Objects are Enough!

Unsubstantiated Assertion

Andrew Black
Portland State University
black@cs.pdx.edu
We question all our beliefs, except for the ones that we really believe, and those we never think to question.

Orson Scott Card, *Speaker for the dead*
What we believe:

- We must have a domain specific language to write a clear and concise solution to a complex problem if
  - we want the domain concepts to stand out, and
  - the boiler-plate to recede into the background
- This “fact” is the very premise of this workshop
Suppose...

... that it’s false
Suppose...

... that the code of *every* (well-constructed) program forms a higher-level, program specific language

[Baniassad 2009]
Suppose...

... that the ordinary mechanisms of an ordinary (well-constructed) programming language are sufficient to express “domain-specific” concepts
Then ...

Objects would be enough for describing distributed algorithms
Objects are Enough — A Position Paper
ECOOP '93 workshop on Object-based Distributed Programming
Andrew P. Black
Digital Cambridge Research Laboratory

I. Objects Encapsulate Location

The idea of using an object to encapsulate the implementation details of a computational structure is central to object-oriented programming. Objects can also be used as the unit of distribution, that is, each object can be confined to a single address space. Alternatively, object boundaries can be taken as orthogonal to distribution boundaries, thus admitting the possibility that parts of an object may be in different address spaces. A mechanism must then be provided for the various parts of the distributed object to communicate with each other.

Our experience is that it is perfectly adequate to let object boundaries also encapsulate location. In other words, we require that the whole of an object be in a single address space. The added complexity of a truly distributed object does not seem to be warranted. If a given object’s state appears to need to be distributed, the desired effect can always be obtained by partitioning the state of the given object into smaller objects, each of which is located in a suitable (possibly remote) address space. The original object's state can now be replaced by references to the new objects.

II. Invocation Encapsulates Communication

In a centralized environment, the insistence of the object model that every operation be represented as a message sent to an object is cumbersome and unnatural. In a distributed environment, the idea that $3 + 4$ means send the message ‘+4’ to the object ‘3’ reflects the communication that is actually necessary to compute the sum. Moreover, it makes us conscious that a while the representations of the two objects may be quite different, a mutually understood way of transmitting the parameter to the target of the invocation must exist.

III. Little Need be added for Distributed programming!

Many people ask what they should add to their programming environment to make it suitable for distributed programming. Another common question is what extra facilities are necessary to make the environment distributed. The answer to both questions is that little need be added;

1. Disallow the testing of equality on object identity

   • It compromises encapsulation

   A number of writers have recently pointed out that enabling clients to ascertain, by comparing the identity of the references to two interfaces, whether they are (or are not) implemented by the same object compromises encapsulation.

   • It is expensive to implement in the absence of global naming

   Not all object systems incorporate a global naming scheme. Alternative arrangements whereby all object names are relative to the object that holds the reference are quite workable; the relative name may be thought of as a path from the referee to the referent. Even if global names exist, it is usually more efficient to allocate a global name to an object only when its name must be passed to a remote location, and for on-machine references to be abbreviated. Thus, many objects may not have or need a global name, and even for those objects that do have such names, finding them may involve non-trivial computation.
Two Examples from Smalltalk

Constructing HTML

SeasideDemo » renderContentOn: html
  html heading: 'Rendering Demo'.
  html heading
    level: 2;
    with: 'Rendering basic HTML: '.
  html div
    class: 'subcomponent';
    with: htmlDemo.
"render the remaining components ..."
Two Examples from Smalltalk

Writing an executable grammar

expression ::= superMessage
  / implicitSelfMessage "implicit self-send"
  / operatorMessage "receiver is operator expression"
  / operatorSuperMessage
  / tuple
implicitSelfMessage ::= messageWithArgs
messageWithArgs ::= (identifier, expression) plus
nullaryExpression ::= nullarySuperMessage / nullaryMessage
nullaryMessage ::= term, (dot, identifierWithoutExpression) star
Third starting point

Related publications


Peter A. Jonsson and Johan Nordlander, *Strengthening Supercompilation For Call-By-Value Languages*. In *Second International Workshop on Metacomputation (META)*, Pereslavl-Zalessky, Russia, July 2010.


