Filter Objects: Programming Models and Applications

Rushikesh K. Joshi
Department of Computer Science & Engineering
Indian Institute of Technology Bombay

Abstractions for (product) aspect capture in the Popular OO Paradigms in Practice

- Classes
- objects
- member functions
- inheritance and dynamic binding
- visibility control
- interobject, interclass relations
- components, component composition

Product Concerns and their programming expressions

- Can separate concerns be expressed separately and be traceable eventually?

- If the answer is yes→
  - Independent development processes for the concerns
  - Independent abstractions (representations)
  - Proceed to integration mechanisms

- Limitations of popular OOP
  - Many cross-cutting concerns posed problem
    - Design reuse vs. implementation reuse
    - Transparencies for system/context variabilities
    - Generic concerns covering multiple abstractions
  - Rise of new paradigms: AOP extensions
    - Context relations, compositional filters, aspect oriented programming

A Static Solution

- Aspect specifications separate from base specifications
- Aspects weaved with bases by a Weaver
- Static (compile time) weaving
- Weaved code looses aspects → no traceability of aspects into first class runtime elements of the language

Limitations of aspects

- Implementation-centric approach
- Lacks Process
- Non-first class abstractions
- Contracts may get violated and encapsulation broken

Some other static approaches

- Overloading, Template classes to more powerful generic specification languages
  - Generic (e.g. XML based) specification applied to base code which is transformed

Some Dynamic Approaches

- Subclassing and Polymorphism
- Using Metalevel protocols: e.g. Smalltalk's metaclasses
- Reflection into PL implementations

Our approach:
Introduce First Class Communication Abstractions (Filter objects)

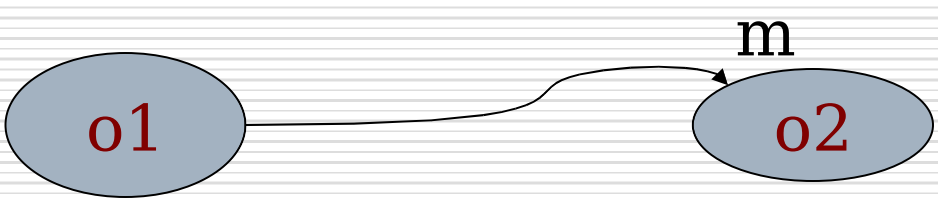
- Honor encapsulation, target messaging

- Communication Aspects are first class entities in base language: objects with member functions and local state

- Abstractions used for Specification are very similar to those available in the base language

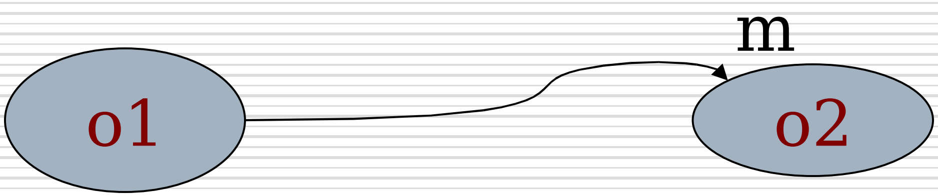
- Weaving is replaced by object level binding at runtime

An Interobject Communication Scenario



Entities(Objects/Components)
Message generation
Message flow
Message delivery
Method Dispatch
Response flow
Response delivery

Targeting messaging, leaving objects as they are



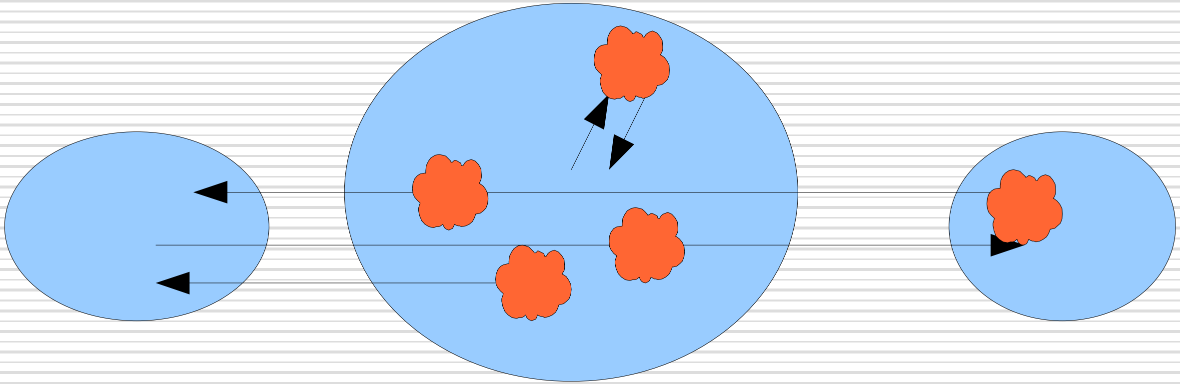
Entities (Objects/Components)
Message generation
Message flow
Message delivery
Methods
Response flow
Response delivery

