

Communicating Processes, Components and Scaleable Systems

Peter Welch
Computing Laboratory
University of Kent at Canterbury
(P.H.Welch@ukc.ac.uk)

IFIP WG2.4, San Miniato, Italy (May 2001)

5-Jun-01

Copyright P.H.Welch

1

Components?



Components must be *composeable* ...
... and they must compose *simply!*

5-Jun-01

Copyright P.H.Welch

2

Components?



Mind you, just because components compose ...
... doesn't always mean that it makes sense ...

Components?



... to compose them ...




*Image courtesy of Philips TASS <<http://www.tass.philips.com/>>

Components?

- If we understand A and B separately, we must be able to deduce *simply* their combined behaviour.

plug together

no surprises

- Semantics $[A + B] = \text{Semantics}[A] + \text{Semantics}[B]$
- A and B must be *composable* ... 


Composition?

- Complex systems are *composed* from *less complex* components ...
- ... which are *composed* from *simpler* components ...
- ... which are *composed* from *simpler* components ...
- ... etc ...
- ... which are *composed* from *simple* components.

Composition?

- Composition rules must be simple and yield no surprises.
- Whatever it is they encapsulate, **components** must have **interfaces** that are *clean, complete* and *explicit*.
- Hardware systems are forced (by physics/geometry) to be built like this.
- Software systems have no such constraints. We think we can do better than nature ... and get into trouble.

Object Orientation?

- OO systems are hierarchies of component networks.
- OO components encapsulate state and algorithms for manipulating that state.
- OO components exist concurrently and interact by message passing across well-defined interfaces.
- OO languages support the above ... **as if!** 

Communicating Processes?

- CSP systems are hierarchies of component networks.
- CSP components encapsulate state and algorithms for manipulating that state.
- CSP components exist concurrently and interact by message passing across well-defined interfaces.
- CSP languages support the above ... ***claim!***



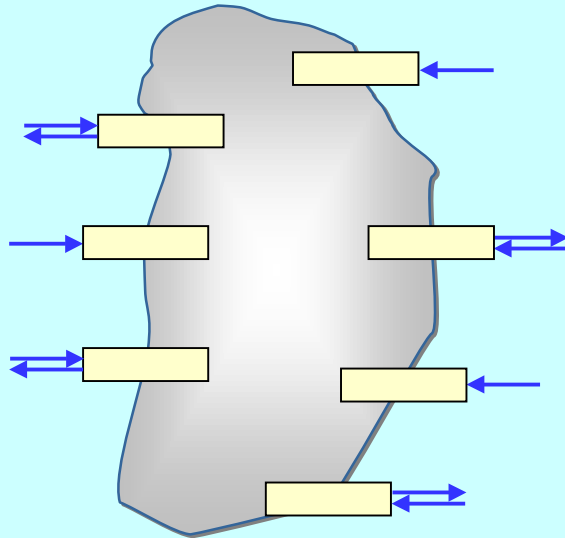
Objects Considered Harmful

- Data encapsulation breaks down all too easily.
- *Private* attributes of an object may themselves be objects.
- All objects live on a universally accessible heap.
- Hence, ***private*** attributes may be ***shared*** (!) between any number of objects - sometimes by design and often by accident (we just have to give the reference away).
- Either way, this contradiction means that local control of an attribute is lost and, with it, local and simple reasoning.

Objects Considered Harmful

What we tell our students:

“An opaque object interacting with a wider system of objects via its formal public interface.”



5-Jun-01

Copyright P.H.Welch

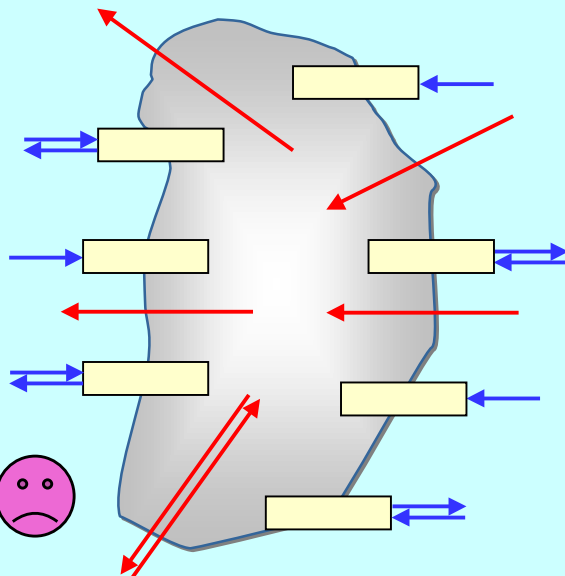
11

Objects Considered Harmful

The truth:

“Undeclared interactions between the object and any other parts of the system ...”

**Documentation
= Source Code**



5-Jun-01

Copyright P.H.Welch

12

Objects Considered Harmful

- Even when the attribute is a primitive data-type, we are not safe ... (*I am grateful to Tom Locke for the following, rather scary, observation*) ...
- ... data encapsulation still breaks down too easily.

