4. Safety Patterns

Prof. O. Nierstrasz
Roadmap

> Idioms, Patterns and Architectural Styles
> Immutability:
  — avoid safety problems by avoiding state changes
> Full Synchronization:
  — dynamically ensure exclusive access
> Partial Synchronization:
  — restrict synchronization to “critical sections”
> Containment:
  — structurally ensure exclusive access
Idioms, Patterns and Architectural Styles

- **Immutability:**
  - avoid safety problems by avoiding state changes

- **Full Synchronization:**
  - dynamically ensure exclusive access

- **Partial Synchronization:**
  - restrict synchronization to “critical sections”

- **Containment:**
  - structurally ensure exclusive access
Idioms, Patterns and Architectural Styles

*Idioms, patterns and architectural styles express best practice in resolving common design problems.*

**Idiom**
- “an implementation technique”
  - function objects, OCF, futures, RPC

**Design pattern**
- “a commonly-recurring structure of communicating components that solves a general design problem within a particular context”
  - Observer, Proxy, Master/Slave

**Architectural pattern**
- “a fundamental structural organization schema for software systems”
  - dataflow, blackboard
Idioms, Patterns and Architectural Styles

Immutability:
— avoid safety problems by avoiding state changes

Full Synchronization:
— dynamically ensure exclusive access

Partial Synchronization:
— restrict synchronization to “critical sections”

Containment:
— structurally ensure exclusive access
Pattern: Immutable classes

**Intent:** Bypass safety issues by *not changing an object’s state* after creation.

**Applicability**

> When objects represent values of simple ADTs
>   — colours (*java.awt.Color*), numbers (*java.lang.Integer*)
> When classes can be separated into mutable and immutable versions
>   — *java.lang.String* vs. *java.lang.StringBuffer*
> When updating by copying is cheap
>   — “hello” + “ ” + “world” → “hello world”
> When multiple instances can represent the same value
>   — i.e., two copies of 712 represent the same integer
Immutability variants

**Variants**

**Stateless methods**
— methods that do not access an object’s state do not need to be synchronized (can be declared static)
— any temporary state should be local to the method

**Stateless objects**
— an object whose “state” is dynamically computed needs no synchronization!

**“Hardening”**
— object becomes immutable after a mutable phase
— expose to concurrent threads only after hardening
Immutable classes — design steps

> Declare a class with instance variables that are never changed after construction.

```java
class Relay {                     // helper for some Server class
    private final Server server_;
    Relay(Server s) {             // blank finals must be initialized in all // constructors
        server_ = s;             // initialized in all
    }
    void doit() {                //
        server_.doIt();
    }
}
```
Design steps ...

> Especially if the class represents an immutable data abstraction (such as `String`), consider overriding `Object.equals` and `Object.hashCode`.

> Consider writing methods that generate new objects of this class. (e.g., `String` concatenation)

> Consider declaring the class as `final`.

> If only some variables are immutable, use synchronization or other techniques for the methods that are not stateless.
Example — immutable complex numbers

```java
public class Complex {
    private final int x, y;
    public Complex(int x, int y) { this.x = x; this.y = y; }
    public Object plus(Complex other) {
        return new Complex(this.x+other.x, this.y+other.y);
    }
    public Object times(Complex other) {
        return new Complex(this.x*other.x - this.y*other.y,
                           this.x*other.y + other.x*this.y);
    }
    public boolean equals(Object o) {
        if (o instanceof Complex) {
            Complex other = (Complex) o;
            return (this.x == other.x) && (this.y == other.y);
        }
        return false;
    }
    ...
}
```

Complex numbers never change state, so are thread-safe by design
Roadmap

> Idioms, Patterns and Architectural Styles
> Immutability:
  — avoid safety problems by avoiding state changes
> Full Synchronization:
  — **dynamically ensure exclusive access**
> Partial Synchronization:
  — restrict synchronization to “critical sections”
> Containment:
  — structurally ensure exclusive access
Pattern: Fully Synchronized Objects

**Intent:** Maintain consistency by *fully synchronizing all methods*. At most one method will run at any point in time.

**Applicability**
- You want to eliminate all possible read/write and write/write conflicts, regardless of the context in which it the object is used.
- All methods can run to completion without waits, retries, or infinite loops.
- You do not need to use instances in a layered design in which other objects control synchronization of this class.
Applicability ...

> You can avoid or deal with liveness failures, by:
   — Exploiting partial immutability
   — Removing synchronization for accessors
   — Removing synchronization in invocations
   — Arranging per-method concurrency
   — ...

More on this later …
Full Synchronization — design steps

> Declare all (public) methods as synchronized
  
  — Do *not allow any direct access to state* (i.e., no public instance variables; no methods that return references to instance variables).

