2.3.2 Sabo and Poruban Approach [SP09]

This detection technique was designed to address the problem that occurs when soon after development begins, issues occur in the traceability of the design pattern due to design patterns being a concept that stretches over different parts of an object or even over various objects. This leads to the high potential that the design pattern implementation of the source code will unintentionally get broken during modifications to the source code hence losing the design patterns structure. The main objectives in resolving this problem are as follows:

1. Enable the clear identification of language constructs that represent the design patterns at the system implementation level.

2. Create the capability to determine whether a pattern is applied correctly.

3. Help in resolving broken patterns.

The above objectives are realized in Sabo and Poruban's approach through the use of Java annotations. As discussed earlier, Java 5 introduces built in annotating functionality which includes built in annotations like JSR175 [Mic06] and custom annotations. These Java annotations help resolve the traceability issues described above as they can be read and processed by development and deployment tools.

Collections of different classes and other constructs within the classes at the implementation level make up the design pattern. These classes get lost because often they implement both the intentions of the design pattern and the intentions of the system. There are in most cases many more system specific intentions than there are design pattern specific intentions. Annotations are useful here because they can be applied to very many source code elements and constructs. The approach here is to use custom annotations to label those constructs that make up the design pattern so that they are easily distinguishable from the rest of the classes and other constructs in the source code.

The aim of using annotations in this approach is to create a set of constraints or rules to ensure that a design pattern already applied to the source code is not broken. In this approach it is assumed that the design pattern has already been used in the source code. Within Java the design pattern constructs are formalized in terms of the following:

1. Classes, interfaces and relationships between the constructs.

2. Enumerations and annotations.

3. Constructors, methods and member variables.

4. Modifiers, parameter lists, return types and exception lists.

The approach here is to store the design pattern information directly into the source code using the Java annotations specifically because annotations are valid language constructs that can express any information.

At a high level this approach will basically, put the constraints or rules, as defined above, for each design pattern construct into the code using annotations. This can be done either manually by the programmer or through a tool that generates the source code, if one is being used. During the compilation process of the code those entities in the code that use the design patterns are compared against the constraints defined in their annotations. If they do not comply with the constraint then the design pattern is considered to be broken. The developer is then notified and provided with possible recommendations based on the specific constraint in the annotation that was broken.

The annotating of the design patterns in the code is achieved in this approach by first and foremost creating a specific annotation type for each design pattern. Constraints for the design pattern are set in the "TargetDesignator" which is a meta-annotation (annotation type definition) and is a single parameter taking an array of constraints. The source code construct being annotated with this annotation must conform to these constraints in order to be considered as correctly conforming to the design pattern that the annotation represents.

These structural constraints are expressed in an extensible language created specifically for its ease in automated processing, which is usually done during compilation as described earlier but can theoretically be done anytime, and its ease in being read by programmers.

Because design patterns can be implemented in different ways, the annotation type definition should be adjustable. This is easy to do in this approach by adjusting the appropriate constraint(s) for different implementations of the design pattern. The ID and role parameters are introduced to aid in identifying different implementations of the design pattern [HLH06]. ID is simply the unique identifier for each distinct implementation of the design pattern.

2.3.3 Meffert Approach [Mef06]

Like in the Sabo and Poruban Approach, the Meffert Approach uses Java annotations to detect design patterns but does not use the built in annotations introduced by JSR175 [Mic06]

in Java 5. The reason for this is that these annotations are too rigid since they can only be used in source code at the declaration level not the statement level. Due to the fact that an annotation can only be applied to one declaration the functionality and expressiveness of the annotations is limited. In Meffert's approach the Java custom annotations are used and accomplish the following goals:

1. Express the source code intentions in the system.

2. Express design pattern intentions in the system.

These annotations can then be used to give advice on a suitable design pattern to implement by basically matching source code intentions to design pattern intentions. This is important because of the complexity of design patterns and their ever increasing number.

The annotation of a section of source code is determined by the developer. First of all only code that fits within the context of the problem (as viewed by the developer) will be considered for the annotation. The annotation can be referred to as the developer's expression of the source code intention.

The design pattern intentions are not in the source code but are in a separate file. Unlike in the Sabo and Poruban Approach [SP09], where design pattern intentions are stored directly in the source code using the Java annotations. The aim in Meffert's approach is to have the design pattern intentions (i.e. the constraint semantics of the design pattern), match up with the source code annotations which are representative of the source code intentions. This is achieved by:

1. Ensuring that the same syntax is used for both the design pattern intentions and the source code intentions.

2. Describing the design pattern intentions in a set of tuples where each tuple contains an intention that corresponds to one or many of the source code annotations.

Other Java annotation application possibilities discussed in this approach include:

1. Selection of design patterns using a tool that will use the above described functionality.

2. Automatically applying the Java annotations in the code which contain the source code intentions as described earlier.

3. Applying selected design patterns to the code.

2.3.4 He, Li and He Approach [HLH06]

The aim in this approach is to resolve traceability issues related to design pattern information in source code by using Java annotations. Two approaches to extract the design pattern information contained within the Java annotations in the Java programs are discussed in this research:

1. Visualization of design pattern instantiation information in a Java program.

2. Automatic checking of the structural properties of a pattern instance in a Java program.

The source code is manually annotated with the design pattern information.

To extract and visualize the design pattern instantiation information, the reflection feature in Java is used. Reflection in the Java programming language can be defined as *"a feature that allows an executing Java program to examine or 'introspect' upon itself, and manipulate internal properties of the program"* [McC98]. Use of the reflection functions of Java enable obtaining related information from the Java classes and methods then, using the getAnnotations() method to extract this annotated information. This information can then be visualized in different formats (e.g. using HTML web pages).

Part of this approach involves checking the structural properties of the design pattern instance(s) in the Java program. This check is made to determine whether the application implemented all the roles contained with the design pattern as well as all the methods with the role(s). This is done by comparing the design pattern information extracted

from the code and comparing it with the design pattern meta-model. This involves the following [HLH06]:

1. Extracting the values from the @PatternRoleImplementation, @PatternPropertyImplementation and @PatternMethodImplementation annotations.

2. Comparing the extracted information in (1) above with meta-model of the design pattern. This includes checking for generalization relationships between classes.

3. Checking whether the program has implemented the generalization and association relationships among the corresponding design pattern roles in the meta-model.

4. Checking whether the program code satisfies the constraints in the meta-model.

2.3.5 Rasool, Philippow and Mader Approach [RPM08]

The aim here is the recovery (extraction) of design patterns from legacy systems based on annotations, regular expressions, Sparx Enterprise Architect Modeling tool (EA) and database queries. The approach used in this research helps reduce the traceability problem, similar to some of the other approaches discussed above. Detection of design patterns in huge legacy systems can help with better maintenance of the system.

In this approach fifty annotations have been created that relate to specific design patterns. These can be placed in the code by the developer to help with the documentation and maintenance of legacy systems. This is a manual process done by the developer and not automated. These annotations can be used for both human and machine readability. The human readability part is used for static analysis and the machine readability for dynamic analysis.

Pattern detection is done by matching the Id of the annotation in the source code to the corresponding Id in the annotation.type file.

In summary the design pattern recovery in this approach is achieved as follows:

1. Manual annotation of the source code.

2. Using EA on the source code, the source code model is obtained which includes various relationships between different elements in the code but unfortunately does not include other valuable information required for design pattern discovery (hence the use of regular expressions as explained later).

3. Features of each design pattern are defined and constraints are created from the concatenations of these features.

4. These constraints are translated into SQL queries and regular expressions which will be used to search for the desired design pattern.

5. Extraction of delegation information, aggregation information and other relationships from the source code using regular expressions due to the limitations in the use of EA as hinted on in point (2) above (mainly inability to obtain delegation information, aggregation information and friend relationships between classes).

6. Use of a prototyping tool they have developed (an Add-In with the VS.Net framework) to perform experiments on various examples.

2.4 Summary

In this chapter we have given a background into our research topic of identifying concurrency design patterns in Java source code. In Section 2.2 we elaborated on what concurrency design patterns are. We discussed in detail the 8 concurrency design patterns that we are specifically targeting. In Section 2.3 we discussed 4 techniques currently in use to detect design patterns using Java annotations. Just to reiterate, these approaches target the creational, structural and behavioral design patterns, whilst our approach targets concurrency design patterns.

We also gave an overview of what Java annotations are in Section 2.3, as they play a very important role in identifying the roles that make up the different concurrency design patterns. In the upcoming chapters, specifically Chapters 3 and 4, we will delve into our technique of detecting and identifying concurrency design patterns.

Chapter 3

Concurrency Design Pattern Detection

3.1 Overview

As mentioned briefly earlier in our approach we use a static analysis technique to detect concurrency design patterns. We are targeting specifically 8 concurrency design patterns which were elaborated on in Chapter 2, namely:

1. Single Threaded Execution (also called Critical Section)
2. Lock Object
3. Guarded Suspension
4. Balking
5. Scheduler
6. Read/Write Lock
7. Producer-Consumer
8. Two-Phase Termination

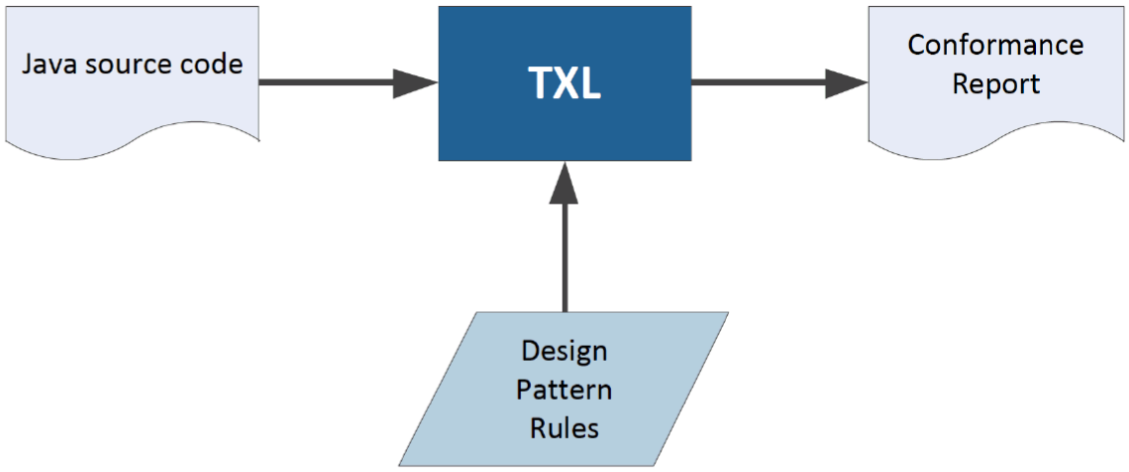

Figure 3.1: Parsing of the Java source code by TXL

We have created 8 TXL programs each of which finds one of the 8 design patterns above. Each of these TXL programs has a set of rules that correspond to the various forces (roles) that determine a specific concurrency design pattern. Figure 3.1 shows the general form in which we are going to use TXL. In brief, the TXL programs will take Java source code which, they will parse based on the design pattern rules established from the design pattern roles and as output, provide conformance reports on whether they were or were not able to detect the design patterns.

In this chapter we will start by providing a background into TXL and why we chose it for our technique. This will be discussed in Section 3.2 below. In Section 3.3 we will discuss our approach in detail, elaborating on how we used TXL in our technique to detect the concurrency design patterns. This will involve a detailed discussion on how we identified the concurrency design pattern roles in the Java source code and created TXL rules to correspond to them and then detect them.

3.2 Background: TXL

TXL is a domain specific language used widely for source code transformation [CCH07]. TXL's basic operation can be described as follows:

"The basic paradigm of TXL involves transforming input to output using a set of transformation rules that describe by example how different parts of the input are to be changed into output. Each TXL program defines its own context-free grammar according to which the input is to be broken into parts, and rules are constrained to preserve grammatical structure in order to guarantee a well-formed result." [CCH07]

A grammar in TXL describes elements in the language that can be transformed. For example the Java.Grm describes the Java programming language as would be used when transforming Java source code. Rules are used to help match elements of the language and transform them to what the user requires. For example, one could transform a set of if statements to case statements. Both of these constructs would be defined in the grammar file and the ".txl" files would contain the rules to do the actual matching of the constructs and then their transformation. Figure 3.2 [CCH07] shows the three phases of the TXL transformation process:

1. Parsing: TXL takes the input and parses it into a tree as defined by the language grammar.

2. Transforming: TXL then transforms the input tree into a new tree using the rules defined in the TXL program.

3. Unparsing: TXL unparses the new tree to produce the desired transformed output.

As noted above, apart from its pattern matching capability, an added advantage of TXL is that it offers parsing for free and does not require the development or use of a separate parser.

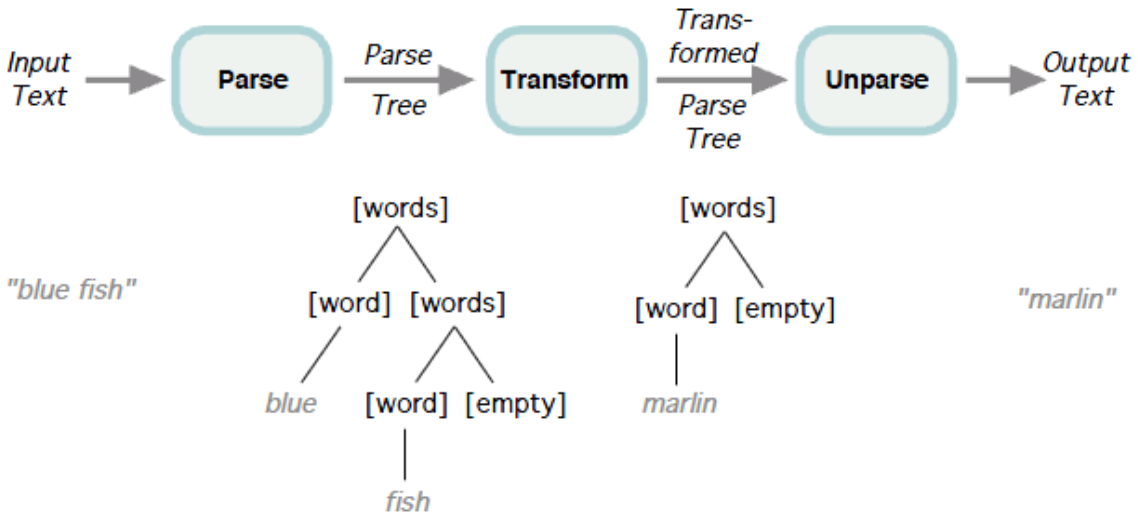

Figure 3.2: Illustration of the three phases of TXL [CCH07]

Given the above features of TXL we decided that it would be an ideal language to enable us to identify concurrency design patterns and transform the source code by adding Java annotations detailing the different roles comprising the design patterns. Our approach will be discussed in more detail shortly but, in brief, we created rules for the different roles that make up the concurrency design patterns. For a specific concurrency design pattern, if all the rules are successfully matched in the Java source code then we deduce that the design pattern exists in that Java program and annotate the code accordingly.

3.2.1 Some important Constructs in the TXL Language

To understand TXL examples in the subsequent parts of this thesis, it is necessary to discuss some of TXLs features in more detail. Specifically we will discuss defines, redefines, functions and rules.

```
redefine variable_declaration
  [variable_declaration2]
    |[attr labelM][repeat modifier][type_specifier]
    [variable_declarators]';[NL]
end redefine

define variable_declaration2
    [repeat modifier][type_specifier][variable_declarators]';[NL]
end define
```

Figure 3.3: Illustration of TXL define and redefine constructs

Defines and Redefines

A **define** is "the basic unit of a TXL grammar" [CCH07]. As the name states it provides a means of actually describing an element in the source code that is going to be parsed and transformed. Figure 3.3, variable_declaration2, shows the definition. Each of those constructs within its definition was in turn defined previously, in this particular case in a Java.Grm file. The Java.Grm file contains most Java source code construct definitions and can be included in individual TXL programs to use for Java source code transformation. In some cases a define can also occur in the TXL file when we want a program to be parsed in a way that aids in the source code transformation.

A **redefine** allows for the overriding of the original define, giving a new description for the element previously defined. As illustrated in Figure 3.3 variable_declaration has been overridden. variable_declaration was initially defined in the Java.Grm file but is now redefined in the TXL file.

Functions and Rules

It is in the function and rule constructs of the TXL program that the matching of elements that were created in the parse tree (during the parsing phase), is done and transformation to a new tree is accomplished. This is achieved by the replace and by clauses in the definition of the function or rule. Functions and rules must at the bare minimum have these two

```
function name
    replace [type]
        pattern
    by
        replacement
end function
```

Figure 3.4: Illustration of a TXL function [CCH07]

```
rule name
    replace [type]
        pattern
    by
        replacement
end rule
```

Figure 3.5: Illustration of a TXL rule [CCH07]

clauses. There are a few exceptions to this however, for example "matching functions" which will be elaborated on later in Section 3.3.3 of this chapter.