5-Jun-01

Copyright P.H.Welch

13

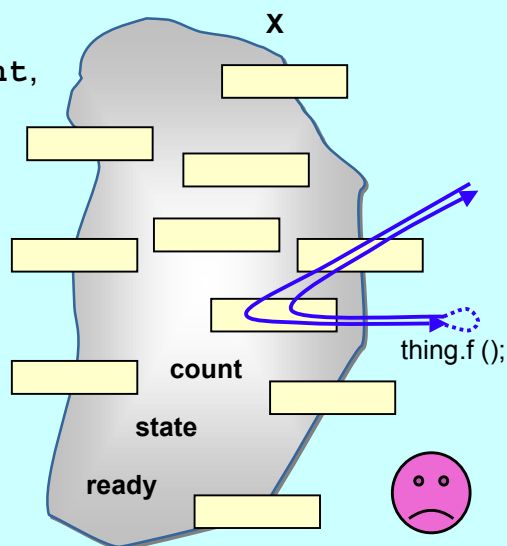
Objects Considered Harmful

Suppose class **x** has a *private* integer field, **count**, and *private* methods that see and change it.

Suppose the following code occurs in one of those methods:

```
count = 42; thing.f();
```

What is the value of **count** after these two statements?

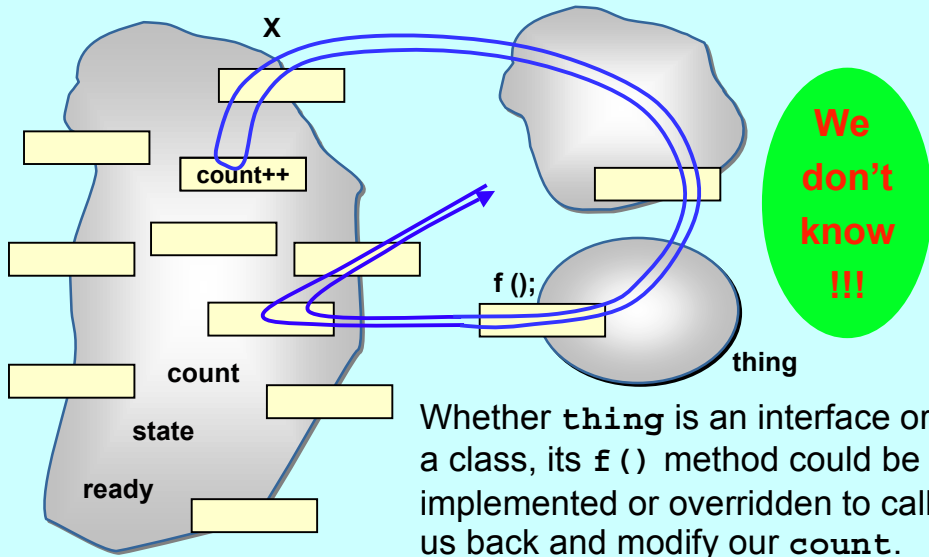


5-Jun-01

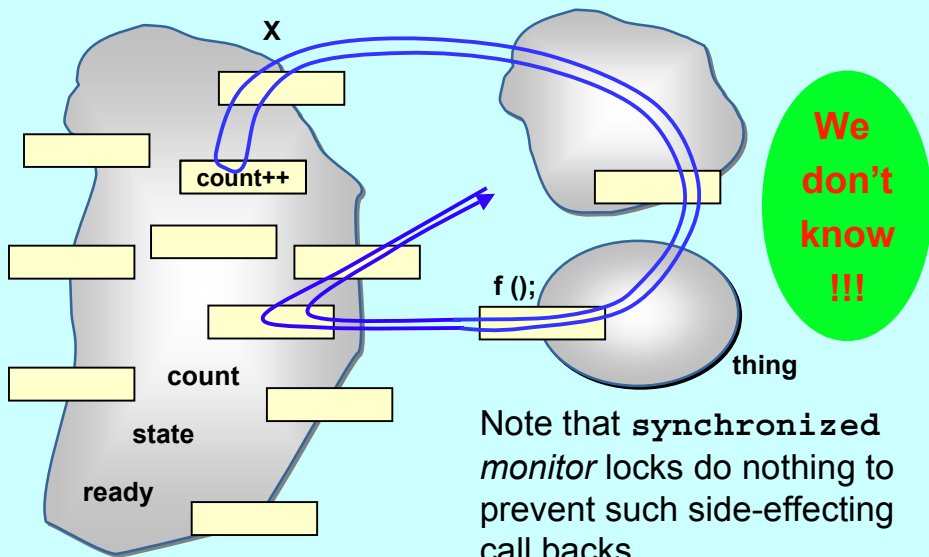
Copyright P.H.Welch

14

Objects Considered Harmful



Objects Considered Harmful



Objects Considered Harmful

We don't know the value of `count` after the following two statements:

```
count = 42; thing.f ();
```

This lack of ability to reason locally about private fields is strangely familiar. In the bad old days, free use of global variables led us into exactly the same mess.

Structured programming led us out of that mire. Is *object orientation* taking us back in?

Processes Considered Good

What is the value of `count` after these two statements?

```
count = 42;  
thing.write (anything);
```

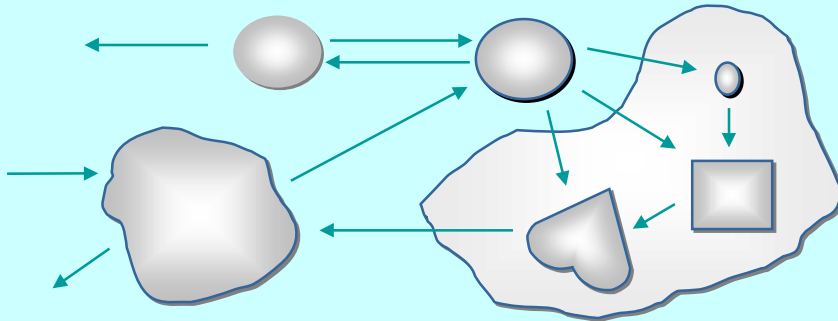
thing →

This time we do know. *What-you-see-is-what-you-get*. The answer is 42.

