

# *Aspects, Wrappers and Events*

**Robert E. Filman**

**Research Institute for Advanced Computer  
Science**

**NASA Ames Research Center**

**[rfilman@mail.arc.nasa.gov](mailto:rfilman@mail.arc.nasa.gov)**

# *Talk Overview*

- ❖ **Chronological Perspective**
- ❖ **Software development**
- ❖ **Object Infrastructure Framework (OIF)**
  - A system developed to simplify building distributed applications by allowing independent implementation of multiple concerns
- ❖ **Aspect-Oriented Programming (AOP)**
  - Mechanisms for independent implementation of multiple concerns
- ❖ **Quantification over Events**
  - Current work on “most general” AOP technologies

# *Distributed Computing*

- ❖ **Developing distributed systems is difficult**
  - **Hard to achieve systems with systematic properties, e.g.:**
    - **Reliability**
    - **Security**
    - **Manageability**
    - **Quality of Service**
    - **Scalability**
  - **Distribution is complex**
    - **Concurrency is complicated**
    - **Distributed algorithms are difficult to implement**
    - **Every policy must be realized in every component**
    - **Existing frameworks are difficult to use**

# *Requirements*

- ❖ ***Functional* requirements**
  - Realizable by writing code in a specific place
- ❖ **Systematic requirements**
  - Requirements achievable by hygienic behavior throughout a system
- ❖ ***Combinatorial* requirements**
  - Requirements that emerge as a measurable property of the system as a whole
- ❖ ***Aesthetic* requirements**
  - Requirements that require human judgment to decide if they're satisfied

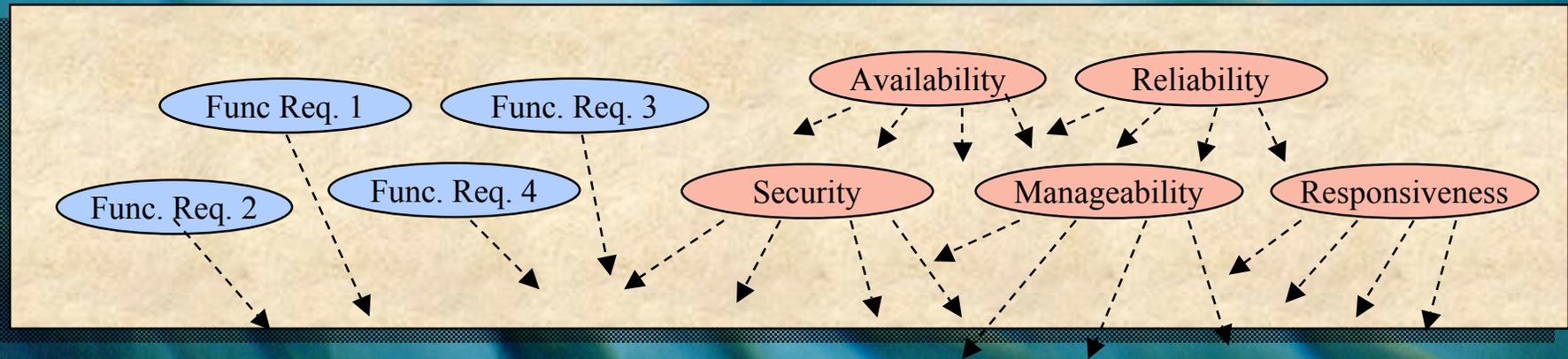
# *Terminology*

- ❖ **Concern**
  - Something one cares about in software development that is realized in code
  - Crosscutting concern
    - A concern whose code intermixes with other behavior in a system
    - Crosscutting is a function of organization and environment
- ❖ **Policy**
  - A “way of doing things” that is to be realized throughout a system
- ❖ **Ility**
  - A desirably overall property of a software system
- ❖ **Systematic behavior**
  - A behavior seen throughout a system
- ❖ **Non-functional requirement**
  - A requirement for a system that is not localized to a particular point in the system
- ❖ **Aspect**
  - The code that realizes a concern

