CS 351
Design of Large Programs
Concurrency Control

Brooke Chenoweth

University of New Mexico

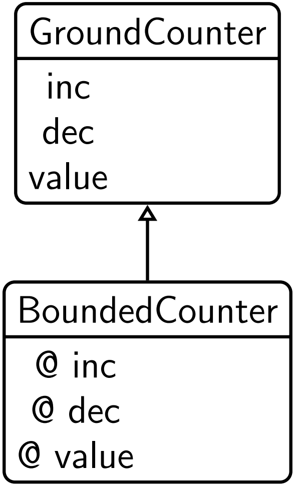
Fall 2024

Concurrency Control Definition

- Layering of synchronization and control policies over base mechanisms
- Key assumption: ground classes have been designed to be amenable to the desired forms of control
- Strategies
    - adding policy control in subclasses
    - controlling delegated actions
    - representing messages as objects

Adding Synchronization Via Subclassing

- Ground level class
    - provides non-public methods
    - enforces no invariants
- Subclass
    - implements the synchronization policy, e.g.,
        - atomic updates
        - bounded counter
    - delegates actions to ground methods



Bounded Counter Code

```
public class GroundCounter {
  protected long count;

  protected GroundCounter(long c) {
    count = c;
  }

  protected long value() { return count; }

  protected void inc() {
    ++count;
  }

  protected void dec() {
    --count;
  }
}
```

Bounded Counter Code

```
public class BoundedCounter extends GroundCounter {
  private final long MIN, MAX;

  public BoundedCounter(final long MIN, final long MAX) {
    super(MIN);
    this.MIN = MIN;
    this.MAX = MAX;
  }

  public synchronized long value() { return super.value(); }

  public synchronized void inc() {
    while (value() >= MAX) {
      try { wait(); } catch(InterruptedException e) {};
    }
    super.inc();
    notifyAll();
  }

  public synchronized void dec() {
    while (value() <= MIN) {
      try { wait(); } catch(InterruptedException e) {};
    }
    super.dec();
    notifyAll();
  }
}
```

Anomalies to be Avoided

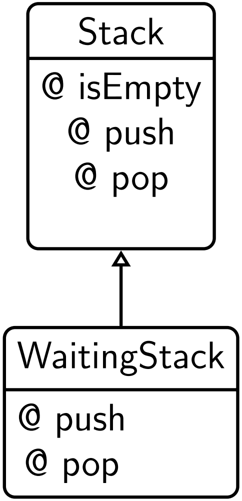
If all relevant variables and methods are declared `protected` the subclass can generally implement the desired policy

Frequent problems

- failing to track all conditions on which the subclass depends
- differences in state representation
- immutable variable in the superclass is being modified
- introducing waits for which the subclass has not matching notifications
- extensions which require a transition from `notify` to `notifyAll`

Layering Guards

- Ground level class
  - ensures atomicity
  - generates exception on empty stack
  - forces a busy wait solution in a concurrent setting
- Subclass
  - eliminates the exception generation
  - adopts a wait/notify protocol
  - delegates select actions to ground methods



Waiting Stack Code

```java
public class Stack {
  public synchronized boolean isEmpty() { /* ... */ }
  public synchronized void push(Object x) { /* ... */ }
  public synchronized Object pop() throws StackEmptyException {
    if (isEmpty()) {
      throw new StackEmptyException();
    } else {
      // ...
    }
  }
}
```

```java
public class WaitingStack extends Stack {
  public synchronized void push(Object x) {
    super.push(x);
    notifyAll();
  }

  public synchronized Object pop() throws StackEmptyException {
    while(isEmpty()) {
      try { wait(); } catch (InterruptedException e) {}
    }
    return super.pop();
  }
}
```

Conflict Sets et al

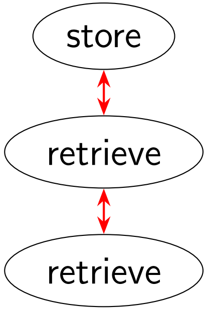
- Design paradigm
  - track the superclass state through the introduction of auxiliary variables
  - block method calls based on the current abstract state of the superclass
- Sample formalizations
  - conflict set
  - conflict graph
  - finite set control

Illustration: Inventory Control

Conflict Set

(store, retrieve)
(retrieve, retrieve)

