Implementing Assertions in Distributed Object Systems

Rushikesh K. Joshi*
Department of Computer Science and Engineering
Indian Institute of Technology, Bombay

*Currently visiting NUS

Assertions in Software Systems

- A Boolean expression placed in a program where its evaluation is always true
- Typically supported as text annotations or embedded executables
- Focus is on *what* part rather than *how* part of the system
- *Detection, classification and Diagnosis* of errors

Applying Assertions: An Example

Insert (value: T)

Before execution, assert:

Count < *capacity*

…….Code for insert ……

After execution, assert:

Count = old count+1

Count <= *capacity*

Values[old count]=value

Assertions in Practice

- Contract view
  - Needs to be enforced by following it as a contract
  - A good design process

- Defensive programming view
  - An assertion expresses programmer's intentions
  - Failure? – handle exception/abort
  - A good debugging process

The contract view

- Example: Meyer's *design by contract* method
- Express contracts
- Assign the responsibilities

    ad-hoc redundant checks are not needed

- Produce contract documentation based on assertions

The Defensive Programming View

- Be on the defensive, check once more, have many assertions
- Criticized for redundancies
- Practical
- Systems built on contracts also support this view!

Assertion Systems

- Native
  - Eiffel
  - JAVA (Only recently)
- Extensions
  - C extensions: APP
  - JAVA extensions: JASS
- Intermediate: C predefined macro

The C Assert Macro

[The C Programming Language]

```
#include <assert.h>
....
void insert (int i) {

     assert (count < CAPACITY);
     .....
}
main () {

     ... insert (element); ...
}
```

Observations

- *Switching off by defining macro NDEBUG ahead of #include*
- Program is (unfortunately) aborted if the assertion expression returns *false*
- *Assertions tightly integrated with functional code*

Eiffel Assertion System

[Meyer]

- Preconditions
  - To be asserted before method execution begins
- Postconditions
  - To be asserted after method execution before returning the result
- Class Invariants
  - To be asserted
    - after every object creation
    - after every method execution
    - i.e. in observable states only,
    - not necessarily during method execution

Monitoring Assertions at Runtime

- Compile time options
  - No assertion checking
  - Preconditions only
  - Pre and post conditions
  - Pre,post conditions and invariants
- Exception handling mechanism required

An Example: DBC in Eiffel

```
insert (value: T) is
require
        count < capacity
do
    -- Actual functional code
ensure
        count = old count+1
        count <= capacity
        values[old count]=value
end
```

The contract

Party	obligations	benefits
Client	call put only on non-full LIFO	get the LIFO modified with element on top
Supplier	insert element on top	no need to deal with a case when LIFO is full

Who checks?

- The parties are expected to abide by the contract

- Weak to strong preconditions possible
  - changes the emphasis of checking them from supplier to client

Drawbacks of this approach

- DBC recommends a demanding style
- Could cause breakage of encapsulation or undesirable exposure of private data
  - e.g. exposure to variable count in above program
- Hence a uniform demanding approach is not practical in our opinion

Where's the problem

- No mechanism to separate concerns
  - of the assertion code
  - functional code of the supplier

- Requirements?

  - Assertions may need access to supplier's data
  - Client code needs to be freed from supplier's concerns
  - Suppliers want to be more *demanding*

JAVA Assertion System

[J2SE v1.4]

assert expression;

If evaluated to false: throws AssertionError

assert exp1: exp2;

passes on value returned by exp2 to constructor of AssertionError

Observations

- JAVA assertions disabled at runtime by default, with compile time options they can be enabled at various granularities
- Improvement over C style assertions: Exceptions over termination
- Assertions not a full DBC facility
- Tightly integrated with functional code

Extended Systems: APP [Rosenblum]

- As annotations
  - /*@ ..... *@/
- Assertions declared with function interfaces
  - Precondition:
    - assume x > 10
  - Postcondition: promise
    - promise *x == in *y
  - Return value constraint:
    - return y where y >0;
- Assertions associated with single statements in function bodies
  - Intermediate constraint
    - assert index <MAX

Inheritance needs to be handled

- Contractor-subcontractor interaction
- A contract declared by the superclass must be adhered to by the subclasses (conceptual compatibility)
- What does it mean to preconditions and postconditions?

