Getting to Know Your Java Object's State of Mind with the State Design Pattern

by Barry A. Burd and Michael P. Redlich

Introduction

This article, the fifth in a series about design patterns, introduces the State design pattern, one of the 23 design patterns defined in the legendary 1995 book *Design Patterns – Elements of Reusable Object-Oriented Software*. The authors of this book, Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides, are known affectionately as the *Gang of Four* (GoF).

Design Patterns

The GoF book defines 23 design patterns. The patterns fall into three categories:

- A creational pattern abstracts the instantiation process.
- A structural pattern groups objects into larger structures.
- A behavioral pattern defines better communication among objects.

The State design pattern fits into the behavioral category.

The State Pattern

According to the GoF book, the State design pattern:

"Allows an object to alter its behavior when its internal state changes. The object will appear to change its class."

A change in object behavior can be accomplished by encapsulating states into separate classes. That explains the first part of the definition. In the second part of the definition, an object can be made to appear to change its class by referencing the state objects as necessary. This can be accomplished through the use of composition, and as always, we will demonstrate this through an example.

Motivation

A finite state machine (FSM) is commonly used for modeling things like a vending machine. To demonstrate the motivation for using the State design pattern, we adapted a subway turnstile application that Robert Martin (affectionately known as "Uncle Bob") used in his June 1998 article, UML:Tutorial: Finite State Machines, published in C++ Report. Our version of a very simple subway turnstile application will be implemented in Java and presented here.

So let's first start with a state transition diagram.