Conflict Graph



Finite State control

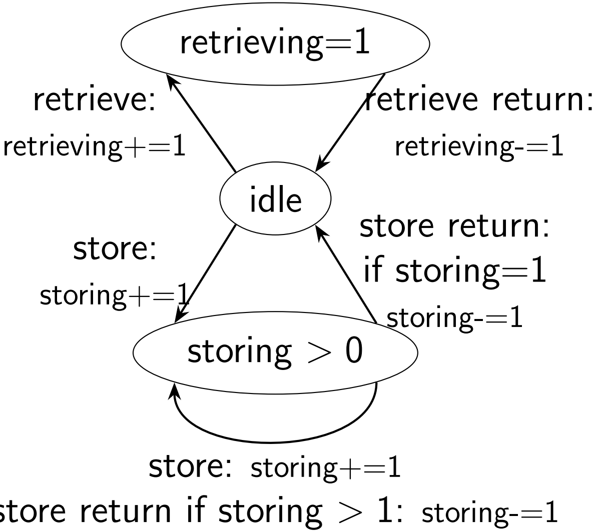


Illustration: Inventory Code

```
public class Inventory extends GroundInventory {
  protected int storing = 0;
  protected int retrieving= 0;

  public void store(String desc, Object item,
                    String supplier) {
    synchronized (this) {
      while (retrieving != 0) {
        try { wait(); } catch (InterruptedException e) {}
      }
      ++storing;
    }

    super.store(desc, item, supplier);

    synchronized (this) {
      if (--storing == 0) {
        notifyAll();
      }
    }
  }
```

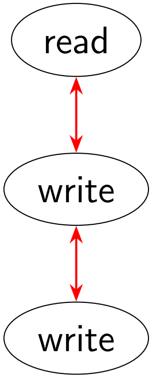
Illustration: Inventory Code

```
public void retrieve(String desc, Object item,
                     String supplier) {
  synchronized (this) {
    while (storing != 0) {
      try { wait(); } catch (InterruptedException e)
    }
    ++retrieving;
  }

  super.retrieve(desc, item, supplier);

  synchronized (this) {
    if (--retrieving == 0) {
      notifyAll();
    }
  }
}
```

Readers and Writers: A Common Design Pattern

- A simple conflict graph
- Readily ensured safety
- Major variations when it comes to liveness and lack of starvation

Conflict Graph



General Pattern

- Track the number of waiting and active readers and writers
- Bracket the read/write operations with before/after control code
- Same design accommodates a wide range of policies

General Pattern Code

```
public abstract class ReadWrite {
  protected int activeReaders = 0;
  protected int activeWriters = 0;
  protected int waitingReaders = 0;
  protected int waitingWriters = 0;

  protected abstract void doRead();
  protected abstract void doWrite();

  public void read() {
    beforeRead();
    doRead();
    afterRead();
  }

  public void write() {
    beforeWrite();
    doWrite();
    afterWrite();
  }
```

Control Code: before/afterRead

```java
protected synchronized void beforeRead() {
  ++waitingReaders;
  while (!allowReader()) {
    try { wait(); } catch (InterruptedException e) {}
  }
  --waitingReaders;
  ++activeReaders;
}

protected synchronized void afterRead() {
  --activeReaders;
  notifyAll();
}
```

Control Code: before/afterWrite

```
protected synchronized void beforeWrite() {
  ++waitingWriters;
  while (!allowWriter()) {
    try { wait(); } catch (InterruptedException e) {}
  }
  --waitingWriters;
  ++activeWriters;
}

protected synchronized void afterWrite() {
  --activeWriters;
  notifyAll();
}
```

Policy: Reading Priority

- Direct enforcement of the conflict set rules
  - readers can read unless a writer is writing
  - a writer can start writing if no other thread is reading or writing
- Writers can be starved if readers continue to use the resource

```
protected boolean allowReader() {
  return activeWriters == 0;
}

protected boolean allowWriter() {
  return activeReaders == 0 && activeWriters == 0;
}
```

Policy: Reading Preemption

- Writer starvation is prevented by blocking new reading threads once a writer is waiting
- Readers can be kept out by multiple writers waiting in line

```
protected boolean allowReader() {
  return waitingWriters == 0 && activeWriters == 0;
}

protected boolean allowWriter() {
  return activeReaders == 0 && activeWriters == 0;
}
```

Policy: Turn Taking

- If both readers and writers are waiting, alternate between readers and writers
- Still, there is no guarantee that no readers or writers are blocked forever