Honest subcontractor view

[Meyer]

- May accept input rejected by the contractor
  - Precondition weakening

- May return a better result than promised by the contractor
  - Postcondition strengthening

An assertion model for inheritance: Eiffel

- Subclasses can refine the contract:
  - **require else** pre-new
    - pre-original or else pre-new
  - **ensure then** post-new
    - Post-original and then post-new

Extended Systems: JASS
[Univ. Oldenburg]

- Assertions as annotations
  - /** ..... **/
- Eiffel like extensions
  - Require, ensure, (class) invariant, loop invariant, loop variant (decreasing and positive)
- Expressions/function calls allowed
  - But they must be side effect free

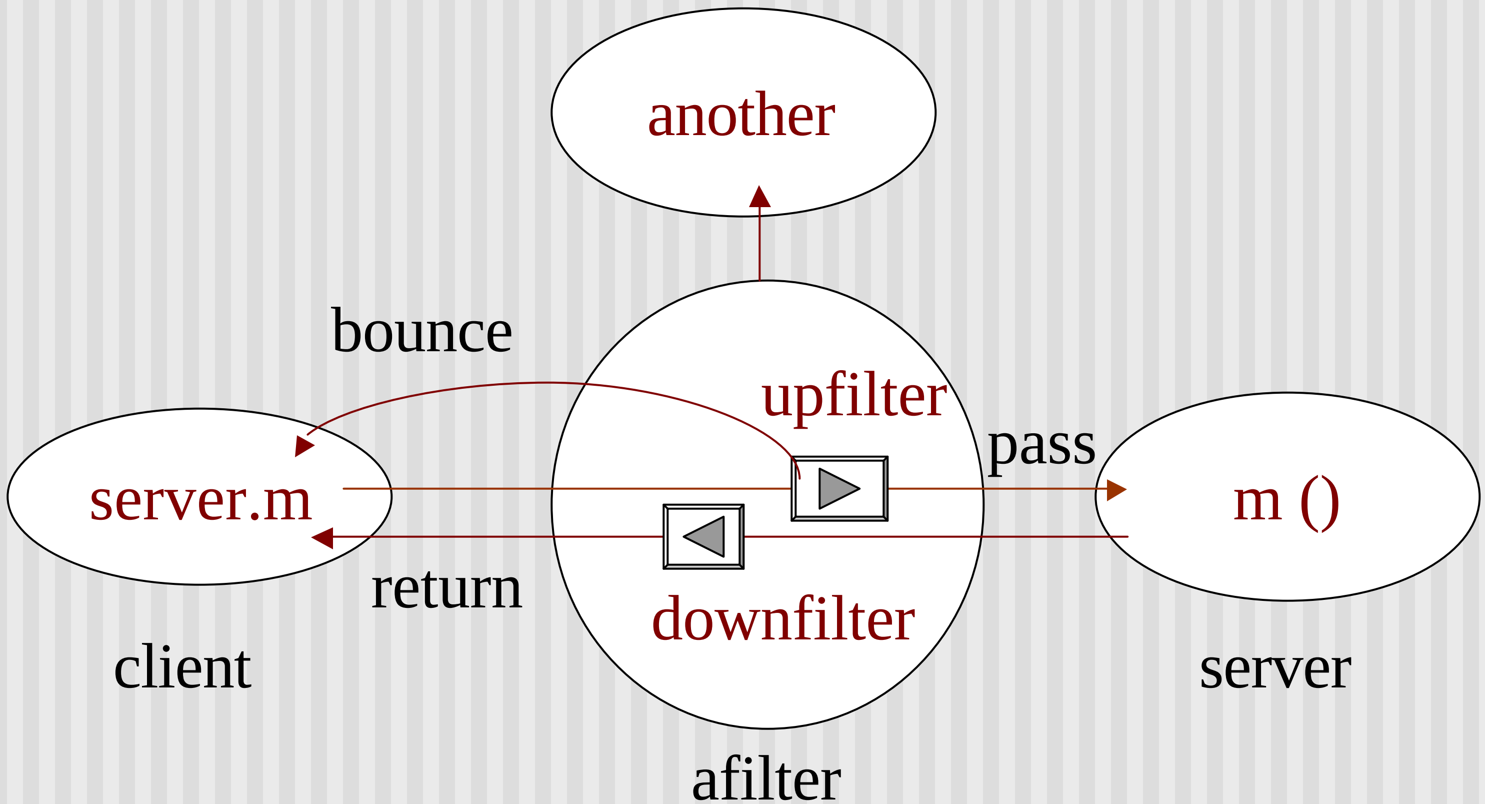
Summarily..