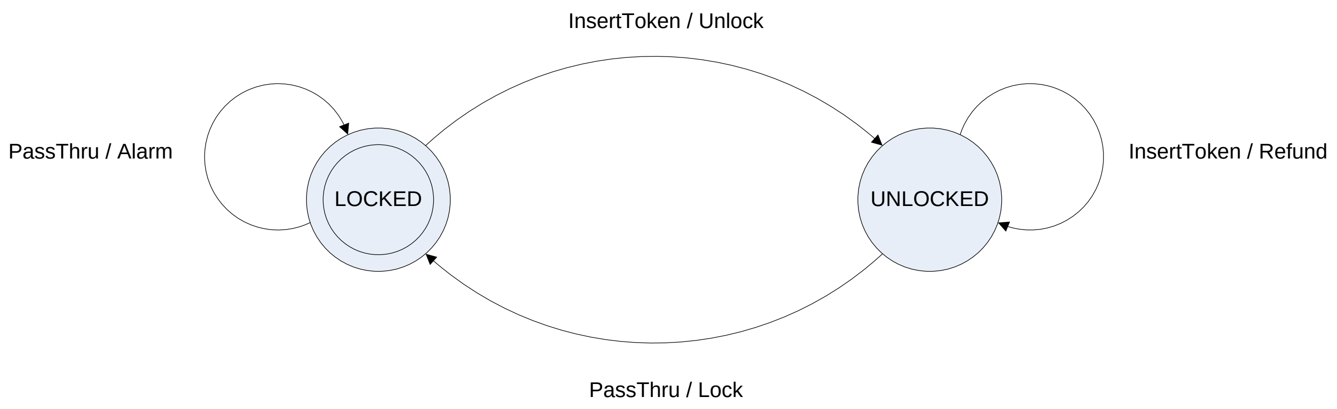

Figure 1: The state transition diagram for the turnstile application

For those of you that may not have studied state transition diagrams for awhile, let's walk through this diagram as a quick refresher. The circles represent the states of the turnstile. The turnstile has **`LOCKED`** and **`UNLOCKED`** states. The **`LOCKED`** state is the initial state since it is denoted with the inner circle. Please note that there are other ways of denoting the initial state. The arrows represent transitions from one state to another. Each transition is labeled with an event that will trigger a corresponding action. For example, in the transition from **`LOCKED`** to **`UNLOCKED`**, **`InsertToken`** is an event and its corresponding action is **`Unlock`**. Implementing such a FSM has traditionally been accomplished using a set of conditionals or using **`switch`**/**`case`** statements. Let's see what this looks like in code using **`switch`**/**`case`** statements as shown in Listing 1.

```
public class TurnstileApp
      {
      private State state;

      private enum State
            {
            LOCKED,
            UNLOCKED
            }

      private enum Event
            {
            InsertToken,
            PassThru
            }

      public TurnstileApp()
            {
            // set initial state of turnstile
            state = State.LOCKED;
            }

      public static void main(String[] args)
            {
            TurnstileApp turnstile = new TurnstileApp();

            // first person inserts a token and passes through the turnstile
            System.out.println(turnstile);
            turnstile.transition(Event.InsertToken);
            turnstile.transition(Event.PassThru);

            // second person attempts to pass through turnstile without inserting a token
            System.out.println(turnstile);
            turnstile.transition(Event.PassThru);
            turnstile.transition(Event.InsertToken);
            turnstile.transition(Event.PassThru);

            // third person attempts to pass through turnstile without initially inserting a 
token.
            System.out.println(turnstile);
            turnstile.transition(Event.InsertToken);
            turnstile.transition(Event.InsertToken);
            turnstile.transition(Event.PassThru);
            }

      public void transition(Event event)
            {
            switch(state)
                  {
                  case LOCKED:
                        switch(event)
                              {
                              case InsertToken:
                                    System.out.println("InsertToken: Accepting token in the " + 
state + " state...");
                                    unlock();
                                    break;
                              case PassThru:
                                    System.out.println("PassThru: Passing through in the " + 
state + " state...");
                                    alarm();
                                    break;
                              }
                        break;
                  case UNLOCKED:
                        switch(event)
                              {
                              case InsertToken:
                                    System.out.println("InsertToken: Accepting token in the " + 
state + " state...");
                                    refund();
                                    break;
                              case PassThru:
                                    System.out.println("PassThru: Passing through in the " + 
state + " state...");
                                    lock();
                                    break;
                              }
                        break;
                  }
            }

      public void lock()
            {
            System.out.println("Locking the turnstile...");
            state = State.LOCKED;
            System.out.println(state);
            }

      public void unlock()
            {
            System.out.println("Unlocking the turnstile...");
            state = State.UNLOCKED;
            System.out.println(state);
            }

      public void alarm()
            {
```

```
            System.out.println("Alert! Cannot pass through without inserting a token!!");
            }

      public void refund()
            {
            System.out.println("Refunding the token...");
            }

      public String toString()
            {
            StringBuffer result = new StringBuffer();
            result.append("\n-- Welcome to the Turnstile Application --\n");
            result.append(state);
            return result.toString();
            }
      }
```

Listing 1: A traditional method of implementing the FSM for the subway turnstile application

The states, **`LOCKED`** and **`UNLOCKED`**, and events, **`InsertCoin`** and **`PassThru`**, are encapsulated in their respective enumerations. The methods, **`lock()`**, **`unlock()`**, **`refund()`**, and **`alarm()`**, represent the actions triggered by one of the events. The **`transition()`** method handles, well, the transition from one state to the next. Notice that each state has its own nested **`switch`**/**`case`** statements. This already seems a bit complicated, but this implementation of the subway turnstile works well with one ***major*** assumption. Say you wanted to expand this application by adding states. Additional **`case`**s to accommodate the additional states must be written, and within these new **`case`**s, each will have their own nested **`switch`**/**`case`** statements to handle the related events and actions. Hmmm, it appears that the **`transition()`** method will now become even more complicated.. We're sure that you would agree that this is not the way to develop the subway turnstile application using an FSM. It is just not resilient to change.

UML Diagram

Figure 1 shows the official UML diagram for the State pattern.

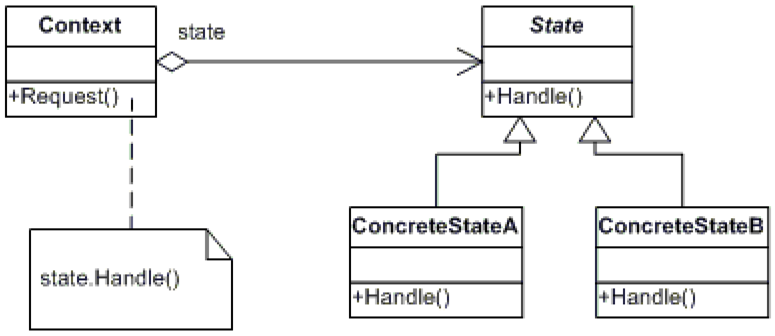

Figure 2: The UML diagram for the State design pattern

The diagram contains four classes:

- The **`Context`** class represents a process that manages a set of states. In the subway turnstile application, we will define a class, **`Turnstile`**, as the context.
- The **`State`** interface declares the methods for encapsulating the transition events.
- A **`ConcreteState`** class implements the **`State`** interface for each of the states in an application and takes care of requests from the **`Context`** class.