# Functional and Non-Functional Requirements

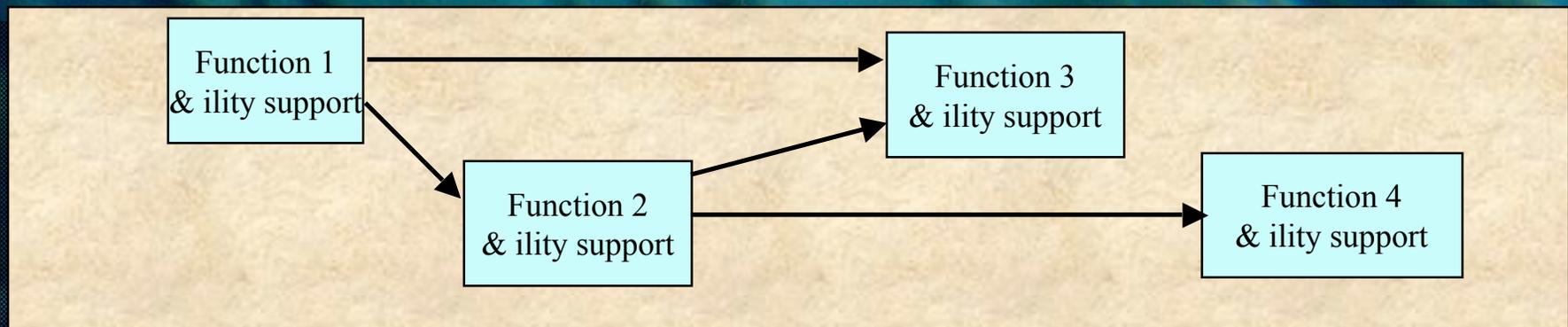
**Functional:**

**Non-functional (ility):**

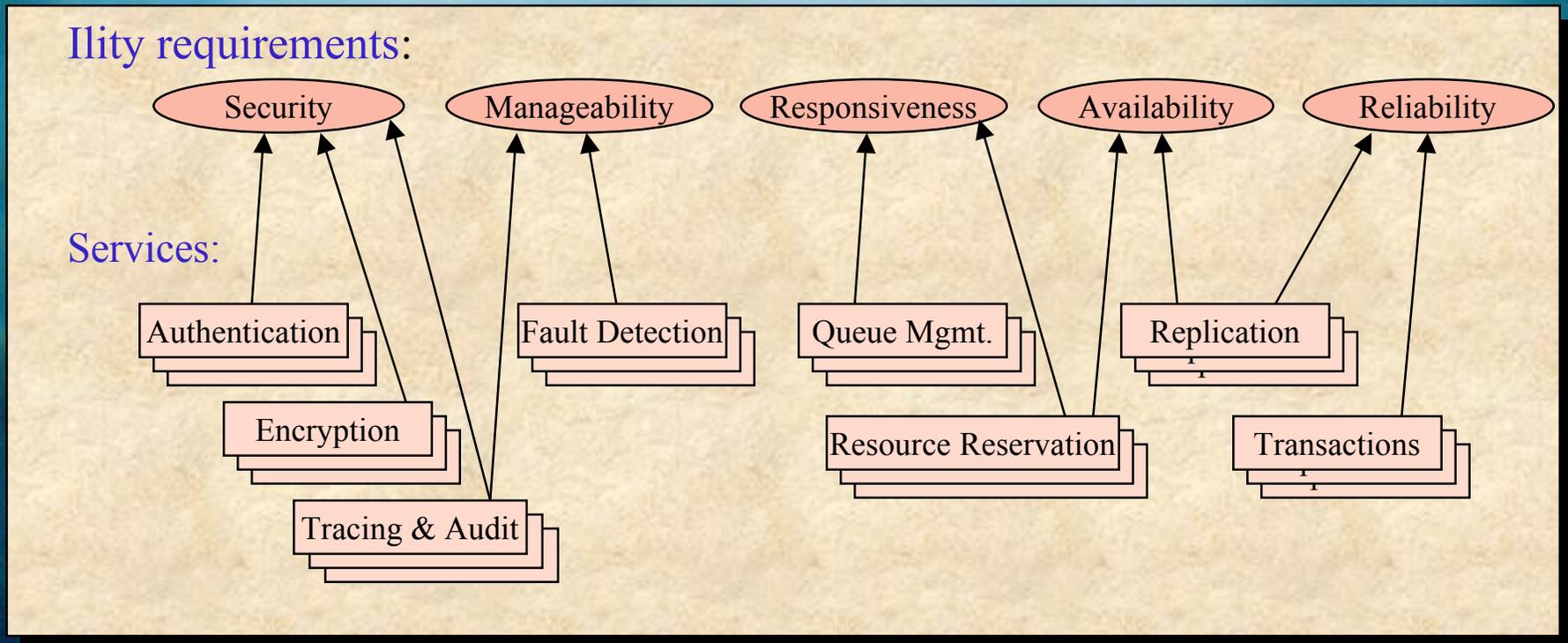


**Functional requirements map to specific components**

**ility requirements map almost everywhere**



# Services and QoS



- ❖ QoS requirements are implemented by combinations of service algorithms.
- ❖ Supporting QoS involves a complex selection from sets of alternative service algorithms.
- ❖ The services must be invoked pervasively.