Timber:

- Made timing explicit
- Explored reactivity as the basic programming model
- Combines the power of a functional language with that of mutable objects
- Supported static timing analysis as well as dynamic adaptivity
Concurrency and Objects

Object: encapsulated, mutable state + identity

serialized execution

asynchronous message send

concurrent execution

serialized execution

Messages: asynchronous send + autonomous response
Reactivity

- Event = method invocation = message send
  - Output event: sending a message
  - Input event: being invoked
- No active input
- Method execution = reaction = non-blocking code sequence
- Objects alternate between transient *activity* and indefinite periods of rest
Reactivity

- Event = method invocation = message send
  - Output event: sending a message
  - Input event: being invoked
- No active input
- Method execution = reaction = non-blocking code sequence
- Objects alternate between transient activity and indefinite periods of rest

Update local state / create new objects / send messages
Blocking input is a bad idea
Blocking input is a bad idea

Blocking message send (or procedure call) is the wrong way to get input

- Program has to choose which message to send
- This represents a premature commitment
  - Order of external events *not* under program control!
  - Events are missed, or reordered!
Blocking input is a bad idea

- Blocking message send (or procedure call) is the wrong way to get input
  - Program has to choose which message to send
  - This represents a premature commitment
    - Order of external events *not* under program control!
    - Events are missed, or reordered!

- Event loop using `select`
  - Not compositional
  - Helps only if all events of interest are encoded uniformly and posted to a single port
Controlling timing

- Each event has a *timeline*, an interval from a *baseline* … to a... *deadline*

- Default timeline is same as that of sender

- Can also be set explicitly:
  - after (10*seconds) send defers baseline
  - before (25*milliseconds) send sets deadline

**Key idea:** code can be *time-dependent, yet platform-independent*. Static analysis determines feasibility.
Accessing the timeline

- Built-in constants *baseline* and *deadline*
  - defined only within methods
  - provide access to the baseline and the deadline for the current method execution
Example: Ping Program

Example from Java documentation
Connects to each of a list of hosts and measures the time required to complete the connection. Uses a selector and two additional threads to demonstrate non-blocking connects and the multithreaded use of a selector.

```java
-> hosts = ["dogbert", "ratbert", "ratberg", "theboss"]

-> ping hosts (Port 515)

dogbert: lookup & connect after 20.018 ms
ratbert: lookup & connect after 41.432 ms
ratberg: NetError "Host name lookup failure" after 70.282 ms
theboss: no response within 2 s
```
ping hosts port env =

    object
    outstanding := hosts

    in let
        client host start peer =
        record
            connect     = action
                env.putStrLn(host ++ ": lookup & connect after "
                                ++ show (baseline-start))

            outstanding := remove host outstanding
                peer.close

            deliver   _ = action done

    neterror e = action
        env.putStrLn(host ++ ": " ++ show e ++ " after "
                       ++ show (baseline-start))

        outstanding := remove host outstanding

        close       = action done

    cleanup = action
        forall h <- outstanding do
            env.putStrLn(h ++ ": no response within " ++ show timeout)
        env.quit

    timeout = 2*seconds

    in record
        main = action
            forall h <- hosts do
                env.inet.tcp.open h port (client h baseline)
            after timeout cleanup
Java Version

http://java.sun.com/j2se/1.4/docs/guide/nio/example/Ping.java

```java
public class Ping {
    static int DESTINATION_PORT = 13;
    static int port = DESTINATION_PORT;

    static class Target {
        InetAddress address;
        SocketChannel channel;
        IOException failure;
        long connectStart;
        long connectFinish;
        boolean alive = false;
    }

    static class Connector extends Thread {
        Selector selector;
        Printer printer;
        LinkedList pending = new LinkedList();

        private void shutdown() {
            shutdown = true;
            pending = null;
        }

        static class Printer extends Thread {
            LinkedList pending = new LinkedList();

            public void run() {
                for (;;) {
                    try {
                        pending.wait(0);
                        synchronized (pending) {
                            pending.notify();
                        }
                    } catch (InterruptedException e) {
                        fail = e;
                    }

                    public void run() {
                        for (;;) {
                            Target t = null;
                            synchronized (pending) {
                                try {
                                    pending.wait();
                                    t = firstpending.removeFirst();
                                } catch (InterruptedException e) {
                                    return;
                                }
                                if (t != null) {
                                    target = t;
                                } else {
                                    if (pending.size() == 0) {
                                        break;
                                    }
                                }
                                t.show();
                            }
                        }
                    }

                    static class Connector extends Thread {
                        Selector selector;
                        Printer printer;
                        LinkedList pending = new LinkedList();

                        private void shutdow
```
Comparison