The structure of a function and a rule are illustrated in Figures 3.4 and 3.5 [CCH07]. In terms of structure, the function and rule, TXL constructs are identical except for their names. In terms of functionality, the only difference between the two is that a function will search for and replace only the first match it finds, but a rule will search for every match of the pattern of elements and replace them all until no more can be found.

It is in the replace clause that elements to be matched are placed and in the by clauses that the elements that they will be replaced by are put. This whole process will be elaborated on in Section 3.3 of this chapter when we are discussing the various functions and rules created to accomplish our design pattern matching.

Other TXL Constructs

The **attr** construct defines the item following it as being an optional attribute. Figure 3.3 shows an example of the **attr** construct being used. In this case in the statement [attr labelM], it defines labelM as an optional attribute.

The **repeat** construct allows for zero or more repetitions of the item following it to be matched. Figure 3.3 shows an example of the **repeat** construct being used. In this case in the statement [repeat modifier], a modifier can exist zero or more times.

For a comprehensive overview of all TXL language constructs see [CCH07]. In Section 3.3 below, we will introduce and elaborate on a few other TXL constructs that we used in developing our TXL programs to accomplish our goal of detecting concurrency design patterns in Java source code and later adding Java annotations to the source code through TXL's matching and transformation capability, to identify where these design patterns are.

3.3 Approach

Our approach entailed first and foremost identifying the roles that comprise the individual design patterns. After determining these roles we created TXL rules that corresponded to them. As we tested our TXL rules against Java source code examples we ended up revising and refining the rules to make them either more rigid or less rigid in identifying the patterns. This will be elaborated on in Section 3.3.4 of this chapter.

3.3.1 Using TXL for Design Pattern Detection

At a high level Figure 3.6 illustrates our detection technique using TXL. The TXL parser takes the following:

1. Our **TXL programs** in which we have created our rules to detect the patterns.
2. The **Java source code** in which we want to identify the design patterns and annotate.
3. The **Java grammar** file that contains all Java source code elements and constructs.

TXL then outputs the Java source code with the design pattern and its respective roles identified and annotated.

Java Grammar file

The Java grammar file as discussed earlier is a file containing defines for almost all Java source code constructs. Rather than define the numerous Java source code constructs that

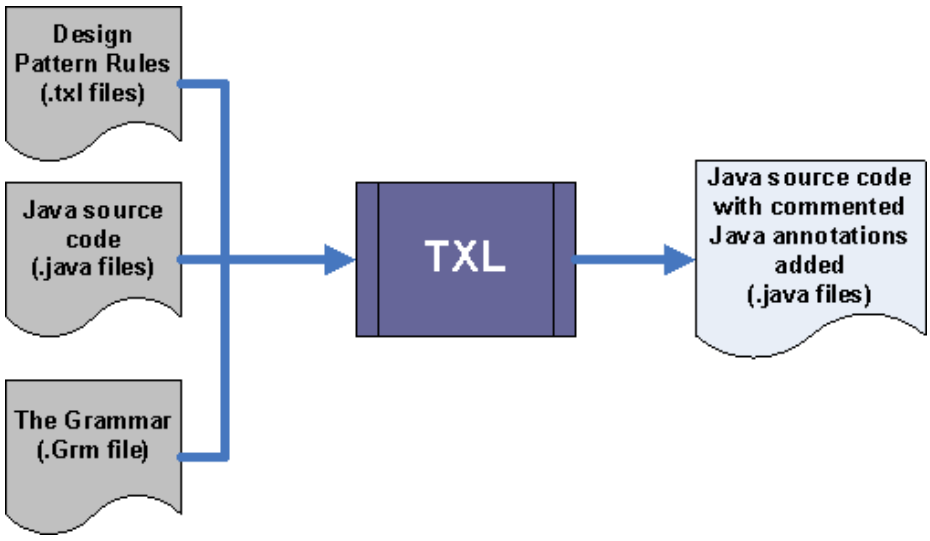

Figure 3.6: Illustration of our Concurrency Design Pattern Detection Technique

we will need in our TXL code for our pattern matching, we have simply used the include statement at the very top of our TXL code to add its defines to our TXL code, as follows: Include "Java.Grm". This grammar file is located in the same location as our TXL code.

Defines and Redefines

To be useful for our design pattern matching, we have had to define new constructs and redefine some of the TXL constructs we included as part of the "Java.Grm" file. The defining and redefining of TXL constructs is done at the top of the TXL file just after the "Java.Grm" file is included.

The first new define we added was labelM, illustrated in Figure 3.7. labelM is used to add the construct MUTATED at the beginning of Java constructs that we match. This is important because most of the pattern matching we are doing is done via the use of TXL "rules". As described earlier "rules" unlike "functions" continue to find a match until none is found. If we do not mutate the construct that is matched by the rule then the TXL program could go into an indefinite loop as the same construct will be matched by the rule

```
define labelM
   'MUTATED
end define
```

Figure 3.7: Illustration of defining new construct, labelM

```
redefine class_declaration
  [class_declaration2]
    | [attr labelM] [class_header] [class_body]
    | [attr labelM] /* [stringlit] */ [NL]
    [class_header] [class_body]
end redefine

define class_declaration2
    [class_header] [class_body]
end define
```

Figure 3.8: Illustration of class_declaration redefine

over and over again.

Therefore, to facilitate the prevention of indefinite attempts at pattern matching we have had to redefine some of the TXL constructs to allow for them to accept the MUTATED keyword at their front. The TXL constructs that we redefined are class_declaration, method_declaration, variable_declaration, while_statement, do_statement and expression_statement. Figure 3.8 illustrates this redefine for the class_declaration TXL construct.

We start by defining a class_declaration2 construct that is the same as class_declaration, the original in the "Java.Grm" file. We then redefine the class_declaration construct. In this case we are saying that a class declaration can be one of the three possibilities as follows:

1. class_declaration2 which is basically what the original class_declaration define was and comprises all the elements that make up a class declaration in Java.

2. The same as (1) above except that it has the keyword MUTATED, that we defined as labelM, in front of it.

3. The same as (1) above except that it has a [NL], representing a new line, in front of

it, any string in front of the [NL] and the keyword MUTATED in front of the string.

3.3.2 Identification of Concurrency Design Pattern Roles

The primary source used to define the concurrency design pattern roles was "Patterns in Java Vol. 1" [Gra02]. Using this text we selected eight of the eleven concurrency design patterns described earlier in Section 2.2.1. For each of these design patterns the text described the specific roles that make up the design pattern.

With this information we proceeded to create a summary document for each design pattern, listing each role that constitutes the pattern. For example for the Single Threaded Execution pattern one role or constraint (actually the only role for it) is that a method has to be synchronized. So, in this case the constraint is that the method has to have the keyword synchronized as one of its modifiers. We ended up refining this role slightly, as will be discussed later in the refinement section of this chapter in regards to the Guarded Suspension design pattern that also implements the Single Threaded Execution design pattern. The tables in Appendix A give an in depth look into all the pattern roles that we identified for each of the eight concurrency design patterns.

From the same text [Gra02] we got specific Java source code examples for each of the eight concurrency design patterns and proceeded to put comments in the source code. These comments were an identification of each role that comprises the pattern and we placed them at the method level for where the role actually occurs. Figure 3.9 illustrates the commenting of the source code with the design pattern roles. This particular example illustrates the Guarded Suspension design pattern.

Because the Guarded Suspension design pattern is a rather elaborate pattern we will be using it as a running example for the rest of the chapter (i.e. for the creation of TXL rules and their refinement, Sections 3.3.3 and 3.3.4 respectively).

```
//*****************************************************//
//*** Guarded Suspension pattern:                   ***//
//*** If a condition that prevents a method from    ***//
//*** executing exists, this design pattern         ***//
//*** allows for the suspension of that method      ***//
//*** until that condition no longer exists .       ***//
//*****************************************************//
import java.util.ArrayList;

public class Queue {
    private ArrayList data = new ArrayList();

    //***Role = 1(Ensuring a method in the class is synchronized.
    //***Contains Role 1a.); ID = 1.
    //***Role = 1a(Ensure there is a notify() or notifyAll() statement.);
    //***ID = 1.
    synchronized public void put(Object obj) {
        data.add(obj);
        notify();
    } // put(Object)

    //***Role = 2(Ensuring a method in the class is synchronized.
    //***Contains Role 2a.); ID = 1.
    //***Role = 2a(Ensuring there is a while statement.
    //***Contains Role 2aa.); ID = 1.
    //***Role = 2aa(Ensuring there is a wait() statement.); ID = 1.
    synchronized public Object get() {
        while (data.size() == 0){
            try {
                wait();
            } catch (InterruptedException e) {
            } // try
        } // while
        Object obj = data.get(0);
        data.remove(0);
        return obj;
    } // get()
} // class Queue
```

Figure 3.9: Illustration of Guarded Suspension design pattern roles in Java source code

3.3.3 Creation of the TXL rules

Once the roles were identified, we proceeded to create actual TXL rules to correspond to the roles. We created a TXL program for each of the eight concurrency design patterns. Each of these TXL programs contains the rules corresponding to the roles identified and listed in Appendix A.

Our first steps for each of the eight TXL programs was to ensure that the TXL rules we

created, correctly corresponded to the roles in the respective concurrency design pattern. We cross checked that the TXL rules matched their respective concurrency design pattern role by creating TXL array-like collections to house the names of the various Java source code components that comprise the design pattern role. These collections would then get populated by these Java source code components as they were located by their respective TXL rules.

After the TXL program completed the detection of the concurrency design pattern, it's last step was to print out these items from the TXL array-like collections to the screen. For example in the case of the Guarded Suspension design pattern we printed out the names of the methods that satisfy Role 1 (i.e. the synchronized methods that have a notify() or notifyAll() statement within them) and Role 2 (i.e. the synchronized methods that have a loop within them and a wait() statement within the loop) to the screen. This way we were certain that the rules did successfully match their respective concurrency design pattern roles.

Main function

The main function in our TXL program to identify the Guarded Suspension design pattern is illustrated in Figure 3.10. We start by declaring global variables using the TXL keyword export. These variables will be used later in our TXL program to, amongst other things, obtain the number of Guarded Suspension design pattern instances and the number of synchronized methods.

After declaring those global variables we use a replace-by clause as is required for all rules and functions. The general practice in TXL program main functions is to replace the program which is represented by the TXL keyword program. Basically the whole program is parsed into a tree and elements within that tree will be replaced by what is contained in the by part of the function. In our case we have created rules, the first being findGuardedSuspensionPattern, that will act on the parsed program tree.

```
function main
  export Counter [number]
    0
  export CountFirstSynchMethIDs [number]
    0
  export CountSecondSynchMethIDs [number]
    0
  export numVarsIDCollection [repeat id]
    -
  export FirstSynchMethIDs [repeat id]
    -
  export SecondSynchMethIDs [repeat id]
    -
    export notifyCollection [repeat expression]
    -
  export waitCollection [repeat expression]
    -
  replace [program]
        P [program]
  by
    P [findGuardedSuspensionPattern] [printPatternNotFound]
        [printOutput] [printFirstSynchMethIDs]
        [printSecondSynchMethIDs] [printNotifyCollection]
        [printWaitCollection]
end function
```

Figure 3.10: Illustration of the main function

The findGuardedSuspensionPattern rule will be elaborated on in the following section but, in summary, it is through this rule that our matching of Java constructs that determine whether the Guarded Suspension design pattern exists, occurs.

The other rules that are called in this main function are the print functions we created, which basically write out our findings to the screen after the pattern matching occurs. We will elaborate on these as well in a later section dedicated specifically to describing them.

Rules Corresponding to Design Pattern Roles

The Guarded Suspension design pattern is comprised of five roles (see Appendix A, Table A.3: Guarded Suspension design pattern Roles, for a description of each). We have created TXL rules that basically correspond directly to these roles.

findGuardedSuspensionPattern rule: The first rule we created and that was introduced

```
rule findGuardedSuspensionPattern
  replace [class_declaration]
      CH [class_header] CB [class_body]
  construct NumVarInstancesFound [class_body]
    CB [findAllNumberVars]
  construct InstanceFoundFirstSynchMeth [class_body]
    CB [find1stSynchMethod1] [find1stSynchMethod2]
  construct InstanceFoundSecondSynchMeth [class_body]
    CB [find2ndSynchMethod1] [find2ndSynchMethod2]
  by
    'MUTATED CH CB
end rule
```

Figure 3.11: Illustration of the findGuardedSuspensionPattern rule

in the previous section is the findGuardedSuspensionPattern rule. This is the rule from which the rules corresponding to the Guarded Suspension design pattern roles emerge. From within it we call the other rules discussed below. This rule is illustrated in Figure 3.11. The rule does a replace right at the class level of the Java program.

findAllNumberVars rule: The first rule called from within the findGuardedSuspensionPattern rule is the findAllNumberVars rule, illustrated in Figure 3.12. This rule does not correspond to a Guarded Suspension design pattern role but simply collects a list of all variables declared within the program and stores them in the global variable numVarsIDCollection. numVarsIDCollection, is one of those global variables that we declared in the main function using the TXL keyword export. As illustrated in Figure 3.12, to populate a global variable from another rule or function, the TXL keyword import is used to first of all obtain the variable and then an export is performed to repopulate it and have its new value available globally.

find1stSynchMethod1 rule: The find1stSynchMethod1 rule illustrated in Figure 3.13 corresponds to role 1 of the Guarded Suspension design pattern and ensures that there is one of two required synchronized methods (guarded methods) in the Java program under analysis. find1stSynchMethod2 is a variation of this rule and will be discussed in the refinement of TXL rules section further below, Section 3.3.4.

hasNotifyOrNotifyAll rule: The hasNotifyOrNotifyAll rule corresponds to role 1a of the

```
rule findAllNumberVars
  construct VARTYPE [type_specifier]
    'short
  construct VARTYPE2 [type_specifier]
    'int
  construct VARTYPE3 [type_specifier]
    'long
  construct VARTYPE4 [type_specifier]
    'float
  construct VARTYPE5 [type_specifier]
    'double
  replace [variable_declaration]
    RM [repeat modifier] TS [type_specifier]
       VDS [variable_declarators] ';
  where
    TS [isVarOfType VARTYPE] [isVarOfType VARTYPE2]
       [isVarOfType VARTYPE3] [isVarOfType VARTYPE4]
       [isVarOfType VARTYPE5]
  deconstruct VDS
    LVD [list variable_declarator+]
  deconstruct LVD
    VN [variable_name] OEVI [opt equals_variable_initializer]
  deconstruct VN
    DN [declared_name] RD [repeat dimension]
  deconstruct DN
    objectID [id] OGP [opt generic_parameter]
  import numVarsIDCollection [repeat id]
  construct newIDCollection [repeat id]
    numVarsIDCollection [. objectID]
  export numVarsIDCollection
    newIDCollection
  by
    'MUTATED RM TS VDS ';
end rule
```

Figure 3.12: Illustration of the findAllNumberVars rule

Guarded Suspension design pattern and determines if there is a notify() or notifyAll() Java statement in the synchronized method being matched in the rules find1stSynchMethod1 and find1stSynchMethod2 from which it is called. This rule is illustrated in Figure 3.14. Figure 3.13 illustrating find1stSynchMethod1, shows how it is called. One important observation to make about this rule is that it takes a parameter. This parameter is of the type method_declarator and basically houses the declaration of the synchronized method being analyzed.

