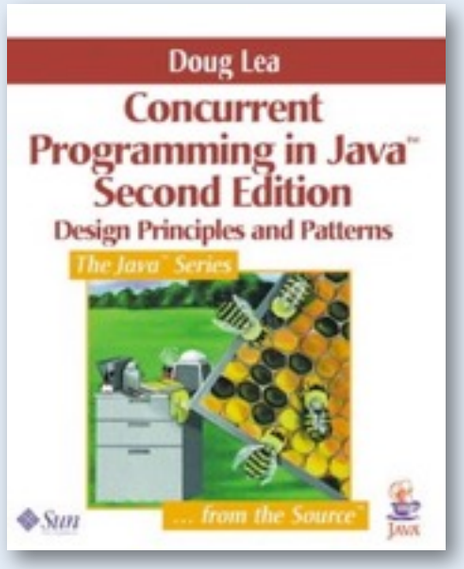
b
UNIVERSITÄT
BERN

4. Safety Patterns & Transactional Memory

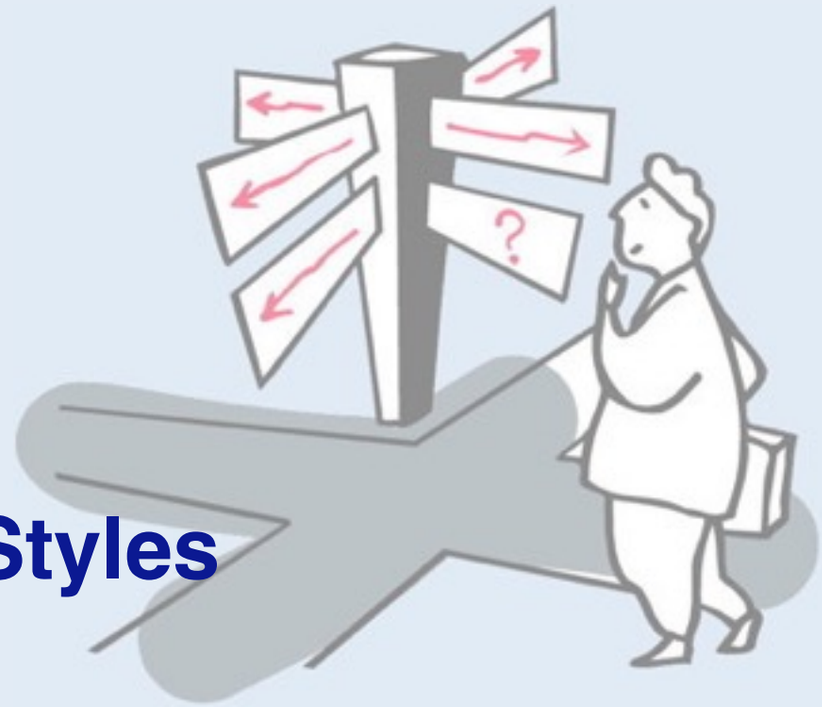
Oscar Nierstrasz

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
- Transactional Memory



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
- Transactional Memory

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

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
- Transactional Memory

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 updating by copying is cheap
  - —"hello" + " " + "world" → "hello world"
- When classes can be separated into mutable and immutable versions
  - —`java.lang.String` vs. `java.lang.StringBuffer`
- When multiple instances can represent the same value
  - —i.e., two copies of 712 represent the same integer

A design pattern consists of extensive *documentation* of a *general solution* to a *recurring design problem*. Typically design patterns require several pages of text to describe them in sufficient detail. Here we only summarize them in a few slides.

Design patterns are usually described in a structured way, as seen here.

There is a descriptive *name* for the pattern. The *intent* summarizes the purpose of the pattern. *Applicability* summarizes the situations in which it can be applied. Additionally there may be *design steps*, *examples*, *variants*, *discussion* and *related patterns* or *synonyms*.

The *immutable class* pattern is one of the simplest ways to ensure safety, since an immutable object cannot be corrupted.

Immutability 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.

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

Most of the patterns are illustrated by running examples in the accompanying examples repository. This is one of few “toy” examples without an running example.

The `Relay` is immutable because it simply delegates requests to a server, and the server identity cannot be changed.

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., color manipulation)

- Consider declaring the class as `final`.

- If only some variables are immutable, use synchronization or other techniques for the methods that are not stateless.

Immutable objects are often used to represent abstract “values”. If these values can be comapred, then you should implement the appropriate equals method. In Java, as in in most object-oriented languiages, when you implement equals you should always reimplement the hashCode method, so that values can be used reliably as keys in a dictionary.