Message Processing + Message Control Layer

- *Message Processing*
  - Determines response by the receiver once the message is dispatched to the receiver
  - Implementation of the component/object's contract
- *Message Control*
  - Activities on/over messages in transit
    - i.e. during to and fro information flow

Possible Message Paths/Moves

Primitives:
Process
Forward
Replace
Force
Delay
Bounce



Class level specification: An Example

```
Class Dictionary {
    ...
}

Class Cache: filter Dictionary {

    ....
}
```

Instance level pairing (not weaving)

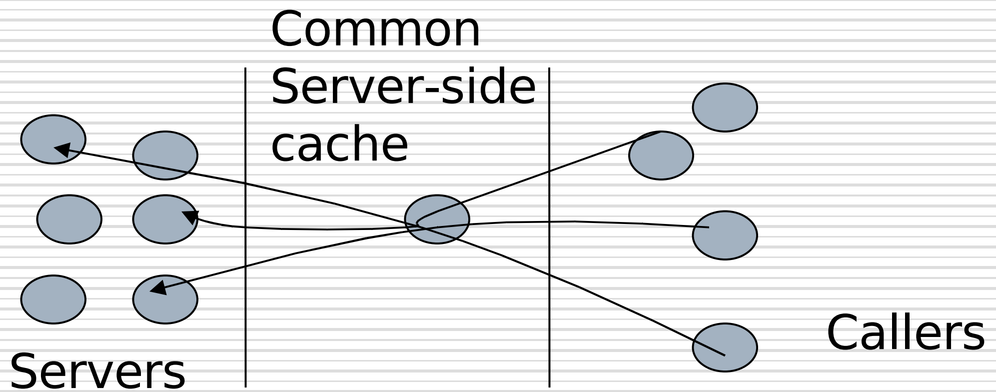
plug and *unplug*

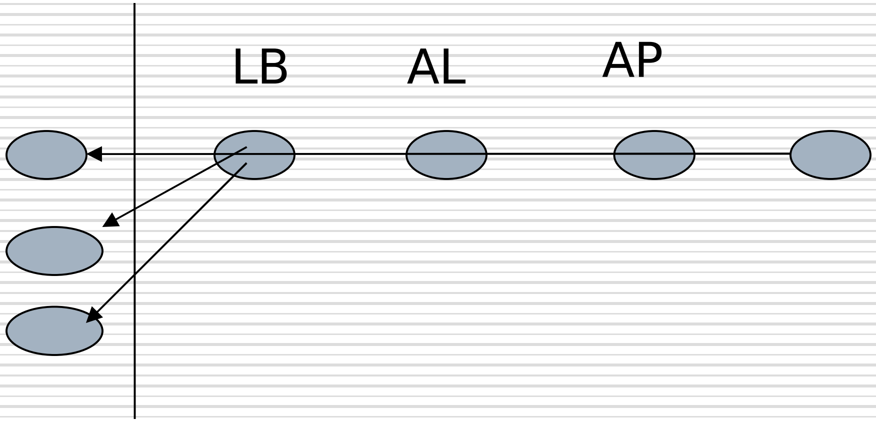
```
main ( ) {
Dictionary *d=..;
Cache *c=..;
      plug d c;
      ...
      unplug d;
}
```

