Nels E. Beckman

Case Studies in Concurrent Object Protocols

This Talk

- We have two concurrent object protocol (typestate) checkers
  - NIMBY & Sync or Swim
- I’ve been using them to verify real programs
  - Interesting encounters
    - Surprising Power (Often, but just a little in this talk)
    - Tricky Patterns (**Mostly**)
- Possible extensions
  - Inspired by interesting encounters

Concurrent Typestate Checkers

- Static typestate checking in multi-threaded Java programs
  - I.e., methods that must be called in a particular order
  - Both extensions of Plural
  - NIMBY
    - Checks programs with atomic blocks
  - Sync or Swim
    - Checks programs with synchronized blocks

Does it work?

- Can we use our tools to verify real Java programs?
- Let’s find out!
  - Search open source code bases
  - Find classes that are used concurrently & define protocols
  - Specify them!
  - Verify them!
  - Note patterns & deficiencies

Encounters

- Blocking_queue
  - Cool use of dimensions!
- Timer & Timer Task
  - A simple protocol
  - Motivates polymorphism over permissions
- Causal Demo
  - Shutdown hooks?!
- Dining Philosophers
  - Effects are *still* hard…

Blocking_queue.java

- A concurrent queue
  - Designed by Allen Holub
  - Used in a number of open-source apps.
    - E.g., Axl Lucene

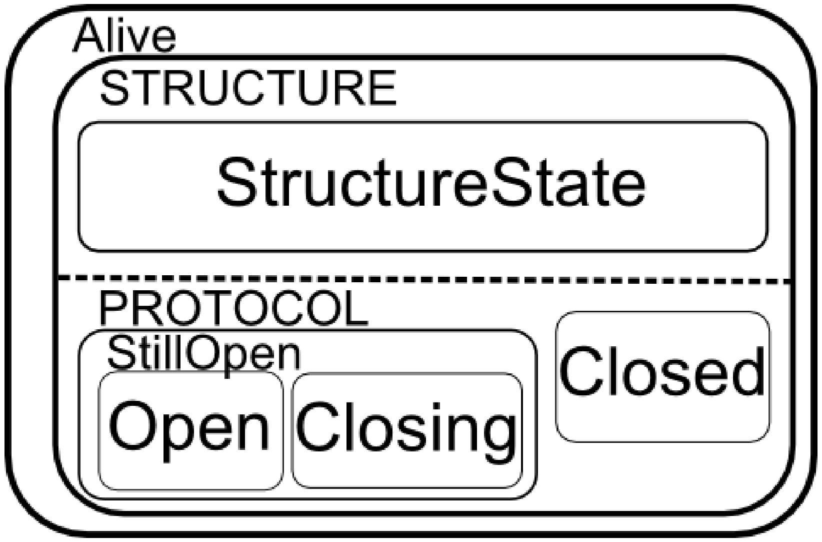
- Specified & verified
  - Client-side & implementation
  - Required interesting use of dimensions
  - 21 annotations in 84 LOC
  - 0 Warnings

Recall...

- Dimensions allow programmers to "divide up an object"
  - Specify certain fields as being grouped together
  - Can be treated as an atom in specs.

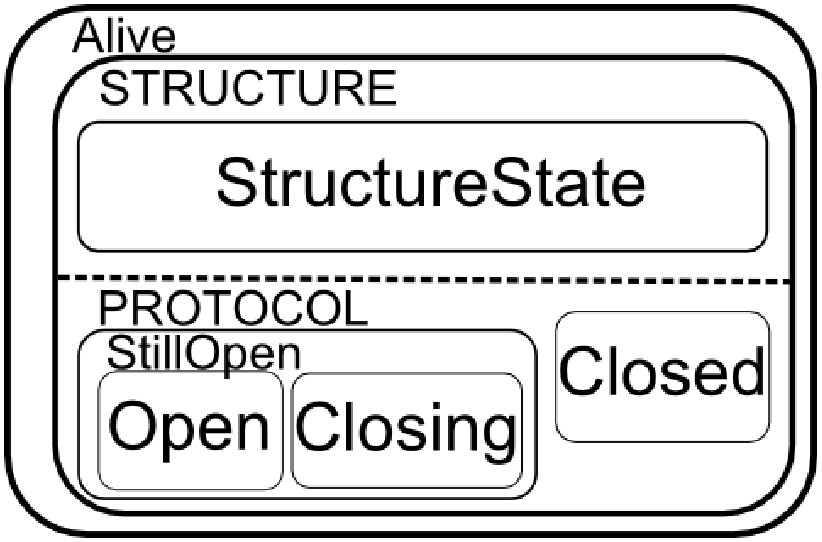
Blocking_queue Specification

```
@Refine({
@States(dim="STRUCTURE", value={"STRUCTURESTATE"}),
@States(dim="PROTOCOL", value= {"CLOSED", "STILLOPEN"}),
@States(refined="STILLOPEN", value={"OPEN", "CLOSING"})
})
```



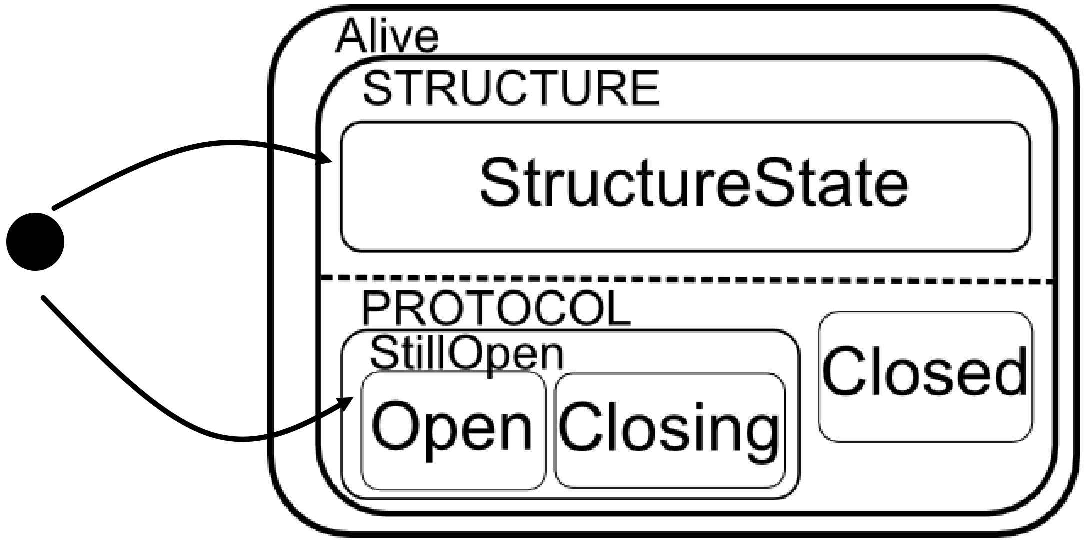
Blocking_queue Specification

```
@In("STRUCTURE")
private LinkedList elements= new LinkedList();
@In("PROTOCOL")
private boolean closed = false;
@In("STRUCTURE")
private boolean reject_enqueue_requests =
                                           false;
```



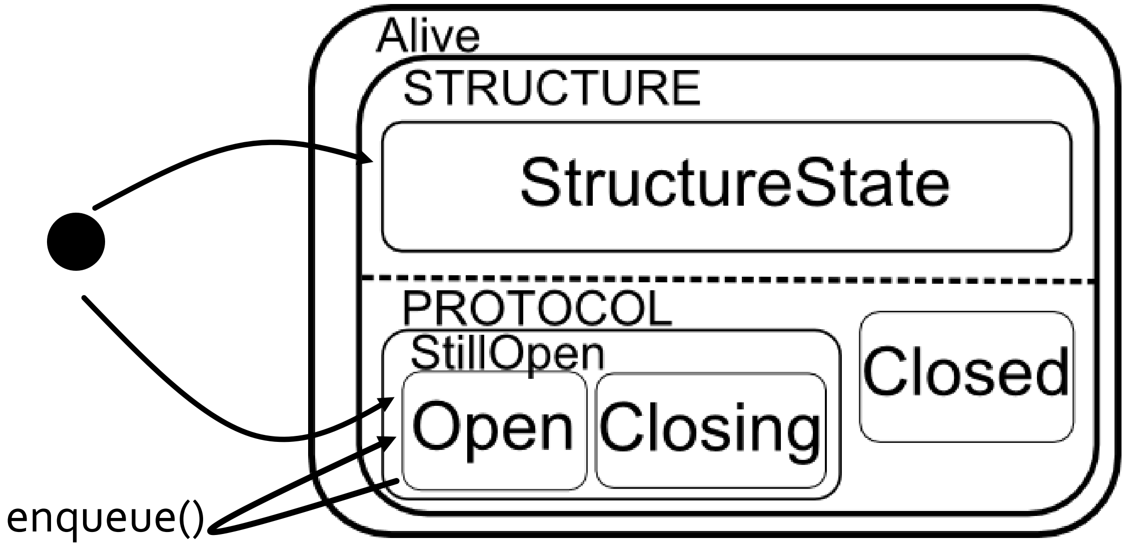
Blocking_queue Specification

```
@Perm(ensures="unique(this!fr) in
  OPEN,STRUCTURESTATE")
public Blocking_queue() {}
```