- Timber version
  - all actions are defined inside *ping* object
    - can safely manipulate *outstanding* in mutual exclusion
  - solution is straightforward:
    - one object, one instance variable

- Java version
  - 10 class variables
  - 3 threads
    - timeout, printing, de-multiplex of connection events
  - Less concurrency (*gethostbyname* bug!)
Position paper

- Limitations from “Objects are enough”
  + Timing & Asynchrony from Timber
  + Syntax from Grace
  + Season to taste
2PC — Coordinator

object coordinator is:

var xmd ← ... — a local map from transaction ids to transaction metadata
var tid ← ... — the unique identifier of this transaction

process is:

... — do the work
(xmd.for tid).cohorts.do:
    each → each.queryToCommit (tid, coordinator)
self.commitPhase tid after tid.queryTimeout

method abort (tid, cohort) is:
(xmd.for tid).addToAborters cohort

method agreement (tid, cohort) is:
(xmd.for tid).addToAgreeable cohort

potential infinite recursion when a cohort fails and never recovers
2PC — Coordinator

method commitPhase tid is:
  if (xmd.for tid).cohorts.size = (xmd.for tid).agreeable.size then:
    (xmd.for tid).cohorts.do:
      each → each ◀ commit (tid, coordinator)
  else :
    (xmd.for tid).cohorts.do:
      each → each ◀ rollback (tid, coordinator)
      tid.markAsAborted

  self ◀ completePhase tid
  after (xmd.for tid).commitTimeout

method ack (tid, cohort) is:
  (xmd.for tid).recordAckFor cohort

method nack (tid, cohort) is:
  (xmd.for tid).recordNackFor cohort
method completePhase tid is:
  if (xmd.for tid).ackers = (xmd.for tid).cohorts
    then:
      (xmd.for tid).markAsCommitted
  else:
    self.completePhase tid
    after (xmd.for tid).commitTimeOut
    − potential infinite recursion when a cohort fails and never recovers
**2PC — Cohort**

**object cohort is:**

**var** xmd ← ... — a local map from transaction ids to transaction metadata

**method queryToCommit (tid, coord) is:**

if (xmd.for tid).willingToCommit
    then:
        coord ← agreement (tid, self)
    else:
        coord ← abort (tid, self)

**method commit (tid, coord) is:**

(xmd.for tid).completeLocalOperations
(xmd.for tid).releaseResources
coord ← ack (tid, self)

**method rollback (tid, coord) is:**

(xmd.for tid).undoLocalOperations
(xmd.for tid).releaseResources
coord ← nack (tid, self)

What happens if the deadline of a method cannot be met? In many cases we can ascertain statically whether a particular deadline can be met on particular hardware, given a knowledge of the hardware execution speed and the code in the method. Recall that methods never block. Of course, this is not always possible, so a method failing to meet its deadline is treated as a failure. Equivalently, a failure is treated as a method failing to meet its deadline.

Emerald provided built-in checkpoint and recovery mechanisms; checkpoints could be written to "stable storage" or to the volatile memory of another node. Hutchinson and Jeffrey. Failure detection was also built-in. It seems to me now that checkpointing, recovery and failure detection should all be "plugable": the language should support the concepts, but the mechanisms should be under programmer control. Emerald atomically checkpointed all of the state of an object; in practice, some of this may be redundant or inconsistent, and it might be better to allow the programmer direct control over exactly what is checkpointed and how it is recovered.

The lesson from Erlang is that it is generally much easier to resume correct operation from a known "good state" than by sifting through the bits of a representation that may contain inconsistencies. Thus, it may make more sense to continuously stream log entries to an active backup rather than to checkpoint one's whole state to passive storage, even though the atomic checkpoint seems like a higher level primitive. Moreover, a conventional atomic checkpoint must be a blocking operation, while sending a log entry is asynchronous, so in our framework they are not interchangeable. Similarly, after years of research on failure detectors, starting with Chandra and Toueg's landmark paper, we now know that, even though all failure detectors are necessarily approximations, the exact character of the approximation matters in rather important ways. Som while it seems reasonable to support failure handling as a language primitive, the determination of what constitutes a failure should probably be left to the designer of the algorithm.
Frequently Asked Questions

• Has your 2PC algorithm been, implemented, tested, or proved correct?
  - No

• Has your proposed language been specified formally, implemented, or subjected to user studies?
  - No

• Do you have any indication that this approach will work in the large?
  - No