First class representation in an OOPL

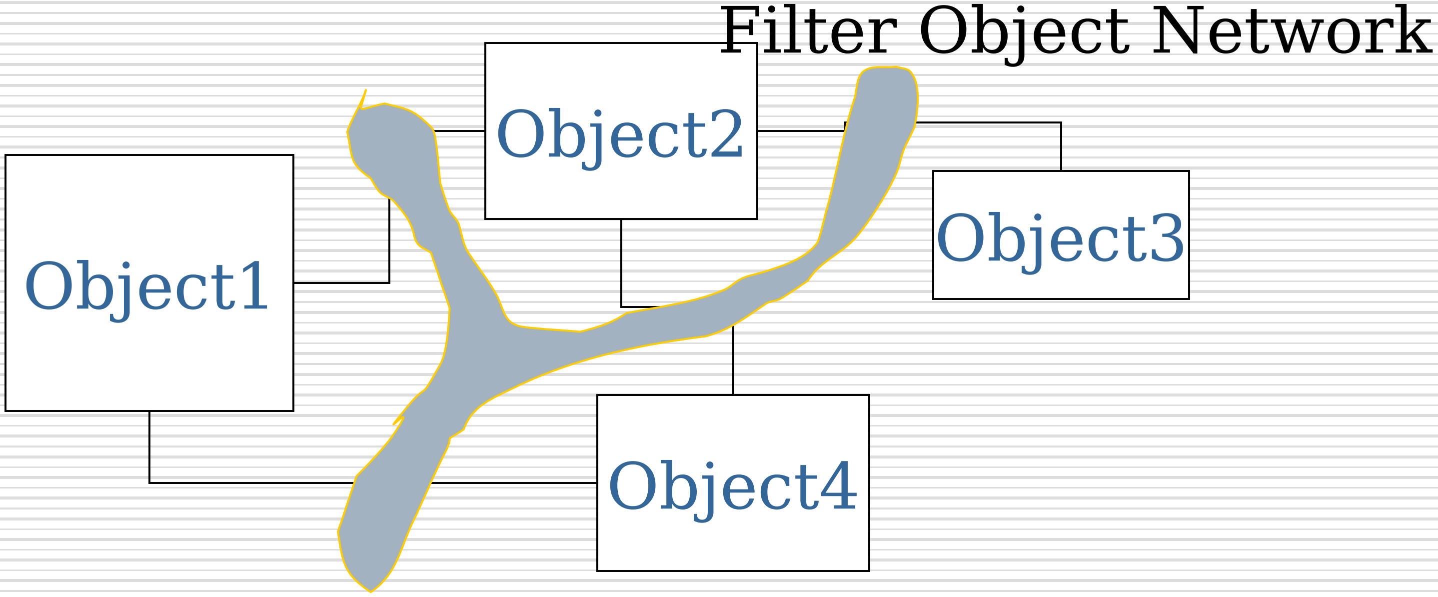
```
Class Dictionary {
    public: Meaning SearchWord( Word);
}
class Cache : filter Dictionary {
upfilter:
    Meaning SearchCache(Word) filters SearchWord;
downfilter:
    Meaning ReplaceCacheEntry (Meaning) filters SearchWord;
public:
    double hitRatio ( );
private:
        … implementation
}
```

Dynamic Grouping and Layering





Orthogonal Collaborative Frameworks: Crosscutting functionalities



Patterns at Messaging Layer

- Message replacement
- Receiver Replacement
- Routing, destination selection
- Repeater
- Message Content Replacement (value transformer)
- Decoration (logger)
- Message hold/delay and synchronize

Replacer

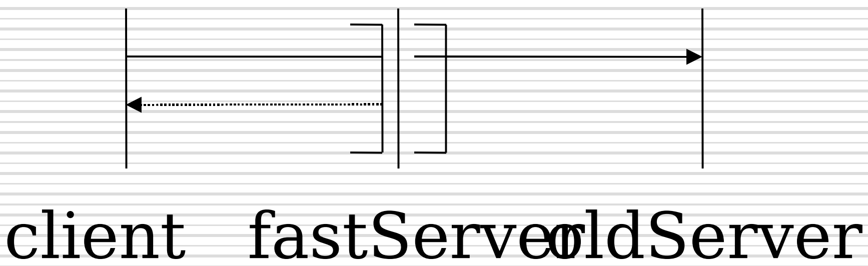
- A filter member function operates as a replacement function to its corresponding server member function

FastServer | oldServer =

filter interface:

    funcReplacer (in) upfilters oldServer :: func (in)

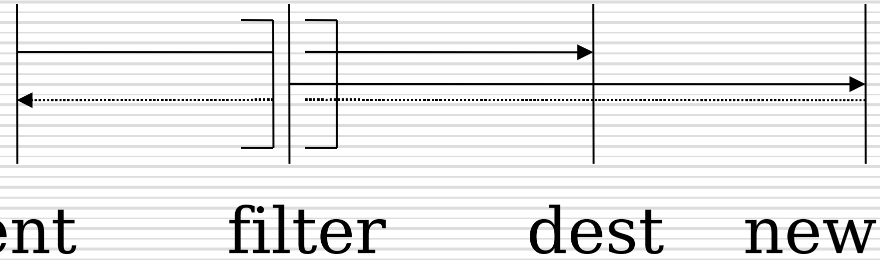
        = [v <-- self.func (in); bounce (v);]