As seen in previous illustrations, including Figure 3.14, the TXL keyword construct is used to create new variables that can be used in the rules for the matching of elements

```
rule find1stSynchMethod1
  construct SYNCH [modifier]
    'synchronized
  replace [method_declaration]
    RM [repeat modifier] TS [type_specifier] MD [method_declarator]
       OT [opt throws] MB [method_body]
  where
    RM [isMethodSynchronized SYNCH]
  deconstruct MB
       BL2 [block]
  deconstruct BL2
    '{
     RDS3 [repeat declaration_or_statement]
    '}
  construct InstanceFound [repeat declaration_or_statement]
    RDS3 [hasNotifyOrNotifyAll MD]
  by
    'MUTATED RM TS MD OT MB
end rule
```

Figure 3.13: Illustration of the find1stSynchMethod1 rule

within the parsed tree or even added to the new trees created via the TXL transformation phase. The TXL keyword deconstruct is just as important in TXL functionality. Unlike construct, deconstruct breaks TXL elements into smaller elements to aid in deeper pattern matching. These new elements created from the deconstruction of larger elements can also be added to new parse trees.

find2ndSynchMethod1 rule: The find2ndSynchMethod1 rule, which is very similar to the find1stSynchMethod1, rule is illustrated in Figure 3.15. This rule corresponds to Role 2 of the Guarded Suspension design pattern and determines whether the second of the two required synchronized methods exists in the Java source code. The find2ndSynchMethod2 rule is a variation of this rule and will be discussed later in Section 3.3.4 of this chapter.

isWhileLpWait rule: The isWhileLpWait rule is called from within rule find2ndSynchMethod1 discussed previously and is illustrated in Figure 3.16. The rule corresponds to both roles 2a and 2aa of the Guarded Suspension design pattern. The purpose of this rule as required by roles 2a and 2aa is to determine whether the second synchronized method being matched has a while loop and whether within the while loop a Java wait() statement exists. Rule

```
rule hasNotifyOrNotifyAll MD [method_declarator]
    replace [expression_statement]
      EX [expression] ';
    construct idNotify [id]
        'notify
    construct idNotifyAll [id]
      'notifyAll
    construct idNotifyExpr [assignment_expression]
      'notify()
    construct idNotifyAllExpr [assignment_expression]
      'notifyAll()
    deconstruct EX
      AE [assignment_expression]
    where
      AE [isAssignmentExpr idNotifyExpr]
        [isAssignmentExpr idNotifyAllExpr]
    deconstruct MD
        MN [method_name] '( LFP [list formal_parameter] ')
        RD [repeat dimension]
    deconstruct MN
      DN [declared_name]
    deconstruct DN
       methodID [id] OGP [opt generic_parameter]
    import FirstSynchMethIDs [repeat id]
    construct newMethodIDs [repeat id]
      FirstSynchMethIDs [. methodID]
    export FirstSynchMethIDs
      newMethodIDs
    import CountFirstSynchMethIDs [number]
    construct PlusOne [number]
      1
    construct NewCount [number]
      CountFirstSynchMethIDs [+ PlusOne]
    export CountFirstSynchMethIDs
      NewCount
    import notifyCollection [repeat expression]
    construct newNotifyCollection [repeat expression]
      notifyCollection [. EX]
    export notifyCollection
      newNotifyCollection
    by
      'MUTATED
      EX ';
    end rule
```

Figure 3.14: Illustration of the hasNotifyOrNotifyAll rule

isDoWhileLpWait is a variation of this rule and will be discussed later in this chapter, in Section 3.3.4.

completeStats function: The completeStats function illustrated in Figure 3.17 is the last of the rules/functions called that originate from the rule findGuardedSuspensionPattern. This

```
rule find2ndSynchMethod1
  construct SYNCH [modifier]
    'synchronized
  replace [method_declaration]
    RM [repeat modifier] TS [type_specifier] MD [method_declarator]
       OT [opt throws] MB [method_body]
  where
    RM [isMethodSynchronized SYNCH]
  deconstruct MB
       BL2 [block]
  deconstruct BL2
    '{
      RDS3 [repeat declaration_or_statement]
    '}
  construct InstanceFound [repeat declaration_or_statement]
    RDS3 [isWhileLpWait MD] [isDoWhileLpWait MD]
  by
    'MUTATED RM TS MD OT MB
end rule
```

Figure 3.15: Illustration of the find2ndSynchMethod1 rule

```
rule isWhileLpWait MD [method_declarator]
  replace [while_statement]
      'while '( EX [expression] ')
      STMT [statement]
  construct waitStmt [statement]
    'wait();
  import numVarsIDCollection [repeat id]
  where
    STMT [hasStmt waitStmt] [hasWaitStmt each numVarsIDCollection]
  construct InstanceFound [method_declarator]
    MD [completeStats MD EX]
  by
    'MUTATED
      'while '( EX ')
    {
      STMT
    }
end rule
```

Figure 3.16: Illustration of the isWhileLpWait rule

function is called from within the isWhileLpWait rule and its variation isDoWhileLpWait.

This rule does not correspond to any of the roles in the Guarded Suspension design pattern. It's sole purpose is to populate many of the global variables that were declared in the main function, with statistics to be printed out. For example:

1. The collection of all Java wait() statements used.

2. The number of both the first and second synchronized methods that are required roles for the Guarded Suspension design pattern.

3. The number of instances of the Guarded Suspension design pattern found in the Java source code being analyzed. This is stored in the Counter variable.

Matching Functions

To aid in our design pattern matching we created functions whose sole purpose is to check for the existence of specific Java constructs. To elaborate on how these matching functions work I will discuss the isMethodSynchronized function that is one of the functions illustrated in Figure 3.18.

The isMethodSynchronized function takes a parameter called SYNCH that is of type modifier and either succeeds if it finds the modifier represented within SYNCH anywhere within the calling construct or fails if it does not.

Figure 3.15, illustrating rule find2ndSynchMethod1, is an example of where the isMethodSynchronized matching function is called. Here the variable SYNCH is populated with the Java construct synchronized which is a valid modifier. This function is called off the construct RM which is a list of modifiers. If synchronized is found to be amongst the modifiers in RM then the function passes, but if synchronized is not amongst the modifiers then the function fails. In Figure 3.18 the isMethodSynchronized function is called as part of a where clause. This ensures that the rule find2ndSynchMethod1 will either proceed or not proceed depending on whether synchronized is or is not found, respectively in the matching function.

Printing Functions

After the rule findGuardedSuspensionPattern is called from within the main function and completes its pattern matching as elaborated on in detail in the last few sections, the print

```
function completeStats MD [method_declarator] EX [expression]
  replace [method_declarator]
    MD
  deconstruct MD
      MN [method_name] '( LFP [list formal_parameter] ')
        RD [repeat dimension]
  deconstruct MN
    DN [declared_name]
  deconstruct DN
     methodID [id] OGP [opt generic_parameter]
  import SecondSynchMethIDs [repeat id]
  construct newMethodIDs [repeat id]
    SecondSynchMethIDs [. methodID]
  export SecondSynchMethIDs
    newMethodIDs
  import waitCollection [repeat expression]
  construct newWaitCollection [repeat expression]
    waitCollection [. EX]
  export waitCollection
    newWaitCollection
  import CountSecondSynchMethIDs [number]
  construct PlusOne [number]
    1
  construct NewCount [number]
    CountSecondSynchMethIDs [+ PlusOne]
  export CountSecondSynchMethIDs
    NewCount
  import CountFirstSynchMethIDs [number]
  construct numZero [number]
  '0
  where not
    CountFirstSynchMethIDs [hasNumber numZero]
  where not
    CountSecondSynchMethIDs [hasNumber numZero]
  import Counter [number]
  construct PlusOneb [number]
    1
  construct NewCountb [number]
    Counter [+ PlusOneb]
  export Counter
    NewCountb
  by
    MD
end function
```

Figure 3.17: Illustration of the completStats function

functions are called. The sole purpose of the print functions as mentioned earlier is to print out messages to the screen. Figure 3.19 and 3.20 illustrate how this is done by using the TXL keyword print.

For the printPatternNotFound function illustrated in Figure 3.19 all we are doing is checking

```
% Function to check if the synchronized modifier is being used.
function isMethodSynchronized SYNCH [modifier]
  match * [modifier]
    SYNCH
end function

% Function to check if the synchronized modifier is being used
% by the "this" keyword.
function isMethodSynchdUsingThis THIS [expression]
  match * [expression]
    THIS
end function

% Function to check if there is a match to a variable ID.
function matchesVarID theID [id]
  match * [id]
    theID
end function
```

Figure 3.18: Illustration of matching functions

```
function printPatternNotFound
  replace [program]
    P [program]

  import Counter [number]

  where
    Counter [= 0]

  construct InstanceFound [stringlit]
    "*** No instances of Guarded Suspension Pattern found. "

  construct InstanceFoundPrint [id]
    _ [unquote InstanceFound] [print]

  by
    P
end function
```

Figure 3.19: Illustration of the printPatternNotFound function

whether the global variable Counter is "0". If it is then that means no instances of the Guarded Suspension design pattern were found and a message to that effect will be printed to the screen.

For the printOutput function illustrated in Figure 3.20, we again check Counter, but this time to see if the value is greater than zero. If it is then that means 1 or more instances

```
% Function print out the number of Guarded Suspension
% design pattern instances found.
function printOutput
  replace [program]
    P [program]
  import Counter [number]
  where
    Counter [> 0]
  construct InstanceFound [stringlit]
    "** Instances of Guarded Suspsension Pattern found = "
  construct InstanceFoundPrint [id]
    _ [unquote InstanceFound] [+ Counter] [print]
  by
    P
end function
```

Figure 3.20: Illustration of the printOutput function

of the Guarded Suspension design pattern were found and a message stating that will be printed to the screen, including the value of Counter.

3.3.4 Refinement of the TXL Rules

After and during the creation of the TXL rules we continuously run the TXL programs against actual Java source code examples found in the text "Patterns in Java Vol. 1" [Gra02]. We used a Java source code example corresponding to each of the concurrency design patterns described in Section 2.2. The only one of these Java source code examples not obtained from the text was the one corresponding to the Single Threaded Execution design pattern. This is because the Single Threaded Execution design pattern does not have the complexity of the other 7 concurrency design patterns we are targeting and can be very easily created (see Figure 2.1 for an example).

Using this process we were able to constantly refine our TXL programs and the TXL rules contained within them, to enable the detection of more Java construct variations for the various design pattern roles. In our refinement of the TXL rules, we not only looked to make the TXL program more general but also wanted to reduce the rate of false positives.

As mentioned in Section 3.3.3, find1stSynchMethod2 and find2ndSynchMethod2 rules are

```
rule find1stSynchMethod2
  construct  THIS [expression]
    'this
  replace [method_declaration]
    RM [repeat modifier] TS [type_specifier] MD [method_declarator]
        OT [opt throws] MB [method_body]
  deconstruct MB
        BL [block]
  deconstruct BL
      '{
      RDS [repeat declaration_or_statement]
    '}
  deconstruct RDS
    STMT [statement]
    RDS2 [repeat declaration_or_statement]
  deconstruct STMT
    SSTMT [synchronized_statement]
  deconstruct SSTMT
      'synchronized '( EX [expression] ')
      BL2 [block]
  where
    EX [isMethodSynchdUsingThis THIS]
  deconstruct BL2
    '{
      RDS3 [repeat declaration_or_statement]
    '}
  construct InstanceFound [repeat declaration_or_statement]
    RDS3 [hasNotifyOrNotifyAll MD]

  by
    'MUTATED RM TS MD OT MB
end rule
```

Figure 3.21: Illustration of the find1stSynchMethod2 rule

variations of find1stSynchMethod1 and find2ndSynchMethod1 respectively. These came about due to a need to refine the matching for a synchronized method. The initial variations simply searched for the Java construct synchronized in the method declaration.

After running our TXL program against other examples of the Guarded Suspension design pattern and failing to find a match, we realized that it was because the methods were being guarded differently. Instead of the Java construct synchronized appearing in the method declaration, it was the Java construct this that was being synchronized. this was basically a reference to the method itself. Figures 3.21 and 3.22 illustrate this refinement in the form of the rules find1stSynchMethod2 and find2ndSynchMethod2 respectively.

```
rule find2ndSynchMethod2
  construct  THIS [expression]
    'this
  replace [method_declaration]
    RM [repeat modifier] TS [type_specifier] MD [method_declarator]
        OT [opt throws] MB [method_body]
  deconstruct MB
        BL [block]
  deconstruct BL
      '{
      RDS [repeat declaration_or_statement]
    '}
  deconstruct RDS
    STMT [statement]
    RDS2 [repeat declaration_or_statement]
  deconstruct STMT
    SSTMT [synchronized_statement]
  deconstruct SSTMT
      'synchronized '( EX [expression] ')
      BL2 [block]
  where
    EX [isMethodSynchdUsingThis THIS]
  deconstruct BL2
    '{
      RDS3 [repeat declaration_or_statement]
    '}
  construct InstanceFound [repeat declaration_or_statement]
    RDS3 [isWhileLpWait MD] [isDoWhileLpWait MD]
  by
    'MUTATED RM TS MD OT MB
end rule
```

Figure 3.22: Illustration of the find2ndSynchMethod2 rule

Figure 3.23 illustrates the isDoWhileLpWait rule which as mentioned earlier is a refinement of the isWhileLpWait rule. As was the case for find1stSynchMethod1 and find2ndSynchMethod1, after running our program against more examples, we determined that the Guarded Suspension design pattern's role 2a, a loop, could occur in various forms. Initially we had just one TXL rule that detected a while statement, as shown in Figure 3.16. So, we created an additional rule, isDoWhileLpWait, as a refinement and variation of it, allowing for do-while loops as well.

```
rule isDoWhileLpWait MD [method_declarator]
  replace [do_statement]
    'do
      STMT [statement]
    'while '( EX [expression] ') ';
  construct waitStmt [statement]
    'wait();
  import numVarsIDCollection [repeat id]
  where
    STMT [hasStmt waitStmt] [hasWaitStmt each numVarsIDCollection]

  construct InstanceFound [method_declarator]
    MD [completeStats MD EX]
  by
    'MUTATED
    'do
    {
      STMT
    }
    'while '( EX ') ';
end rule
```

Figure 3.23: Illustration of the isDoWhileLpWait rule

3.4 Summary

In this chapter we have used the Guarded Suspension design pattern as a running example to illustrate how and why we created the TXL rules; and how these rules corresponded to the roles that we initially identified. We followed the very same process in creating TXL programs for the other seven concurrency design patterns described in Section 3.1. Similar refinements as described in Section 3.3.4 were also required on these seven concurrency design pattern TXL programs after running them against their corresponding Java source code examples.

After numerous updates to our TXL programs we were able to detect all 8 of our targeted concurrency design patterns named earlier in Section 3.1. The next step in our research was to add Java annotations to our Java source code examples using the transformative features of TXL. Adding Java annotations would complete the identification aspect of our research by pointing out, right in the source code, where the various roles that comprise the design patterns exist. This stage in our research will be discussed in Chapter 4.

Chapter 4

Annotation of Design Patterns

4.1 Overview

After completing the development and necessary refinements for the detection of the concurrency design patterns in the Java source code, as discussed in Chapter 3, we proceeded to modify our TXL programs to enable them to actually transform the Java source code examples by adding commented Java annotations to them.

These commented Java annotations identify specifically where in the Java source code the various concurrency design pattern roles that constitute the respective concurrency design patterns exist. Before discussing the implementation of the commented Java annotations in this chapter we will, in Section 4.2, illustrate the Java annotations we created to identify the Guarded Suspension design pattern. A complete list of all the Java annotations we created for all 8 of the concurrency design patterns we targeted, can be found in Appendix B.

Further in Section 4.2, we will illustrate one of our Java source code examples that has been transformed by our TXL rules and now has Java annotations added to identify the presence of a concurrency design pattern and it's roles, specifically the Guarded Suspension design pattern. In Section 4.3, we will discuss our implementation of the commented Java

annotations in the Java source code using TXL and then finally end with a summary of this chapter.

4.2 Annotation Specifications

The custom Java annotations we created, correspond directly to the concurrency design pattern roles illustrated in the tables in Appendix A. Table 4.1 shows the annotation specifications for the Guarded Suspension design pattern. As mentioned earlier, a complete list of the Java annotations we created for all 8 of the concurrency design patterns we are targeting is illustrated in the tables in Appendix B.

Table 4.1: Guarded Suspension Design Pattern Annotation Specifications

Role ID:	Annotation:
1	@GuardendSuspensionPattern(ID=1,role=1,comment="Ensuring a method in the class is synchronized - guarded.")
1a	@GuardendSuspensionPattern(ID=1,role=1a,comment="Ensure there is a nofify() or notifyAll() statement.")
2	@GuardendSuspensionPattern(ID=1,role=2,comment="Ensuring a method in the class is synchronized - guarded.")
2a	@GuardendSuspensionPattern(ID=1,role=2a,comment="Ensuring there is a while statement.")
2aa	@GuardendSuspensionPattern(ID=1,role=2aa,comment="Ensuring there is a wait() statement.")