- There are many more variations of the themes discussed
- Most commercial integrations are of two kinds
  - Simple assertion statement
    - Terminates/or throws exception
  - Design by Contract – preconditions, postconditions and invariants
    - Throws exceptions
- Implementations in presence of Inheritance: yet to stabilize

Our Approach

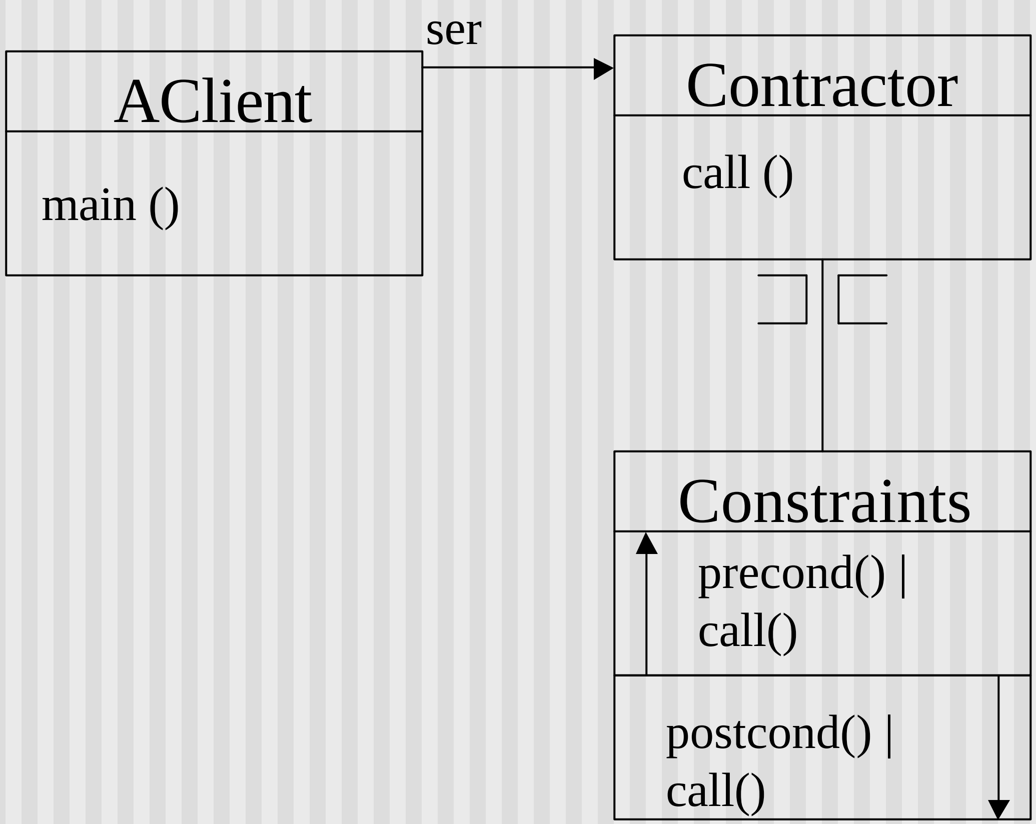
- Separate concerns of functional code from the assertion system
  - Transparent Pluggable Filter Objects
- Predefined control points
  - Interception points
- Modularity to assertion code
  - Filter objects are instances of classes
- Runtime control
  - Pluggable at runtime