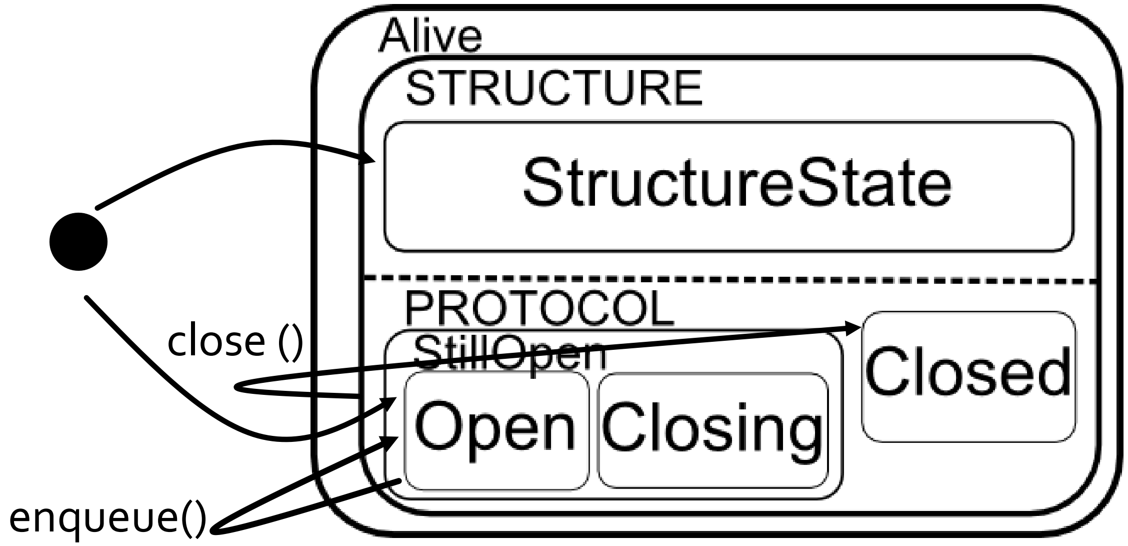
Blocking_queue Specification

```
@Share(guarantee="STRUCTURE")
@Full(requires="OPEN", ensures="OPEN",
  guarantee="PROTOCOL")
public synchronized final
  void enqueue( Object new_element )
  throws Closed
```



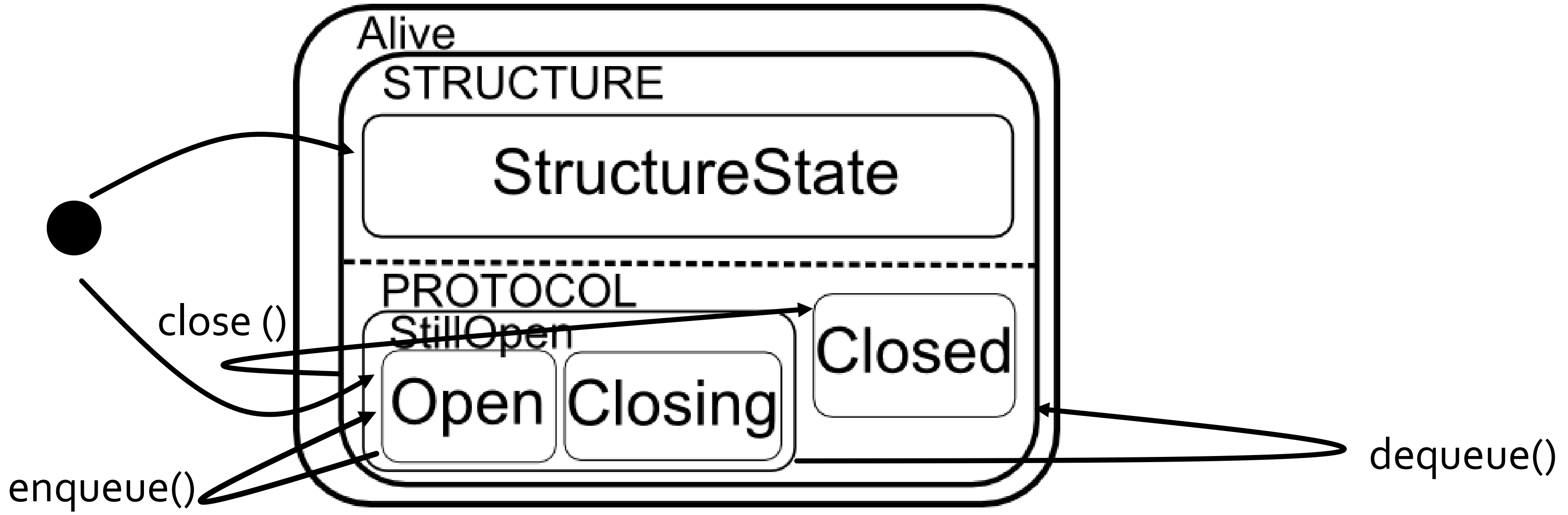
Blocking_queue Specification

```
@Full(value="PROTOCOL",
  ensures="CLOSED")
@Share(guarantee="STRUCTURE")
public synchronized void close()
```



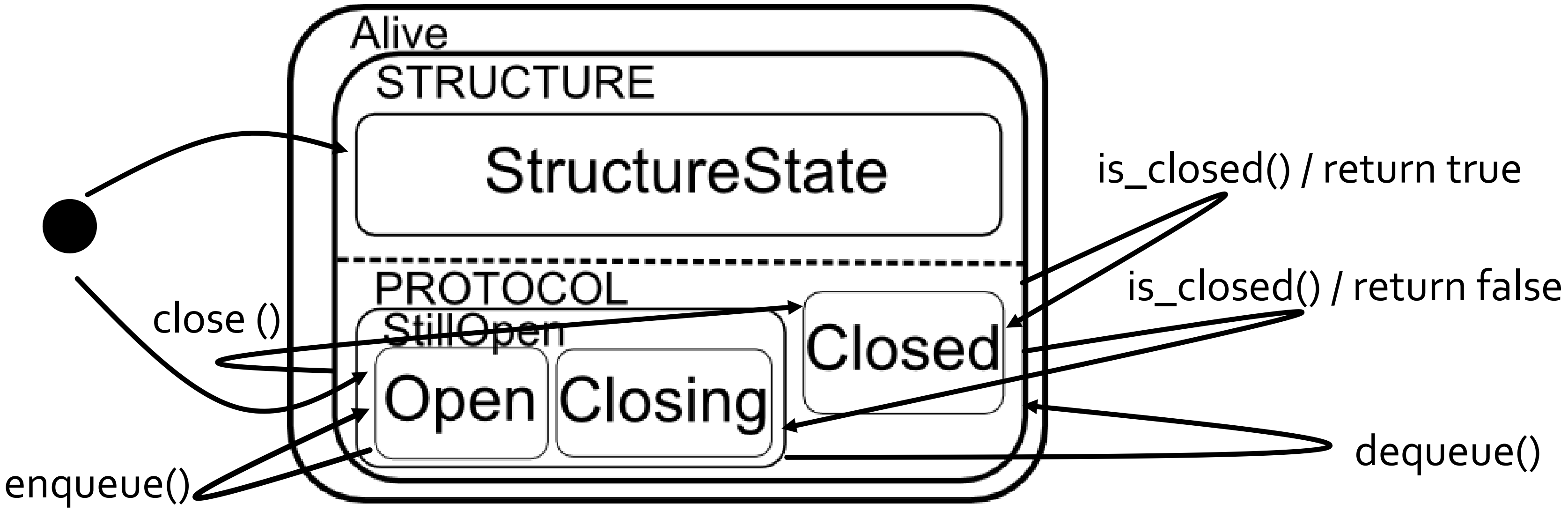
Blocking_queue Specification

```
@Share(guarantee="STRUCTURE")
@Pure( guarantee="PROTOCOL",
  requires="STILLOPEN")
public synchronized final
   Object dequeue( )
   throws Closed
```



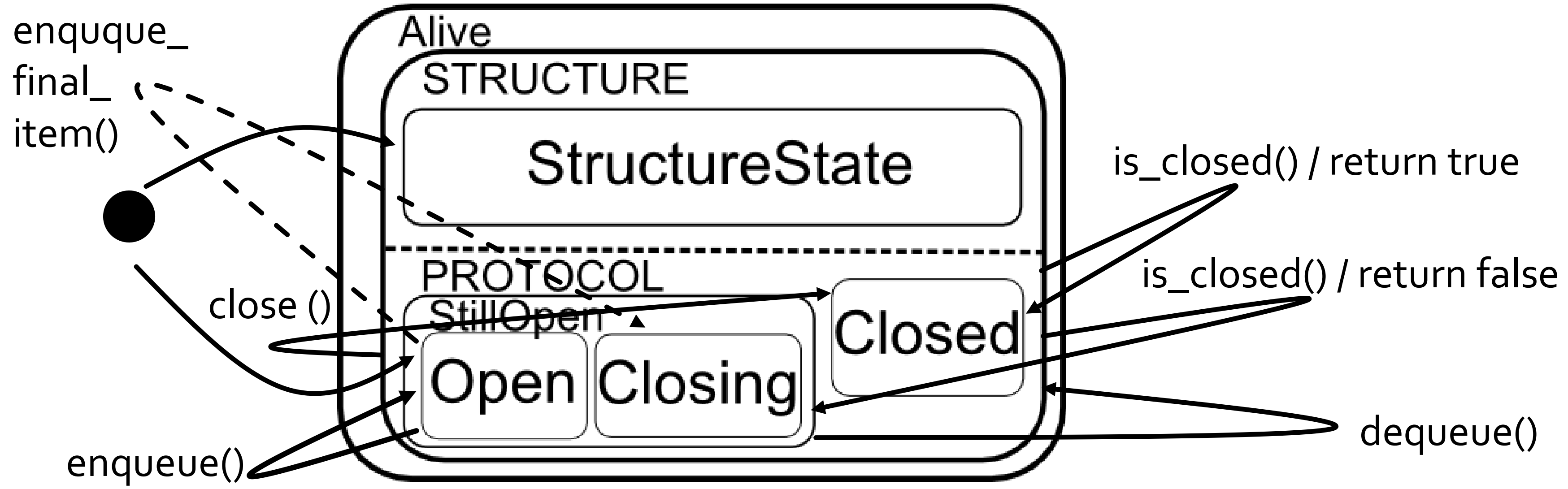
Blocking_queue Specification

```
@Pure(guarantee="PROTOCOL")
@TrueIndicates("CLOSED")
@FalseIndicates("STILLOPEN")
public final synchronized
   boolean is_closed()
```