Policy: Turn Taking

```java
private boolean canWrite;

public synchronized void afterRead() {
  canWrite = true; // ...previous afterRead code...
}
public synchronized void afterWrite() {
  canWrite = false; // ...previous afterWrite code...
}

protected boolean allowReader() {
  if (waitingWriters > 0 && waitingReaders > 0) {
    return !canWrite && activeWriters == 0;
  }
  return waitingWriters == 0 && activeWriters == 0;
}

protected boolean allowWriter() {
  if (waitingWriters > 0 && waitingReaders > 0) {
    return canWrite && activeWriters == 0;
  }
  return activeReaders == 0 && activeWriters == 0;
}
```

Policy: Custom Scheduler

- Each reader/writer makes a request and gets a ticket number
- Each request and its type (read/write) is placed in a queue according with some policy that optimizes throughput and ensures fairness
- A writer starts working when its ticket is first in the queue
- A reader starts working when its ticket is part of a sequence of read requests at the front of the queue
- Tickets are returned upon completion of the operation
- Ticket counter is reset when no tickets are out

Basic Adapter Concept

- An *adapter* is a specialized wrapper that offers a new interface based on an existing class
- The main advantages are
    - flexibility
        - a range of services based on an existing class
        - dynamic changes of the base class in use
    - reduced coding effort
    - reuse of legacy code
- Adapters do not have access to protected fields and methods
    - this is distinct from subclassing

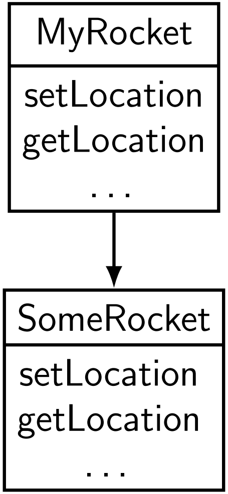
Example: Rocket

- Internal state
    - location (point in 3D space, meters)
    - velocity (3D vector, meters/sec)
    - flight time (seconds)
- Methods
    - read/update location
    - read/update velocity
    - increment flight time

Delegation

The interface is the same as that of the ground class

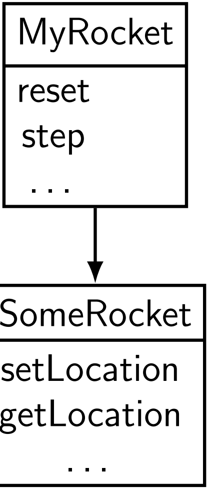
- actions are simply redirected
- returned values are passed up
- What is the gain?
  - flexibility



Refactoring

Different methods are provided and coded in terms of the ground class

```
public void reset() {
  setFlightTime(0);
  setLocation(new PosPoint());
  setVelocity(new VelVec());
}

public void step(VelVec vel) {
  flightTime++;
  velocity = vel;
  location.add(velocity);
}
```



Superposition

Superposition – a process by which variables are introduced to monitor the state of the ground class

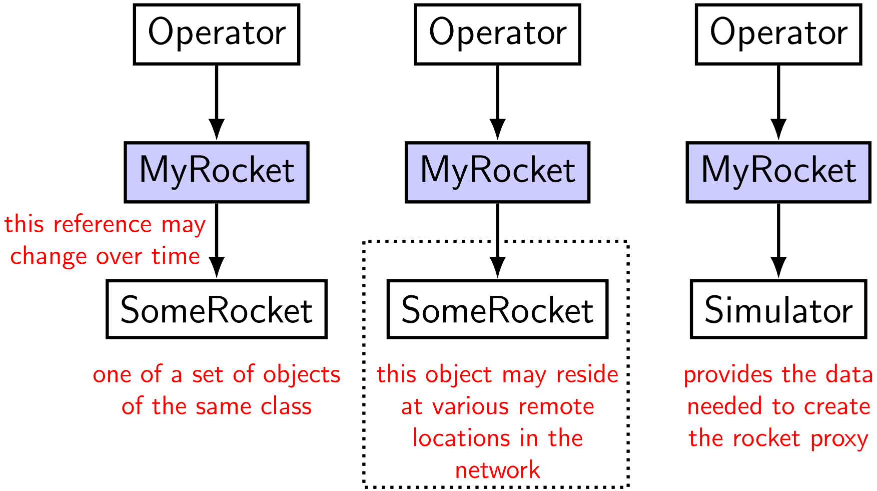
- to extend functionality without actually affecting the state
- to facilitate reasoning about the computation

```
public void step(VelVec vel) {
  flightTime++;
  velocity = vel;
  location.add(velocity);
  if(velocity.compare(MAX_SAFE_VEL) > 0) {
    unsafeCount++;
  }
}
```