The only way `count` can be changed is if *this process* changes it - and it doesn't! Local analysis is sufficient. We don't need to worry about what lies beyond the `thing` channel. Our intuitive understanding about the sequence of instructions has been honored.

Nature has very large numbers of independent agents, interacting with each other in regular and chaotic patterns, at all levels of scale:

... nuclear ... human ... astronomic ...



5-Jun-01

Copyright P.H.Welch

19

The Real(-Time) World and Concurrency

Computer systems - to be of use in this world - need to model that part of the world for which it is to be used.

If that modeling can reflect the natural concurrency in the system ... it should be **simpler**.

Yet concurrency is thought to be an **advanced** topic, **harder** than serial computing (which therefore needs to be mastered first).

5-Jun-01

Copyright P.H.Welch

20

This tradition is **WRONG!**

... which has (radical) implications on how we should educate people for computer science ...

... and on how we apply what we have learnt ...



What we want from Concurrency

- A powerful tool for **simplifying** the description of systems.
- Performance that spins out from the above, but is **not** the primary focus.
- A model of concurrency that is mathematically clean, yields no engineering surprises and scales well with system complexity.

Java Monitors - NOT THIS!

- Easy to learn - but very difficult to apply ... safely ...
- Monitor methods are *tightly interdependent* - their semantics compose in *complex ways* ... the whole skill lies in setting up and staying in control of these complex interactions ...
- Threads have no structure ... there are no *threads within threads* ...
- Big problems when it comes to scaling up complexity ...

Java Monitors - NOT THIS!

- No guarantee that any **synchronized** method will ever be executed ... (e.g. *stacking* JVMs)
- Even if we had above promise (e.g. *queueing* JVMs), standard design patterns for **wait / notify** fail to guarantee liveness (“*Wot, no chickens?*”)

See:

<http://www.hensa.ac.uk/parallel/groups/wotug/java/discussion/3.html>

<http://www.nist.gov/itl/div896/emaildir/rt-j/msg00385.html>

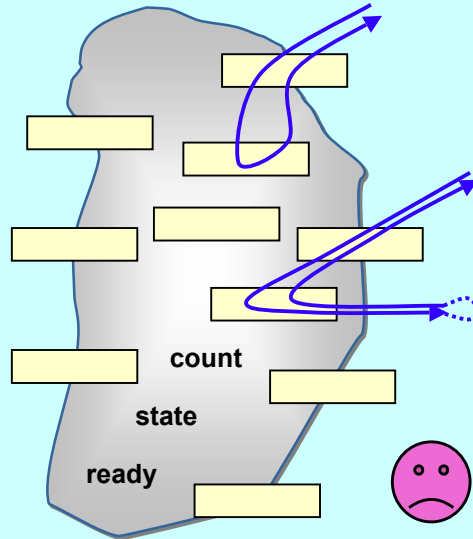
<http://www.nist.gov/itl/div896/emaildir/rt-j/msg00363.html>

Objects Considered Harmful

Most objects are dead - they have no life of their own.

All methods have to be invoked by an external thread of control - they have to be *caller oriented* ...

... a somewhat curious property of so-called *object oriented* design.

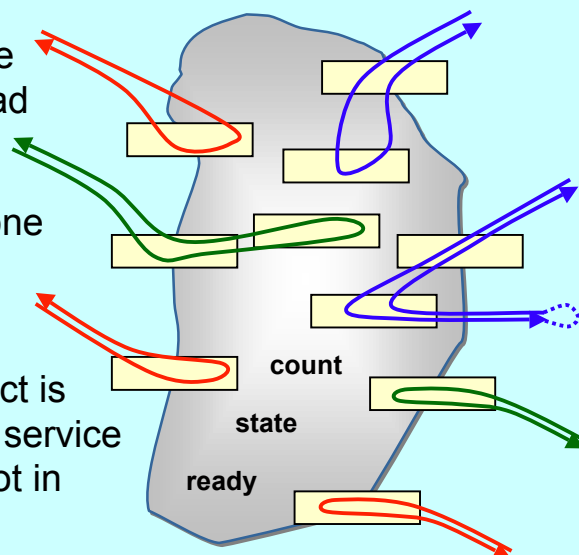


Objects Considered Harmful

The object is at the mercy of *any* thread that sees it.

Nothing can be done to prevent method invocation ...

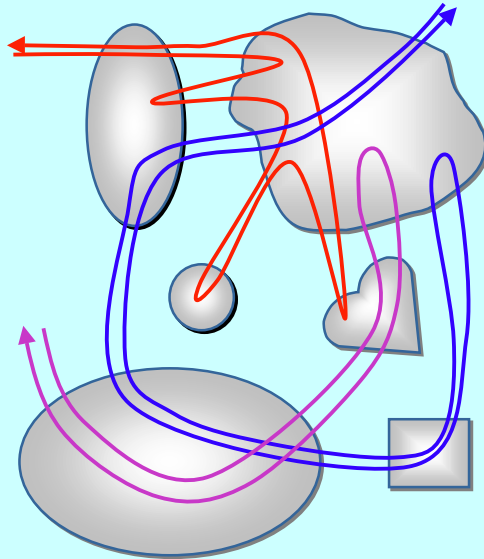
... even if the object is not in a fit state to service it. The object is not in control of its life.



Objects Considered Harmful

Each single thread of control snakes around objects in the system, bringing them to life *transiently* as their methods are executed.

Threads cut across object boundaries leaving spaghetti-like trails, *paying no regard to the underlying structure*.