Figure 2 shows a State UML diagram that's specific to the subway turnstile application:

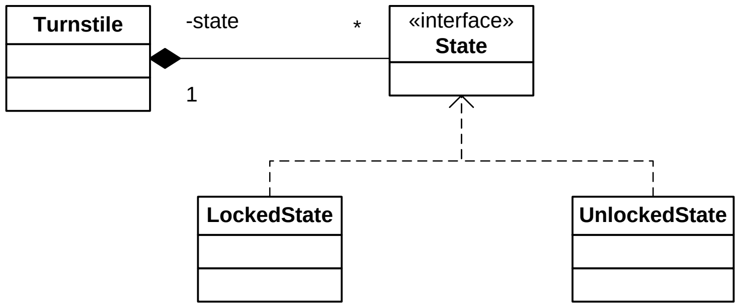

Figure 2: The UML diagram for the car ordering application.

The source code for the diagram of Figure 2 is shown in Listings 2 to 5:

```
public interface State
      {
      public void insertToken();
      public void passThru();
      }
```

Listing 2: The State interface

```
public class LockedState implements State
      {
      private final Turnstile turnstile;

      public LockedState(Turnstile turnstile)
            {
            this.turnstile = turnstile;
            }

      public void insertToken()
            {
            System.out.println("insertToken(): Accepting token in the " + turnstile.getState() +
" state...");
            turnstile.unlock();
            }

      public void passThru()
            {
            System.out.println("passThru(): Passing through in the " + turnstile.getState() + "
state....");
            turnstile.alarm();
            }

      public String toString()
            {
            return "LOCKED";
            }
      }
```

Listing 3. The LockedState (ConcreteState) class

```
public class UnlockedState implements State
```

```
      {
      private final Turnstile turnstile;

      public UnlockedState(Turnstile turnstile)
            {
            this.turnstile = turnstile;
            }

      public void insertToken()
            {
            System.out.println("insertToken(): Accepting token in the " + turnstile.getState() +
" state...");
            turnstile.refund();
            }

      public void passThru()
            {
            System.out.println("passThru(): Passing through in the " + turnstile.getState() + "
state...");
            turnstile.lock();
            }

      public String toString()
            {
            return "UNLOCKED";
            }
      }
```

Listing 4. The UnlockedState (ConcreteState) class

```
public class Turnstile
      {
      private State state;
      private final State lockedState;
      private final State unlockedState;

      public Turnstile()
            {
            lockedState = new LockedState(this);
            unlockedState = new UnlockedState(this);

            // set initial state of turnstile
            state = lockedState;
            }

      public State getState()
            {
            return state;
            }

      public void setState(State state)
            {
            this.state = state;
            }

      public void insertToken()
            {
            state.insertToken();
            }

      public void passThru()
            {
            state.passThru();
            }

      public void lock()
            {
            System.out.println("Locking the turnstile...");
            setState(lockedState);
            System.out.println(getState());
            }

      public void unlock()
            {
            System.out.println("Unlocking the turnstile...");
            setState(unlockedState);
            System.out.println(getState());
            }

      public void alarm()
            {
            System.out.println("Alert! Cannot pass through without inserting a token!!");
            }

      public void refund()
            {
            System.out.println("Refunding the token...");
            }

      public String toString()
            {
            StringBuffer result = new StringBuffer();
            result.append("\n-- Welcome to the Turnstile Application --\n");
            result.append(getState());
            return result.toString();
            }
      }
```

Listing 5. The Turnstile (Context) class

The **`State`** interface in Listing 2 has been established to declare the methods for transition events. These methods, **`insertToken()`** and **`passThru()`**, replace the enumerations **`InsertToken`** and **`PassThru`** from the FSM implementation. Each state is now represented by the classes, **`LockedState`** and **`UnlockedState`**, which implement the **`State`** interface. Notice that these two concrete classes store an instance of **`Turnstile`**. This allows you to call the action-related methods for those states. The **`Turnstile`** class, the context in this application, creates the objects, **`lockedState`** and **`unlockedState`**, representing each state, and a generic state variable, **`state`**, which will be used as the current state. The action methods, **`lock()`**, **`unlock()`**, **`refund()`**, and **`alarm()`**, are defined along with **`insertToken()`** and **`passThru()`** methods that are basically a convenience for the current state to call it's corresponding event method. As you can see, state changes occur in the actions methods as necessary.