Transparent Pluggable Filter Objects



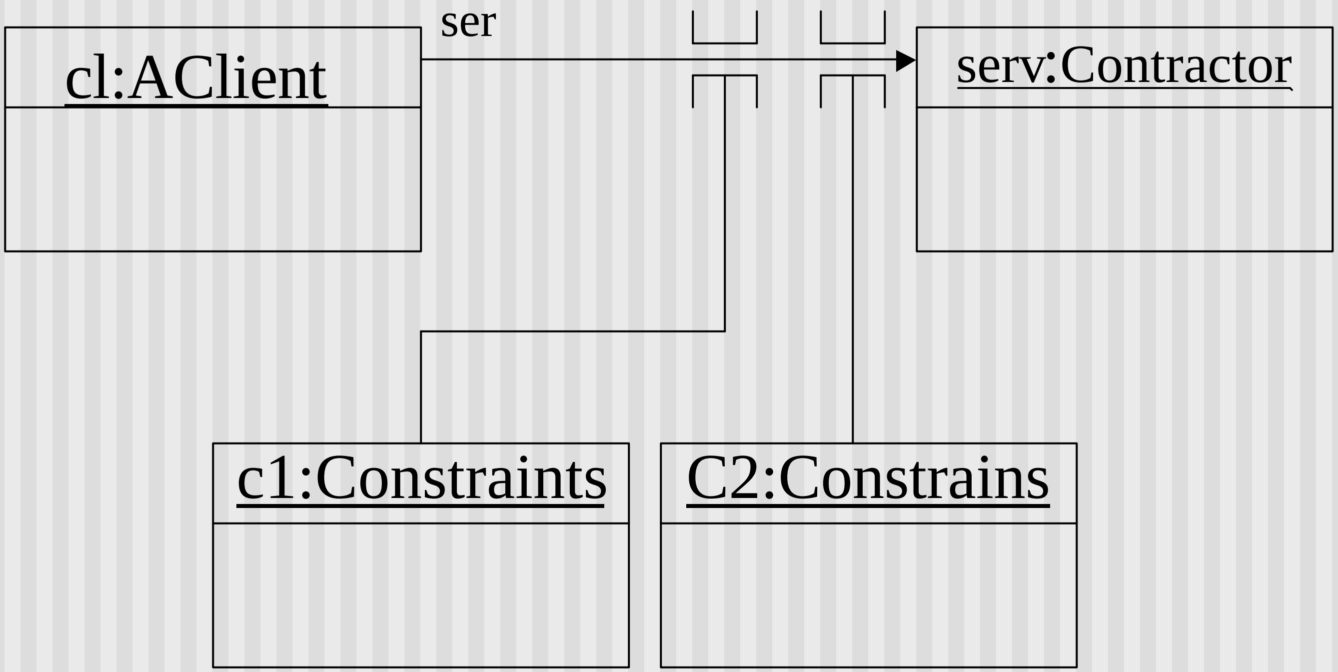
Interclass Relationship

Class Diagram

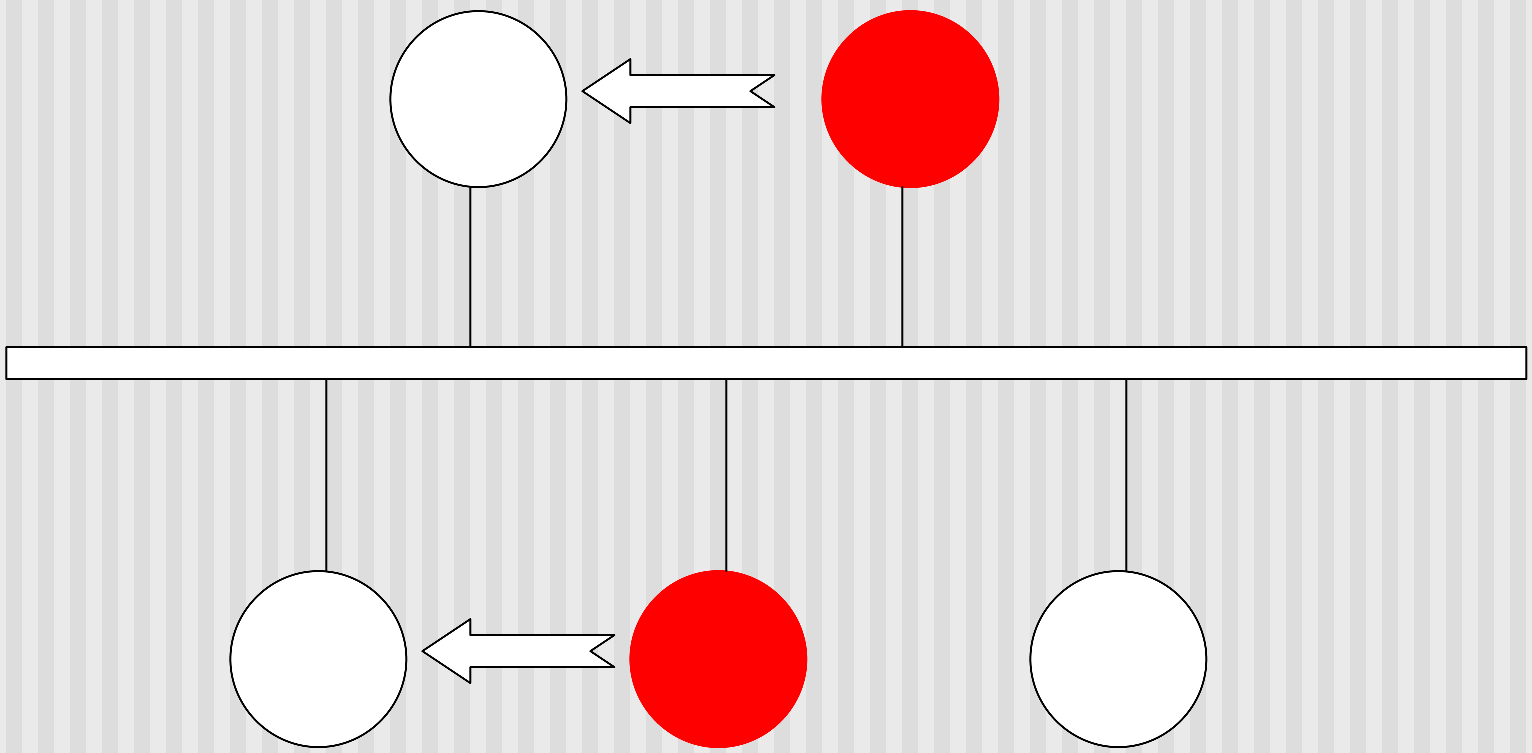


Filter Relationship

Object Diagram



A Distributed System Scenario:

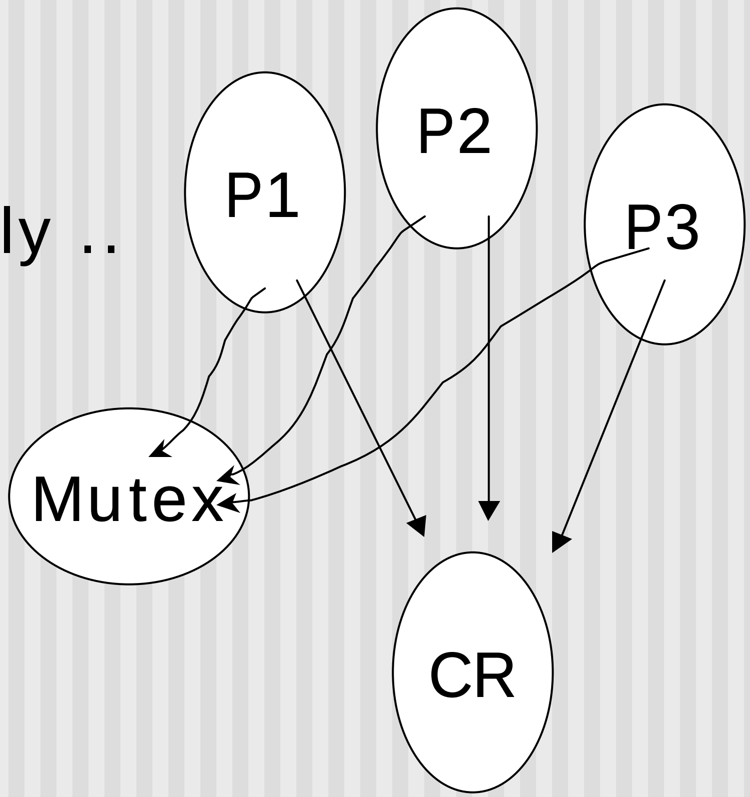


Objects on a CORBA Bus

A Critical Resource Component

```
Class CriticalResource {
   public void exwrite() {
        .. Functional code only ..
   };
   ...
}
```

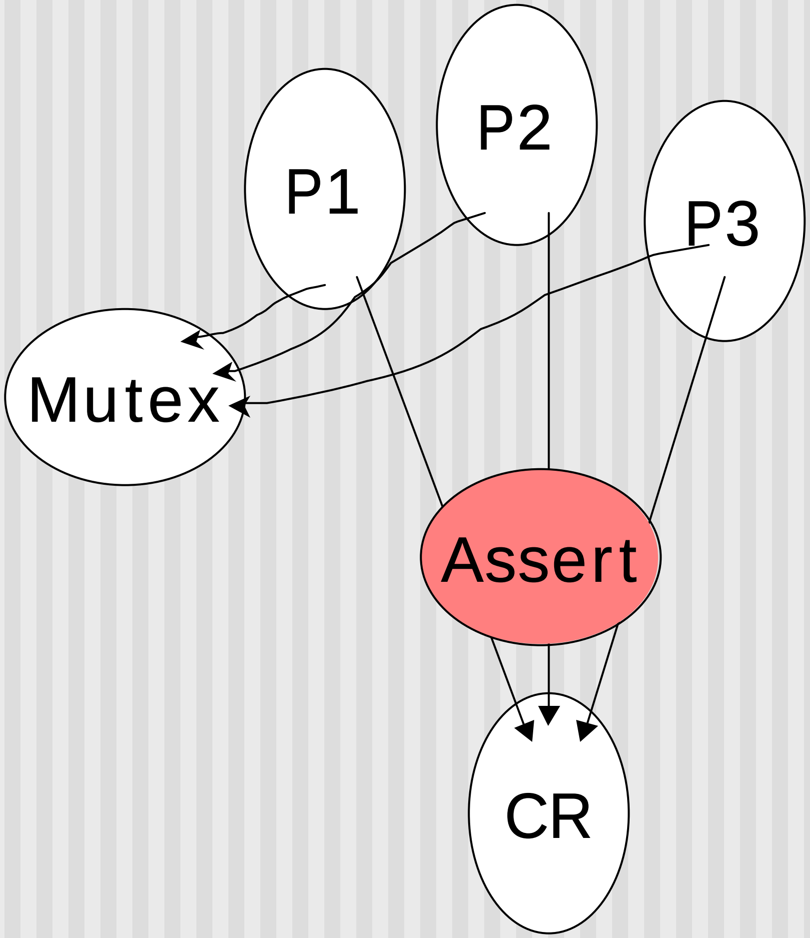
Assert mutually exclusive access to CR



Introducing a Transparent Filter Object

- The Assert filter traps calls to CR and asserts mutually exclusive access

- No need to change existing code.
- Assert is an independent component



A Critical Resource Filter Component

```
Class CRFilter : filter CriticalResource {
    boolean up;
    CRFilter () {up=true;}
    upfilter: void assertCS() filters exwrite() {
                    if (!up) FailAction();
    }
    upfilter: void update () filters exwrite() {
                    up = false;
    }
...
}
```