Communicating Sequential Processes (CSP)

A mathematical theory for specifying and verifying complex patterns of behaviour arising from interactions between concurrent objects.

CSP has a formal, and *compositional*, semantics that is in line with our informal intuition about the way things work.

Claim

Why CSP?

- Encapsulates fundamental principles of communication.
- Semantically defined in terms of structured mathematical model.
- Sufficiently expressive to enable reasoning about deadlock and livelock.
- Abstraction and refinement central to underlying theory.
- Robust *and commercially supported* software engineering tools exist for formal verification.

Why CSP?

- CSP libraries available for Java (**JCSP, CTJ**).
- Ultra-lightweight kernels* have been developed yielding **sub-100-nanosecond** overheads for context switching, process startup/shutdown, synchronized channel communication and high-level shared-memory locks.
- Easy to learn and easy to apply ...

* not yet available for JVMs.

Why CSP?

- After 5 hours teaching
 - ◆ exercises with 20-30 threads of control
 - ◆ regular and irregular interactions
 - ◆ appreciating and eliminating race hazards, deadlock, etc.

- CSP is (parallel) architecture neutral
 - ◆ message-passing
 - ◆ shared-memory



So, what is CSP?

CSP deals with **processes**, **networks** of processes and various forms of **synchronisation / communication** between processes.

A network of processes is also a process - so CSP naturally accommodates layered network structures (**networks of networks**).

We do not need to be mathematically sophisticated to work with CSP. That sophistication is pre-engineered into the model. We benefit from this simply by using it.

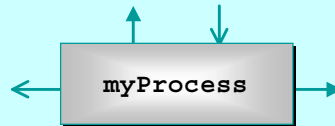
Processes



myProcess

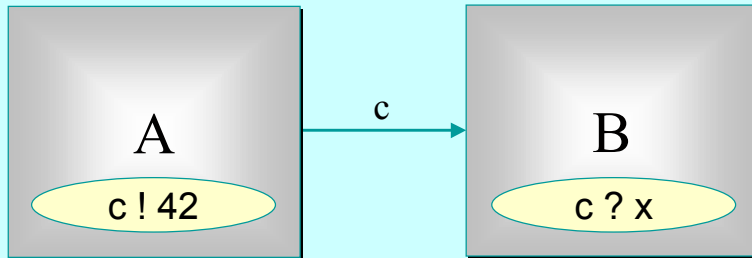
- A **process** is a component that encapsulates some data structures and algorithms for manipulating that data.
- Both its data and algorithms are **private**. The outside world can neither see that data *nor execute those algorithms!* [They are not *objects*.]
- The algorithms are executed by the process in its own thread (or threads) of control.
- So, how does one process interact with another?

Processes



- The simplest form of interaction is *synchronised* message-passing along **channels**.
- The simplest forms of channel are **zero-buffered** and **point-to-point** (i.e. *wires*).
- But, we can have **buffered** channels (*blocking/overwriting*).
- And **any-1**, **1-any** and **any-any** channels.
- And structured multi-way synchronisation (e.g. **barriers**) ...
- And high-level (e.g. **CREW**) *shared-memory* locks ...

Synchronised Communication

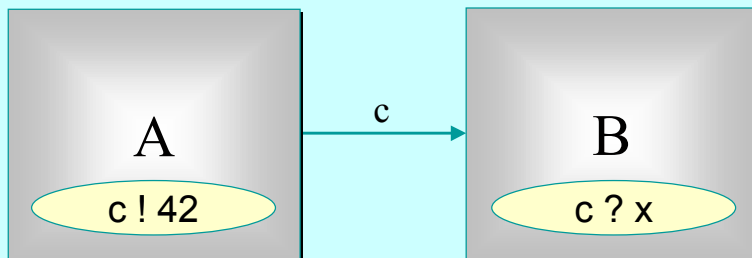


A may write on c at any time, but has to wait for a *read*.

B may read from c at any time, but has to wait for a *write*.

$A(c) \parallel B(c)$

Synchronised Communication



Only when both *A* and *B* are ready can the communication proceed over the channel c .

$A(c) \parallel B(c)$

Putting CSP into practice ...

JCSP

<http://www.cs.ukc.ac.uk/projects/ofa/jcsp/>

5-Jun-01

Copyright P.H.Welch

37

CSP for Java (JCSP) 1.0-rc1

[All Classes](#)

Packages

- [jcsp.awt](#)
- [jcsp.lang](#)
- [jcsp.pluginplay](#)
- [jcsp.pluginplay.ints](#)
- [jcsp.util](#)
- [jcsp.util.ints](#)

Overview Package Class [Tree](#) [Deprecated](#) [Index](#) [Help](#)

PREV NEXT [FRAMES](#) [NO FRAMES](#)

CSP for Java™ (JCSP) 1.0-rc1 API Specification

This document is the specification for the JCSP core API.

See: [Description](#)

Packages	
jcsp.awt	This provides CSP extensions for all java.awt components -- GUI events and widget configuration map to channel communications.
jcsp.lang	This provides classes and interfaces corresponding to the fundamental primitives of CSP.
jcsp.pluginplay	This provides an assortment of <i>plug-and-play</i> CSP components to wire together (with <code>Object</code> -carrying wires) and reuse.
jcsp.pluginplay.ints	This provides an assortment of <i>plug-and-play</i> CSP components to wire together (with <code>int</code> -carrying wires) and reuse.
jcsp.util	This provides classes and interfaces to customise the semantics of <code>Object</code> channels.
jcsp.util.ints	This provides classes and interfaces to customise the semantics of <code>int</code> channels.