Proxy – A Useful Design Pattern

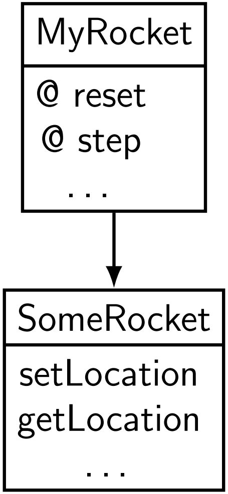
- Proxy – an object that stands in place of another
    - displays an appropriate veneer
    - delegates all actions
- The original object may be a
    - concrete object – present in the system and having the same interface
    - abstract object – the result of refactoring

Illustration: Proxy Examples


Operator
MyRocket
this reference may change over time
SomeRocket
one of a set of objects of the same class
Operator
MyRocket
SomeRocket
this object may reside at various remote locations in the network
Operator
MyRocket
Simulator
provides the data needed to create the rocket proxy

Synchronized Adapters

- In the presence of concurrency, objects designed to function in a single threaded environment need to be protected
- The wrapper can provide the needed synchronization (when the ground object is private)



Synchronized Adapters with Access Control

- An adapter can also introduce blocking of threads in a way that is sensitive to the state of the ground object
- Illustration: block thread waiting for a specific target velocity to be reached and release it when
  - a reset has been issued (return false)
  - the target velocity has been reached or exceeded (return true)

Synchronized Adapters with Access Control

```
public synchronized void step(VelVec vel) {
  // ...
  notifyAll();
}

public synchronized void reset() {
  // ...
  notifyAll();
}

public synchronized boolean targetVelocity(VelVec vel) {
  // mag returns integer magnitude of vector
  while (mag(velocity) != 0 && mag(velocity) < mag(vel)) {
    try { wait(); } catch (InterruptedException e) { }
  }
  if(mag(velocity) == 0) return false; // reset issued
  else return true; // target velocity reached
}
```

Synchronized Adapters with Access Control

```
public synchronized void step(VelVec vel) {
  // ...
  notifyAll();
}

public synchronized void reset() {
  // ...
  notifyAll();
}

public synchronized boolean targetVelocity(VelVec vel) {
  // mag returns integer magnitude of vector
  while (mag(velocity) != 0 && mag(velocity) < mag(vel)) {
    try { wait(); } catch (InterruptedException e) { }
  }
  if(mag(velocity) == 0) return false; // reset issued
  else return true; // target velocity reached
}
```

Does this work as expected?

What's wrong?

When `notifyAll` is invoked in a synchronized method M_1:

1. An arbitrarily chosen waiting thread T is removed from the waiting set
2. T blocks while re-obtaining the lock
   - T blocks AT LEAST until M_1 releases the lock – if another synchronized method M_2 acquires the lock first, T continues to block!
3. T resumes from the point of wait

A thread in `step` may change shared state before T resumes from the wait in `targetVelocity`!

Synchronized Adapters with Access Control

```
public synchronized void step(VelVec vel) {
  // ...
  notifyAll();
}

public synchronized void reset() {
  // ...
  tracking = false;
  notifyAll();
}

public synchronized boolean targetVelocity(VelVec vel) {
  tracking = true; // remains true unless reset
  while (tracking && mag(velocity) < mag(vel)) {
    try { wait(); } catch (InterruptedException e) { }
  }
  if(!tracking) return false;
  tracking = false;
  return true;
}
```

Extending Atomicity

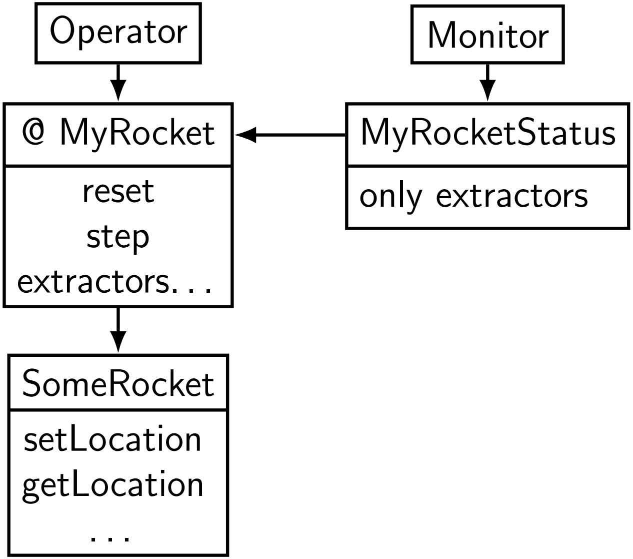
- An adapter can augment existing methods with additional functionality while making the entire operation atomic
- Illustration: augment the step method to include a logging action

```
public synchronized void step(VelVec vel) {
  flightTime++;
  velocity = vel;
  location.add(velocity);
  Log.addEntry(flightTime, location);
}

class Log {
  public static synchronized void addEntry(int time,
                                           PosPoint loc) {
    // store/print entry...
  }
}
```