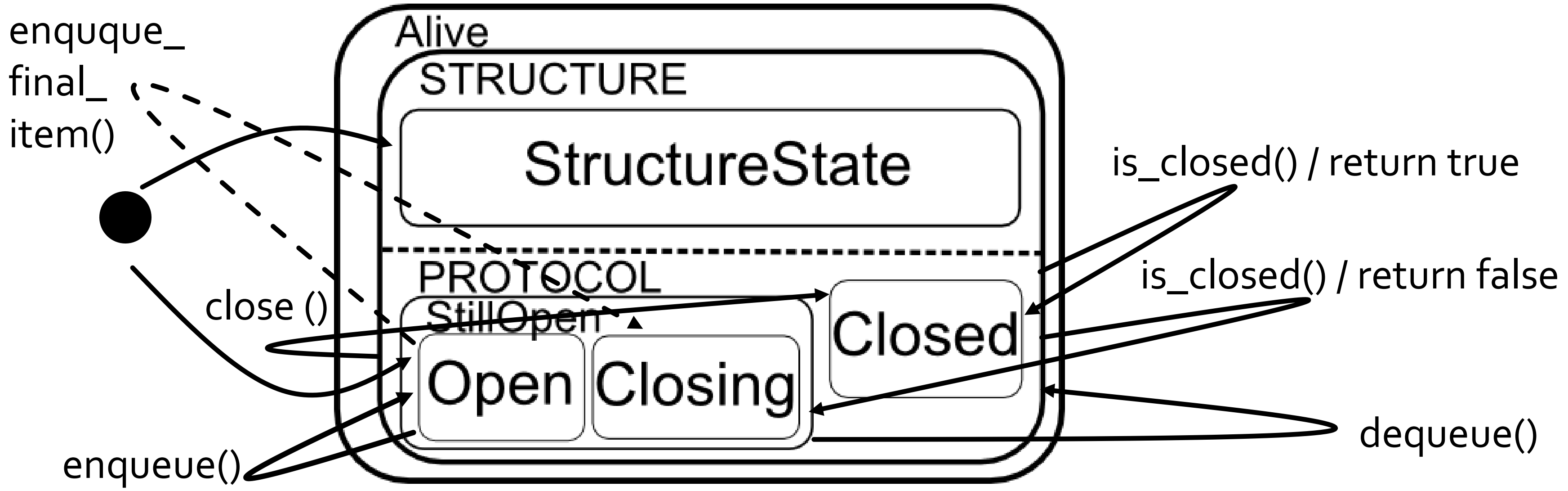
Blocking_queue Specification

```
@Full(requires="OPEN", guarantee="PROTOCOL",
  returned=false)
@Share(guarantee="STRUCTURE")
public synchronized final
  void enqueue_final_item(Object new_element)
  throws Closed
```



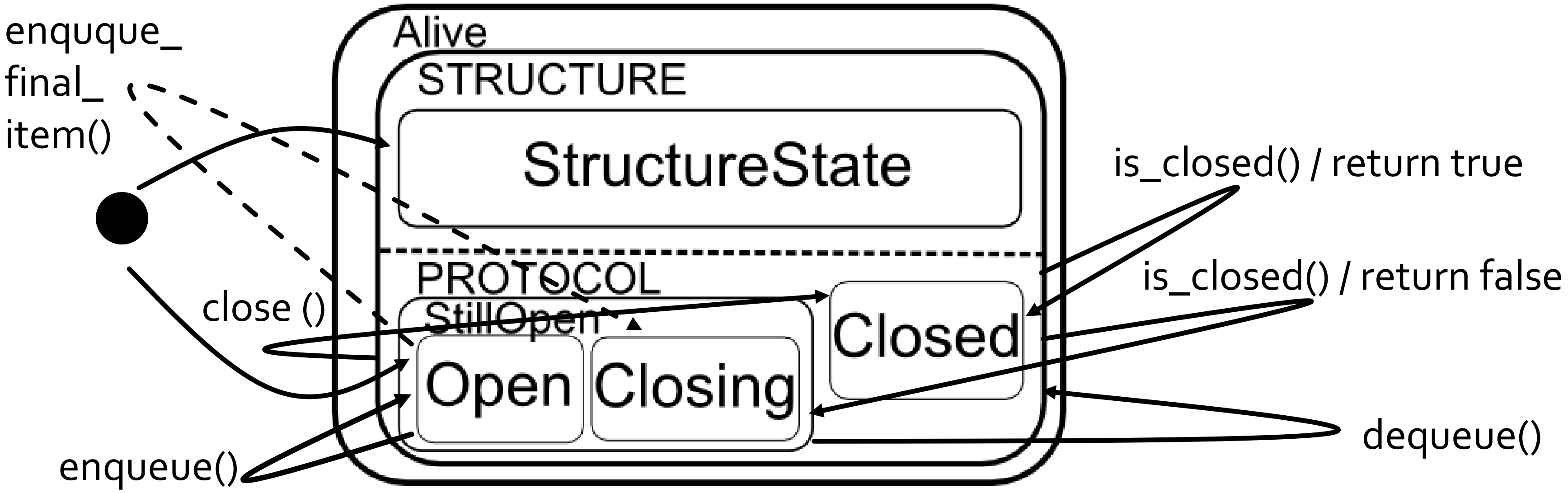
Blocking_queue Specification

```
@Share(guarantee="STRUCTURE")
@Pure( guarantee="PROTOCOL",
  requires="STILLOPEN")
public synchronized final
  Object dequeue( )
  throws Closed
```



Blocking_queue Specification

```
@ClassStates({
@State(name="STRUCTURE",
inv="share(elements) * reject_enqueue_requests ==
   true => full(this,PROTOCOL) in CLOSING"),
@State(name="STILLOPEN", inv="closed == false"),
@State(name="CLOSED", inv="closed == true")
})
```



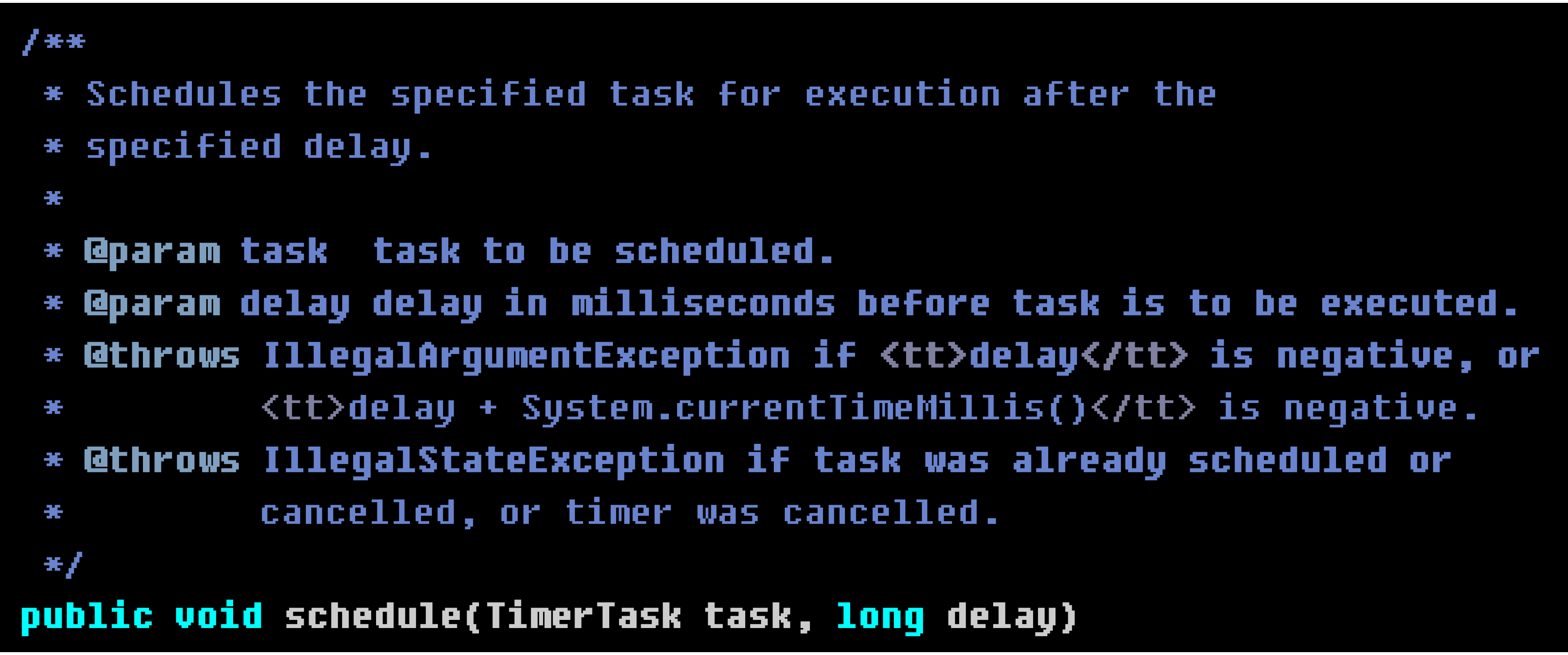
Blocking_queue Summary

- Dimensions used to separate protocol & underlying data structure
  - Conceptually two objects? Not really…
- One dimension can "store" a permission to the other dimension
  - (Modifying cannot be unpacked from read-only)

java.util.TimerTask & Timer

- A task meant to be executed at some time in the future.
  - Should be scheduled with Timer.schedule()
  - TimerTask.run() will be called
  - TimerTask defines four states, VIRGIN, SCHEDULED, EXECUTED, CANCELED
- Timer provides several schedule methods…
  - But all require a TimerTask in the VIRGIN or EXECUTED state!

java.util.Timer



```
/**
 * Schedules the specified task for execution after the
 * specified delay.
 *
 * @param task  task to be scheduled.
 * @param delay delay in milliseconds before task is to be executed.
 * @throws IllegalArgumentException if <tt>delay</tt> is negative, or
 *         <tt>delay + System.currentTimeMillis()</tt> is negative.
 * @throws IllegalStateException if task was already scheduled or
 *         cancelled, or timer was cancelled.
 */
public void schedule(TimerTask task, long delay)
```

Twine: Timer Case Study

- TwineGUI
  - Extends the TimerTask
  - Refreshes the display screen every second
  - Only the timer thread accesses fields of the object
    - So candidate for unique permission