Figures 4.1 and 4.2 illustrate a Queue class that makes use of the Guarded Suspension design pattern. Figure 4.1 is the class before the commented Java annotations have been added and Figure 4.2 is the same class but after it has been parsed and transformed by our TXL rules. The various roles that comprise the Guarded Suspension design pattern have been identified and commented Java annotations have been inserted in the Java source code to identify where specifically these roles exist within the code.

```
import java.util.ArrayList;

public class Queue {
    private ArrayList data = new ArrayList();

    synchronized public void put(Object obj) {
        data.add(obj);
        notify();
    } // put(Object)

    synchronized public Object get() {
        while (data.size() == 0){
            try {
                wait();
            } catch (InterruptedException e) {
            } // try
        } // while
        Object obj = data.get(0);
        data.remove(0);
        return obj;
    } // get()
} // class Queue
```

Figure 4.1: Illustration of the Guarded Suspension design pattern before Annotations

```
import java.util.ArrayList;

public class Queue {
    private ArrayList data = new ArrayList ();

    /* ``@GuardedSuspensionPatternAnnotation(patternInstanceID=1, roleID=1, roleD
escription='Ensuring a method in the class is synchronized - guarded')" */
    synchronized public void put (Object obj) {
        data.add (obj);
        /* ``@GuardedSuspensionPatternAnnotation(patternInstanceID=1, roleID=1a,
roleDescription='Ensure there is a notify() or notifyAll() statement.')" */
        notify ();}

    /* ``@GuardedSuspensionPatternAnnotation(patternInstanceID=1, roleID=2, roleD
escription='Ensuring a method in the class is synchronized - guarded')" */
    synchronized public Object get () {

        /* ``@GuardedSuspensionPatternAnnotation(patternInstanceID=1, roleID=2a,
roleDescription='Ensuring there is a while statement.')" */
        while (data.size () == 0) {
        {
            try {
                /* ``@GuardedSuspensionPatternAnnotation(patternInstanceID=1, roleID
                    =2aa,
                roleDescription='Ensuring there is a wait() statement.')" */
                wait ();} catch (InterruptedException e) {
            }
        }} Object obj = data.get (0);
        data.remove (0); return obj;
    }
}
```

Figure 4.2: Illustration of the Guarded Suspension design pattern after Annotations

4.3 Implementing Commented Annotations using TXL

To add the commented Java annotations to the Java source code we further refined our TXL rules to allow for the transformation of the Java source code. The transformation is basically the addition of the commented Java annotations above the different Java constructs that make up the different roles of the concurrency design pattern. We will illustrate this further using the Guarded Suspension design pattern which we have already used in Chapter 3 to illustrate various components of our technique.

4.3.1 TXL Rules adding Commented Java Annotations

These TXL rules are the same rules we described in Section 3.3.3 of this thesis, except that they have now been refined to enable the adding of commented Java annotations immediately above the areas where the design pattern roles have been detected in the Java source code. In short these TXL rules have now been modified to add the commented Java annotations above specific Java source code constructs to accurately identify exactly where a design pattern role exists.

findGuardedSuspensionPattern rule

Figure 4.3 illustrates the refined version of Figure 3.11, the findGuardedSuspensionPattern rule. Recall from Section 3.3.3 that it is from this rule that the other rules corresponding to various Guarded Suspension design pattern roles emerge. Namely the following four rules discussed in the next few sections: find1stSynchMethod1; find1stSynchMethod2; find2ndSynchMethod1 and find2ndSynchMethod2.

Other rules emerge from these four rules and correspond to additional Guarded Suspension design pattern roles. All these rules, the part they play in adding the Java annotations to the Java source code and in identifying the exact area where the design pattern roles exist, will be elaborated on shortly in the next few sections.

```
rule findGuardedSuspensionPattern
  replace [class_declaration]
      CH [class_header] CB [class_body]
  construct NumVarInstancesFound [class_body]
    CB [findAllNumberVars]
  construct TransformedClassBody [class_body]
    CB [find1stSynchMethod1] [find1stSynchMethod2]
       [find2ndSynchMethod1] [find2ndSynchMethod2]
  import Counter [number]
  where
    Counter [> 0]
  by
    `MUTATED CH TransformedClassBody
end rule
```

Figure 4.3: Illustration of Transforming findGuardedSuspensionPattern rule

As illustrated in Figure 4.3, these four rules are passed the entire body of the class being examined for the design pattern using the following TXL statement: CB [find1stSynchMethod1] [find1stSynchMethod2] [find2ndSynchMethod1] [find2ndSynchMethod2].

These rules, as will be discussed shortly, transform the class body code by adding the Java annotations. As shown in Figure 4.3, this transformed class body is assigned to a newly constructed variable called TransformedClassBody. TransformedClassBody is then used in the by section of the rule, replacing the original class body, CB.

find1stSynchMethod1 and find1stSynchMethod2 rules

These two rules are simply refinements of Figure 3.13 and 3.21, respectively. Figure 4.4 illustrates the refined rule find1stSynchMethod1. As discussed briefly earlier, the refinement is to allow for these TXL rules to not only identify the respective roles that constitute the design pattern, but also modify the Java source code to add Java annotations identifying exactly where these roles (role 1 in the case of these two rules) that make up the Guarded Suspension design pattern lie.

As discussed in Section 3.3.3, these two rules correspond to role 1 of the Guarded Suspension design pattern which simply ensures that a method in the class is synchronized and contains role 1a which, is a notify() or notifyAll() statement within the synchronized method. The difference between these 2 rules (find1stSynchMethod1 and find1stSynchMethod2) is that they simply handle the two different ways a synchronized method can be defined in Java - also discussed earlier in Section 3.3.3.

If the Java method being examined meets all constraints within the TXL rule then processing within the rule will continue through to the end and transform the method it is working on by adding the commented Java annotations just above the method's declaration. The addition of the commented Java annotations is done at the end of the rule in the by clause as illustrated in Figure 4.4. These constraints are established using the where clauses within the rule. The constraints being checked for here are RM [isMethodSynchronized SYNCH], and tmpRole1Passed [$>$ 0]. As described earlier in Section 3.3.3, isMethodSynchronized is a matching function that tries to establish whether the Java synchronized construct is present in the variable being passed to it, in this case RM, which is a list of all the modifiers in the method's declaration. TXL variable tmpRole1Passed serves as a flag that is set in the rule hasNotifyOrNotifyAll, if that rule succeeds. Rule hasNotifyOrNotifyAll is called from within find1stSynchMethod1 and find1stSynchMethod2 but before this flag is checked. Rule hasNotifyOrNotifyAll will be discussed in detail below.

```
rule find1stSynchMethod1
  construct SYNCH [modifier]
    `synchronized
  replace [method_declaration]
    RM [repeat modifier] TS [type_specifier] MD [method_declarator]
        OT [opt throws] MB [method_body]
  where
    RM [isMethodSynchronized SYNCH]
  deconstruct MB
        BL2 [block]
  deconstruct BL2
    `{
      RDS3 [repeat declaration_or_statement]
    `}
  construct TransformedRDS3 [repeat declaration_or_statement]
    RDS3 [hasNotifyOrNotifyAll MD]
  import tmpRole1Passed [number]
  where
    tmpRole1Passed [> 0]
  construct TransformedBL2 [block]
    `{
      TransformedRDS3
    `}
  construct TransformedMB [method_body]
    TransformedBL2
  construct GuardedSuspensionAnnotation1pt1 [stringlit]
    ``@GuardedSuspensionPatternAnnotation(patternInstanceID="
  construct GuardedSuspensionAnnotation1pt2 [stringlit]
    ``, roleID=1, roleDescription=`Ensuring a method in the class is synchronized -
        guarded')"
  import CountFirstSynchMethIDs [number]
  export tmpRole1Passed
    0
  by
    `MUTATED /* GuardedSuspensionAnnotation1pt1 [+ CountFirstSynchMethIDs]
        [+ GuardedSuspensionAnnotation1pt2] */ RM TS MD OT TransformedMB
end rule
```

Figure 4.4: Illustration of Transforming find1stSynchMethod1 rule

hasNotifyOrNotifyAll rule

This rule hasNotifyOrNotifyAll corresponds to and identifies Role 1a, the presence of a notify() or notifyAll() statement. Rule hasNotifyOrNotifyAll is illustrated in Figure 4.5 (continued in Figure 4.6) and is a refinement of the one illustrated in Figure 3.14.

As mentioned earlier, the hasNotifyOrNotifyAll rule is called from the find1stSynchMethod1 and find1stSynchMethod2 rules. It checks for the existence of the notify() or notifyAll() statements within the synchronized method. The existence of either one of these 2 Java constructs is established using AE [isAssignmentExpr idNotifyExpr] [isAssignmentExpr idNotifyAllExpr] in the where clause in this rule, illustrated in Figure 4.5 (continued in Figure 4.6). Function isAssignmentExpr is a matching function that checks for the assignment expression passed to it, in this case notify() or notifyAll().

If this constraint - the check for notify() or notifyAll() - is passed then processing in this rule is passed down to where the expression is transformed by adding the commented Java annotations. This is similar to what has been discussed in the previous section and will also occur in the next couple of rules to be discussed.

```
rule hasNotifyOrNotifyAll MD [method_declarator]
  replace [expression_statement]
    EX [expression] `;
  construct idNotify [id]
    `notify
  construct idNotifyAll [id]
    `notifyAll
  construct idNotifyExpr [assignment_expression]
    `notify()
  construct idNotifyAllExpr [assignment_expression]
    `notifyAll()
  deconstruct EX
    AE [assignment_expression]
  where
    AE [isAssignmentExpr idNotifyExpr] [isAssignmentExpr idNotifyAllExpr]
  deconstruct MD
      MN [method_name] `( LFP [list formal_parameter] `) RD [repeat dimension]
  deconstruct MN
    DN [declared_name]
  deconstruct DN
     methodID [id] OGP [opt generic_parameter]
```

Figure 4.5: Illustration of Transforming hasNotifyOrNotifyAll rule

```
  import FirstSynchMethIDs [repeat id]
  construct newMethodIDs [repeat id]
    FirstSynchMethIDs [. methodID]
  export FirstSynchMethIDs
    newMethodIDs
  import CountFirstSynchMethIDs [number]
  construct PlusOne [number]
    1
  construct NewCount [number]
    CountFirstSynchMethIDs [+ PlusOne]
  export CountFirstSynchMethIDs
    NewCount
  import notifyCollection [repeat expression]
  construct newNotifyCollection [repeat expression]
    notifyCollection [. EX]
  export notifyCollection
    newNotifyCollection
  import tmpRole1Passed [number]
  construct tmpCount [number]
    tmpRole1Passed [+ PlusOne]
  export tmpRole1Passed
    tmpCount
  construct GuardedSuspensionAnnotation1apt1 [stringlit]
    ``@GuardedSuspensionPatternAnnotation(patternInstanceID="
  construct GuardedSuspensionAnnotation1apt2 [stringlit]
    ``, roleID=1a, roleDescription=`Ensure there is a notify() or notifyAll()
        statement.')"
  by
    `MUTATED
    /* GuardedSuspensionAnnotation1apt1 [+ CountFirstSynchMethIDs]
        [+ GuardedSuspensionAnnotation1apt2] */
    EX `;
end rule
```

Figure 4.6: Illustration of Transforming hasNotifyOrNotifyAll rule Continued

```
rule find2ndSynchMethod1
  construct SYNCH [modifier]
    `synchronized
  replace [method_declaration]
    RM [repeat modifier] TS [type_specifier] MD [method_declarator]
        OT [opt throws] MB [method_body]
  where
    RM [isMethodSynchronized SYNCH]
  deconstruct MB
        BL2 [block]
  deconstruct BL2
    `{
      RDS3 [repeat declaration_or_statement]
    `}
  construct TransformedRDS3 [repeat declaration_or_statement]
    RDS3 [isWhileLpWait MD] [isDoWhileLpWait MD]
  import tmpRole2Passed [number]
  where
    tmpRole2Passed [> 0]
  construct TransformedBL2 [block]
    `{
      TransformedRDS3
    `}
  construct TransformedMB [method_body]
    TransformedBL2
  construct GuardedSuspensionAnnotation2pt1 [stringlit]
    ``@GuardedSuspensionPatternAnnotation(patternInstanceID="
  construct GuardedSuspensionAnnotation2pt2 [stringlit]
    ``, roleID=2, roleDescription=`Ensuring a method in the class is synchronized -
        guarded')"
  import Counter [number]
  export tmpRole2Passed
    0
  by
    `MUTATED /* GuardedSuspensionAnnotation2pt1 [+ Counter]
        [+ GuardedSuspensionAnnotation2pt2] */ RM TS MD OT TransformedMB
end rule
```

Figure 4.7: Illustration of Transforming find2ndSynchMethod1 rule

find2ndSynchMethod1 and find2ndSynchMethod2 rules

Like find1stSynchMethod1 and find1stSynchMethod2 rules discussed earlier, these 2 rules establish that a synchronized method exists. They are refinements of the same named rules discussed in Section 3.3.3 that simply detected Role 2 (ensuring that a second method in the class is synchronized) in the Java source code, but did not transform the Java source code by adding annotations to specifically identify where the rule exists. Rule find2ndSynchMethod1 is illustrated in Figure 4.7.

These refined rules, find2ndSynchMethod1 and find2ndSynchMethod2, identify and add annotations to the second synchronized method required for the Guarded Suspension design pattern to be identified. Like in find1stSynchMethod1 and find1stSynchMethod2, these 2 rules differ only in that they handle the two different ways a synchronized method can be defined in Java.

The rules find2ndSynchMethod1 and find2ndSynchMethod2 work very similarly to the rules find1stSynchMethod1 and find1stSynchMethod2, discussed earlier, in terms of checking constraints and proceeding further in the rule to adding the commented Java annotations, if those constraints are met. One of these constraints, tmpRole2Passed [$>$ 0], that is checked for in find2ndSynchMethod1 and find2ndSynchMethod2, is set in the rule completeStats, which is called from within rules isWhileLpWait and isDoWhileLpWait, which in turn are called from these 2 rules, find2ndSynchMethod1 and find2ndSynchMethod2.

isWhileLpWait and isDoWhileLpWait rules

These 2 rules, isWhileLpWait and isDoWhileLpWait, correspond to the Guarded Suspension design pattern's Roles 2a (ensuring there is a loop) and 2aa (ensuring that within that loop there is a Java wait() statement). Like with the other rules described above, this rule is a refinement of the same named rules discussed in Section 3.3.3. With this refinement being the addition of TXL code to annotate the Java source code being examined. Both rules are illustrated in Figures 4.8 and 4.9.

If a loop is found using either of these 2 rules and a Java wait() statement found using the constraint STMT [hasStmt waitStmt] [hasWaitStmt each numVarsIDCollection], then the rule will continue down to the transformation, creating the commented Java annotations and adding them to the Java source code. If Role 2a and 2aa are established as discussed above then 2 sets of commented Java annotations will be added. The first above the loop for Role 2a and the second above the statement where the Java wait() construct is, for Role 2aa.

```
rule isWhileLpWait MD [method_declarator]
  replace [while_statement]
      `while `( EX [expression] `)
      STMT [statement]
  construct waitStmt [statement]
    `wait();
  import numVarsIDCollection [repeat id]
  where
    STMT [hasStmt waitStmt] [hasWaitStmt each numVarsIDCollection]
  construct GuardedSuspensionAnnotation2apt1 [stringlit]
    ``@GuardedSuspensionPatternAnnotation(patternInstanceID="
  construct GuardedSuspensionAnnotation2apt2 [stringlit]
    ``, roleID=2a, roleDescription=`Ensuring there is a while statement.')"
  construct GuardedSuspensionAnnotation2aapt1 [stringlit]
    ``@GuardedSuspensionPatternAnnotation(patternInstanceID="
  construct GuardedSuspensionAnnotation2aapt2 [stringlit]
    ``, roleID=2aa, roleDescription=`Ensuring there is a wait() statement.')"
  construct InstanceFound [method_declarator]
    MD [completeStats MD EX]
  import Counter [number]
  by
    `MUTATED /* GuardedSuspensionAnnotation2apt1 [+ Counter]
        [+ GuardedSuspensionAnnotation2apt2] */
      `while `( EX `)
    {
      /* GuardedSuspensionAnnotation2aapt1 [+ Counter]
           [+ GuardedSuspensionAnnotation2aapt2] */
      STMT
    }
end rule
```

Figure 4.8: Illustration of Transforming isWhileLpWait rule

```
rule isDoWhileLpWait MD [method_declarator]
  replace [do_statement]
    `do
      STMT [statement]
    `while `( EX [expression] `) `;
  construct waitStmt [statement]
    `wait();
  import numVarsIDCollection [repeat id]
  where
    STMT [hasStmt waitStmt] [hasWaitStmt each numVarsIDCollection]
  construct GuardedSuspensionAnnotation2apt1 [stringlit]
    ``@GuardedSuspensionPatternAnnotation(patternInstanceID="
  construct GuardedSuspensionAnnotation2apt2 [stringlit]
    ``, roleID=2a, roleDescription=`Ensuring there is a while statement.')"
  construct GuardedSuspensionAnnotation2aapt1 [stringlit]
    ``@GuardedSuspensionPatternAnnotation(patternInstanceID="
  construct GuardedSuspensionAnnotation2aapt2 [stringlit]
    ``, roleID=2aa, roleDescription=`Ensuring there is a wait() statement.')"
  construct InstanceFound [method_declarator]
    MD [completeStats MD EX]
  import Counter [number]
  by
    `MUTATED /* GuardedSuspensionAnnotation2apt1 [+ Counter]
        [+ GuardedSuspensionAnnotation2apt2] */
    `do
    {
      /* GuardedSuspensionAnnotation2aapt1 [+ Counter]
           [+ GuardedSuspensionAnnotation2aapt2] */
      STMT
    }
    `while `( EX `) `;
end rule
```

Figure 4.9: Illustration of Transforming isDoWhileLpWait rule

4.4 Summary

In this chapter we have discussed how using our identification technique, we inserted commented custom Java annotations to our Java source code examples, identifying the roles that comprise the 8 different concurrency design patterns that we are targeting (see Section 2.2.1 for a description of each concurrency design pattern). We started by defining the custom Java annotations in Section 4.2. These Java annotations were created for each role in each concurrency design pattern.