Listing 6 is the client application:

```
public class TurnstileApp
      {
      private TurnstileApp()
            {
            // using default private constructor in lieu of creating a Singleton object
            }

      public static void main(String[] args)
            {
            Turnstile turnstile = new Turnstile();
            // first person inserts a token and passes through the turnstile
            System.out.println(turnstile);
            turnstile.insertToken();
            turnstile.passThru();

            // second person attempts to pass through turnstile without inserting a token
            System.out.println(turnstile);
            turnstile.passThru();
            turnstile.insertToken();
            turnstile.passThru();

            // third person attempts to pass through turnstile without initially inserting a
token.
            System.out.println(turnstile);
            turnstile.insertToken();
            turnstile.insertToken();
            turnstile.passThru();
            }
      }
```

Listing 6: The TurnstileApp class – our client application

The client application starts by creating an instance of the **`Turnstile`** class. And as you already know, **`Turnstile`** creates the objects for each state. The initial state of the turnstile is **`LOCKED`**.

In the first scenario, a person approaches the turnstile and inserts his/her token into the turnstile. The statement,

```
turnstile.insertToken();
```

captures this event by calling the **`insertToken()`** method defined in the **`Turnstile`** class:

```
public void insertToken()
      {
      state.insertToken();
      }
```

Since the current state is **`LOCKED`**, the **`state`** variable, currently assigned to the **`lockedState`** variable, calls the **`insertToken()`** method that is defined in the **`LockedState`** class.

```
public void insertToken()
      {
      System.out.println("insertToken(): Accepting token in the " + turnstile.getState() + "
state...");
      turnstile.unlock();
      }
```

Here is where things change. The **`unlock()`** method to execute the **`Unlock`** action (as shown in the state transition diagram) is called to unlock the turnstile.

```
public void unlock()
      {
      System.out.println("Unlocking the turnstile...");
      setState(unlockedState);
      System.out.println(getState());
      }
```

The **`setState()`** method passes in the object, **`unlockedState`**, and assigns it to the variable, **`state`**, as the current state. As control returns to the client application, the turnstile is now unlocked. The next event,

```
turnstile.passThru();
```

is called, and is defined as

```
public void passThru()
      {
      state.passThru();
      }
```

This time, the **`passThru()`** method is called from within the **`UnlockedState`** class. Since the turnstile is **`UNLOCKED`** state, all is well. The person passes through the turnstile without incident, and the turnstile then returns to the **`LOCKED`** state.

In one of the other scenarios defined in the client application, a person attempts to pass through the turnstile in the locked position without inserting a token. In this case, the **`passThru()`** method defined in the **`LockedState`** class sends and alert and remains in the locked position. So the application basically delegates which event-related method is called based on the current state. That is, the statement,

```
state.passThru();
```

when **`state = lockedState`** is different from when **`state = unlockedState`**.

Figure 3 shows the output of the subway turnstile application.

```
-- Welcome to the Turnstile Application --
LOCKED
insertToken(): Accepting token in the LOCKED state...
Unlocking the turnstile...
UNLOCKED
passThru(): Passing through in the UNLOCKED state...
Locking the turnstile...
LOCKED

-- Welcome to the Turnstile Application --
LOCKED
passThru(): Passing through in the LOCKED state....
Alert! Cannot pass through without inserting a token!!
insertToken(): Accepting token in the LOCKED state...
Unlocking the turnstile...
UNLOCKED
passThru(): Passing through in the UNLOCKED state...
Locking the turnstile...
LOCKED

-- Welcome to the Turnstile Application --
LOCKED
insertToken(): Accepting token in the LOCKED state...
Unlocking the turnstile...
UNLOCKED
insertToken(): Accepting token in the UNLOCKED state...
Refunding the token...
passThru(): Passing through in the UNLOCKED state...
Locking the turnstile...
LOCKED
```

Figure 3: The output of the subway turnstile application

What Happens When You Add States to the Application?