Document: Done

5-Jun-01

Copyright P.H.Welch

38

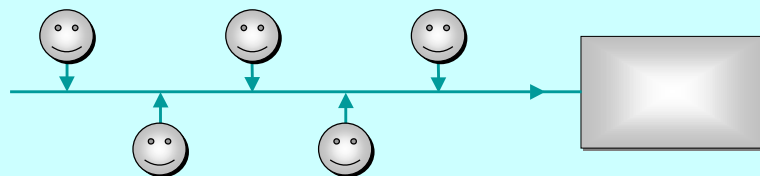
Shared Channels

- So far, all our channels have been point-to-point, zero-buffered and synchronised (i.e. standard CSP primitives);
- JCSP also offers multi-way shared channels (in the style of `occam3` and the `KRoC` shared channel library);
- JCSP also offers buffered channels of various well-defined forms.

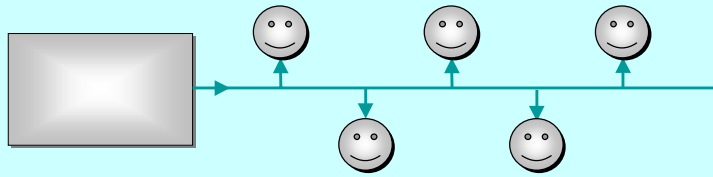
One2OneChannel



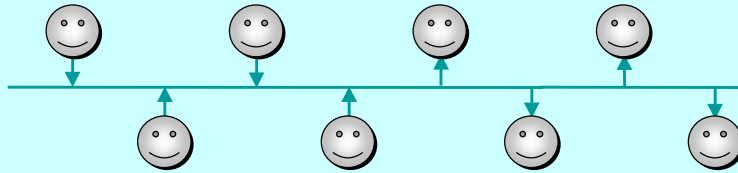
Any2OneChannel



One2AnyChannel

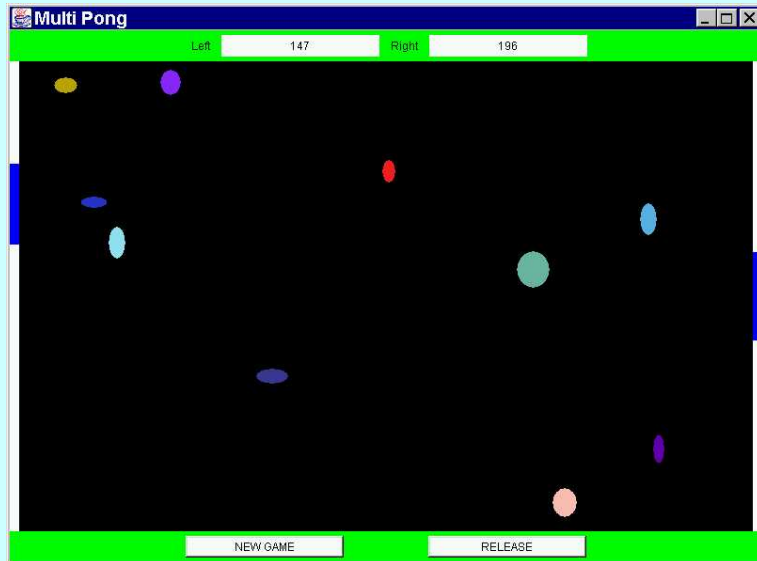


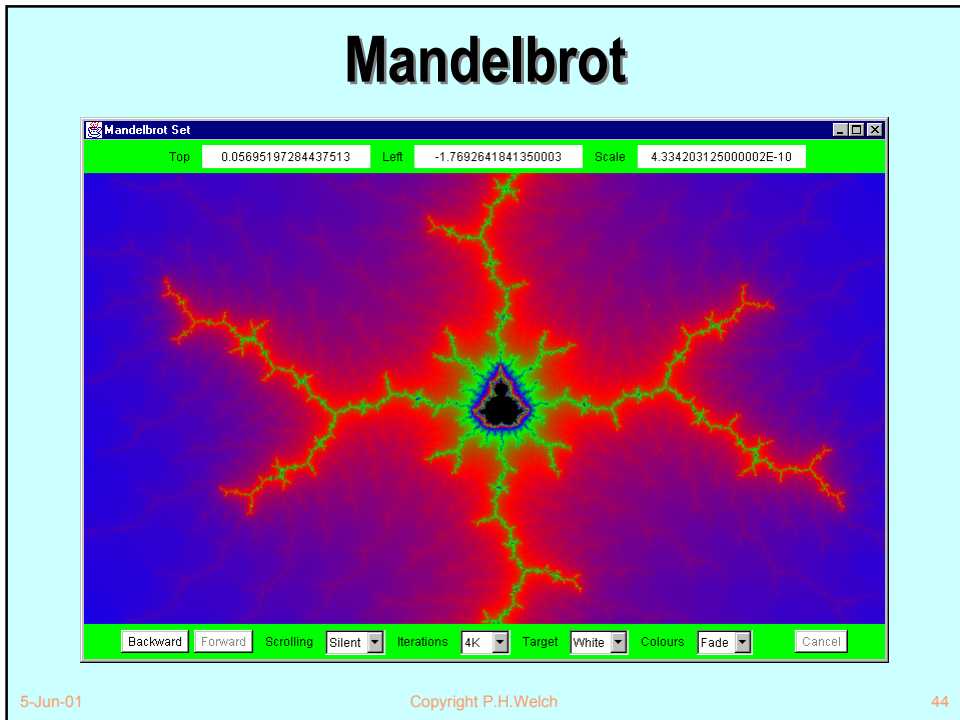
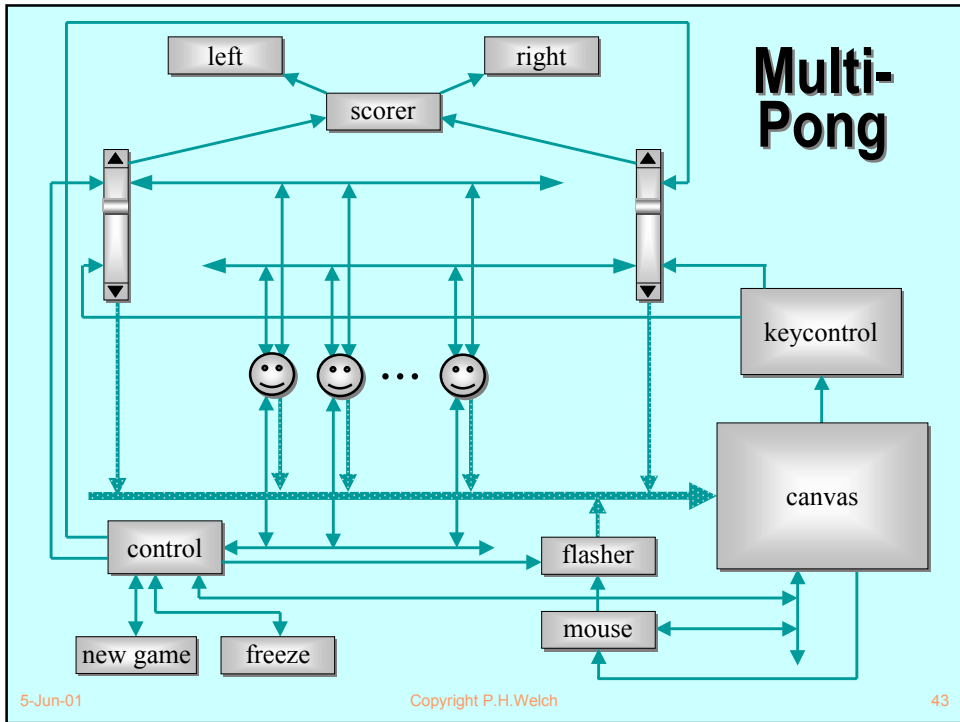
Any2AnyChannel



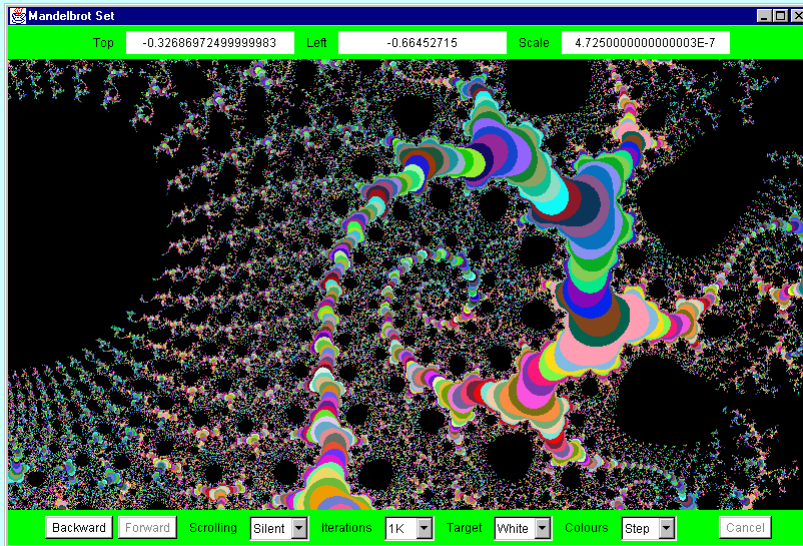
No ALTING!

Multi-Pong

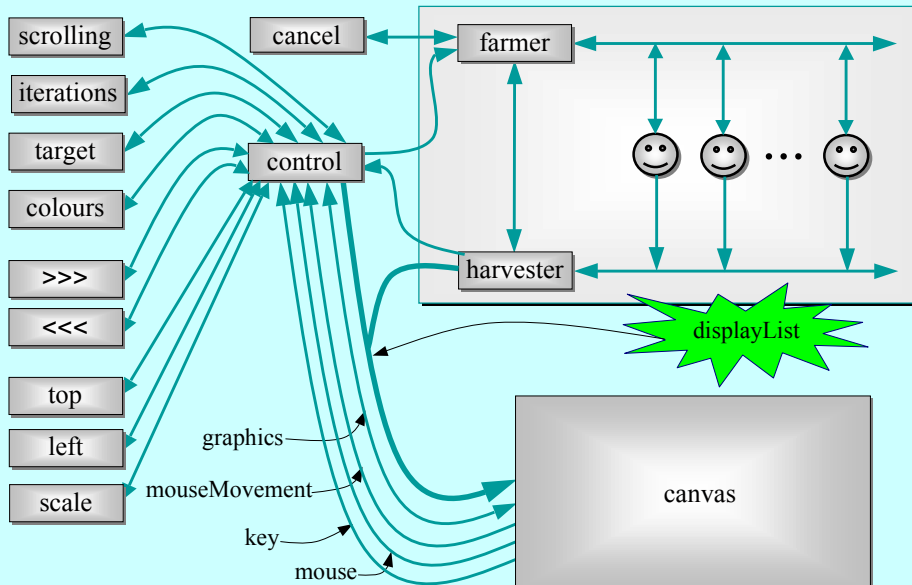




Mandelbrot



Mandelbrot



Good News!