Router

- A filter member function operates as a router function

balancer | searchEngine =
filter interface:
    searchRouter (item) upfilters SearchEngine ::search (item)
        = [newDest <-- self.nextDest();
            v<--newDest.search(item); bounce (v);]



Repeater

- A filter member function dispatches the filtered invocation to multiple servers

enrollFilter | centralEnroller =

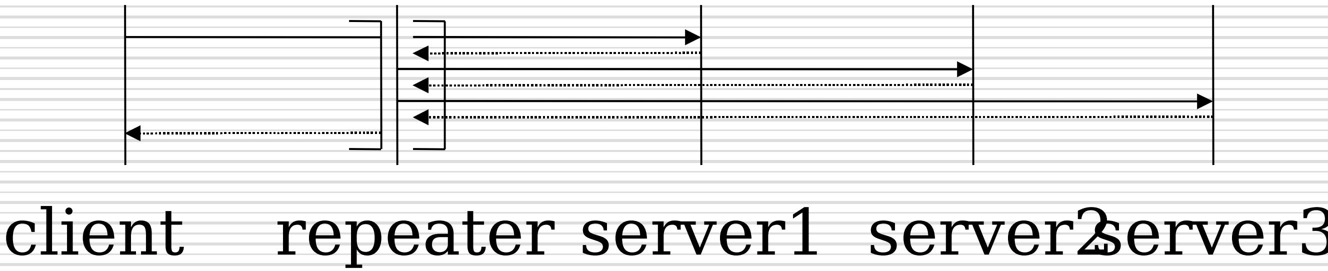
filter interface:

libEnroll (student) upfilters centralEnroller :: enroll (student)

= [if (student.dept == civil) civilLib-->enroll (student);

if (student.status == minor)minorBody-->
enroll(student);

pass;]



Two Models for Distributed Middlewares

- Need-to-filter principle: A server is declared as *Filterable Server*

```
interface Filterable {
attach (in Object filter)
detach ();
};
interface Server : Filterable {
service ();
}
```

- *Filter Object aware Middleware (e.g. MICO extensions)*

Dynamic Functional Evolution
(functional cross-cut)

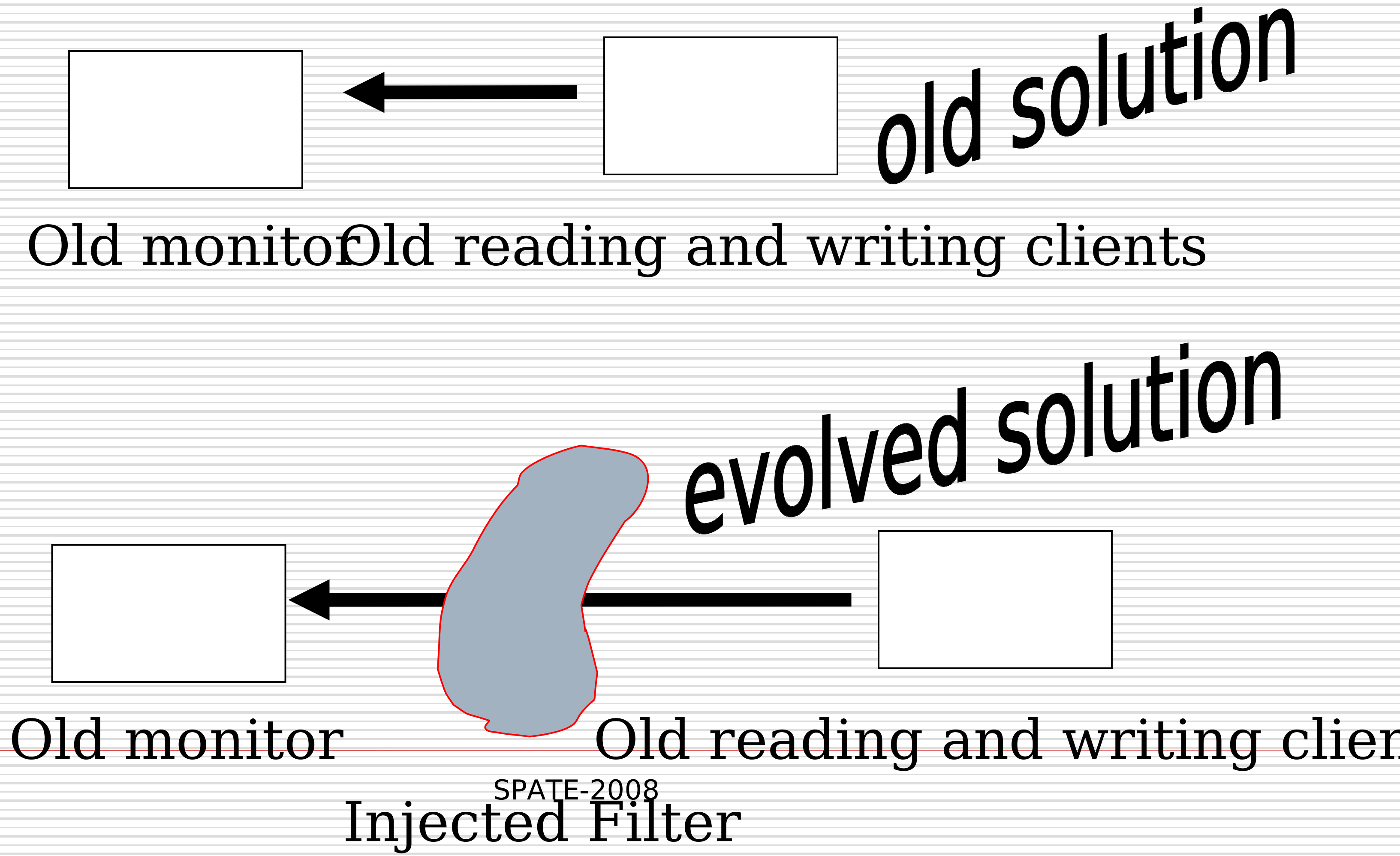
- A Readers and Writers Solution
  - (Hansen 1978)

```
process resource
s: int
proc StartRead   when s>0 : s++; end
proc EndRead     if s >1: s--; end
proc StartWrite  when s==1: s--; end
proc EndWrite    if s==0: s++; end
s=1;
```