Inject Code

- Code that creates and injects transparent objects in an existing system

```
....
CRFilter crf = new CRFilter();
resource1.plug(crf);
....
resource1.unplug(crf);
```

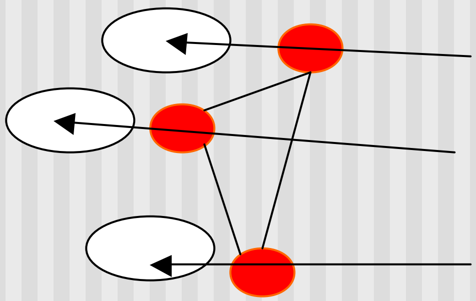
Implementing Design by contract through Assertion Objects

- Preconditions
  - As upfilters
    - On arguments
    - On server state*
- Postconditions
  - As downfilters
    - On return result
    - On server state*
- Invariants
  - On method boundaries
    - On messages
    - On server state*

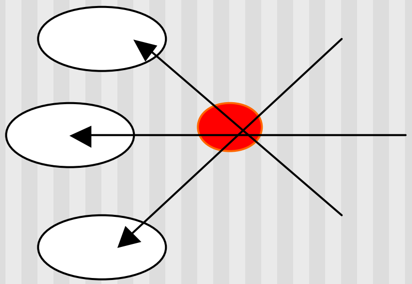
*access required

Collaboration, Sharing and Runtime Reconfiguration

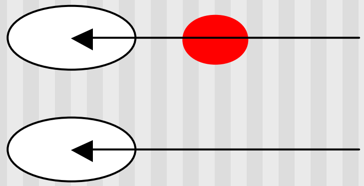
- Collaborating Assertions
  - Since they are full-fledged objects, collaboration is possible



- Shared Assertions
  - plugged to multiple servers



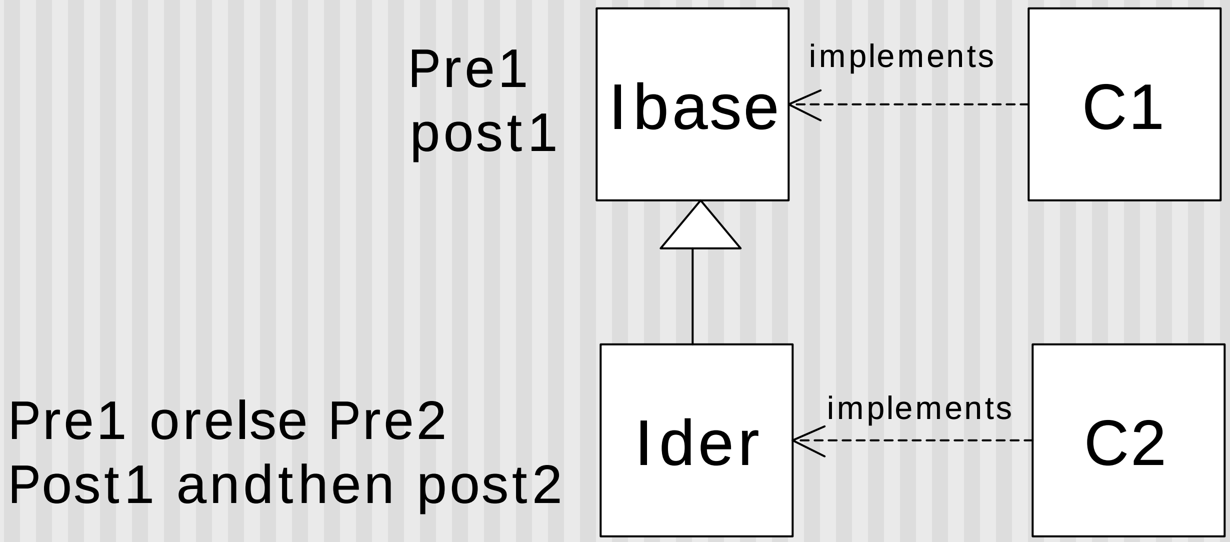
- Runtime configuration
  - Switch on and off