TwineGUI

```
@Refine(
  @States({“Virgin”,”Scheduled”,”Executed”,”Cancelled”}))
public abstract class TimerTask {...}

public class TwineGUI extends TimerTask {
  @Perm(ensures=“unique(this!fr) in
    Virgin")
  public TwineGUI()
  ...
}

public class Timer {
  public void scheduleAtFixedRate(
  @Unique(requires="Virgin", returned=false) TimerTask task,
   long delay, long period)
  ...
}
```

TwineGUI.init()

```
@Unique(requires="Virgin",
  returned=false)
public void init(Resolver r) {
   ...
   // Refresh display periodically
   timer = ((TwineResolver)r).timer;
   timer.scheduleAtFixedRate(this,
      REFRESH_INTERVAL,
      REFRESH_INTERVAL);
}
```

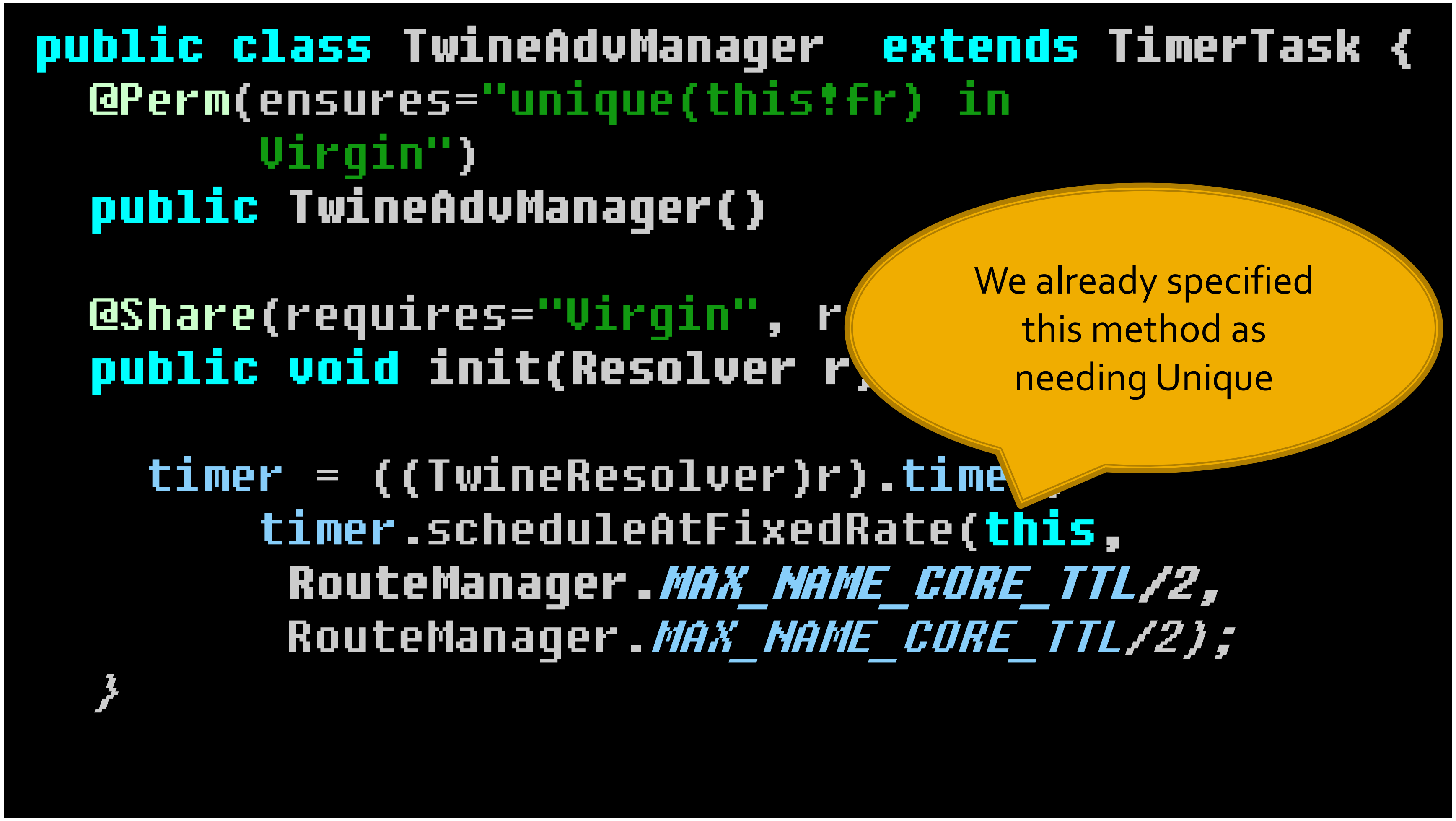
TwineGUI.run()

```
 @Unique
 public void run() {
  if ( !TwineResolver.DISPLAY )  return;

  if ( text == null )  displayWindow();

  Enumeration elements;

  text.replaceRange(prefix + nameTree.toPrettyString(),0,text.getText().length());
  text.append("\n\n ----> Directly connected: \n");

  for ( elements = nameTree.getNameRecords(); elements.hasMoreElements();;) {

    NameRecord nr = (NameRecord)elements.nextElement();
    boolean mine = (nr.getINRuid() == INRuid);

    if ( mine ) text.append(" - " + nr.getID());
  }

}
```

TwineAdvManager

- Another timer task
- Manages “advertisements”
  - Essentially description of a remove service
  - Timer periodically marks advertisements as outdated
  - Other threads add new advertisements
  - All threads will need modifying access
  - I.e., Share

TwineAdvManager



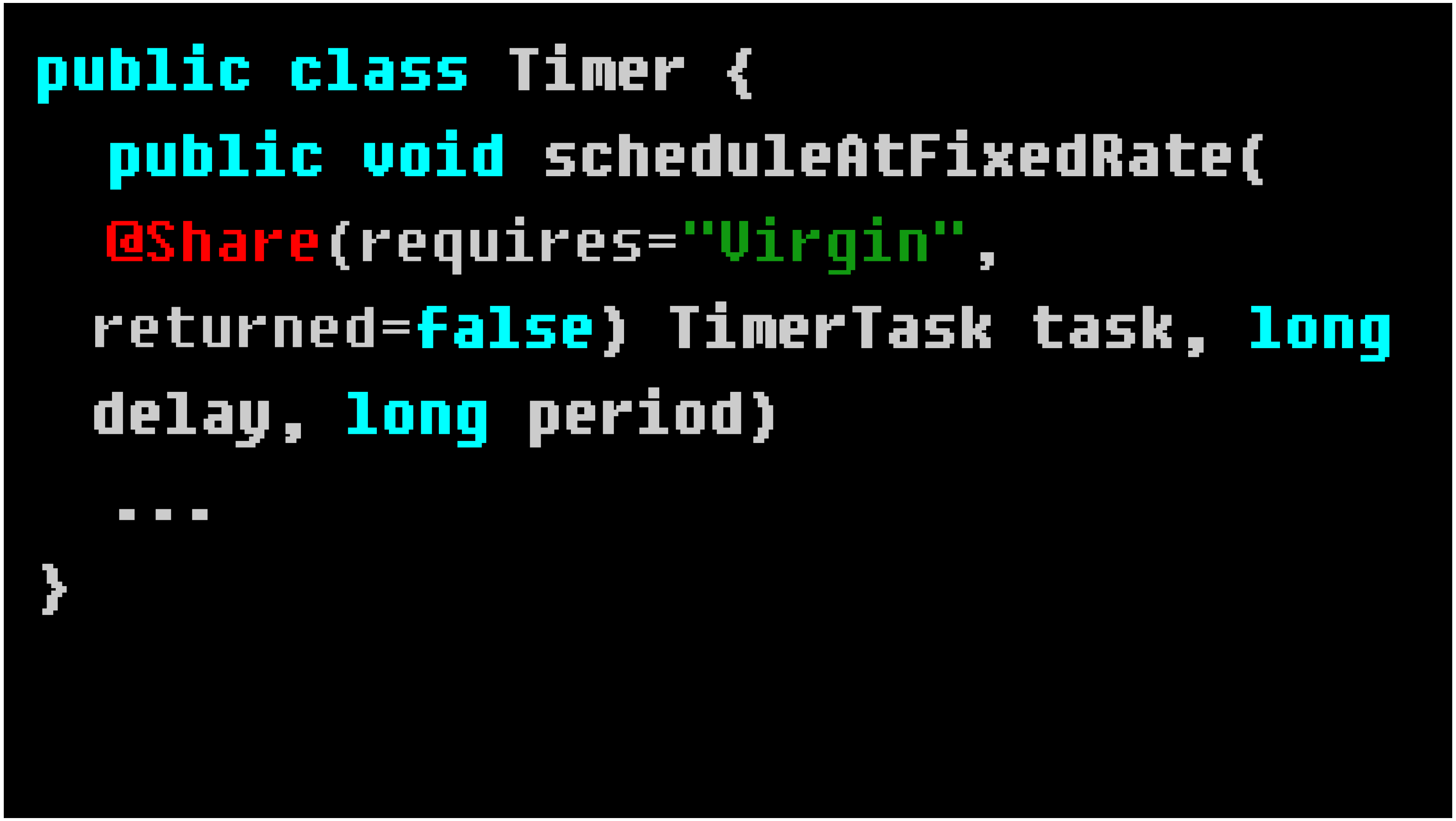
```
public class TwineAdvManager  extends TimerTask {
  @Perm(ensures="unique(this!fr) in
        Virgin")
  public TwineAdvManager()

  @Share(requires="Virgin", r
  public void init(Resolver r

    timer = ((TwineResolver)r).time
        timer.scheduleAtFixedRate(this,
         RouteManager.MAX_NAME_CORE_TTL/2,
         RouteManager.MAX_NAME_CORE_TTL/2);
  }
```

We already specified
this method as
needing Unique

Respecify Timer?