The good news is that we can worry about each process on its own. A process interacts with its environment *through its channels*. It does not interact directly with other processes.

Some processes have *serial* implementations - these are just like traditional serial programs.

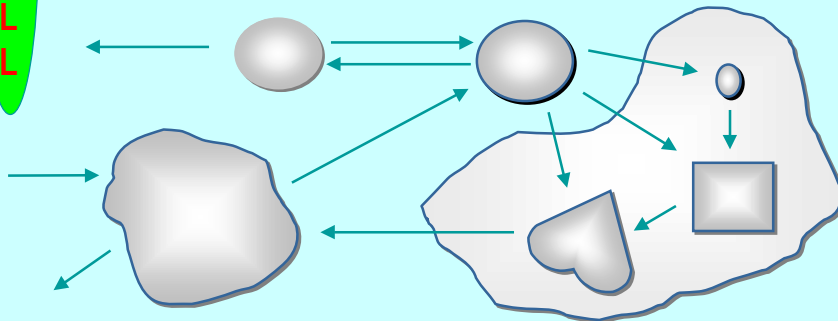
Some processes have *parallel* implementations - i.e. networks of sub-processes.

Our skills for serial logic sit happily alongside our new skills for concurrency - there is no conflict. This will scale!

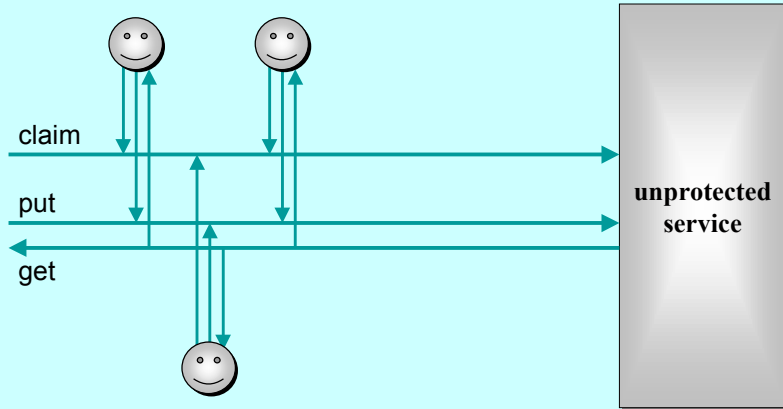
Nature has very large numbers of independent agents, interacting with each other in regular and chaotic patterns, at all levels of scale:

RECALL

... nuclear ... human ... astronomic ...

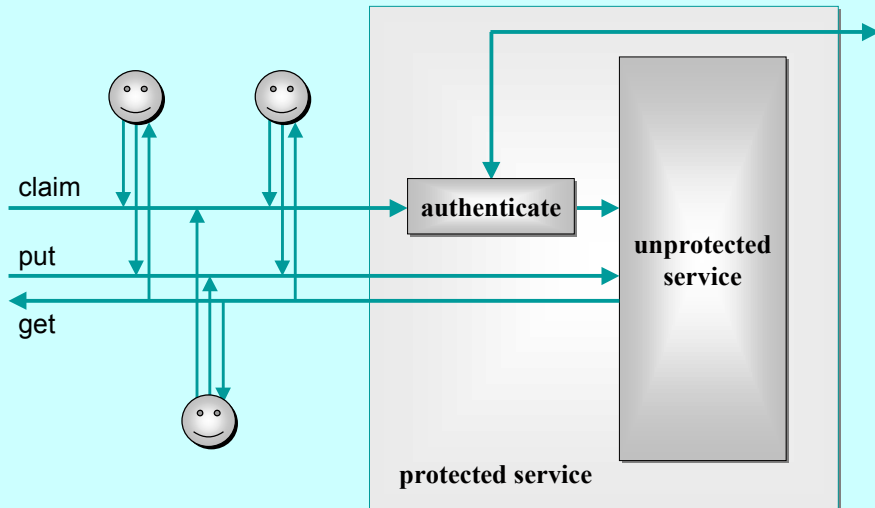


Extending Functionality



Extend service to authenticate claimants? Inheritance/overriding - NO!

Extending Functionality



Extend service by adding the necessary components - reuse the old intact.

Extending Functionality

Intercepting external channels and splicing in extra components modifies the services provided to *external* clients. The services provided by the original component - only now seen *internally* - are unchanged. That original component is still there, completely unchanged.

This is not the case with *method overriding* in OO. That changes the *internal* behaviour of the original superclass - internal invocations of the overridden method going to the subclass. To find out if this is happening, ***look at the source code ...***

But ... Language Matters?

- Support these ideas with a library (e.g. JCSP, CCSP) plus usage patterns (e.g. to control aliasing problems) ...
 - ◆ less development effort; (done it)
 - ◆ less upheaval for users - *much* easier to sell;
 - ◆ lets everyone try immediately - gets exposure and feedback;
 - ◆ long term solution? (build tools to check correct patterns)
- Support these ideas with a language ...
 - ◆ ideas relevant for wide application? (Yes!)
 - ◆ significant software engineering benefits? (encapsulation works => scalability)
 - ◆ direct syntactic expression of the model (=> simpler to use);
 - ◆ enforce correct patterns - the compiler is the tool for this;
 - ◆ generate efficient code - the compiler knows what's going on.