Let's say you want to a new state, say an **`ALARM`** state, to the subway turnstile application. Hmmm, that sounds like change. No problem! Design principles are here to guide you! The state transition diagram needs to be changed first. Let's take a look at Figure 4:

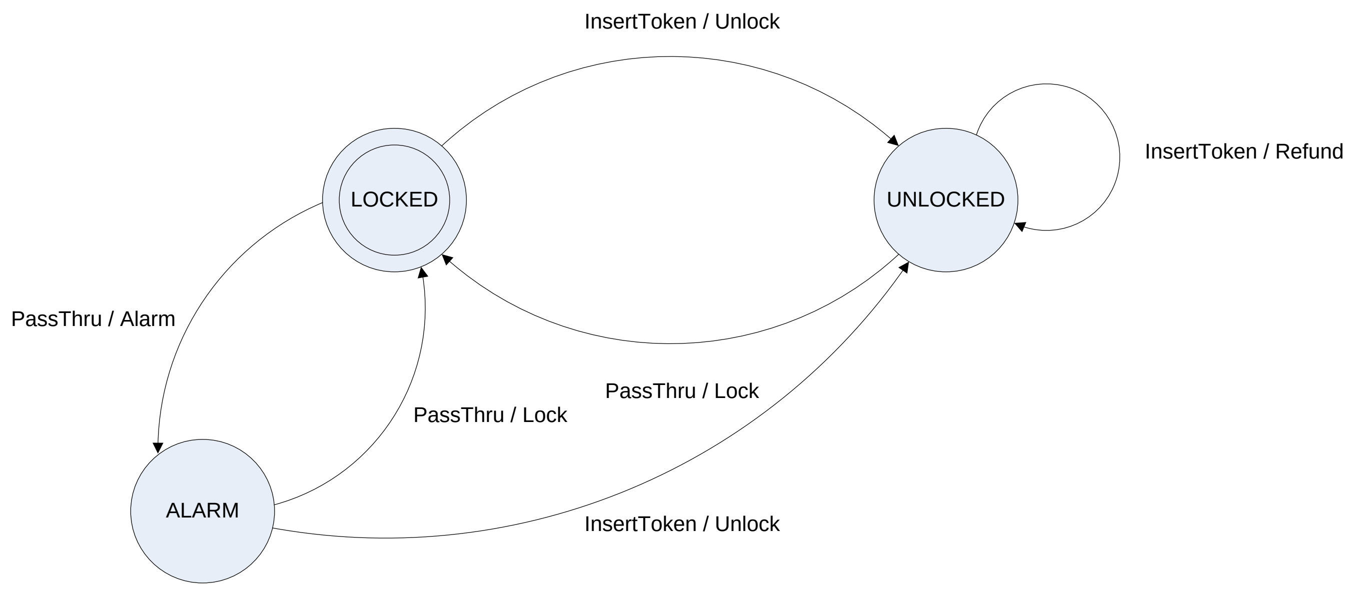

Figure 4: The revised state transition diagram for the subway turnstile application

In this new diagram, when someone attempts to pass through the turnstile in the **`LOCKED`** state, the turnstile now transitions to the **`ALARM`** state, and you can assume that the turnstile is still locked. If the person then inserts a token, the turnstile unlocks, and proceeds to the **`UNLOCKED`** state. However, if the person still insists on pushing on the turnstile bar to pass through without inserting a token, the turnstile goes back into **`LOCKED`** state.

OK, so now that there's a new state transition diagram to use as a model, let's add an **`AlarmState`** class to the application:

```
public class AlarmState implements State
      {
      private final Turnstile turnstile;

      public AlarmState(Turnstile turnstile)
            {
            this.turnstile = turnstile;
            }

      public void insertToken()
            {
            System.out.println("insertToken(): Accepting token in the " + turnstile.getState() +
" state...");
            turnstile.unlock();
            }

      public void passThru()
```

```
            {
            System.out.println("passThru(): Passing through in the " + turnstile.getState() + "
state...");
            turnstile.lock();
            }

      public String toString()
            {
            return "ALARM";
            }
      }
```

Listing 7. The AlarmState (ConcreteState) class

Since the **`Turnstile`** class is the context of the application, this is the only place to make the necessary changes. An instance variable, **`alarmState`**, representing the **`ALARM`** state is added first:

```
private State state;
private final State lockedState;
private final State unlockedState;
private final State alarmState; // new instance variable added
```

Next, the **`alarmState`** object is created in the constructor along with the original states:

```
public Turnstile()
      {
      lockedState = new LockedState(this);
      unlockedState = new UnlockedState(this);
      alarmState = new AlarmState(this); // create the alarmState object

      // set initial state of turnstile
      state = lockedState;
      }
```