Note that these points are only suggestions, not rules. For example, declaring a class as final ensures that its immutability won't be invalidated by subclasses, but it also impedes extensibility.

Example — immutable complex numbers

```
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

This example illustrates the main points. Instance variables are final. Public methods return new instances of this class. We override equals (and hashCode).

We do not declare the class as final, so it is possible to extend the class, though this might break immutability.

Recall, source code of all examples is available from the course examples git repo:

```
git clone git://scg.unibe.ch/lectures-cp-examples
```

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
- Transactional Memory

Pattern: Fully Synchronized Objects

Intent: Maintain consistency by *fully synchronizing all public 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.

In this pattern, all public methods are mutually exclusive, and make no use of `wait` or `notify`.

Full Synchronization in Java — 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.

Be very careful if the constructor passes the pseudo-variable `this` to an external method before construction is complete. This can be very evil.

Note that static variables belong to the class, and not to the object. To ensure that access is synchronized, you can use a `synchronized` block that uses the result of `getClass()` as a lock. (All accessors must use the same lock!)

Design steps ...

- 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

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

Normally you should apply this pattern only when all operations can always run to completion without waiting for a synchronization condition.

If this is not possible, then the only solution is to “balk”, i.e., to give up.

Since this means client requests may fail, you should provide an interface that allows clients to check the condition themselves before requesting service.

Example: a BalkingBoundedCounter

```
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?

We will see a number of different implementation of `BoundedCounter` objects, each illustrating a different design pattern for synchronization.

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

(Ask yourself what possible race conditions would exist.)

BusyWaitingClient

```
public class BalkingBoundedCounterTest {
   ...
   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

The problem with a balking design is that clients are forced to busy-wait if a request fails. In such a design there is no guarantee that the client will eventually succeed.

Example: an ExpandableArray

```
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 = new Array(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

The challenge here will be how to support atomic operations over the entire array.

Example ...

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

Bundling Atomicity

> Consider adding synchronized methods that perform *sequences of actions as a single atomic action*

```
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?

In this solution, the ExpandableArray passes its all elements to a Mutator object within a synchronized updateAll method. This guarantees that all updates will occur within a single critical section.

What could go wrong here?

Testing atomicity

```
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
                    } }); } };
    }
```

Each mutator thread tries to set all array values to its unique id

These Mutator objects just set all elements of the array to a single value. To test whether `updateAll` is truly atomic, we introduce a `Thread.yield()` and a random `sleep`. Since the `yield` occurs within a synchronized `updateAll` method, the lock is not released, and atomicity is ensured. (A lock is only released with a `wait`, not a `yield`.)

Testing atomicity

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

We test that the array contains one set of unique values.

If we remove synchronization from BatchArray, the test may fail.

This method tests that `updateAll` is truly atomic by checking that all values in the array are the same, i.e., that the two mutators did not somehow interleave.

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
- Transactional Memory

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.

This pattern relaxes full synchronization by reducing the scope of the critical sections. The caveat is that you must reason very carefully about which code belongs in a critical section or you may endanger safety. Don't apply this pattern unless you are sure of what you are doing and you are certain it is needed (i.e., to improve liveness).

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

The strategy here is to start with a fully synchronized design, and then gradually introduce partial synchronization where it is safe to do so. This is wiser than starting with an unsynchronized (and unsafe) design and gradually introducing critical sections.

Note that access to immutable values is *always* safe, so no synchronization is needed there.

When replacing method synchronization by block synchronization, take care to introduce a *single synchronized block*. If the object is in an inconsistent state between two synchronized blocks, you risk violating the class invariant and another synchronized public method may incorrectly start in between.

Example: LinkedCells

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

    public LinkedCell (double value, LinkedCell next) {
        this.value = value;
        this.next  = next;
    }
    public synchronized double value() {
        return value;
    }
    public synchronized void setValue(double v) {
        value = v;
    }
    public LinkedCell next() {          // not synched!
        return next;                    // next is immutable
    }
```

LinkedCell

In Java, updating an integer value is guaranteed to be atomic, but updating a double is not. In these example we therefore use doubles to properly motivate the need for synchronized access.

Note that the `next` variable is immutable in this linked list example. The accessor therefore does not need to be synchronized.

Example ...

```
    public double sum() {                   // add up all element values
        double v = value();                 // get via synchronized accessor
        if (next() != null) {
            v += next().sum();
        }
        return v;
    }
    public boolean includes(double x) {     // search for x
        synchronized(this) {                // synch to access value
            if (value == x) {
                return true;
            }
        }
        if (next() == null) {
            return false;
        } else {
            return next().includes(x);
        }
    }
}
```

Interestingly the `sum` method does not need to be synchronized at all. The linked list is traversed using immutable links that need no synchronized access. The values themselves are mutable, but they are read using a synchronized accessor.