In Section 4.3 we discussed how we implemented these commented custom Java annotations in our Java source code examples using TXL. To briefly reiterate, we refined the same TXL rules we created to detect the concurrency design pattern roles, in Chapter 3, to transform the Java source code, by adding the commented custom Java annotations where a specific concurrency design pattern role was detected.

Adding the commented custom Java annotations to the Java source code when our 8 targeted concurrency design patterns are detected, is the last step in the identification of each of the 8 concurrency design patterns. In Chapter 5, we will discuss how we evaluated our detection and identification technique.

Chapter 5

Evaluation

5.1 Overview

In Chapters 3 and 4 we elaborated on our concurrency design pattern detection and identification technique. We discussed how we run our 8 concurrency design pattern identification programs on the 8 corresponding Java source code examples and in so doing refined our programs ensuring that this detection and identification was successful. The 8 concurrency design patterns we targeted are elaborated on in Section 2.2.

In this chapter we will discuss how we evaluated our 8 concurrency design pattern identification programs. In Section 5.2 we will elaborate on our evaluation procedure. We will then give an in depth discussion on our results and how we obtained them, in Section 5.3. We will also elaborate on how differences in the structural nature (*"Structural patterns deal with the composition of classes or objects"* [GHJV95]) and behavioral nature (*"Behavioral patterns characterize the ways in which classes or objects interact and distribute responsibility"* [GHJV95]) of the different concurrency design patterns we targeted, influenced the outcome of the results. We will end with a discussion of the possible threats to validity on our evaluation methodology, in Section 5.4.

5.2 Evaluation Methodology

The purpose of our evaluation is to establish how effective our static analysis technique is at identifying concurrency design patterns. We run our concurrency design pattern identification programs against 17 other Java source code examples. To perform an unbiased evaluation, we obtained these 17 examples from open source projects and from examples developed by our colleagues independently.

In regards to the open source projects, we performed our evaluation on individual Java files within the projects that utilized the 8 concurrency design patterns we were targeting and not all the files in all the projects. Table 5.1 lists all 17 Java source code examples, the concurrency design patterns expected and the sources from where they were obtained.

The experimental environment chosen for this evaluation was a single laptop machine containing an Intel(R) Core(TM)2 Duo CPU P8600 with 2.4GHz and 2.39GHz processors and 2.99GB of RAM. We followed the following steps in our evaluation methodology:

1. We automated the process by creating 8 batch files to run each of the 8 concurrency design pattern identification programs.

2. We run each batch file against each of the 17 Java source code examples.

3. By running each concurrency design pattern identification program against each of the 17 Java source code examples we were able to get a thorough analysis of the effectiveness of our static analysis technique and it's performance rate.

Table 5.1: List of Java source code examples used for Evaluation

Examples:	Expected Concurrency Design Pattern:	Description (Source):
Vector.java	Single Threaded Execution	From Sun Microsystems source code for java.util.Vector
Data.java	Balking	From the svn at http://oliverlee.googlecode.com/svn
RequestQueue.java	Balking	From the svn at http://oliverlee.googlecode.com/svn
RequestQueue2.java	Balking	From the svn at http://oliverlee.googlecode.com/svn
SaverThread.java	Balking	From the svn at http://oliverlee.googlecode.com/svn
ChangerThread.java	Balking	From the svn at http://oliverlee.googlecode.com/svn
ReadWriteLockTXLEval.java	Read/Write Lock	Coded by fellow student Kevin Jalbert
BlockingQueue.java	Guarded Suspension	From package nl.justobjects.pushlet.core found at http://www.pushlets.com/src/index.html
EventQueue.java	Guarded Suspension	From package nl.justobjects.pushlet.core found at http://www.pushlets.com/src/index.html
RequestQueue3.java	Guarded Suspension	From the svn at http://oliverlee.googlecode.com/svn
ServerThread.java	Guarded Suspension	From the svn at http://oliverlee.googlecode.com/svn
CancellableThread.java	Two Phase Termination	From package ORG.oclc.util found at http://opensitesearch.sourceforge.net/docs/helpzone/api/overview-summary.html
Component.java	Lock Object	From Sun Microsystems source code for package java.awt
Queue.java	Producer Consumer	From package org.apache.commons.threadpool found at http://commons.apache.org/dormant/threadpool/apidocs/org/apache/commons/threadpool/ package-summary.html
MTQueue.java	Producer Consumer	From package org.exoplatform.services.threadpool.impl found at http://www.clickblocks.org
ProducerConsumerTXLEval.java	Producer Consumer	Coded by fellow student Ben Waters
SchedulerTXLEval.java	Scheduler	Coded by fellow student Dave Kelk

5.3 Results

In discussing our results we will elaborate on 2 main areas, as enumerated below:

1. The effectiveness in identifying the 8 targeted concurrency design patterns.

2. Performance Rates of the TXL Programs.

5.3.1 Effectiveness in Identifying the Concurrency Design Patterns

In the sections that follow we will discuss the success and failure rates of each of our 8 concurrency design pattern detecting programs. We will elaborate on why our concurrency design pattern detection programs were more successful at detecting and identifying certain concurrency design patterns, than others.

Single Threaded Execution Design Pattern Success Rates

As discussed in Section 2.2.2 the Single Threaded Execution design pattern is used by almost all the other concurrency design patterns and is hence the most present in our 17 Java source code examples. Of all the concurrency design patterns, the Single Threaded Execution design pattern is also the most basic in structure. It has just one role and that is for the method to be synchronized. For this reason it has one of the highest detection rates using our static analysis technique. Table 5.2 shows the results from our TXL program's detection and identification of the Single Threaded Execution design pattern on the 17 java source code examples.

As shown in the result table there were cases where the Single Threaded Execution design pattern was not detected. After a manual walk through of the Java source code examples and an in depth look at our TXL program, we discovered that we had not put into consideration that the synchronized(this) statement can appear anywhere in the method body in order for the method to be synchronized and hence be an instance of the Single

Threaded Execution design pattern. We had only put into consideration the scenario where the synchronized(this) statement appears at the start of the method definition. Adjustments to the TXL code to accommodate for this can be easily added.

Table 5.2: Single Threaded Execution Design Pattern Success Rates

Program:	Instances Present:	Total Instances Found:	Partial Instances Found:	Total Match (%):	Partial Match (%):	False Positives:
Vector.java	37	36	0	97	0	0
Data.java	2	2	0	100	0	0
RequestQueue.java	2	2	0	100	0	0
RequestQueue2.java	2	2	0	100	0	0
SaverThread.java	0	0	-	-	-	0
ChangerThread.java	0	0	-	-	-	0
ReadWriteLockTXLEval.java	2	2	0	100	0	0
BlockingQueue.java	7	7	0	100	0	0
EventQueue.java	8	8	0	100	0	0
RequestQueue3.java	2	2	0	100	0	0
ServerThread.java	0	0	-	-	-	0
CancellableThread.java	3	2	0	66	0	0
Component.java	44	32	0	73	0	0
Queue.java	5	5	0	100	0	0
MTQueue.java	4	4	0	100	0	0
ProducerConsumerTXLEval.java	2	2	0	100	0	0
SchedulerTXLEval.java	2	2	0	100	0	0
Total:	122	108	0	88.5	0	0

Balking Design Pattern Success Rates

The Balking design pattern is discussed in detail in Section 2.2.5 and details about the roles that comprise it are shown in Section A.4 of the Appendix. In brief, the Balking design pattern works at the method level and in order for it to be detected 3 roles must be satisfied, namely:

1. Ensuring the method is synchronized - guarded.

2. Ensure an if statement that tests a flag right at the start of the synchronized method.

3. Ensuring an if statement or an else statement that tests the flag in 2 above does an immediate return - balking.

Table 5.3 shows the results from running our Balking design pattern detecting program against the 17 Java source code examples. Given the 3 roles required above only 1 of the programs had an instance of the Balking design pattern and that 1 was successfully detected and identified by the detecting program.

There were some interesting findings during the evaluation procedure for the Balking design pattern:

1. RequestQueue.java and RequestQueue2.java ended up being correctly detected as containing instances of the Guarded Suspension design pattern not the Balking design pattern despite the source (http://oliverlee.googlecode.com/svn) claiming that balking design pattern instances exist in both.

2. Despite the source (http://oliverlee.googlecode.com/svn) claiming that the source code files “SaverThread.java” and “ChangerThread.java” have instances of the Balking design pattern, none existed (determined by a manual walk through of the code) and none were detected.

Table 5.3: Balking Design Pattern Success Rates

Program:	Instances Present:	Total Instances Found:	Partial Instances Found:	Total Match (%):	Partial Match (%):	False Positives:
Vector.java	0	0	-	-	-	0
Data.java	1	1	0	100	0	0
RequestQueue.java	0	0	-	-	-	0
RequestQueue2.java	0	0	-	-	-	0
SaverThread.java	0	0	-	-	-	0
ChangerThread.java	0	0	-	-	-	0
ReadWriteLockTXLEval.java	0	0	-	-	-	0
BlockingQueue.java	0	0	-	-	-	0
EventQueue.java	0	0	-	-	-	0
RequestQueue3.java	0	0	-	-	-	0
ServerThread.java	0	0	-	-	-	0
CancellableThread.java	0	0	-	-	-	0
Component.java	0	0	-	-	-	0
Queue.java	0	0	-	-	-	0
MTQueue.java	0	0	-	-	-	0
ProducerConsumerTXLEval.java	0	0	-	-	-	0
SchedulerTXLEval.java	0	0	-	-	-	0
Total:	1	1	0	100	0	0

Guarded Suspension Design Pattern Success Rates

Table 5.4 shows the results from running our Guarded Suspension design pattern detection program on the 17 Java source code examples. The Guarded Suspension design pattern is discussed in detail in Section 2.2.4 and all the roles that comprise it can be seen in Section A.3 of the Appendix. In summary the Guarded Suspension design pattern encompasses 2 synchronized methods within a class. Each of which corresponds to the 2 main roles that make up the design pattern. The first synchronized method (Role 1) will contain a notify() or notifyAll() statment and the second synchronized method (Role 2) will contain a loop and within the loop a wait() statement.

Table 5.4: Guarded Suspension Design Pattern Success Rates

Program:	Instances Present:	Total Instances Found:	Partial Instances Found:	Total Match (%):	Partial Match (%):	False Positives:
Vector.java	0	0	-	-	-	0
Data.java	0	0	-	-	-	0
RequestQueue.java	1	1	0	100	0	0
RequestQueue2.java	1	1	0	100	0	0
SaverThread.java	0	0	-	-	-	0
ChangerThread.java	0	0	-	-	-	0
ReadWriteLockTXLEval.java	1	0	1	0	50	0
BlockingQueue.java	2	0	1	0	50	0
EventQueue.java	3	0	1	0	50	0
RequestQueue3.java	1	1	0	100	0	0
ServerThread.java	0	0	-	-	-	0
CancellableThread.java	0	0	-	-	-	0
Component.java	0	0	-	-	-	0
Queue.java	1	1	0	100	0	0
MTQueue.java	0	0	-	-	-	0
ProducerConsumerTXLEval.java	0	0	-	-	-	0
SchedulerTXLEval.java	0	0	-	-	-	0
Total:	10	4	3	40	30	0

Before running our Guarded Suspension detection program against all 17 Java source code examples, we manually analyzed the Java source code to determine the number of Guarded Suspension instances detected. Table 5.4 illustrates this in the "Instances Present" column. There were a total of ten instances of the Guarded Suspension design pattern present, contained within seven of the programs. Four of these ten instances were detected by our program. In some cases partial instances of the Guarded Suspension design pattern were detected which, is also shown in Table 5.4.

For the three instances of the Guarded Suspension design pattern that were expected but were not found we re-analyzed the code and made the following discoveries:

1. For the "ReadWriteLockTXLEval.java" program the synchronized method that should have satisfied Role 2 of the Guarded Suspension design pattern, the wait() statement that existed, was in the form this.wait() and yet our program was searching for specifically a lone wait() statement. In this particular case our tool was a bit too rigid. Adjustments in the TXL code can certainly be made to accommodate for this scenario.

2. In the case of "BlockingQueue.java" and "EventQueue.java", the 2 main roles that comprise an instance of the Guarded Suspension design pattern are found within the same methods. So, our detection program detects Role 1 only, each time it identifies these methods and not Role 2 because the methods have already been mutated. As discussed in Section 3.3.1, we mutate Java source code constructs that have been identified as roles of the concurrency design pattern as they are detected. Because these methods have already been detected as Role 1, they get mutated and hence will not be analyzed again and not have their Role 2 aspects detected.

3. In all 3 Java source code examples discussed in 1 and 2 above, all Role 1s of the 2 roles that comprise the Guarded Suspension design pattern were detected. So there was at least a 50% match of the Guarded Suspension design pattern instances in these cases.

Lock Object Design Pattern Success Rates

Of the 17 Java source code examples we used to evaluate our technique, we were expecting the Lock Object design pattern to exist in just one and this was identified correctly. The Lock Object design pattern had a 100% success rate because it is more structural in nature than most of the other concurrency design patterns we were targeting. Being more structural in nature makes it a more suitable candidate for identification using a static analysis technique like ours. The results are shown in Table 5.5. The Lock Object design pattern is described in detail in Section 2.2.3 and the roles that comprise it can be found in Section A.2

of the Appendix. In summary though, it is made up of 3 major roles:

1. Role 1: Lock object - a static object in the class.
2. Role 2: Lock object method - a static method in the same class as Role 1 which returns an instance of the Lock object, Role 1.
3. Role 3: Synchronized calls to method Role 2. We consider each synchronized call to the Role 2 method, an instance of the Lock Object design pattern.

Table 5.5: Lock Object Design Pattern Success Rates

Program:	Instances Present:	Total Instances Found:	Partial Instances Found:	Total Match (%):	Partial Match (%):	False Positives:
Vector.java	0	0	-	-	-	0
Data.java	0	0	-	-	-	0
RequestQueue.java	0	0	-	-	-	0
RequestQueue2.java	0	0	-	-	-	0
SaverThread.java	0	0	-	-	-	0
ChangerThread.java	0	0	-	-	-	0
ReadWriteLockTXLEval.java	0	0	-	-	-	0
BlockingQueue.java	0	0	-	-	-	0
EventQueue.java	0	0	-	-	-	0
RequestQueue3.java	0	0	-	-	-	0
ServerThread.java	0	0	-	-	-	0
CancellableThread.java	0	0	-	-	-	0
Component.java	36	36	0	100	0	0
Queue.java	0	0	-	-	-	0
MTQueue.java	0	0	-	-	-	0
ProducerConsumerTXLEval.java	0	0	-	-	-	0
SchedulerTXLEval.java	0	0	-	-	-	0
Total:	36	36	0	100	0	0

Producer Consumer Design Pattern Success Rates

The Producer Consumer design pattern is of a very behavioral nature and this explains the low rate of identifying instances of it within the Java source code examples in which we expected to find it. The results of our evaluation of the program identifying the Producer Consumer design pattern can be found in Table 5.6. When a design pattern is more behavioral in nature there tends to be a very large number of ways it can be applied. A dynamic analysis technique would be more suited for more behavioral design patterns like this one.