  — Constructors cannot be marked as synchronized in Java — use a synchronized block in case a constructor passes this to multiple threads.

  — Methods that access static variables must either do so via static synchronized methods or within blocks of the form `synchronized(getClass()) { ... }`.

> Ensure that *every public method leaves the object in a consistent state*, even if it exits via an exception.
Design steps ...

> Keep methods *short* so they can atomically run to completion.

> State-dependent actions must rely on *balking*:
  — Return failure (i.e., exception) to client if preconditions fail
  — If the precondition does not depend on state (e.g., just on the arguments), then check outside synchronized code
  — Provide public accessor methods so that clients can check conditions before making a request
Example: a BalkingBoundedCounter

```java
public class BalkingBoundedCounter implements BalkingCounter {
    protected long count = BoundedCounter.MIN; // from MIN to MAX

    public synchronized long value() { return count; }
    public synchronized void inc() throws BalkingException {
        if (count >= BoundedCounter.MAX) {
            throw new BalkingException("cannot increment");
        } else {
            ++count;
        }
        checkInvariant();
    }
    public synchronized void dec() throws BalkingException {
        ...
    }
}
```

**NB: Client may need to busy-wait**

What safety problems could arise if this class were not fully synchronized?
BusyWaitingClient

```java
public abstract class BusyWaitingClient extends Thread {
    BusyWaitingClient() { this.start(); }
    public void run() {
        boolean succeeded = false;
        while (!succeeded) {
            try {
                action();
                succeeded = true;
            } catch (BalkingException e) {
                Thread.yield();
            }
        }
        abstract void action() throws BalkingException;
    }
}
```

Busy-wait loop could starve
Example: an ExpandableArray

```java
public class ExpandableArray<Value> {
    protected Value[] data; // the elements
    protected int size; // the number of slots used
    static final int DEFAULT_SIZE = 10;
    public ExpandableArray(int initialSize) {
        data = newArray(initialSize); // reserve some space
        size = 0;
    }
    ...
    public synchronized Value at(int i) throws NoSuchElementException {
        if (i < 0 || i >= size) {
            throw new NoSuchElementException();
        } else {
            return data[i];
        }
    }
    ...
}
```

All public operations are synchronized
Example ...

```java
... public synchronized void append(Value x) { // add at end
    if (size >= data.length) { // need a bigger array
        Object[] olddata = data; // so increase ~50%
        data = newArray(3 * (size + 1) / 2);
        System.arraycopy(olddata, 0, data, 0, olddata.length);
    }
    data[size++] = x;
}
public synchronized void removeLast() throws NoSuchElementException { // remove the last
    if (size == 0) {
        throw new NoSuchElementException();
    } else {
        data[--size] = null;
    }
}
```
Bundling Atomicity

> Consider adding synchronized methods that perform

\textit{sequences of actions as a single atomic action}

```java
public interface Mutator<Value> {
    public Value update(Value x);
}
public class BatchArray<Value> extends ExpandableArray<Value> {
    public BatchArray(int initialSize) { super(initialSize); }
    public BatchArray() { super(); }
    public synchronized void updateAll(Mutator<Value> p) {
        for (int i = 0; i < size; ++i) {
            data[i] = p.update(data[i]);
        }
    }
}
```

What possible liveness problems does this introduce?
public class BatchArrayTest {
    private BatchArray<Integer> ba;
    private Thread t1, t2;
    private static final int ARRAYSIZE = 20;

    public BatchArrayTest() {
        ba = new BatchArray<Integer>();
        for (int i = 1; i <= ARRAYSIZE; ++i) {
            ba.append(new Integer(i));  // all values different
        }
        t1 = mutatorThread(1, ba);
        t2 = mutatorThread(2, ba);
    }

    private Thread mutatorThread(final int id, final BatchArray<Integer> ba) {
        return new Thread() {
            public void run() {
                ba.updateAll(new Mutator() {
                    public Object update(Object x) {
                        Thread.yield();       // yielding has no effect
                        randomSleep();         // makes no difference
                        return id;             // set all values to my id
                    }
                });
            }
        };
    }

    ...
@Test
public void testNoInterference() {
    t1.start();
    t2.start();
    try {
        t1.join();
        t2.join();
    } catch (InterruptedException e) {
        System.err.println("Could not join mutator threads!");
        e.printStackTrace();
    }
    for (int i = 0; i < ba.size(); i++) {
        assertEquals(ba.at(1), ba.at(i));
    }
}
Roadmap

> Idioms, Patterns and Architectural Styles
> Immutability:
>   — avoid safety problems by avoiding state changes
> Full Synchronization:
>   — dynamically ensure exclusive access
> Partial Synchronization:
>   — restrict synchronization to “critical sections”
> Containment:
>   — structurally ensure exclusive access
Pattern: Partial Synchronization