Evolution Requirement

- Solve the same problem with additional constraint that further reader requests should be delayed as long as there are writers waiting or using the resource

The Approach



Evolution using Filter Process

```
process problemSolver: filter resource
www : int
upfilter:
     SW_Ufilter filters StartWrite
     SR_Ufilter filters StartRead
downfilter:
     EW_Dfilter filters EndWrite
proc SW_Ufilter:        www++; pass; end
proc EW_Dfilter:        www---; end
proc SR_Ufilter:        when www==0: pass; end
www=0;
```

Dining Philosophers with Deadlocks

```
process Fork [i:0..N-1];

s: int
owner: int
 proc Pickup (ph)  when $s = 1$ : $s=s-1 $; $onwer=ph$;  endproc
 proc PutDown (ph) if $s = 0$ : $s=s+1;$ $owner=none$;  endproc
s = 1; owner=none;

process Philosopher [i:0..N-1];
 proc PickForks()    Fork [i].pickup(i); Fork[(i+1) % N].pickup(i);  endproc
 proc Eat()  eat; endproc
 proc PutDownForks()  Fork[i].putdown(i);  Fork [(i+1)% N].putdown(i);  endproc
proc Think()  think; endproc
..... phil cycle ..
end process
```

Filter Process for Fork Processes

```
process ForkFilter[i:0..N-1] :  filter Fork[0..N-1];
upfilter proc PickFilter (p)  filters Pickup() if (i=p) Monitor.pickup (i); pass;
      endproc
downfilter proc PutFilter (p) filter PutDown() Monitor.putdown(i) endproc
end process

process Monitor ;
s[N]: int;
proc Pickup (p) when (s[p] AND s[(p+1)%N]) : s[p]=s[(p+1)%N]=0;
endproc
proc Putdown (f)  s[f]=1  endproc
for (i=0; i<N; i=i+1)  s[i]=1;
endprocess
```

Summary of Implementations and Models

1. Models for C++/COM components

3. TJF (Translator for Java Filters)

5. Middleware: MICO kernel extensions

   Filter aware middleware

- Implementations of Filter Objects in Distributed Systems over AspectJ+RMI
- Distributed Filter Processes (Unimplemented)
- Extensions to theory of objects, operational semantics and a light weight language with an interpreter of sigmaF calculus

Ongoing work

- Calculus, Semantics (Abadi/Cardelli style) and implementations
- Applications, Notations, Methods and Patterns
- Component Adapters