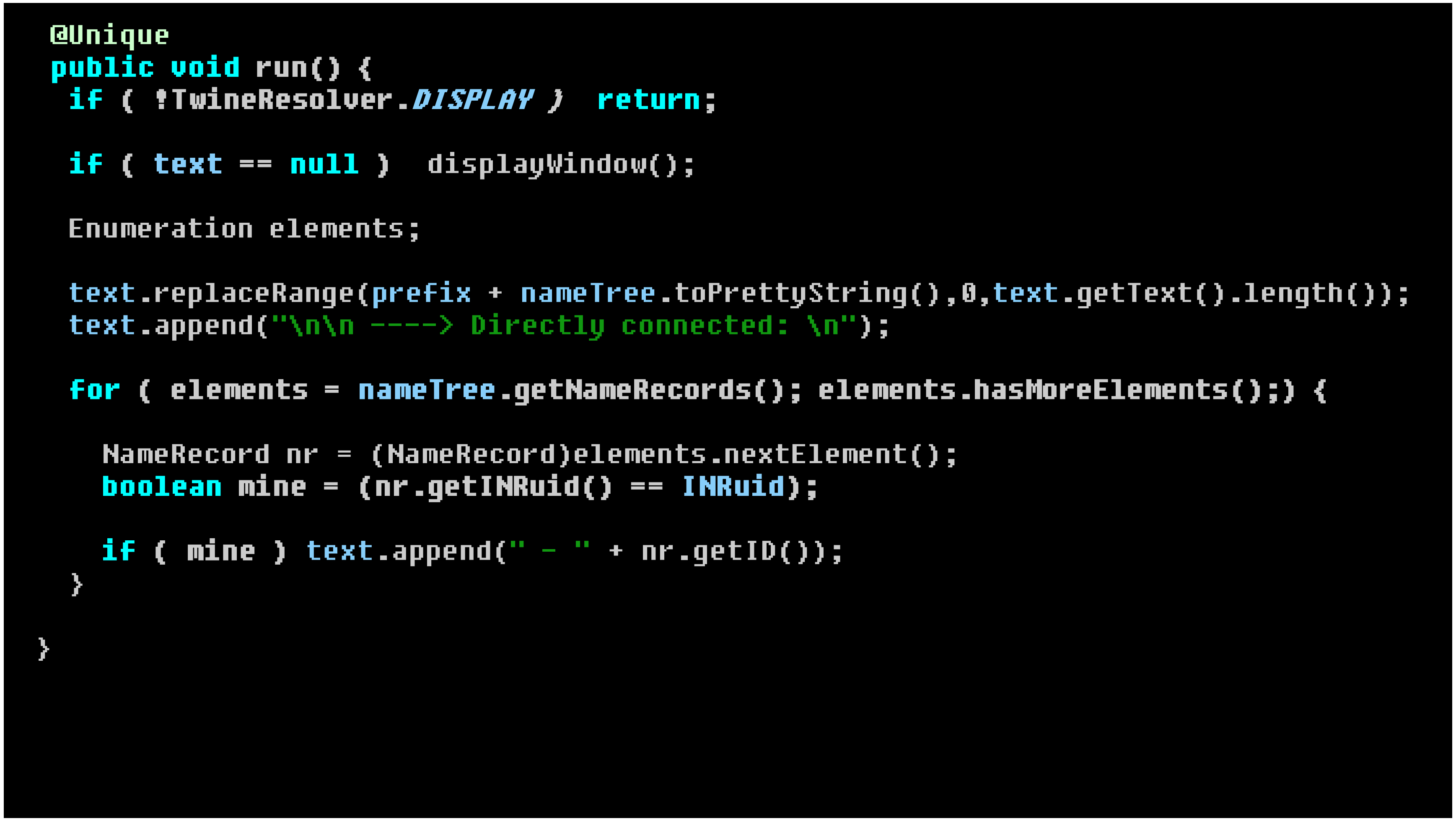
```
public class Timer {
  public void scheduleAtFixedRate(
  @Unique(requires="Virgin",
  returned=false) TimerTask task, long
  delay, long period)
   ...
}
```

Respecify Timer?



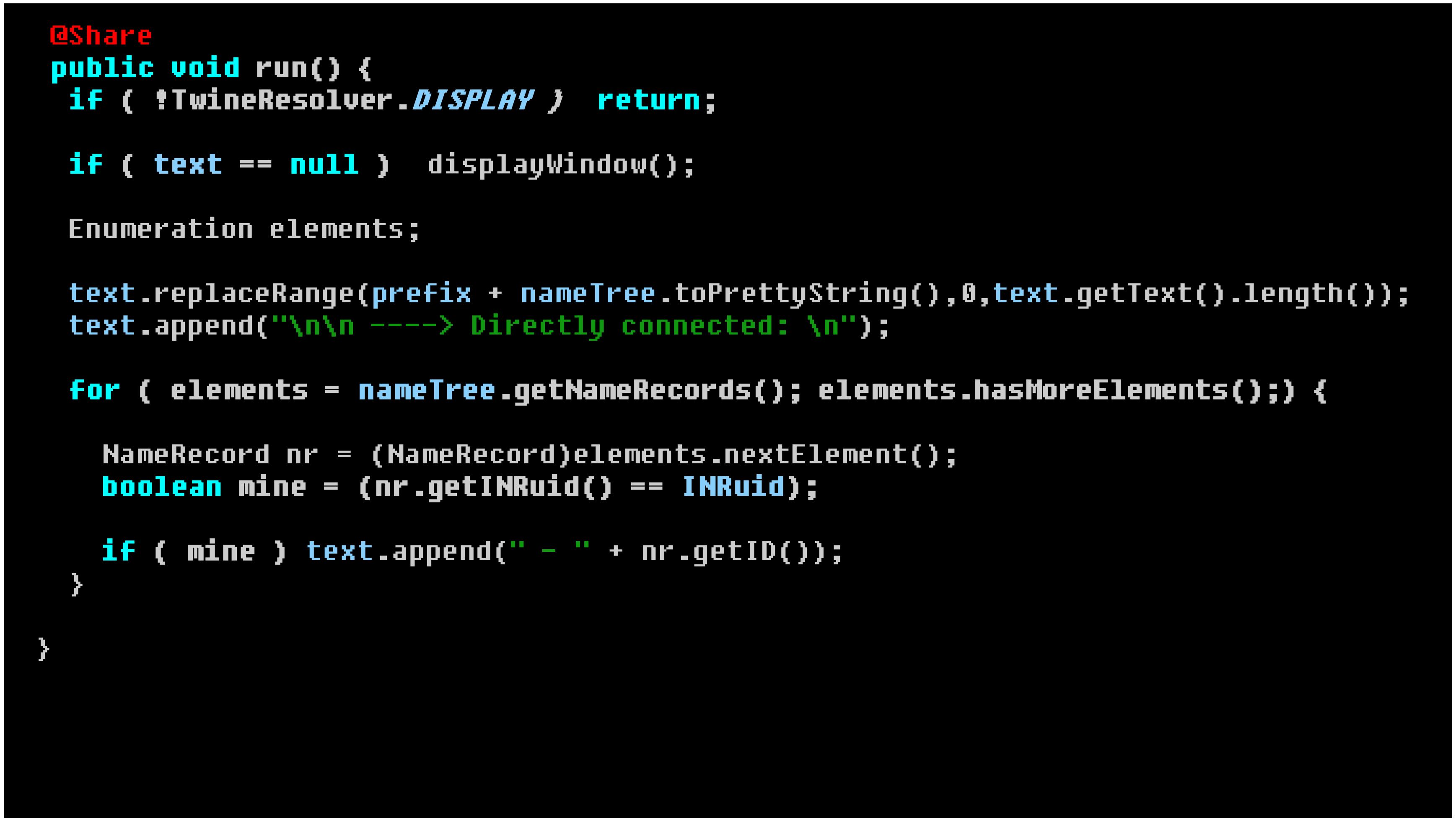
```
public class Timer {
  public void scheduleAtFixedRate(
  @Share(requires="Virgin",
 returned=false) TimerTask task, long
 delay, long period)
   ...
}
```

But Change Propagates



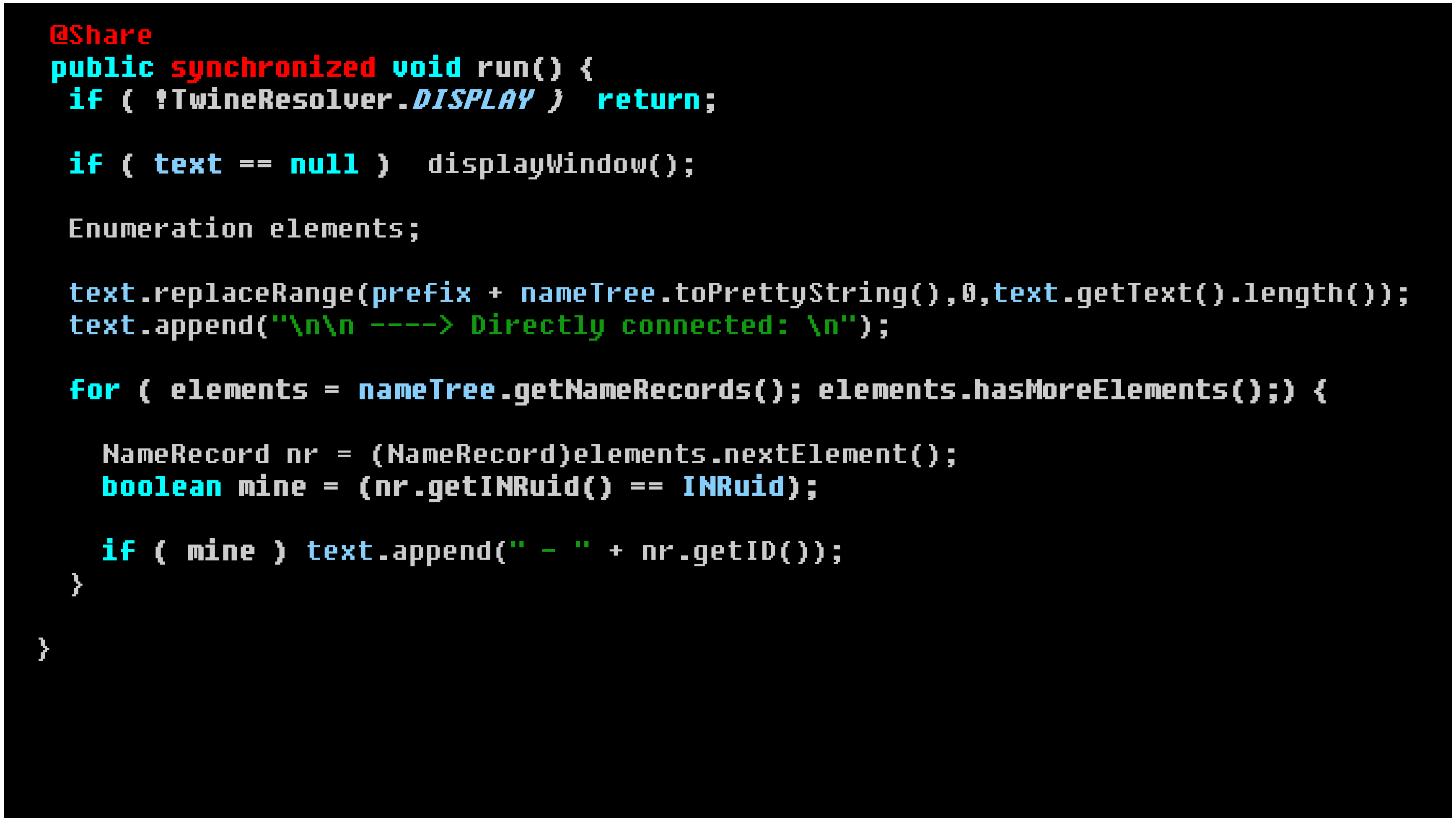
```
 @Unique
 public void run() {
  if ( !TwineResolver.DISPLAY )  return;

  if ( text == null )  displayWindow();

  Enumeration elements;

  text.replaceRange(prefix + nameTree.toPrettyString(),0,text.getText().length());
  text.append("\n\n ----> Directly connected: \n");

  for ( elements = nameTree.getNameRecords(); elements.hasMoreElements();;) {

    NameRecord nr = (NameRecord)elements.nextElement();
    boolean mine = (nr.getINRuid() == INRuid);

    if ( mine ) text.append(" - " + nr.getID());
  }

}
```