# *The Problem: Managing the Service Space*

- ❖ **Compatible support for quality requirements (security, consistency, responsiveness, etc.) is a key component integration problem**
- ❖ **Quality requirements impose difficult upgrade requirements on component subsystems**
  - **Algorithms that support quality requirements are usually intertwined with the subsystem functional logic.**
  - **Separately developed subsystems may have chosen different algorithms (for encryption, transactions, etc.)**

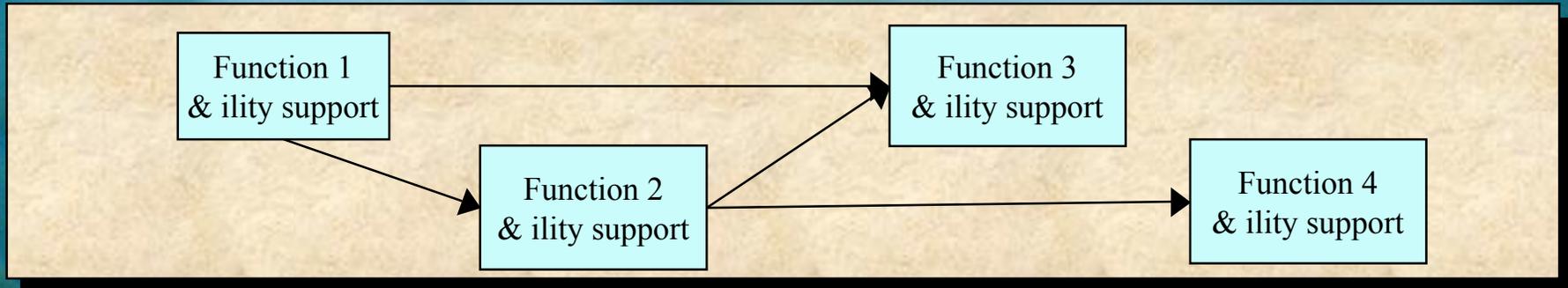
# *Research Hypothesis*

- ❖ **ilities can be achieved by inserting services into the communication path between functional components**
  - On both sides of the communication divide
- ❖ **Frameworks that automate service insertion can systematically achieve non-functional requirements**
  - Object Infrastructure Framework (OIF)

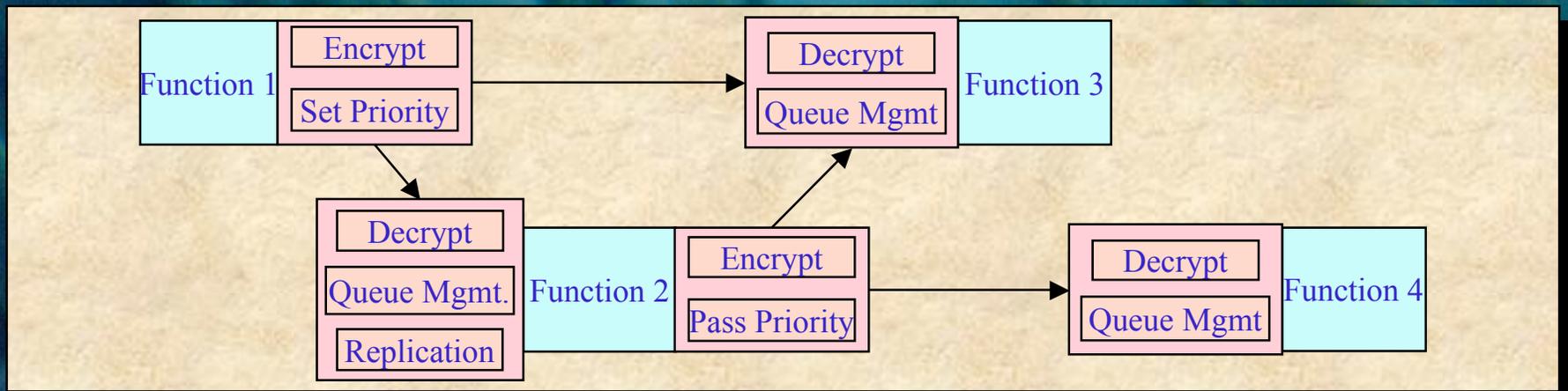


# Architecture with Services in Component Communications

Traditional designs mix ility support within functional components:



Separate service functionality from functional logic by inserting the services into the communications paths among components.