Table 5.6: Producer Consumer Design Pattern Success Rates

Program:	Instances Present:	Total Instances Found:	Partial Instances Found:	Total Match (%):	Partial Match (%):	False Positives:
Vector.java	0	0	-	-	-	0
Data.java	0	0	-	-	-	0
RequestQueue.java	0	0	-	-	-	0
RequestQueue2.java	0	0	-	-	-	0
SaverThread.java	0	0	-	-	-	0
ChangerThread.java	0	0	-	-	-	0
ReadWriteLockTXLEval.java	0	0	-	-	-	0
BlockingQueue.java	0	0	-	-	-	0
EventQueue.java	0	0	-	-	-	0
RequestQueue3.java	0	0	-	-	-	0
ServerThread.java	0	0	-	-	-	0
CancellableThread.java	0	0	-	-	-	0
Component.java	0	0	-	-	-	0
Queue.java	1	0	1	0	33	0
MTQueue.java	1	0	1	0	33	0
ProducerConsumerTXLEval.java	1	0	0	0	0	0
SchedulerTXLEval.java	0	0	-	-	-	0
Total:	3	0	2	0	66.67	0

The Producer Consumer design pattern is comprised of 3 main roles which encompass 3 classes. These 3 roles are a Producer class, a Consumer class and a Queue class, all of which

interact with one another. A more detailed discussion of the Producer Consumer design pattern can be found in Section 2.2.8 of this thesis. Details about these 3 roles and the sub-roles within them can be found in Section A.7 of the Appendix. As mentioned above there are numerous ways in which any of these 3 classes can be set up to interact with one another. In our TXL program to identify the Producer Consumer design pattern we put into consideration just a few of these ways.

Of the 3 Java source code examples in which we were expecting to find the Producer Consumer design pattern, two of them - the "Queue.java" and "MTQueue.java" files - comprise just the Queue class (Role 2) of the Producer Consumer design pattern which is the most in depth of the 3 roles (see Section A.7 of the Appendix). Despite the highly behavioral nature of this pattern, our Producer Consumer design pattern detection program was able to successfully detect a structural aspect of this design pattern - Role 2a - in both. Role 2a is the array list to house the produced objects and is 1 of the 3 sub-roles that comprise Role 2. Our tool did not detect any part of the Producer Consumer design pattern in "ProducerConsumerTXLEval.java", which has 1 complete instance of it.

Read/Write Lock Design Pattern Success Rates

Like the Producer Consumer design pattern the Read/Write Lock design pattern is very behavioral in nature. Because of this there was difficulty in identifying instances of it or even partial instances of it in "ReadWriteLockTXLEval.java". "ReadWriteLockTXLEval.java" is the one example we manually analyzed and identified as having 1 complete instance of the Read/Write Lock design pattern, but neither this instance nor any of its 3 roles were detected by our program. Table 5.7 illustrates the results of our evaluation on the Read/Write Lock design pattern detecting program.

The Read/Write Lock design pattern is comprised of 3 main roles corresponding to 3 methods all within 1 class. These are a ReadLock method, a WriteLock method and a Done method. These 3 roles and the sub-roles under them are described in Section A.6 of the

Appendix. A full description of the Read/Write Lock design pattern and how it works is described in Section 2.2.7.

Table 5.7: Read/Write Lock Design Pattern Success Rates

Program:	Instances Present:	Total Instances Found:	Partial Instances Found:	Total Match (%):	Partial Match (%):	False Positives:
Vector.java	0	0	-	-	-	0
Data.java	0	0	-	-	-	0
RequestQueue.java	0	0	-	-	-	0
RequestQueue2.java	0	0	-	-	-	0
SaverThread.java	0	0	-	-	-	0
ChangerThread.java	0	0	-	-	-	0
ReadWriteLockTXLEval.java	1	0	0	0	0	0
BlockingQueue.java	0	0	-	-	-	0
EventQueue.java	0	0	-	-	-	0
RequestQueue3.java	0	0	-	-	-	0
ServerThread.java	0	0	-	-	-	0
CancellableThread.java	0	0	-	-	-	0
Component.java	0	0	-	-	-	0
Queue.java	0	0	-	-	-	0
MTQueue.java	0	0	-	-	-	0
ProducerConsumerTXLEval.java	0	0	-	-	-	0
SchedulerTXLEval.java	0	0	-	-	-	0
Total:	1	0	0	0	0	0

Scheduler Design Pattern Success Rates

The Scheduler design pattern is another one of the concurrency design patterns that is very behavioral in nature. Like in the Read/Write Lock design pattern and the Producer Consumer design pattern before it, this certainly had an impact on the results. As explained earlier, the more behavioral a pattern is, the more difficult it is to detect instances of it as they are numerous varied ways that the behavioral pattern can be applied.

Like in the Producer Consumer design pattern and the Read/Write Lock design pattern we targeted a few common ways that the design pattern can be applied however, like in those same 2 patterns, the way the Scheduler design pattern was applied in the example we used for this evaluation varied in many ways from the constructs we were targeting to detect the design pattern in the example. Just so that we are clear, our technique targeted one way in which the Scheduler design pattern can be applied whilst the Java source code example had the Scheduler design pattern applied in a different manner.

Table 5.8: Scheduler Design Pattern Success Rates

Program:	Instances Present:	Total Instances Found:	Partial Instances Found:	Total Match (%):	Partial Match (%):	False Positives:
Vector.java	0	0	-	-	-	0
Data.java	0	0	-	-	-	0
RequestQueue.java	0	0	-	-	-	0
RequestQueue2.java	0	0	-	-	-	0
SaverThread.java	0	0	-	-	-	0
ChangerThread.java	0	0	-	-	-	0
ReadWriteLockTXLEval.java	0	0	-	-	-	0
BlockingQueue.java	0	0	-	-	-	0
EventQueue.java	0	0	-	-	-	0
RequestQueue3.java	0	0	-	-	-	0
ServerThread.java	0	0	-	-	-	0
CancellableThread.java	0	0	-	-	-	0
Component.java	0	0	-	-	-	0
Queue.java	0	0	-	-	-	0
MTQueue.java	0	0	-	-	-	0
ProducerConsumerTXLEval.java	0	0	-	-	-	0
SchedulerTXLEval.java	1	0	1	0	100	0
Total:	1	0	1	0	100	0

Table 5.8 shows our results regarding the program to detect the Scheduler design pattern. Only “SchedulerTXLEval.java” had a complete instance of the Scheduler design pattern

and our program identified only it's Role 3. A detailed description of the Scheduler design pattern can be found in Section 2.2.6 and details about the 4 roles that comprise it can be found in Section A.5 of the Appendix. However, to better understand our results, we will summarize all 4 roles below:

1. Role 1: Scheduler class.
2. Role 2: Request class that implements the Schedule Ordering interface (Role 3).
3. Role 3: Schedule Ordering interface class (this was the only one identified).
4. Role 4: Processor class that delegates scheduling of the request object's processing to the Scheduler class, one at a time.

Two Phase Termination Design Pattern Success Rates

Like with the previous concurrency design patterns discussed, namely the Producer Consumer design pattern, the Read/Write Lock design pattern and the Scheduler design pattern, the Two Phase Termination design pattern is also very behavioral. As our results in Table 5.9 show, no instances or even partial instances of the Two Phase Termination design pattern were identified in "CancellableThread.java". As shown in Table 5.1, "CancellableThread.java" is the one Java source code example of the 17 we used that has 1 instance of the Two Phase Termination design patten.

Of the 8 design patterns we are targeting the Two Phase Termination is probably the most varied in ways that in can be applied. A detailed description of this design pattern can be found in Section 2.2.9. The Two Phase Termination design pattern can encompass constructs within multiple classes, numerous methods within one class or just a few methods within a class, so to try and statically identify all the varied forms this concurrency design pattern can take is a huge challenge. Our selection of roles that make up the Two Phase Termination design pattern is based on the example in the text [Gra02] which is a common

way to two-phase terminate a thread. These roles are elaborated on in Section A.8 of the Appendix.

Table 5.9: Two Phase Termination Design Pattern Success Rates

Program:	Instances Present:	Total Instances Found:	Partial Instances Found:	Total Match (%):	Partial Match (%):	False Positives:
Vector.java	0	0	-	-	-	0
Data.java	0	0	-	-	-	0
RequestQueue.java	0	0	-	-	-	0
RequestQueue2.java	0	0	-	-	-	0
SaverThread.java	0	0	-	-	-	0
ChangerThread.java	0	0	-	-	-	0
ReadWriteLockTXLEval.java	0	0	-	-	-	0
BlockingQueue.java	0	0	-	-	-	0
EventQueue.java	0	0	-	-	-	0
RequestQueue3.java	0	0	-	-	-	0
ServerThread.java	0	0	-	-	-	0
CancellableThread.java	1	0	0	0	0	0
Component.java	0	0	-	-	-	0
Queue.java	0	0	-	-	-	0
MTQueue.java	0	0	-	-	-	0
ProducerConsumerTXLEval.java	0	0	-	-	-	0
SchedulerTXLEval.java	0	0	-	-	-	0
Total:	1	0	0	0	0	0

Summary of Results for the Concurrency Design Pattern Detection Programs

As noted in the results shown for the effectiveness of our technique in identifying the concurrency design patterns, there were zero false positives. This result was mainly because we pursued a more rigid rather than general approach in creating our TXL rules. Section 3.3 discusses our approach in more detail however, in brief, because our TXL rules were more rigid in detecting the design pattern roles they ended up having no spurious results but

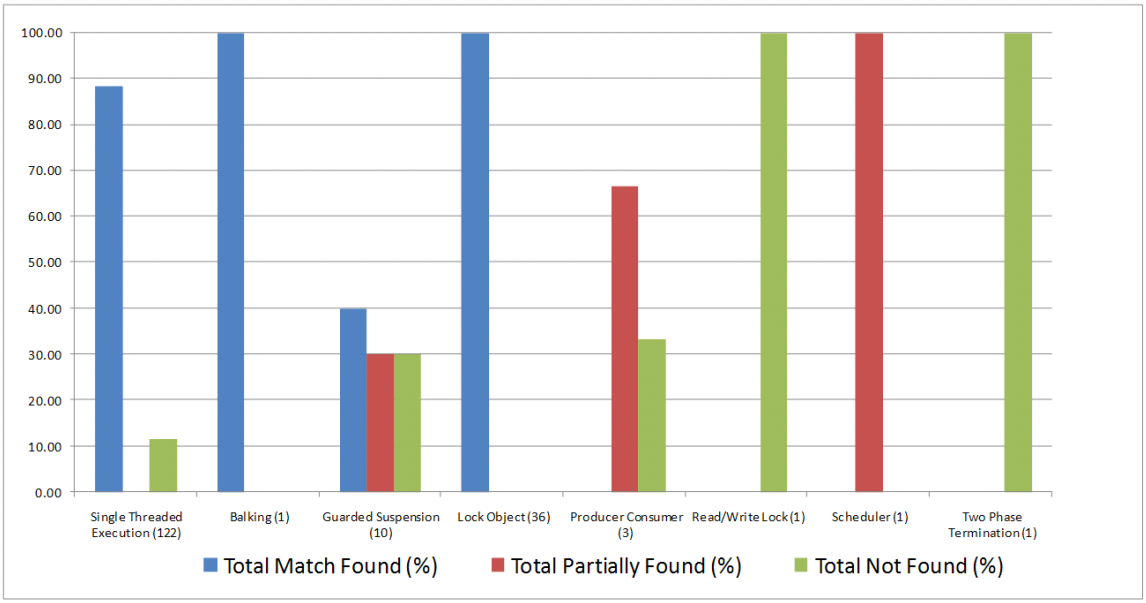

Figure 5.1: Illustration of our Concurrency Design Pattern Detection Results Summary

also ended up not detecting some instances of the design patterns that were present in our examples. Figure 5.1 summarizes the detection results described above for all 8 of our concurrency design pattern detection programs.

5.3.2 Performance Rates of the TXL Programs

In gathering our results we were also able to capture the performance i.e. the length of time it took for each of our 8 programs to complete the analysis on each of the 17 Java source code examples. Table 5.10 shows these results. The notable differences in the duration of the running of the 8 concurrency design pattern detection programs against the Java source code examples is when the size of the program came into play. The larger the Java source code example, the longer it took our 8 detection programs to complete the identification process.

Table 5.10: Performance Rates of Concurrency Design Pattern detection programs

Program/ Pattern Detection Tool:	Lines of Code	Single Threaded Execution	Balking	Guarded Suspension	Lock Object	Producer Consumer	Read/ Write Lock	Scheduler	Two Phase Termination
Vector.java	1028	11ms	14ms	9ms	12ms	12ms	13ms	15ms	13ms
Data.java	39	2ms	2ms	3ms	2ms	1ms	3ms	3ms	3ms
RequestQueue.java	21	2ms	3ms	3ms	3ms	3ms	3ms	3ms	1ms
RequestQueue2.java	25	1ms	1ms	2ms	3ms	3ms	3ms	3ms	1ms
SaverThread.java	21	2ms	3ms	1ms	2ms	3ms	4ms	3ms	2ms
ChangerThread.java	24	1ms	1ms	1ms	2ms	3ms	3ms	3ms	2ms
ReadWriteLockTXLEval.java	48	1ms	2ms	1ms	2ms	4ms	4ms	3ms	2ms
BlockingQueue.java	193	4ms	3ms	4ms	3ms	3ms	3ms	5ms	3ms
EventQueue.java	242	3ms	3ms	3ms	5ms	3ms	4ms	5ms	3ms
RequestQueue3.java	15	1ms	5ms	1ms	2ms	2ms	3ms	3ms	1ms
ServerThread.java	21	2ms	1ms	2ms	3ms	3ms	4ms	3ms	3ms
CancellableThread.java	82	1ms	3ms	3ms	3ms	3ms	3ms	3ms	1ms
Component.java	9453	1s 14ms	1s 52ms	1s 31ms	1s 47ms	1s 44ms	1s 39ms	1s 27ms	1s 26ms
Queue.java	148	1ms	2ms	3ms	3ms	3ms	3ms	5ms	3ms
MTQueue.java	98	3ms	3ms	1ms	3ms	3ms	3ms	5ms	3ms
ProducerConsumerTXLEval.java	55	2ms	1ms	3ms	3ms	3ms	3ms	3ms	3ms
SchedulerTXLEval.java	162	2ms	3ms	3ms	1ms	3ms	5ms	5ms	3ms

5.4 Threats to Validity

The threats to the validity of our experiment fall under the category referred to as "external validity" [WRH+00], which is defined as follows:

"The external validity is concerned with generalization. If there is a causal relationship between the construct of the cause, and the effect, can the result of the study be generalized outside the scope of our study? Is there a relation between the treatment and the outcome?" [WRH+00]

There were two threats to the validity of the evaluation of our technique that fall under the umbrella of external validity. These were related to those programs that could be utilized for our evaluation:

1. **Java test programs size:**

   The first threat was the size of our Java source code programs. As shown in table 5.10 the size of our test programs were relatively small. This was so mainly because we had to perform a manual analysis of each to determine whether the various concurrency design pattern roles actually existed within each program. Having smaller programs was a more practical choice to make this exercise possible. Despite this we tried to offset this threat by adding 2 relatively large Java programs, namely "Vector.java" with 1028 lines of code and "Component.java" with 9453 lines of code.

2. **Java test programs source:**

   The second threat to validity was that most of the 17 Java program examples we used were from open source projects online. These open source Java programs represent only a small subset of the kinds of concurrency Java programs that exist. There are numerous concurrency Java programs used in industry and other areas that are not open source and hence were not accessible to us. We tried offsetting this threat by having

our colleagues create 3 of the test programs on their own, namely "ReadWriteLockTXLEval.java", "ProducerConsumerTXLEval.java" and "SchedulerTXLEval.java".

5.5 Summary

In this chapter we have elaborated on the evaluation of our technique in identifying concurrency design patterns in Java source code. We discussed the methodology of our evaluation in Section 5.2 and then proceeded to have an in depth discussion of our results in Section 5.3. We ended with a discussion on the threats to the validity of our evaluation in Section 5.4, which included a discussion on how we tried to offset these threats.

The conclusion to our evaluation is that the more structural the concurrency design pattern is, the greater the success in identifying the design pattern. As for the actual performance of our 8 concurrency design pattern detection programs, as discussed in Section 5.3.2, this seems to have varied greatly only with the difference in the sizes of the Java test programs. The more lines of code the Java program had, the longer the processing time, but even then the processing for the largest program was under 2 seconds, which in our opinion is a rather fast processing time considering this program, Component.java, had over 9453 lines of code.

Chapter 6

Conclusion and Future Work

6.1 Overview

Our research into identifying and annotating concurrency design patterns in Java source code was quite challenging but both very exciting and fruitful. In Chapter 1 we started by elaborating on what our motive was in pursuing our research area. We proceeded in the same chapter to discuss the problem at hand and hypothesize on it.

In Chapter 2 we gave a detailed background into exactly what concurrency design patterns are. We elaborated and illustrated examples on each of the 8 concurrency design patterns that we are targeting, namely:

1. Single Threaded Execution (also called Critical Section)
2. Lock Object
3. Guarded Suspension
4. Balking
5. Scheduler
6. Read/Write Lock
7. Producer-Consumer
8. Two-Phase Termination

In Chapter 2 we gave a background into what Java annotations are. We discussed 4 existing design pattern detection techniques that use Java annotations. We elaborated on the purpose of each technique and how the Java annotations were implemented in each technique. We also discussed why we chose to implement the Java annotations as commented Java annotations due to various custom Java annotation shortcomings.