But Change Propagates



```
 @Share
 public void run() {
  if ( !TwineResolver.DISPLAY )  return;

  if ( text == null )  displayWindow();

  Enumeration elements;

  text.replaceRange(prefix + nameTree.toPrettyString(),0,text.getText().length());
  text.append("\n\n ----> Directly connected: \n");

  for ( elements = nameTree.getNameRecords(); elements.hasMoreElements();;) {

    NameRecord nr = (NameRecord)elements.nextElement();
    boolean mine = (nr.getINRuid() == INRuid);

    if ( mine ) text.append(" - " + nr.getID());
  }

}
```

But Change Propagates



```java
@Share
public synchronized void run() {
  if ( !TwineResolver.DISPLAY )  return;

  if ( text == null )  displayWindow();

  Enumeration elements;

  text.replaceRange(prefix + nameTree.toPrettyString(),0,text.getText().length());
  text.append("\n\n ----> Directly connected: \n");

  for ( elements = nameTree.getNameRecords(); elements.hasMoreElements();) {

    NameRecord nr = (NameRecord)elements.nextElement();
    boolean mine = (nr.getINRuid() == INRuid);

    if ( mine ) text.append(" - " + nr.getID());
  }

}
```

How Can We Resolve This?

- We want:
  - Reusable classes need
    - Specifications that work for many different aliasing contexts
    - Synchronization only if necessary
- Possible solutions:
  - “Unique dimensions”
    - Small tweak to existing system
  - Parametric permission polymorphism
    - Probably more useful in general

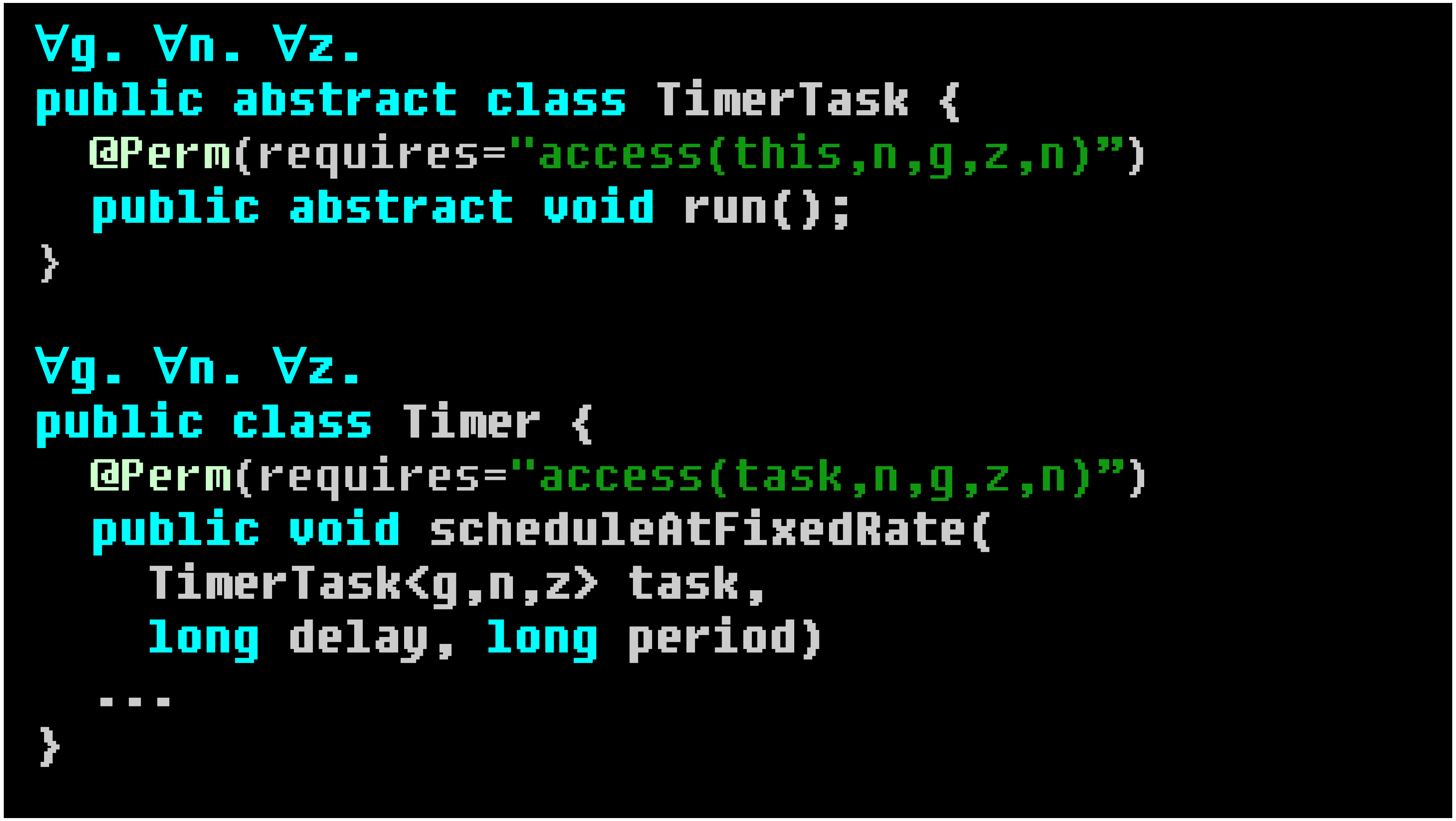
Solution With Unique Dimensions

```
@Refine({
  @States(dim="TLOCAL", value={"TLocalState"}),
  @States(dim="TSHARE", value= {"TShareState"})})
public abstract class TimerTask {
  @Unique(guarantee="TLOCAL")
  @Share(guarantee="TSHARE")
  public abstract void run();
}

public class Timer {
  public void scheduleAtFixedRate(
    @Unique(requires="Virgin",
            guarantee="TLocal"
            returned=false)
    @Share(guarantee="TShare"
           returned=false)
    TimerTask task, long delay, long period)
  ...
}
```

Why Does This Work?

- Each subclass can map fields into appropriate dimensions
  - If all fields are in TLocal, no synchronization necessary
- Downsides
  - Most specifications will mention both dimensions, unwieldy
  - Not a direct encoding
  - Only works for subclassing

Solution With Polymorphism (Proposed) (I)



```
∀g. ∀n. ∀z.
public abstract class TimerTask {
  @Perm(requires="access(this,n,g,z,n)")
  public abstract void run();
}

∀g. ∀n. ∀z.
public class Timer {
  @Perm(requires="access(task,n,g,z,n)")
  public void scheduleAtFixedRate(
    TimerTask<g,n,z> task,
    long delay, long period)
  ...
}
```

Solution With Polymorphism (Proposed) (II)

```
public class TwineGUI extends
  TimerTask<{alive->1},alive,1> {

  public void init(
    Timer<{alive->1},alive,1> timer) {
    timer.scheduleAtFixedRate(this,
      REFRESH_INTERVAL,
      REFRESH_INTERVAL);
  }

  @Unique
  public void run() { ... }
}
```

Comments on Polymorphism

- Allows class to be used in different aliasing contexts
  - Works without subtyping
  - An obvious extension to any type system
  - Enables other useful patterns
    - E.g., collections generic over permission kinds
- Likely to be included in my thesis work