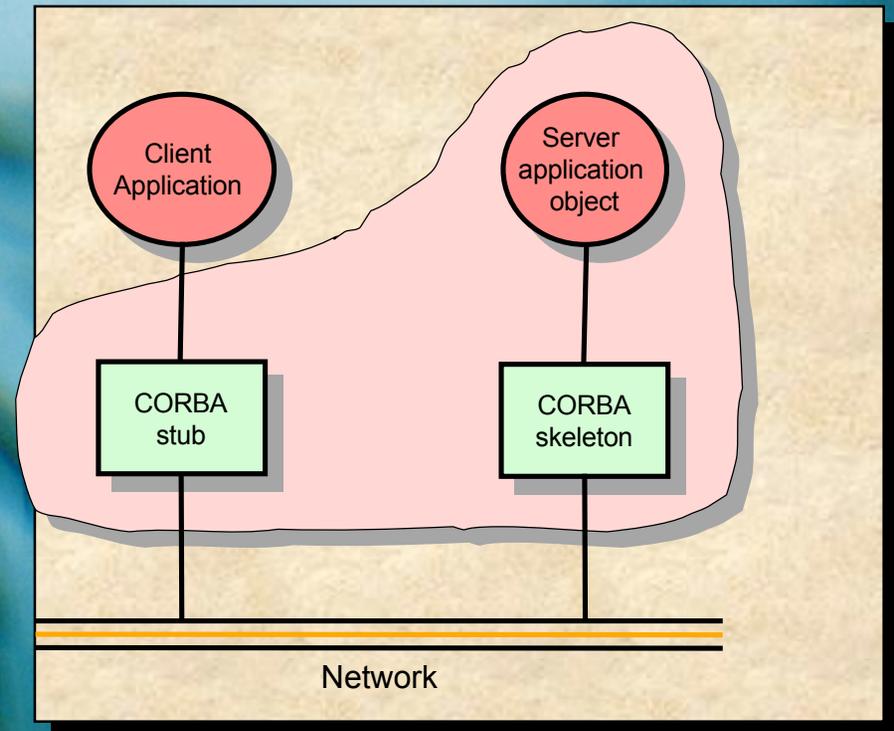
# *Distributed Object Technology*

- ❖ **Many kinds of approaches:**
  - Socket based
  - Message based.
  - Remote Procedure Call
  - Object based.
- ❖ **Distributed Object Technology allows for OO applications to be implemented using *some* objects that do not reside in the same address space (e.g.. machine).**
  - Key semantic of DOT is providing “location transparency”.
- ❖ **CORBA: Objects provide services described in an Interface Definition Language (IDL)**
  - CORBA allows object-oriented applications to be written in multiple languages.

# Using Stubs and Skeletons as Proxies

A client application makes a method invocation on the stub.

The Stub “implements” the IDL defined interface by being a *proxy* for the actual implementation object.

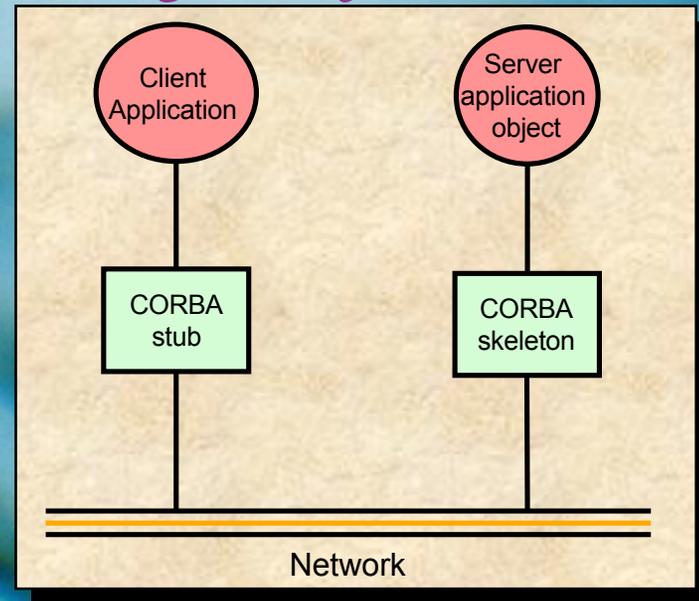


Similarly, the Skeleton acts as a *proxy* for the client application (from the implementation’s perspective).