In Chapter 3 is where we thoroughly discussed our technique in detecting concurrency design patterns in Java source code. We gave an in depth background into the TXL language, explaining how it works, important aspects of it and why it was the ideal choice for what we were trying to achieve. We also discussed our approach in detecting the concurrency design patterns using TXL. This included a discussion into how we identified the roles that comprise each of the 8 concurrency design patterns, how we created the TXL rules to correspond to each of the respective roles that comprise the individual concurrency design patterns and how we refined the TXL rules after running them repetitively on examples of the respective concurrency design patterns from [Gra02].

In Chapter 4 we went on to discuss the final parts of our identification of concurrency design patterns technique. We started by elaborating on all the Java annotations we created to correspond to each of the roles that comprise the 8 different concurrency design patterns we were targeting. We then discussed how we actually implemented these Java annotations, commented, into the Java source code using TXL.

Chapter 5 is where we discussed in detail the evaluation of our technique. This started with a discussion on our evaluation methodology where amongst other procedural matters we introduced our 17 Java source code examples. We then proceeded to discuss our results from evaluating each of the 8 concurrency design pattern detection and identification programs on the 17 Java source code examples. This included success rates and performance rates. We ended Chapter 5 with a discussion on the threats to the validity of our technique.

6.2 Contributions

We have made mainly 2 contributions to the Software Engineering community through our research discussed in this thesis. These contributions are as follows:

1. **Development of a novel technique for the detection and annotation of concurrency design patterns:**

   Our main contribution is the creation of a novel technique to automatically identify concurrency design patterns in Java source code. As discussed in Section 2.3 there are just a few techniques that use Java annotations in identifying design patterns, but even then we could not find any that targeted concurrent software design patterns specifically.

   As elaborated on in Section 1.1 of this thesis, the research into a technique to detect concurrent software is very timely due to the growing importance of concurrent software in Software Engineering. As discussed in Section 1.1, the maintenance of concurrent software is a rather arduous process. The identification of concurrent software design patterns in concurrent software is a major step in easing the maintenance of it. In Section 6.4 below, we introduce future work to further help in the maintenance of concurrent software by checking to ensure that the commented Java annotations we have introduced through our technique, match up to the Java source code constructs they are annotating.

2. **Identifying of limitations in using static analysis in concurrency design pattern detection:**

   In working on this research we discovered that the differences in the structural and behavioral aspects of concurrency design patterns should be a major determinant in establishing the success (or failure) rate of a concurrency design pattern being identified statically. A static analysis technique based on pattern matching like TXL,

is best applied to identifying a more structural concurrency design pattern, whilst a dynamic analysis technique is probably more suited to identifying the more behavioral concurrent software design patterns.

This contribution was established in our evaluation of our static analysis technique against the 8 concurrency software design patterns we targeted. Four of these were more structural whilst the other 4 were more behavioral. We clearly showed in Section 5.3 that the more structural the design pattern was the higher the success rate of identifying it statically and the more behavioral it was the less the success rate.

6.3 Limitations

1. Because our technique in identifying concurrency design patterns was a static analysis technique, it had a low success rate in identifying those concurrency design patterns that were of a more behavioral variety. This is very clear in our results in Section 5.3 of this thesis.

2. As elaborated on in Section 3.2 of this thesis, TXL has many strengths that made it an excellent choice for our concurrency design pattern identification technique however, TXL is limited in it's syntax. Various constructs that are available in common object oriented languages are lacking with TXL. For example if-statments and loops that could offer great functionality are not available in TXL. We overcame syntax limitations but with quite a bit of difficulty and a whole lot more code. This limitation in TXL has been noted by it's developers [TC06] and an updated version 10.5i was launched in July 2011 [Lab], but unfortunately after we had already completed most of the development on our TXL programs.

3. As shown in Chapter 3 another limitation is the tradeoff between either having our concurrency design pattern detection programs being more rigid in their detection

or more flexible. This affects our concurrency detection programs precision and recall [WF11]. This is especially true for concurrency design patterns that can be implemented in a large variety of ways. In making the concurrency design pattern detection programs more rigid they will be less capable of detecting the different ways the concurrency design pattern can be detected hence will detect less instances of the concurrency design patterns (high precision but low recall). In making the programs more flexible they will detect more instances of the concurrency design patterns but with also a higher rate of false positives (high recall but low precision).

Adjustments to the precision and recall of the concurrency design pattern detection programs can be done by either making updates to the grammar that defines the various Java elements or making updates to the specific rules we created to correspond to the various concurrency design pattern roles.

As elaborated earlier in Section 3.2.1 updates to the grammar can be done by using TXL's redefine construct to change the definition of various Java elements to ease the process of detecting the various ways a concurrency design pattern role can be defined. For example the Java element synchronized could be redefined to not only be synchronized but also synchronized(this).

Making updates to the rule would involve either reducing the number of Java elements being searched for in the detection process (which would increase the recall but reduce precision) or increasing the number of Java constructs being searched for (hence increasing the precision but reducing the recall).

6.4 Future Work

6.4.1 Identification of additional concurrency design patterns

As discussed in Section 2.2.1 of this thesis, for the purposes of illustrating our technique we targeted 8 concurrency design patterns. There are other concurrency design patterns that have been established and we would like to add these to the list of those to be identified. In addition to this, for the concurrency design patterns that are more behavioral, we would like to gather more ways that they are applied and add this to our detection and identification programs so that we can improve on the low success rates in identifying them.

6.4.2 Uncomment and use the Java Annotations after JSR 308

Recall our discussion in Section 2.3.1 of this thesis, regarding the lack of Java annotation capability at the statement level of Java source code and how hopefully when JSR 308 becomes supported by Java, these Java annotations will be enabled at the statement level. When JSR 308 passes we would like to modify our concurrency design pattern identification programs to have the commented Java annotations added but uncommented. We would then create a Java program and use Java's reflection [McC98] capability to help with the maintenance of the concurrency design patterns within the Java source code.

6.4.3 Maintenance of Concurrency Design Patterns in Java source code

As discussed briefly in Section 6.2 above we would like to expound on our technique to not only identify the concurrency design patterns but to also detect and report if established ones have been broken. This can be achieved by creating TXL programs for each concurrency design pattern that correspond to the ones we created to detect the respective concurrency design patterns.

These new TXL programs will attempt to match the commented Java annotations to the Java constructs being annotated. These commented Java annotations would have been

added by the original technique to identify (as discussed in this thesis) the various roles that make up the concurrency design pattern. If all or even some of the commented Java annotations do not match the Java source code they are annotating, then that would mean that the concurrency design pattern that was identified in the Java source initially, may be broken.

6.5 Conclusion

In Section 6.1 we gave an overview of our research. Here we briefly summarized the notable items in the previous chapters of this thesis. We proceeded with a discussion of the contributions we have made through this body of research in Section 6.2. In Section 6.3 we discussed some limitations we experienced in our technique. We concluded with a discussion on future work regarding our research into the identification of concurrency design patterns in Java source code.

The identification of concurrency design patterns in not only Java source code but any programming language that uses concurrency, can play a major part in the maintenance of concurrent software. Concurrency design patterns as described in Section 1.1 offer a well tried and tested starting point to developing error free concurrent software. To identify these concurrency design patterns in a system and establish when they have been broken, could in effect also reduce the proliferation of hard to find concurrency related bugs. As shown in this thesis, our static analysis technique, can achieve this.

In conclusion and just to reiterate, the more structural the concurrency design pattern is, the greater the success in identifying the design pattern. Of the 8 concurrency design patterns that we targeted, the ones that were more structural (as described in Section 5.3), had a higher success rate in being identified. These were:

1. Single Threaded Execution design pattern
2. Balking design pattern

3. Guarded Suspension design pattern
4. Lock Object design pattern

The more behavioral the concurrency design pattern is, the less the success rate in identifying it. Of the 8 concurrency design patterns that we targeted, the more behavioral ones (as described in Section 5.3) and hence the ones with very low success rates in being even partially identified were:

1. Producer Consumer design pattern
2. Read/Write Lock design pattern
3. Scheduler design pattern
4. Two Phase Termination design pattern

Bibliography

[CCH07] James R Cordy, Ian H Carmichael, and Russell Halliday. The TXL programming language version 10.5. Technical report, Queen's University, Novemeber 2007.

[Ern10] Michael D Ernst. Type annotations specification (JSR 308). Web page: `http://types.cs.washington.edu/jsr308/java-annotation-design.html#other-annotations` (last accessed: November 5, 2011), 2010.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[Gra02] Mark Grand. *Patterns in Java, Volume 1*. Wiley Publishing, Inc., 2002.

[HLH06] Chengwan He, Zheng Li, and Keqing He. Identification and extraction of design pattern information in Java program. In *Proc. of 9th ACIS International Conference on Software Engineering, Artificial Intelligence, Networking, and Parallel/Distributed Computing (SNPD '08)*, page 828834, 2006.

[Jam05] Javid Jamae. Learn to use the new annotation feature of Java 5.0. Web page: `http://www.devx.com/Java/Article/27235/0/page/1` (last accessed: November 5, 2011), 2005.

[Lab] Software Technology Laboratory. Web page: `http://www.txl.ca/index.html` (last accessed: November 5, 2011).

[McC98] Glen McCluskey. Using Java reflection. Web page: `http://java.sun.com/developer/technicalArticles/ALT/Reflection/` (last accessed: November 5, 2011), 1998.

[Mef06] Klaus Meffert. Supporting design patterns with annotations. In *Proc. of 13th Annual IEEE International Symposium and Workshop on Engineering of Computer Based Systems*, pages 437–445, 2006.

[Mic06] Sun Microsystems. JSR 175: A metadata facility for the Java programming language. Web page: `http://jcp.org/en/jsr/detail?id=175` (last accessed: November 5, 2011), 2006.

[RPM08] Ghulam Rasool, Ilka Philippow, and Patrick Mader. Design pattern recovery based on annotations. In *Advances in Engineering Software*, page 123, 2008.

[SP09] Miroslav Sabo and Jaroslav Poruban. Preserving design patterns using source code annotations. In *Journal of Computer Science*, pages 519–526, 2009.

[TC06] Adrian D Thurston and James R Cordy. Evolving TXL. In *Proc. of 6th International Workshop on Source Code Analysis and Manipulation, Philadelphia, September 2006*, pages 117–126, 2006.

[WF11] Inc. Wikimedia Foundation. Precision and recall. Web page: `http://en.wikipedia.org/wiki/Precision_and_recall` (last accessed: November 5, 2011), 2011.

[WRH+00] Claes Wohlin, Per Runeson, Martin Host, Magnus C Ohlsson, Bjorn Regnell, and Anders Wesslen. *Experimentation in Software Engineering An Introdution.* Kluwer Academic Publishers, 2000.

Appendix A

Summary of Concurrency Design Pattern Roles

A.1 Single Threaded Execution Design Pattern

Table A.1: Single Threaded Execution Design Pattern Roles

Role ID:	Role Description:
1	Use of the Synchronized keyword as one of the method declarations modifiers. This is Java's way of implementing a "guarded method".

A.2 Lock Object Design Pattern

Table A.2: Lock Object Design Pattern Roles

Role ID:	Role Description:
1	Creation of a static object in a class - the Lock Object.
2	Creation of a static method in the same class as Role 1.
2a	Return of the Role 1 Lock Object within the Role 2 method (hence the reason it is 2a).
3	Synchronized calls to the Role 2 method.

A.3 Guarded Suspension Design Pattern

Table A.3: Guarded Suspension Design Pattern Roles

Role ID:	Role Description:
1	Ensuring a method in the class is synchronized and contains Role 1a below.
1a	Ensuring there is a notify() or notifyAll() statement within the Role 1 method.
2	Ensuring a method in the class is synchronized and contains Role 2a below.
2a	Ensuring there is a loop within the Role 2 method and contains Role 2aa below.
2aa	Ensuring there is a wait() statement within the Role 2a loop.

A.4 Balking Design Pattern

Table A.4: Balking Design Pattern Roles

Role ID:	Role Description:
1	Ensuring a method in the class is synchronized.
2	Ensuring the Role 1 method contains an if-then or if-then-else statement that tests a flag right at the start of the synchronized method.
3	Ensuring the Role 2 if statement or it's "else" does an immediate return - the Balking.

A.5 Scheduler Design Pattern

Table A.5: Scheduler Design Pattern Roles

Role ID:	Role Description:
1	Scheduler object class that contains Roles 1a and 1b below.
1a	Method with a parameter that is an instance of ScheduleOrdering object Role 3. Contains Roles 1aa,1ab, 1ac and 1ad.
1aa	New thread creation outside of any critical section.
1ab	Critical section creation by synchronization of this Scheduler object Role 1. Contains Role 1aba below.
1aba	Within Role 1ab a check to see whether the designated runningThread is null. If true proceed with Role 1abaa and 1abab. If false proceed with Role 1abac and 1abad.
1abaa	Assign thread Role 1aa (current thread) to the designated runningThread.
1abab	Return to calling Processor object Role 4.
1abac	Add thread Role 1aa to the list of waiting threads.
1abad	Add instance of ScheduleOrdering object Role 3 (that was passed into method Role 1a) into the list of waiting SchedulingOrdering object requests.
1ac	Critical section creation by synchronization of thread Role 1aa. Contains Role 1aca.
1aca	A loop within critical section Role 1ac to check if the new thread Role 1aa is NOT the same as the designated runningThread. If true proceed with Role 1acaa. If false then new thread Role 1aa is allowed to continue to run and proceeds to Role 1ad.
1acaa	New thread Role 1aa is placed in a waiting state until method Role 1b wakes it up using notifyAll().
1ad	Critical section creation by synchronization of this Scheduler object Role 1. Contains Role 1ada. Contains Role 1adb.
1ada	Remove current thread (Role 1aa) from the arraylist of waiting threads.
1adb	Remove current instance of the requested ScheduleOrdering object (Role 3), that was passed into method Role 1a, from the arraylist of waiting SchedulingOrdering object requests. Corresponds to the list of waiting threads.
1b	Synchronized method called when the current thread is finished with resource. Contains Role 1ba.
1ba	Critical section creation by synchronization of thread Role 1aa. Contains Role 1baa.
1baa	notifyAll() to wake up other waiting threads.

Table A.6: Scheduler Design Pattern Roles Continued

Role ID:	Role Description:
2	Request object - implements the ScheduleOrdering interface Role 3.
2a	private boolean method that helps in determining the order in which the request objects will occur.
3	Schedule Ordering interface implemented by the Role 2 Request object. Contains Role 3a.
3a	public boolean method that helps in determining the order in which the request objects will occur.
4	Processor object - delegates scheduling of the request objects processing to the Scheduler object one at a time. Contains Roles 4a and 4b.
4a	Creation of an instance of the Scheduler object (Role 1) outside of any method within Processor class (Role 4).
4b	Method with a parameter that is an instance of the Request object (Role 2) that carries out the main required functionality. Contains Role 4ba and 4bb.
4ba	Call to the method (Role 1a) of the instance (Role 4a) of the Scheduler object (Role 1). Occurs before any processing in method Role 4b
4bb	Call to the method (Role 1b) of the instance (Role 4a) of the Scheduler object (Role 1). Occurs after all processing in method Role 4b.

A.6 Read/Write Lock Design Pattern

Table A.7: Read/Write Lock Design Pattern Roles

Role ID:	Role Description:
1	Synchronized method to issue a read lock. Contains Roles 1a and 1b.
1a	Boolean check if the designated writeLockedThread has the write lock. If true i.e. a thread has the write lock then processing continues to Role 1aa and then Role 1ab. If false then processing continues to 1b.
1aa	Increment designated waitingForReadLock counter variable by 1.
1ab	Loop iteratively checking if the designated writeLockedThread has the write lock. As long as true i.e. a thread has the write lock Role 1aba occurs. When condition becomes false processing continues to Role 1ac.
1aba	wait() is called to pause further processing.
1ac	Decrement designated waitingForReadLock counter variable by 1.
1b	Increment designated outstandingReadLocks counter variable by 1.