Continuing Work

- A CSP model for the Java monitor mechanisms (**synchronized**, **wait**, **notify**, **notifyAll**) has been built.
- This enables *any* Java threaded system to be analysed in CSP terms - e.g. for formal verification of freedom from deadlock/livelock.
- Confidence gained through the formal proof of correctness of the JCSP channel implementation:
 - ◆ a JCSP channel is a non-trivial monitor - the CSP model for monitors transforms this into an even more complex system of CSP processes and channels;
 - ◆ using FDR, that system has been proven to be a refinement of a single CSP channel and *vice versa* - **Q.E.D.**

Continuing Work

- Higher level synchronisation primitives (e.g. *CALL channels*, *barriers*, *buckets*, ...) that capture good patterns of working with low level CSP events.
- Proof rules and design tool support for the above.
- CSP kernels and their binding into JVMs to support JCSP (or *CoreJCSP* ... ?).
- **Communicating Threads for Java (CTJ):**
 - ◆ this is another Java class library based on CSP principles;
 - ◆ developed at the University of Twente (Netherlands) with special emphasis on real-time applications - it's excellent;
 - ◆ CTJ and JCSP share a common heritage and reinforce each other's on-going development - we do talk to each other!

Summary

WYSIWYG

Plug-n-Play

- CSP has a **compositional** semantics.
- CSP concurrency can **simplify** design:
 - ◆ data encapsulation within processes does not break down (unlike the case for objects);
 - ◆ channel interfaces impose clean decoupling between processes (unlike method interfaces between objects)
- JCSP enables direct Java implementation of CSP design.

Summary

- CSP kernel overheads are very small (around 50 *nanoseconds* on a 500 MHz. P3).
- Rich mathematical foundation:
 - ◆ 20 years mature - recent extensions include simple priority semantics;
 - ◆ higher level design rules (e.g. *client-server*, *resource allocation priority*, *IO-par*) with formally proven guarantees (e.g. freedom from deadlock, livelock, process starvation);
 - ◆ commercially supported tools (e.g. FDR).
- We don't need to be mathematically sophisticated to take advantage of CSP. It's built-in. Just use it!



Summary

- Process Oriented Design (processes, syncs, alts, parallel, layered networks).
- **WYSIWYG:**
 - ◆ each process considered individually (own data, own control threads, external synchronisation);
 - ◆ leaf processes in network hierarchy are ordinary *serial* programs - all our past skills and intuition still apply;
 - ◆ *concurrency* skills sit happily alongside the old serial ones.
- Race hazards, deadlock, livelock, starvation problems: we have a rich set of design patterns, theory, intuition and tools to apply.



Summary

- **Move away from:**
 - ◆ passive objects with method interfaces;
 - ◆ not in control of their own lives (e.g. cannot refuse calls);
 - ◆ endemic aliasing;
 - ◆ reuse by inheritance and overriding (which is dependent on superclass source code).
- **Move towards:**
 - ◆ active components - servicing and generating events;
 - ◆ strong encapsulation behind event (channel) interfaces;
 - ◆ strictly controlled aliasing;
 - ◆ hierarchic network structures (and interface structures);
 - ◆ reuse by plug-and-play (e.g. channel interception/splicing);
 - ◆ a CSP concurrency model

Summary

■ *Benefits:*

- ◆ explicit connections, minimal coupling, flexibility and reuse;
- ◆ strong encapsulation (state and execution threads);
- ◆ escape from uncontrolled aliasing and overriding problems;
- ◆ local reasoning about components - scalability;
- ◆ clean compositional semantics - scalability;
- ◆ more understandable systems - scalability;
- ◆ ubiquitous concurrency (for when you want to exploit it - e.g. for performance and response times) - scalability;
- ◆ static guarantees against concurrency errors - scalability.

Status

■ *Libraries and Run-Time kernels:*

- ◆ JCSP (soon *distributed*-JCSP, working on binding the `occam` kernel into a JVM);
- ◆ CCSP (with the `occam` kernel);

■ *Languages:*

- ◆ major extensions to `occam` (dynamic process linking, distributed dynamic connections, mobile data, mobile processes, GUI/graphics support, ...);
- ◆ researching entirely new language (combining *safe* OO practice and CSP);
- ◆ many open issues (efficient space-time implementation, controlled aliasing, deadlock analysis, ...)
- ◆ want wider experience of programming in the paradigm (**invitation**).

Acknowledgements

- Paul Austin - the original developer of JCSP (p_d_austin@hotmail.com).
- Andy Bakkers and Gerald Hilderink - the CTJ library (bks@el.utwente.nl, G.H.Hilderink@el.utwente.nl).
- Jeremy Martin - for the formal proof of correctness of the JCSP channel (Jeremy.Martin@comlab.ox.ac.uk).
- David Wood, Tom Locke, Fred Barnes and Jim Moores - UKC team ({dcw,ts12,frmb2,jm40}@ukc.ac.uk).
- Nan Schaller (ncs@cs.rit.edu), Chris Nevison (chris@cs.colgate.edu) and Dyke Stiles (dyke.stiles@ece.usu.edu) - for pioneering the teaching.
- The **WoTUG** community - its workshops, conferences and people.

URLs

- CSP** www.comlab.ox.ac.uk/archive/csp.html
- JCSP** www.cs.ukc.ac.uk/projects/ofa/jcsp/
- KRoC** www.cs.ukc.ac.uk/projects/ofa/kroc/
- FDR** www.formal.demon.co.uk/FDR2.html
- java-threads@ukc.ac.uk**
www.cs.ukc.ac.uk/projects/ofa/java-threads/
- WoTUG**
wotug.ukc.ac.uk/