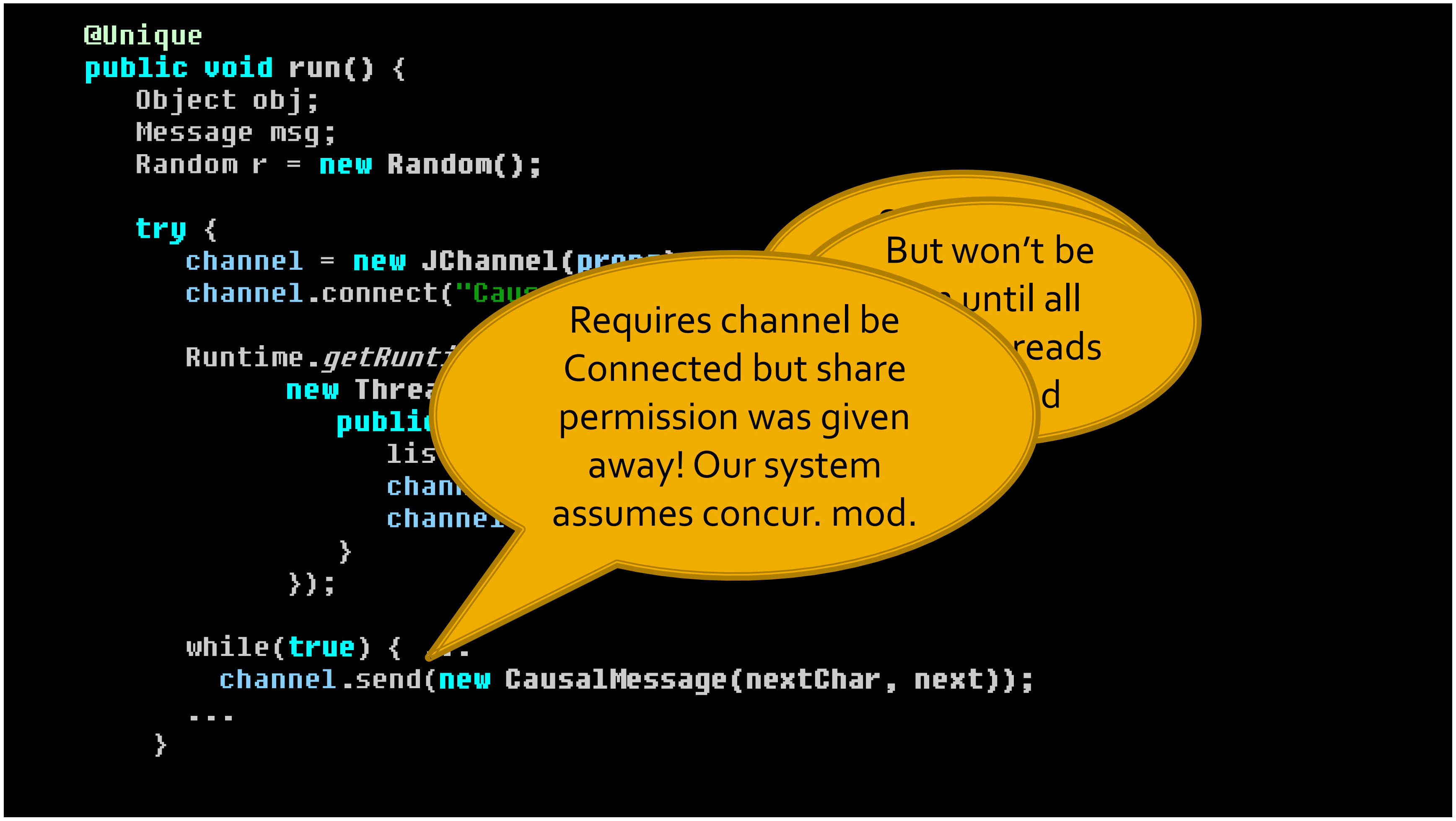
Timer Summary

- Good case study
  - Timer often used in concurrent applications
- Simple protocol
  - But we learned a lot
    - Used in different sharing contexts
  - Motivates some extensions
    - “Unique dimensions”
    - Permission polymorphism

CausalDemo.java

- Test class from JGroups
  - A middleware for writing distributed applications
  - Uses the Channel interface
    - Abstracts a network connection
    - Defines a simple protocol (Unconnected, connected, closed)
  - We cannot verify the correct use of Channel!
    - Makes use of Java's "shut-down hook"
    - Requires modifying permission but doesn't use it until end of process

CausalDemo.run()



```
@Unique
public void run() {
   Object obj;
   Message msg;
   Random r = new Random();

   try {
      channel = new JChannel(prop
      channel.connect("Caus

      Runtime.getRunti
            new Threa
               publi
                  lis
                  chan
                  channe
               }
            });

      while(true) { ...
        channel.send(new CausalMessage(nextChar, next));
      ...
   }
```

But won't be
until all
reads
d

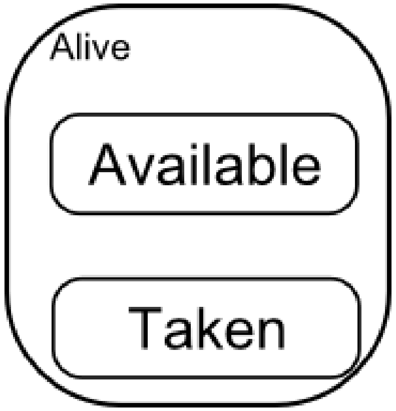
Requires channel be Connected but share permission was given away! Our system assumes concur. mod.

Dining Philosophers

- Classic concurrency challenge problem
- Made quite simple with atomic blocks
  - (So NIMBY, rather than Sync or Swim)
- But verification is tricky
  - Punch line: We cannot effectively track two shared objects
- Note: Didn't find this online…
  - But wanted to see if I could prove it correct

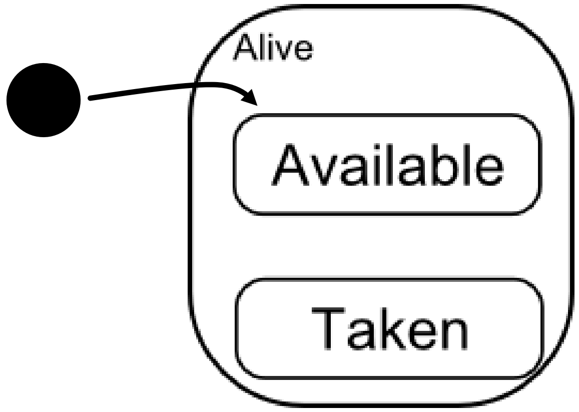
Fork Protocol

```
@States({"Available", "Taken"})
@ClassStates({
@State(name="Available", inv="available == true"),
@State(name="Taken", inv="available == false")
})
class Fork {
  @In("alive")
  private boolean available;
```



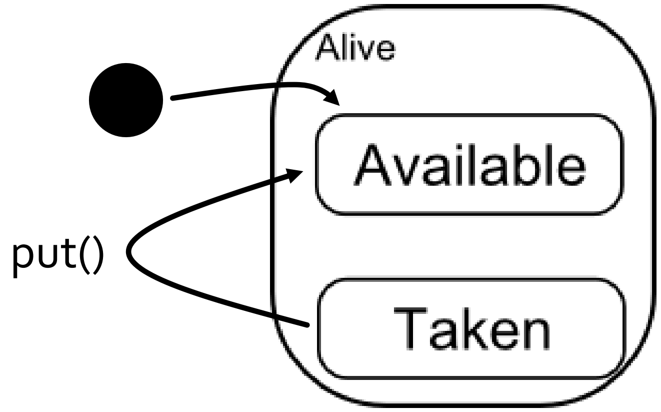
Fork Protocol

```
@Perm(ensures="unique(this!fr) in
  Available")
public Fork()
```



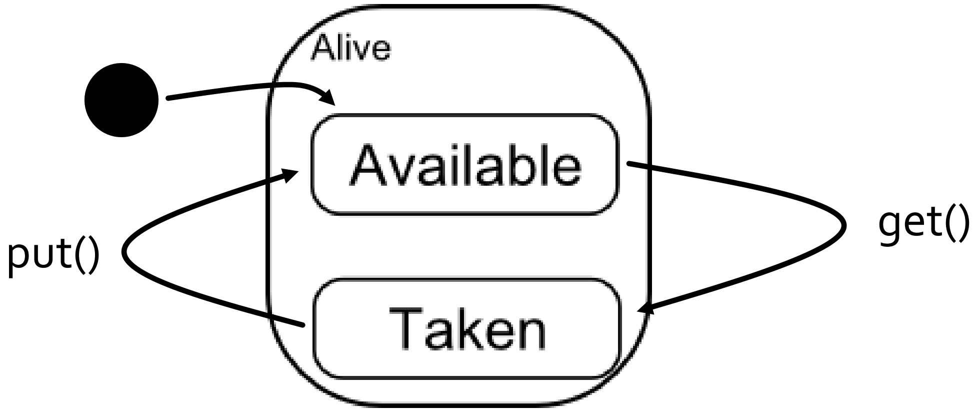
Fork Protocol

```
@Share(requires="Taken",
  ensures="Available")
void put()
```



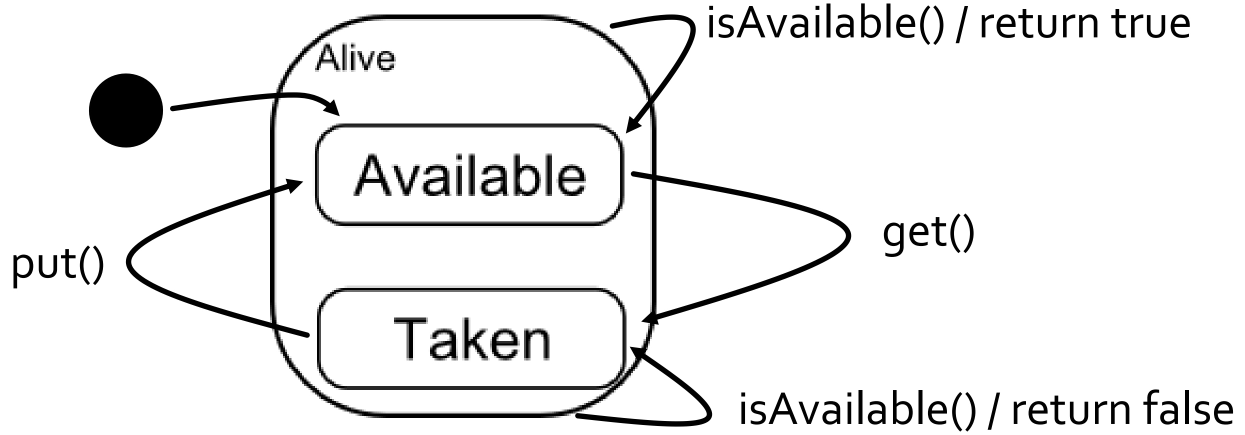
Fork Protocol

```
@Share(requires="Available",
  ensures="Taken")
void get()
```