Table A.8: Read/Write Lock Design Pattern Roles Continued

Role ID:	Role Description:
2	Method to issue a write lock. Contains Roles 2a, 2b, 2c and 2d.
2a	New thread creation outside of any critical section.
2b	Critical section creation by synchronization of this writelock method. Contains Role 2ba.
2ba	Within Role 2b a check whether the designated writelockedthread is null and designated outstandingReadLocks counter variable is zero. If true proceed with Role 2baa and 2bab. If false proceed with Role 2bac and 2bad.
2baa	Assign the current thread to the designated writelockedthread.
2bab	Return to the calling object that is using this method Role 2 of an instance of this object Role 1.
2bac	Make thread Role 2a the current thread.
2bad	Add thread Role 2a to arraylist.
2c	Critical section creation by synchronization of thread Role 2a. Contains Role 2ca.
2ca	A loop within critical section Role 2c to check if the new thread Role 2a is NOT the same as the designated writelockedthread. If true proceed with Role 2caa. If false then new thread Role 2a is allowed to continue to run and proceeds to Role 2d.
2caa	New thread Role 2a is placed in a waiting state until method Role 3 wakes it up using a notifyAll().
2d	Critical section creation by synchronization of this writelock method. Contains Role 2da.
2da	Remove current thread (Role 2a) from the arraylist of waiting threads.
3	Synchronized method called when the current thread is finished with resource. Contains Role 3a.
3a	notifyAll() to wake up other waiting threads.

A.7 Producer-Consumer Design Pattern

Table A.9: Producer-Consumer Design Pattern Roles

Role ID:	Role Description:
1	Producer class - supply objects to be consumed by the Role 3, the Consumer class. Contains Roles 1a, 1b and 1c.
1a	Local instance of Role 2, the Queue.
1b	Local instance of produced object.
1c	Call to push method of Role 1a, the local instance of the Queue. Pushes Role 1b, the produced object.
2	Queue class - buffer between producer and consumer classes. Contains Roles 2a, 2b and 2c.
2a	Array list to house the produced objects.
2b	Synchronized method to push the produced objects into queue. Contains Roles 2ba, 2bb and 2bc.
2ba	One of the parameters of Role 2b must have produced object.
2bb	Adding the produced object, Role 2ba to Role 2a, the arraylist.
2bc	Notification that the thread has completed.
2c	Synchronized method to pull the produced objects from queue to be consumed. Contains Roles 2ca, 2cb, 2cc and 2cd.
2ca	Loop to check if queue is empty by checking size of Role 2a.
2caa	Wait statement.
2cb	Creating instance of produced object and assigning it the 1st value in the arraylist Role 2a.
2cc	Remove the assigned value in Role 2cb from the arraylist Role 2a.
2cd	Returning the produced object - to be consumed by Role 3.
3	Consumer class - use objects to be produced by the Role 1, the Producer class. Contains Roles 3a, 3b and 3c.
3a	Local instance of Role 2, the Queue.
3b	Local instance of consumed object.
3c	Call to pull method of Role 3a, the local instance of the Queue. Pulls Role 3b, the object to be consumed.

A.8 Two-Phase Termination Design Pattern

Table A.10: Two-Phase Termination Design Pattern Roles

Role ID:	Role Description:
1	Thread(s) declaration - thread(s) that will be checked for an interrupt in Role 2.
2	Method running the process. Contains Roles 2a and 2b.
2a	In a loop checking the latch - thread in Role 1 being checked for Role 2aa.
2aa	Thread in Role 1 being checked if it has been interrupted.
2b	After the loop, a call to Role 4 that shuts down the thread.
3	Method that will contain functionality to set the latch - interrupt the thread in Role 1. Contains Role 3a.
3a	Actually setting the latch to true - interrupting the thread in Role 1.
4	Method that will contain functionality to stop the thread in Role 1. Contains Role 4a.
4a	Actually stopping of the thread in Role 1.

Appendix B

Concurrency Design Pattern Annotation Specifications

B.1 Single Threaded Execution Design Pattern

Table B.1: Single Threaded Execution Design Pattern Annotations

Role ID:	Annotation:
1	@SingleThreadedExecutionPattern(ID=1,role=1,comment="Method must be synchronized")

B.2 Lock Object Design Pattern

Table B.2: Lock Object Design Pattern Annotations

Role ID:	Annotation:
1	@LockObjectPattern(ID=1,role=1,comment="Creation of static object in a class - Lock Object.")
2	@LockObjectPattern(ID=1,role=2,comment="Creation of static method in the same class as Role 1 - getLockObject().")
2a	@LockObjectPattern(ID=1,role=2a,comment="Return of Lock Object, Role 1.")
3	@LockObjectPattern(ID=1,role=3,comment="Synchronized calls to method Role 2.")

B.3 Guarded Suspension Design Pattern

Table B.3: Guarded Suspension Design Pattern Annotations

Role ID:	Annotation:
1	@GuardendSuspensionPattern(ID=1,role=1,comment="Ensuring a method in the class is synchronized - guarded.")
1a	@GuardendSuspensionPattern(ID=1,role=1a,comment="Ensure there is a nofify() or notifyAll() statement.")
2	@GuardendSuspensionPattern(ID=1,role=2,comment="Ensuring a method in the class is synchronized - guarded.")
2a	@GuardendSuspensionPattern(ID=1,role=2a,comment="Ensuring there is a while statement.")
2aa	@GuardendSuspensionPattern(ID=1,role=2aa,comment="Ensuring there is a wait() statement.")

B.4 Balking Design Pattern

Table B.4: Balking Design Pattern Annotations

Role ID:	Annotation:
1	@BalkingPattern(ID=1,role=1,comment="Ensuring method is synchronized - guarded.")
2	@BalkingPattern(ID=1,role=2,comment="Ensure an if statement that tests a flag right at the start of the synchronized method.")
3	@BalkingPattern(ID=1,role=3,comment="Ensuring one of the if or else tests of the flag in Role 2 does an immediate return - Balking.")

B.5 Scheduler Design Pattern

Table B.5: Scheduler Design Pattern Annotations

Role ID:	Annotation:
1	@SchedulerPattern(ID=1,role=1,comment="Scheduler object/class.")
1a	@SchedulerPattern(ID=1,role=1a,comment="Method with a parameter that is an instance of ScheduleOrdering object Role 3.")
1aa	@SchedulerPattern(ID=1,role=1aa,comment="New thread creation outside of any critical section.")
1ab	@SchedulerPattern(ID=1,role=1ab,comment="Critical section creation by synchronization of this Scheduler object Role 1.")
1aba	@SchedulerPattern(ID=1,role=1aba,comment="Within Role 1ab a check to whether the designated runningThread is null.")
1abaa	@SchedulerPattern(ID=1,role=1abaa,comment="Assign thread Role 1aa (current thread) to the designated runningThread.")
1abab	@SchedulerPattern(ID=1,role=1abab,comment="Return to calling Processor object Role 4.")
1abac	@SchedulerPattern(ID=1,role=1abac,comment="Add thread Role 1aa to the list of waiting threads.")
1abad	@SchedulerPattern(ID=1,role=1abad,comment="Add instance of ScheduleOrdering object Role 3 (that was passed into method Role 1a) into the list of waiting SchedulingOrdering object requests.")
1ac	@SchedulerPattern(ID=1,role=1ac,comment="Critical section creation by synchronization of thread Role 1aa.")
1aca	@SchedulerPattern(ID=1,role=1aca,comment="A loop within critical section Role 1ac to check if the new thread Role 1aa is NOT the same as the designated runningThread.")
1acaa	@SchedulerPattern(ID=1,role=1acaa,comment="New thread Role 1aa is placed in a waiting state until method Role 1b wakes it up using nofityAll().")
1ad	@SchedulerPattern(ID=1,role=1ad,comment="Critical section creation by synchronization of this Scheduler object Role 1.")
1ada	@SchedulerPattern(ID=1,role=1ada,comment="Remove current thread (Role 1aa) from the list of waiting threads.")
1adb	@SchedulerPattern(ID=1,role=1adb,comment="Remove current instance of the requested ScheduleOrdering object (Role 3), that was passed into method Role 1a, from the arraylist of waiting SchedulingOrdering object requests. Correspond to the list of waiting threads.")

Table B.6: Scheduler Design Pattern Annotations Continued

Role ID:	Annotation:
1b	@SchedulerPattern(ID=1,role=1b,comment="Synchronized method called when the current thread is finished with resource.")
1ba	@SchedulerPattern(ID=1,role=1ba,comment="Critical section creation by synchronization of thread Role 1aa.")
1baa	@SchedulerPattern(ID=1,role=1baa,comment="NotifyAll to wake up other waiting threads.")
2	@SchedulerPattern(ID=1,role=2,comment="Request object - implements the ScheduleOrdering interface Role 3.")
2a	@SchedulerPattern(ID=1,role=2a,comment="private boolean method that helps in determining the order in which the request objects will occur.")
3	@SchedulerPattern(ID=1,role=3,comment="Schedule Ordering interface implemented by the Role 2 Request object.")
3a	@SchedulerPattern(ID=1,role=3a,comment="public boolean method that helps in determining the order in which the request objects will occur.")
4	@SchedulerPattern(ID=1,role=4,comment="Processor object - delegates scheduling of the request objects processing to the Scheduler object one at a time.")
4a	@SchedulerPattern(ID=1,role=4a,comment="Creation of an instance of the Scheduler object (Role 1) outside of any method within Processor class(Role 4).")
4b	@SchedulerPattern(ID=1,role=4b,comment="Method with a parameter that is an instance of the Request object (Role 2) that carries out the main required functionality.")
4ba	@SchedulerPattern(ID=1,role=4ba,comment="Call to the method (Role 1a) of the instance (Role 4a) of the Scheduler object (Role 1). Occurs before any processing in method Role 4b.")
4bb	@SchedulerPattern(ID=1,role=4bb,comment="Call to the method (Role 1b) of the instance (Role 4a) of the Scheduler object (Role 1). Occurs after all processing in method Role 4b.")

B.6 Read/Write Lock Design Pattern

Table B.7: Read/Write Lock Design Pattern Annotations

Role ID:	Annotation:
1	@ReadWriteLockPattern(ID=1,role=1,comment="Synchronized method to issue a read lock.")
1a	@ReadWriteLockPattern(ID=1,role=1a,comment="Boolean check if the designated writelockedthread has the write lock.")
1aa	@ReadWriteLockPattern(ID=1,role=1aa,comment="Increment designated waitingForReadLock counter variable by 1.")
1ab	@ReadWriteLockPattern(ID=1,role=1ab,comment="Loop iteratively checking if the designated writeLockedThread has the write lock.")
1aba	@ReadWriteLockPattern(ID=1,role=1aba,comment="wait() is called to pause further processing.")
1ac	@ReadWriteLockPattern(ID=1,role=1ac,comment="Decrement designated waitingForReadLock counter variable by 1.")
1b	@ReadWriteLockPattern(ID=1,role=1b,comment="Increment designated outstandingReadLocks counter variable by 1.")

Table B.8: Read/Write Lock Design Pattern Annotations Continued

Role ID:	Annotation:
2	@ReadWriteLockPattern(ID=1,role=2,comment="Method to issue a write lock.")
2a	@ReadWriteLockPattern(ID=1,role=2a,comment="New thread creation outside of any critical section.")
2b	@ReadWriteLockPattern(ID=1,role=2b,comment="Critical section creation by synchronization of this writelock method.")
2ba	@ReadWriteLockPattern(ID=1,role=2ba,comment="Within Role 2b a check to whether the designated writeLockedThread is null and designated outstandingReadLocks counter variable is zero.")
2baa	@ReadWriteLockPattern(ID=1,role=2baa,comment="Assign the current thread to the designated writeLockedThread.")
2bab	@ReadWriteLockPattern(ID=1,role=2bab,comment="Return to the calling object that is using this method Role 2 of an instance of this object Role 1 .")
2bac	@ReadWriteLockPattern(ID=1,role=2bac,comment="Make thread Role 2a the current thread.")
2bad	@ReadWriteLockPattern(ID=1,role=2bad,comment="Add thread Role 2a to the arraylist.")
2c	@ReadWriteLockPattern(ID=1,role=2c,comment="Critical section creation by synchronization of thread Role 2a.")
2ca	@ReadWriteLockPattern(ID=1,role=2ca,comment="A loop within critical section Role 2c to check if the new thread Role 2a is NOT the same as the designated writeLockedThread.")
2caa	@ReadWriteLockPattern(ID=1,role=2caa,comment="New thread Role 2a is placed in a waiting state until method Role 3 wakes it up using a nofityAll().")
2d	@ReadWriteLockPattern(ID=1,role=2d,comment="Critical section creation by synchronization of this writelock method.")
2da	@ReadWriteLockPattern(ID=1,role=2da,comment="Remove current thread (Role 2a) from the arraylist of waiting threads.")
3	@ReadWriteLockPattern(ID=1,role=3,comment="Synchronized method called when the current thread is finished with resource.")
3a	@ReadWriteLockPattern(ID=1,role=3a,comment="NotifyAll to wake up other waiting threads.")

B.7 Producer-Consumer Design Pattern

Table B.9: Producer-Consumer Design Pattern Annotations

Role ID:	Annotation:
1	@ProducerConsumerPattern(ID=1,role=1,comment="Producer class - supply objects to be consumed by the Role 3, the Consumer class.")
1a	@ProducerConsumerPattern(ID=1,role=1a,comment="Local instance of Role 2, the Queue.")
1b	@ProducerConsumerPattern(ID=1,role=1b,comment="Local instance of produced object.")
1c	@ProducerConsumerPattern(ID=1,role=1c,comment="Call to push method of Role 1a, the local instance of the Queue. Pushes Role 1b, the produced object.")
2	@ProducerConsumerPattern(ID=1,role=2,comment="Queue class - buffer between producer and consumer classes.")
2a	@ProducerConsumerPattern(ID=1,role=2a,comment="Array list to house the produced objects.")
2b	@ProducerConsumerPattern(ID=1,role=2b,comment="Synchronized method to push the produced objects into queue.")
2ba	@ProducerConsumerPattern(ID=1,role=2ba,comment="One of the parameters of Role 2b must have produced object.")
2bb	@ProducerConsumerPattern(ID=1,role=2bb,comment="Adding the produced object, Role 2ba to Role 2a, the arraylist.")
2bc	@ProducerConsumerPattern(ID=1,role=2bc,comment="Nofification that the thread has completed.")
2c	@ProducerConsumerPattern(ID=1,role=2c,comment="Synchronized method to pull the produced objects from queue to be consumed.")
2ca	@ProducerConsumerPattern(ID=1,role=2ca,comment="Loop to check if queue is empty by checking size of Role 2a.")
2caa	@ProducerConsumerPattern(ID=1,role=2caa,comment="Wait statement.")
2cb	@ProducerConsumerPattern(ID=1,role=2cb,comment="Creating instance of produced object and assigning it the 1st value in the arraylist Role 2a.")
2cc	@ProducerConsumerPattern(ID=1,role=2cc,comment="Remove the assigned value in Role 2cb from the arraylist Role 2a.")
2cd	@ProducerConsumerPattern(ID=1,role=2cd,comment="Returning the produced object - to be consumed by Role 3.")

Table B.10: Producer-Consumer Design Pattern Annotations Continued

Role ID:	Annotation:
3	@ProducerConsumerPattern(ID=1,role=3,comment="Consumer class - use objects to be produced by the Role 1, the Producer class.")
3a	@ProducerConsumerPattern(ID=1,role=3a,comment="Local instance of Role 2, the Queue.")
3b	@ProducerConsumerPattern(ID=1,role=3b,comment="Local instance of consumed object.")
3c	@ProducerConsumerPattern(ID=1,role=3c,comment="Call to pull method of Role 3a, the local instance of the Queue. Pulls Role 3b, the object to be consumed.")

B.8 Two-Phase Termination Design Pattern

Table B.11: Two-Phase Termination Design Pattern Annotations

Role ID:	Annotation:
1	@TwoPhaseTerminationPattern(ID=1,role=1,comment="Thread(s) declaration - thread(s) that will be checked for an interrupt in Role 2.")
2	@TwoPhaseTerminationPattern(ID=1,role=2,comment="Method running the process.")
2a	@TwoPhaseTerminationPattern(ID=1,role=2a,comment="In a loop checking the latch - thread in Role 1 being checked for Role 2aa.")
2aa	@TwoPhaseTerminationPattern(ID=1,role=2aa,comment="Thread in Role 1 being checked if it has been interrupted.")
2b	@TwoPhaseTerminationPattern(ID=1,role=2b,comment="After the loop, a call to Role 4 that shuts down the thread.")
3	@TwoPhaseTerminationPattern(ID=1,role=3,comment="Method that will contain functionality to set the latch - interrupt the thread in Role 1.")
3a	@TwoPhaseTerminationPattern(ID=1,role=3a,comment="Actually setting the latch to true - interrupting the thread in Role 1.")
4	@TwoPhaseTerminationPattern(ID=1,role=4,comment="Method that will contain functionality to stop the thread in Role 1.")
4a	@TwoPhaseTerminationPattern(ID=1,role=4a,comment="Actually stopping of the thread in Role 1.")