Finally, the **`alarm()`** method, which originally didn't specify a change of state, now does so to get to the **`ALARM`** state:

```
public void alarm()
      {
      System.out.println("Alert! Cannot pass through without inserting a token!!");
      setState(alarmState); // set the state to ALARM state
      System.out.println(getState()); // send the state status to the console
      }
```

That's it! If you run the client application again, you will get the slightly different output as shown in Figure 5:

```
-- Welcome to the Turnstile Application --
LOCKED
insertToken(): Accepting token in the LOCKED state...
Unlocking the turnstile...
UNLOCKED
passThru(): Passing through in the UNLOCKED state...
Locking the turnstile...
LOCKED

-- Welcome to the Turnstile Application --
LOCKED
passThru(): Passing through in the LOCKED state....
Alert! Cannot pass through without inserting a token!!
ALARM
insertToken(): Accepting token in the ALARM state...
```

```
Unlocking the turnstile...
UNLOCKED
passThru(): Passing through in the UNLOCKED state...
Locking the turnstile...
LOCKED

-- Welcome to the Turnstile Application --
LOCKED
insertToken(): Accepting token in the LOCKED state...
Unlocking the turnstile...
UNLOCKED
insertToken(): Accepting token in the UNLOCKED state...
Refunding the token...
passThru(): Passing through in the UNLOCKED state...
Locking the turnstile...
LOCKED
```

Figure 5: The output of the revised subway turnstile application

What Your Java Object's State of Mind?

As you can see, the State design pattern is an excellent way to implement FSM-type applications. If you take some time to compile and run the original version of the application, i.e., the one using all of those pesky **`switch`**/**`case`** statements, you will notice that the output looks exactly the same (well, it did until we added the **`ALARM`** state). This was done intentionally, of course, but we wanted to make the point that despite having evolved the application from using complicated **`switch`**/**`case`** statements to using the State design pattern, the subway turnstile application functions ***exactly*** the same. The behavior of each state has been localized into its own class, and we've closed each state for modification, but we were able to allow the subway turnstile application to be open for extension when the **`ALARM`** state was added. For those of you that have followed along our series of design patterns, you probably already know recognize this as the ***Open-Closed Principle*** that we've demonstrated in previous articles.

Resources

Design Patterns – Elements of Reusable Object-Oriented Software
http://www.awprofessional.com/bookstore/product.asp?isbn=0201633612&rl=1
Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides
ISBN 0-201-63361-2

Head First Design Patterns
http://www.oreilly.com/catalog/hfdesignpat/
Eric & Elisabeth Freeman
ISBN 0-596-00712-4

UML Tutorial: Finite State Machines
http://rd13doc.cern.ch/Atlas/DaqSoft/sde/uml/UMLFSM.pdf
Robert Martin
C++ Report, June 1998

Object-Oriented Software Construction
http://vig.prenhall.com/catalog/academic/product/0,1144,0136291554.html,00.html
Bertrand Meyer
ISBN 0-13-629155-4

Data & Object Factory
http://www.dofactory.com/Patterns/Patterns.aspx

About the Authors

Barry Burd is a professor in the Department of Mathematics and Computer Science at Drew University in Madison, New Jersey. When he's not lecturing at Drew University, Dr. Burd leads training courses for professional programmers in business and industry. He has lectured at conferences in America, Europe, Australia and Asia. He is the author of several articles and books, including “Java 2 For Dummies” and “Eclipse For Dummies,” both published by Wiley.

Michael P. Redlich is a Senior Research Technician (formerly a Systems Analyst) at ExxonMobil Research & Engineering, Co. in Clinton, New Jersey with extensive experience in developing custom web and scientific laboratory applications. Mike also has experience as a Technical Support Engineer for Ai-Logix, Inc. where he developed computer telephony applications. He holds a Bachelor of Science in Computer Science from Rutgers University. In his spare time, Mike facilitates the ACGNJ Java Users Group and serves as ACGNJ Secretary. Mike has also co-written several articles for Java Boutique, and his computing experience includes computer security, relational database design and development, object-oriented design and analysis, C/C++, Java, Visual Basic, FORTRAN, Pascal, MATLAB, HTML, XML, ASP, VBScript, and JavaScript in both the PC and UNIX environments.