The `includes` method recursively searches the linked list for a specific value. Each cell tests if it holds the searched for value, and if not, delegates to the next cell. The only mutable value accessed is the `value` variable, so only here do we need a synchronized block.

Can we get rid of the synchronized block here too by using the synchronized accessor method instead of directly reading the value variable?

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**
- Transactional Memory

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.

This is an important pattern to recognize. Even in a highly concurrent application, certain objects may only be manipulated within a given thread at a time. Such objects are “owned” by or “contained” within another object. As long as the outer object is synchronized, then the contained objects do not need to be.

As we shall see shortly, things get a little trickier when the contained objects may move from one container to another.

Applicability (2)

- 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.

You must take care that references to contained objects not leak outside their containers. If they do, all bets are off!

Aside: the same holds true for "`private`" instance variables in plain Java applications. If a reference to a `private` variable leaks out, it can be accessed and manipulated from outside.

The last point concerns designs in which owned objects may be transferred from one container to another. If containers have to incrementally acquire resources, this can lead to liveness problems (upcoming lecture).

Contained Objects — Design steps

- Define the *interface* for the outer host object.
  - The host could be, e.g., an Adaptor, 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 (2)

- 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

A particular danger arises when contained objects represent values that may be returned to clients. In this case, either you must make sure that ownership is transferred (see next slide), or that a *copy* is returned instead.

Managed Ownership

- Model contained objects as *physical resources*
  1. If you have one, then you can do something that you couldn't do otherwise
  2. If you have one, then no one else has it
  3. If you give one to someone else, then you no longer have it
  4. 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:

```
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;
    }
}
```

Ownership

Usage: `var = rv.exchange(var);`

Without synchronization we can provoke a race condition. Consider possible interleavings of two threads invoking `exchange` at the same time ...

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
- **Transactional Memory**

Transactional Memory

Intent: Move the management of the synchronization from the programmer to the runtime by only specifying what pieces of code must happen atomically.

```
// Insert a node into a doubly-linked list atomically
atomic {
     if (node_count == MAX_SIZE)
          abort;
     newNode->prev = node;
     newNode->next = node->next;
     node->next->prev = newNode;
     node->next = newNode;
     node_count++;
}
```

Revisiting the Account Example

```
class Account {
 float balance;
 synchronized void deposit(float amt) {
    balance += amt;
 }
 synchronized void withdraw(float amt) {
    if(balance < amt)
      throw new OutOfMoneyError();
    balance -= amt;
 }
 void transfer_wrong1(Acct other, float amt) {
    other.withdraw(amt);
    // race condition: wrong sum of balances
    this.deposit(amt);
 }
 synchronized void transfer_wrong2(Acct other, float amt) {
    // can deadlock with parallel reverse-transfer
    other.withdraw(amt);
    this.deposit(amt);
 }
}
```

```
class Account {
 float balance;
 void deposit(float amt) {
   atomic { balance += amt; }
 }
 void withdraw(float amt) {
   atomic {
     if(balance < amt)
       throw new OutOfMoneyError();
     balance -= amt;
   }
 }
void transfer(Acct other,
              float amt) {
  atomic {
    other.withdraw(amt);
    this.deposit(amt);
  }
}
```

Transactional Memory (contd.)

- Alternative to lock-based synchronization
  - Simpler than synchronization
  - You don't lock on a particular object; just say **atomic**
- Transaction = piece of code that executes R/W to shared memory
  - Inspired from database transactions
  - ACI semantics (but D from databases)
    - *Atomicity*
    - *Consistency*
    - *Isolation*

Transactional Memory (contd.)

- Threads are optimistic
  - write to a memory location without worrying about other threads
  - keep log of changes
  - make the changes permanent with a *commit* **if** no conflicts
- Can be either SW, HW, or SW + HW
  - Hardware
    - *Instructions: start transaction, end transaction, rollback*
    - *A rollback would undo all the loads and stores from the beginning of the transaction*
    - *Problem: HW has a hard time with long transactions...*
  - Software
    - *slow*
    - *easier to experiment with algorithms, approaches*

Transactional Memory (contd.)

- Applicability
  - When your job mix allows it: situations where there are frequent reads and infrequent writes
  - Otherwise, overhead of rollback and of keeping logs too large
  - When you do not have side effects in transactions

An interesting perspective

- Analogy between TM and GC (http://homes.cs.washington.edu/~djg/papers/analogy_oopsla07.pdf)
  - *system takes care of it without the programmer ever knowing or caring about*
  - *in some cases manual management will be more performant than the automated version*

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?*