Fork Protocol

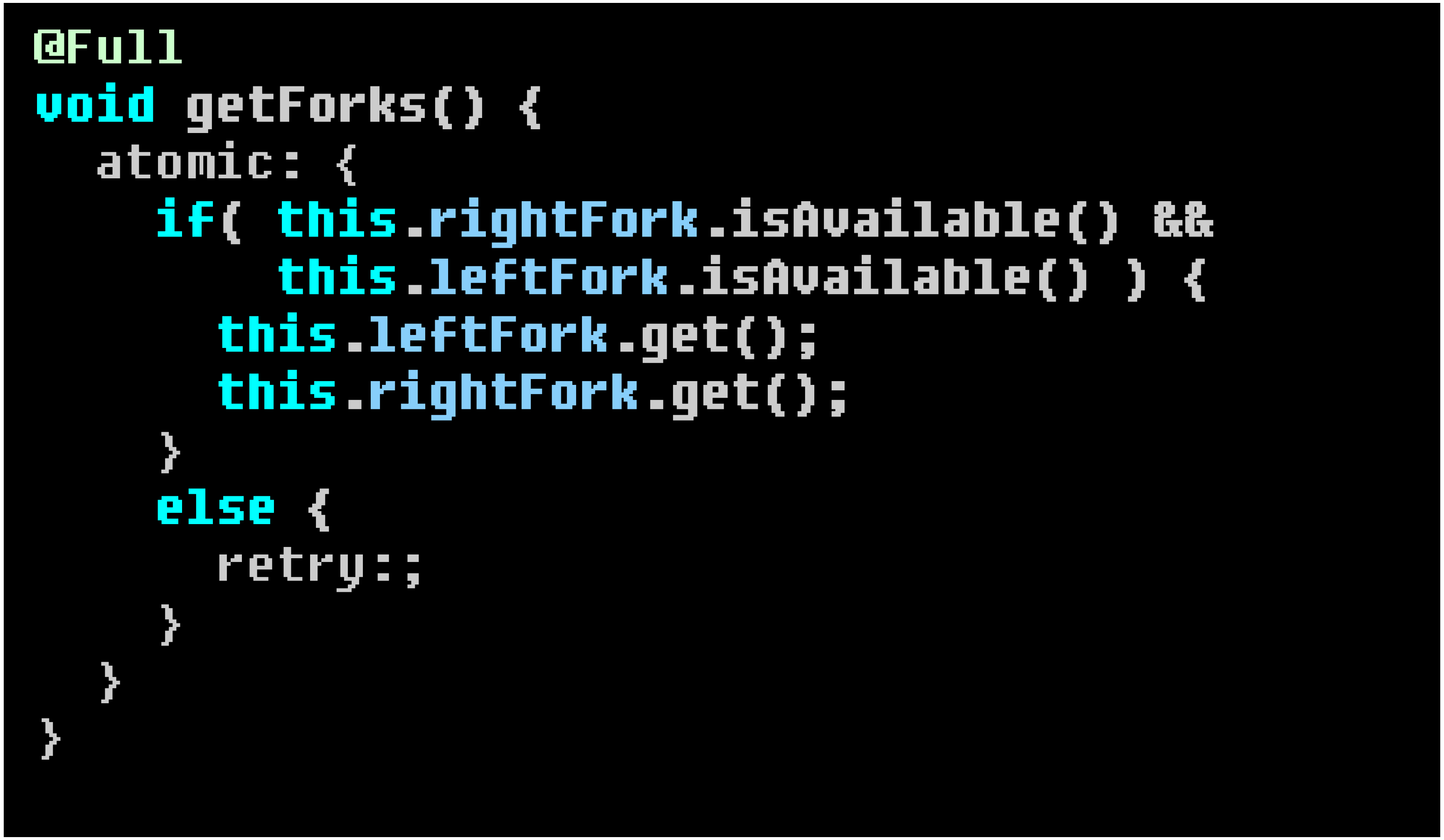
```
@Pure
@TrueIndicates("Available")
@FalseIndicates("Taken")
boolean isAvailable()
```



Philosopher Specification

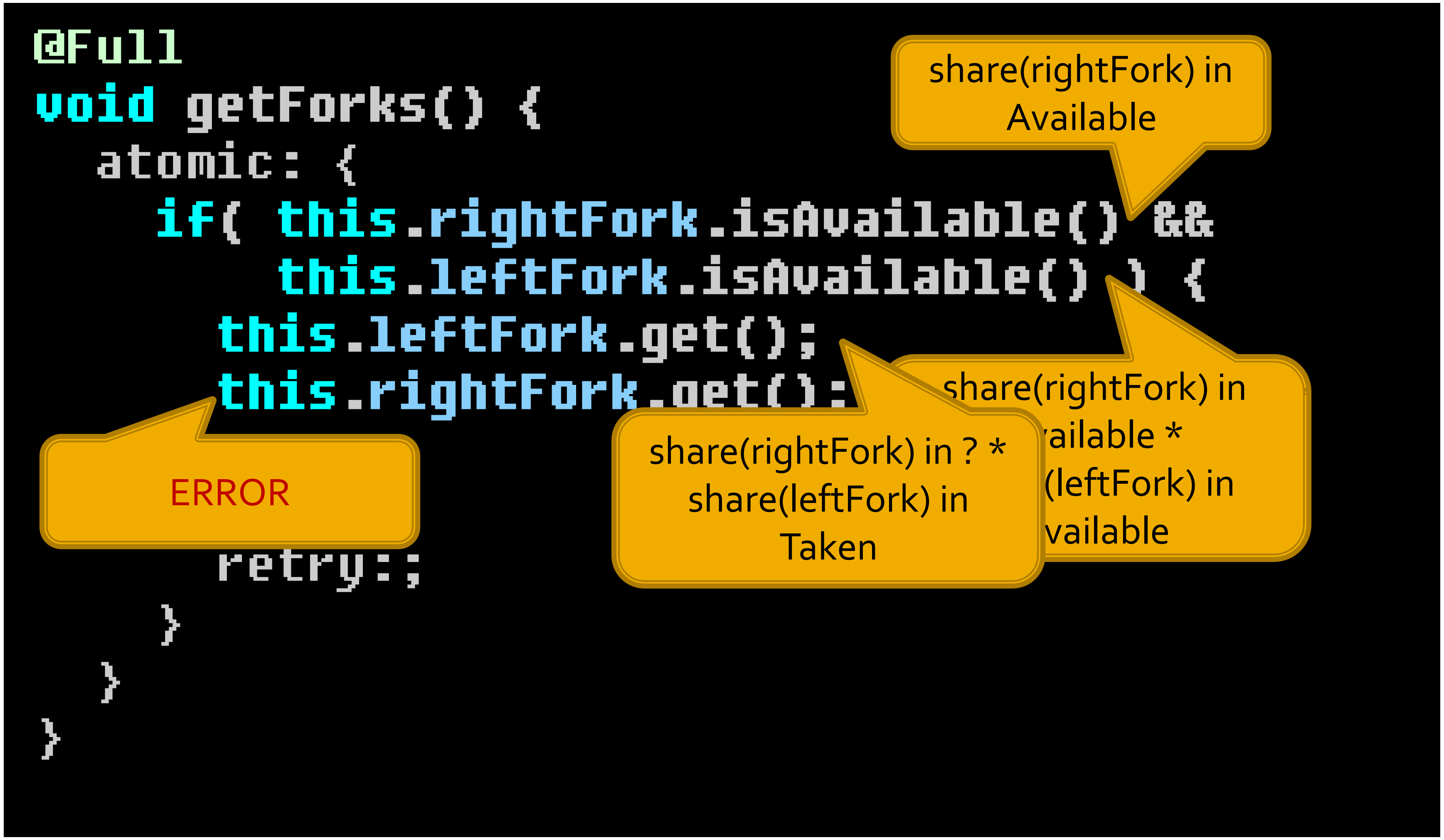
```
@ClassStates(@State(name="alive",
  inv="share(leftFork) *
  share(rightFork)"))
class Philosopher extends Thread {
   Fork leftFork;
   Fork rightFork;

   @Perm(ensures="unique(this!fr")
   public Philosopher(
    @Share(returned=false) Fork leftFork,
    @Share(returned=false) Fork rightFork)
```

Philosopher



```
@Full
void getForks() {
  atomic: {
    if( this.rightFork.isAvailable() &&
        this.leftFork.isAvailable() ) {
      this.leftFork.get();
      this.rightFork.get();
    }
    else {
      retry:;
    }
  }
}
```

Philosopher



```
@Full
void getForks() {
  atomic: {
    if( this.rightFork.isAvailable() &&
        this.leftFork.isAvailable() ) {
      this.leftFork.get();
      this.rightFork.get();

      retry:;
    }
  }
}
```

share(rightFork) in Available
ailable * (leftFork) in vailable
share(rightFork) in ? * share(leftFork) in Taken
share(rightFork) in
ERROR

Share

- Share permissions are still hard to reason about
  - Atomic blocks don’t change that
- Why does program work?
  - Each thread has one permission to each fork
    - Atomic block makes permission Unique
  - Or, programmer knows one fork won’t change another

Wouldn't it be nice…

```
@ClassStates({
  @State(name="alive",
    inv="share(leftFork) *
         share(rightFork)"),
  @State(name="EATING", inv="leftFork
    in Taken * rightFork in Taken")
})
class Philosopher extends Thread {
  Fork leftFork;
  Fork rightFork;
```

Philosopher's Summary

- Share permissions are difficult to reason with
  - Still must account for "plain old" modification
  - Can't go inside state invariants
- Possible solutions
  - Perm. that is unique in atomic block
  - More descriptive effects system
- May end up unsolved in my thesis

Summary

- Blocking_queue
  - Cool use of dimensions!
- Timer & Timer Task
  - A simple protocol
  - Motivates polymorphism over permissions
- Causal Demo
  - Shutdown hooks?!
- Dining Philosophers
  - Effects are *still* hard…