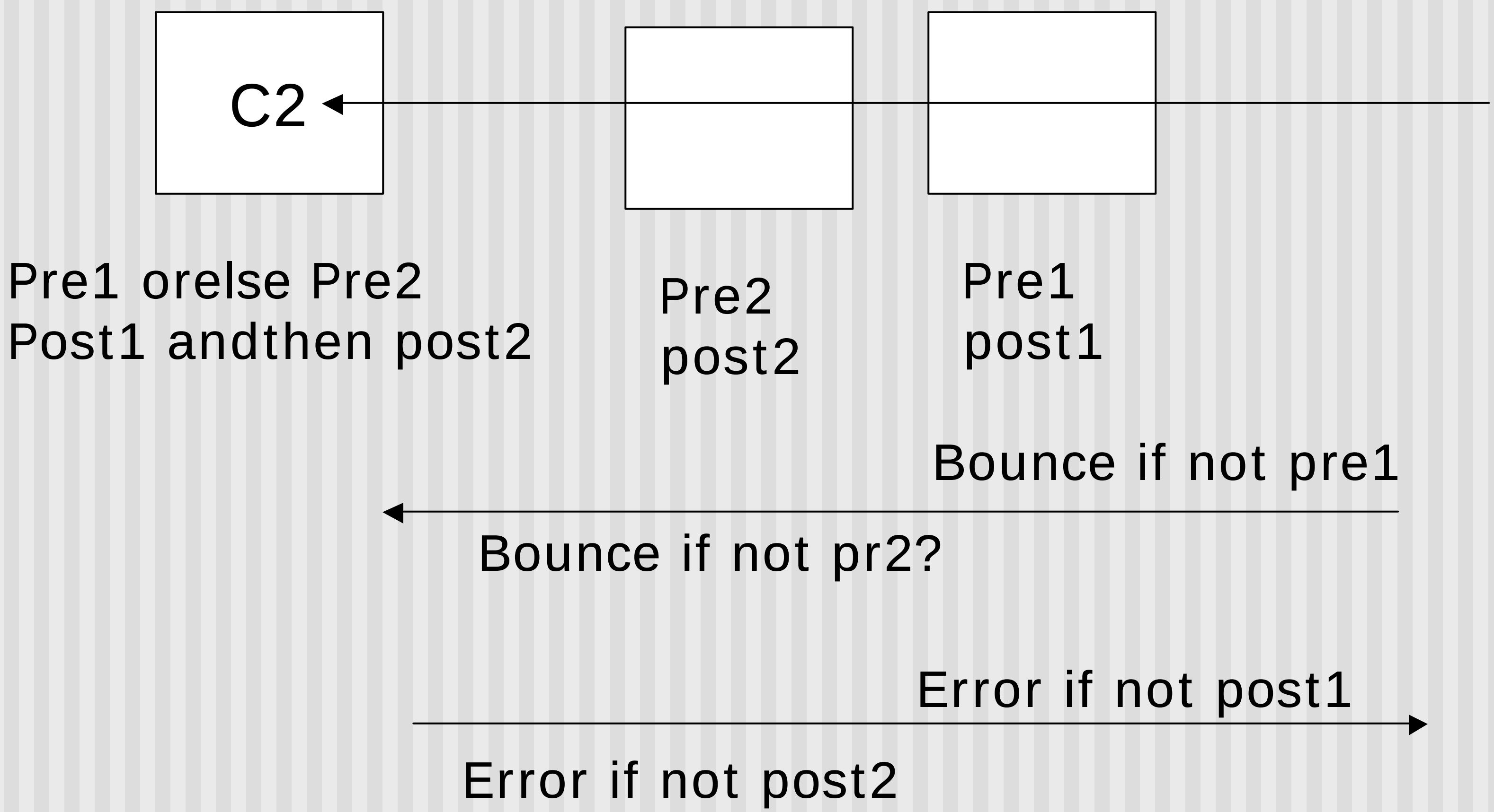
Beyond Assertions→ State Monitors

- Traditional assertion systems do not recommend assertions which keep state, in certain cases, such usage is eliminated
- With separation of assertion code from component's functional code, cause for interference is removed
- keep local state and act as state monitors