**Intent:** Reduce overhead by *synchronizing only within* “critical sections”.

**Applicability**
> When objects have both mutable and immutable instance variables.
> When methods can be split into a “critical section” that deals with mutable state and a part that does not.
Partial Synchronization — design steps

- **Fully synchronize** all methods
- Remove synchronization for
  - accessors to *atomic or immutable values*
  - methods that access mutable state through a single other, already synchronized method
- Replace method synchronization by *block synchronization* for methods where access to mutable state is restricted to a single, critical section
Example: LinkedCells

```java
public class LinkedCell {
    protected double value; // NB: doubles are not atomic!
    protected final LinkedCell next; // fixed

    public LinkedCell (double val, LinkedCell next) {
        value = val;
        this.next = next;
    }

    public synchronized double value() {
        return value;
    }

    public synchronized void setValue(double v) {
        value = v;
    }

    public LinkedCell next() { // not synced!
        return next; // next is immutable
    }

    ...
}
```
Example ...

```java
public double sum() {
    double v = value();
    if (next() != null) {
        v += next().sum();
    }
    return v;
}

public boolean includes(double x) {
    synchronized(this) {
        if (value == x) {
            return true;
        }
    }
    if (next() == null) {
        return false;
    } else {
        return next().includes(x);
    }
}
```
Roadmap

> Idioms, Patterns and Architectural Styles
> Immutability:
  — avoid safety problems by avoiding state changes
> Full Synchronization:
  — dynamically ensure exclusive access
> Partial Synchronization:
  — restrict synchronization to “critical sections”
> Containment:
  — structurally ensure exclusive access
Pattern: Containment

**Intent:** Achieve safety by avoiding shared variables. *Unsynchronized objects are “contained” inside other objects that have at most one thread active at a time.*

**Applicability**
- There is no need for shared access to the embedded objects.
- The embedded objects can be conceptualized as exclusively held resources.
Applicability ...

> Embedded objects must be structured as **islands** — communication-closed sets of objects reachable only from a single unique reference.
  
  — They *cannot* contain methods that *reveal their identities* to other objects.

> You are willing to *hand-check designs* for compliance.

> You can deal with or *avoid indefinite postponements* or deadlocks in cases where host objects must transiently acquire multiple resources.
**Contained Objects — design steps**

> Define the *interface* for the outer host object.
  > The host could be, e.g., an Adaptor, a Composite, or a Proxy, that provides synchronized access to an existing, unsynchronized class

> Ensure that the *host is fully synchronized*, or is in turn a contained object.
Design steps ...

> Define instances variables that are *unique references* to the contained objects.
  — Make sure that these references *cannot leak* outside the host!
  — Establish *policies* and implementations that ensure that acquired references are really unique!
  — Consider methods to duplicate or *clone contained objects*, to ensure that copies are unique
Managed Ownership

> Model contained objects as *physical resources*
  — If you have one, then you can do something that you couldn't do otherwise
  — If you have one, then no one else has it
  — If you give one to someone else, then you no longer have it
  — If you destroy one, then no one will ever have it

> If contained objects can be passed among hosts, define a *transfer protocol*
  — Hosts should be able to acquire, give, take, exchange and forget resources
  — Consider using a dedicated class to manage transfer
A minimal transfer protocol class

A simple buffer for transferring objects between threads:

```java
public class ResourceVariable<Resource> {
    protected Resource resource;
    public ResourceVariable(Resource resource) {
        this.resource = resource;
    }
    public synchronized Resource exchange(Resource newResource) {
        Resource oldResource = resource;
        resource = newResource;
        return oldResource;
    }
}
```

Use as follows:
```
var = rv.exchange(var);
```
What you should know!

> Why are immutable classes inherently safe?
> Why doesn’t a “relay” need to be synchronized?
> What is “balking”? When should a method balk?
> When is partial synchronization better than full synchronization?
> How does containment avoid the need for synchronization?
Can you answer these questions?

- When is it all right to declare only some methods as synchronized?
- When is an inner class better than an explicitly named class?
- What could happen if any of the ExpandableArray methods were not synchronized?
- What liveness problems can full synchronization introduce?
- Why is it a bad idea to have two separate critical sections in a single method?
- Does it matter if a contained object is synchronized or not?
License

http://creativecommons.org/licenses/by-sa/2.5/

You are free:
• to copy, distribute, display, and perform the work
• to make derivative works
• to make commercial use of the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor.

Share Alike. If you alter, transform, or build upon this work, you may distribute the resulting work only under a license identical to this one.

• For any reuse or distribution, you must make clear to others the license terms of this work.
• Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.