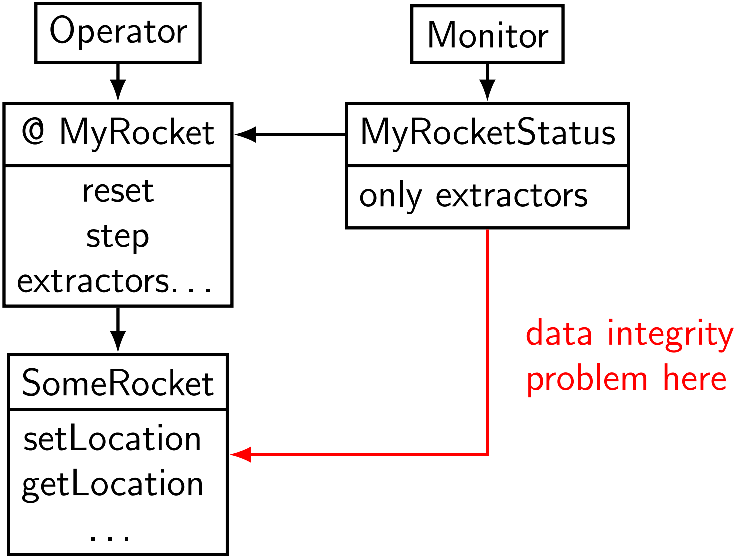
Read Only Adapters

- It is often the case that data needs to be protected against unauthorized modification
- A read-only wrapper gives full access to the object data without the risk of being modified

Read Only Adapters



Read Only Adapters



Programming Concerns

- The methods of the immutable object should be declared final
- The base object B should not be leaked to users of the immutable object X

```
public final class MyRocketStatus {
  private final int flightTime;
  private final PosPoint location;
  public MyRocketStatus(MyRocket rocket) {
    flightTime = rocket.getFlightTime();
    location = rocket.getLocation();
  }
  public int getFlightTime() { return flightTime; }
  public PosPoint getLocation() { return location; }
}

public class Log {
  public static synchronized void addEntry(MyRocketStatus stat) {
    // store/print status...
  }
}
```

Acceptor

- receives requests in the form of messages
- interprets the messages within the context of a system
- by considering pending requests
- by analyzing the program state
- by enforcing execution rules
- initiates actions performed by underlying ground objects

```
public interface Acceptor {
  public void accept(MessageType m);
}
```

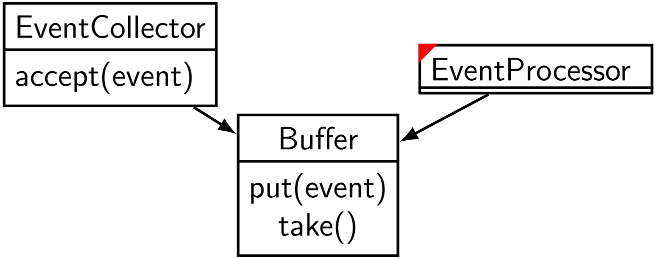
Design Strategies

- An acceptor may
  - employ its own private ground objects
  - delegate actions to ground objects
  - employ a dispatch table
  - generate a thread for each response
  - maintain state information
  - schedule multiple agents
  - maintain history logs
  - filter and modify messages
- Acceptors may work in tandem

Event Loops

Key concepts and components

- event – the class of messages accepted
- buffer – holder of messages to be processed
- collector – recipient of events and known to event producers
- processor – event dispatcher knowing the ground objects



Sample Code Using Bounded Buffer

```
public class EventCollector implements Acceptor {
  protected BlockingQueue<Event> buff;

  public EventCollector() {
    buff = new ArrayBlockingQueue<>(100);
    new EventProcessor(buff);
  }

  public synchronized void accept(Event e) {
    buff.put(e);
  }
}
```

Sample Code Using Bounded Buffer

```
public class EventProcessor implements Runnable, Acceptor {
  protected BlockingQueue<Event> buff;
  public EventProcessor(BlockingQueue<Event> b) {
    buff = b;
    new Thread(this).start();
  }

  public void run() {
    while (true) {
      accept(buff.take());
    }
  }

  public void accept(Event e) {
    switch (e.eventCode()) {
      // dispatch...
    }
  }
}
```