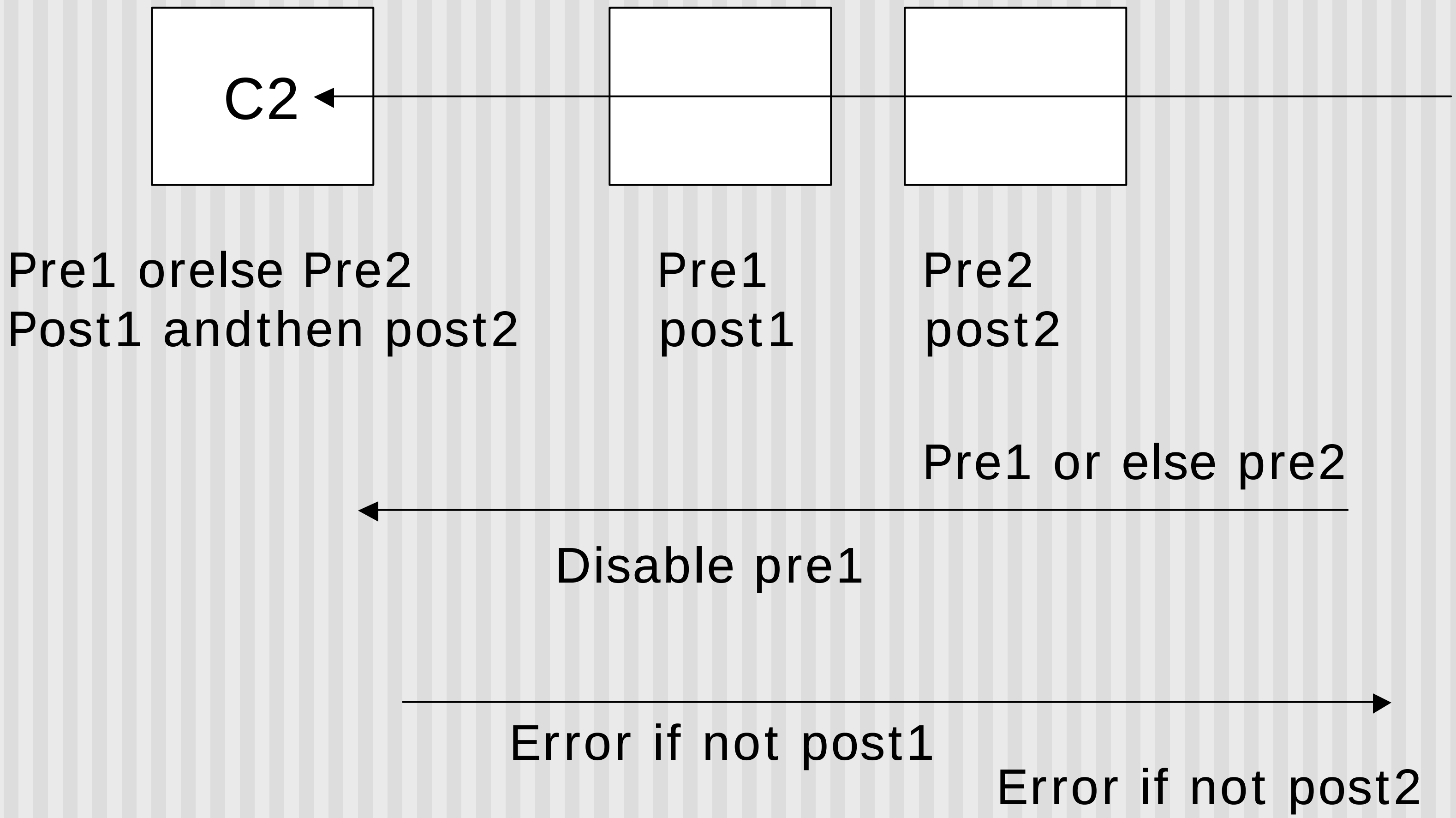
Handling Inheritance



Reusing Assertion Objects – Feature Interaction Problem



Reusing Assertion Objects – Solution



Publications related to this talk

- Design by contract for COM Components
  - Sonal Bhagat, Rushikesh K.Joshi, behavioral contracts for COM components, in proceedings of *information system technology and its applications (ISTA 2001), lecture notes in informatics (LNI) - proceedings, volume P-2,* ISBN 3-88579-331-8, pp. 45-51, June 2001.
- Pluggable Filter Objects in Distributed Systems
  - R.K. Joshi and Neeraj Agrawal, AspectJ based implementation of dynamically pluggable filter objects in distributed environment, proceedings of 2nd German workshop on AOSD, Feb 2002.
  - G. Srirami Reddy, Rushikesh K. Joshi, Filter Objects for Distributed Object Systems, *Journal of Object Oriented Programming*, vl. 13, No. 9, January 2001, pages: 12-17.
  - Pranav Nabar, Amit Padalkar, R.K. Joshi, Filter Object Framework for MICO, communicated
- Design and Implementation of Pluggable Assertions
  - Document in preparation.

References

- Tony Hoare, assertions: A personal perspective. Draft: June 6, 2001.
- David Rosenblum: A practical approach to programming with assertions, IEEE TSE, Jan. 1995.
- Bertrand Meyer, design by contract, in advances in object-oriented software engineering, prentice hall int. (UK) ltd., 1992.

Current Status

- C++ [SPE 97]
- JAVA [SPE review]
- MICO – user level [JOOP 2001]
- A Mechanism for COM [ISTA 2001]
- MICO – kernel level [new]
- AspectJ based implementation [AOSD2002]