The implementation object needs to have the actual code for each method defined the IDL interface.

# *What do Proxies do for you?*

The stub “implements” the IDL defined interface. The operation’s arguments are put into a request object, marshaled and passed over the network to the server machine.



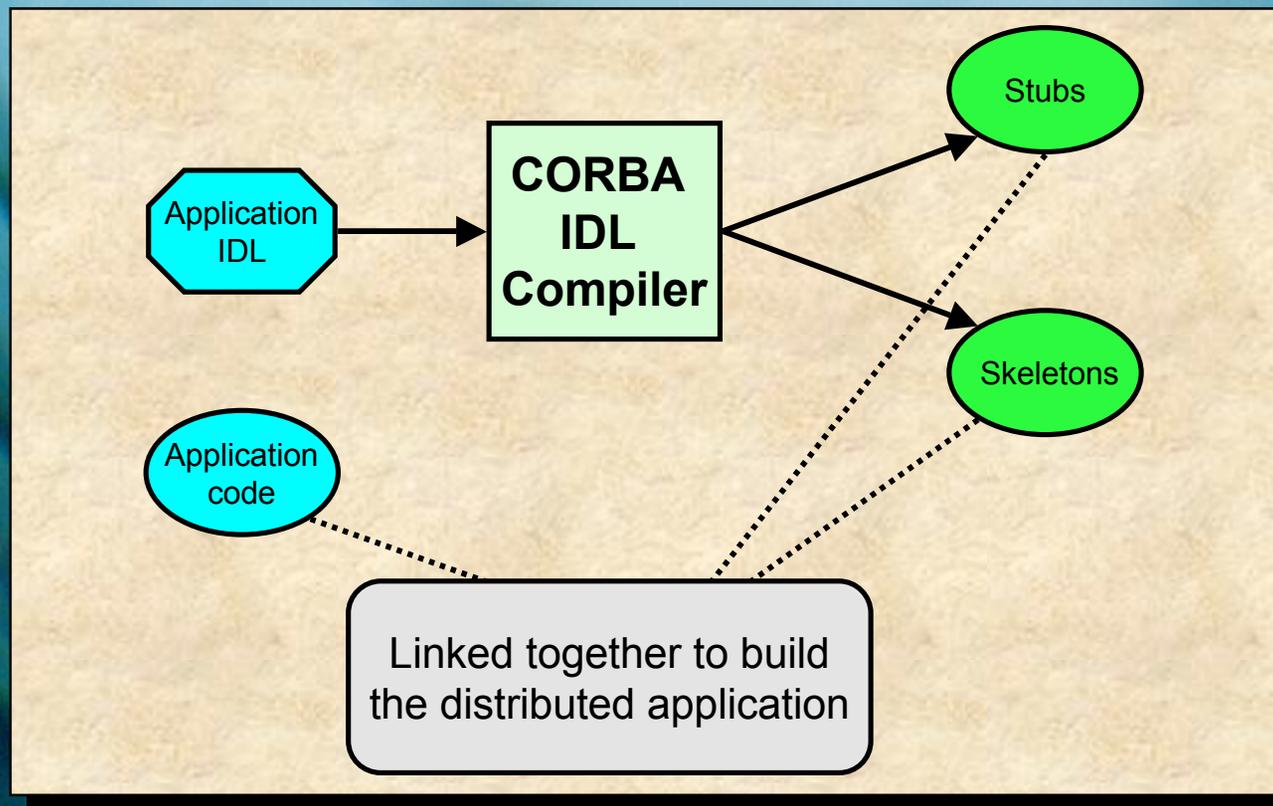
**Marshaling handles all of the issues relating to transmission and translation of the various data types.**

The ORB on the server machine demarshals the client’s request, and invokes the skeleton. The skeleton calls the appropriate implementation object method. The process is applied in reverse to the return value.

# *IDL Example*

```
module Bank {  
  
    exception overdraft {string msg;};  
  
    interface Account {  
        float balance();  
        float withdrawal(in float amount)  
            raises( overdraft );  
        oneway void deposit(in float amount);  
    };  
  
    interface AccountManager {  
        Account open(in string name);  
    };  
};
```

# *Compiling Stubs and Skeletons*



**The IDL compiler produces both client-side stubs and server side skeletons.**

# *Distributed Object Proxies*

- ❖ **What they do:**
  - Provide object location transparency
  - Hide details of communication protocols
- ❖ **What they do *not* do:**
  - Handle partial failures (reliability).
  - Security related issues.
  - Quality of service issues.
  - ...

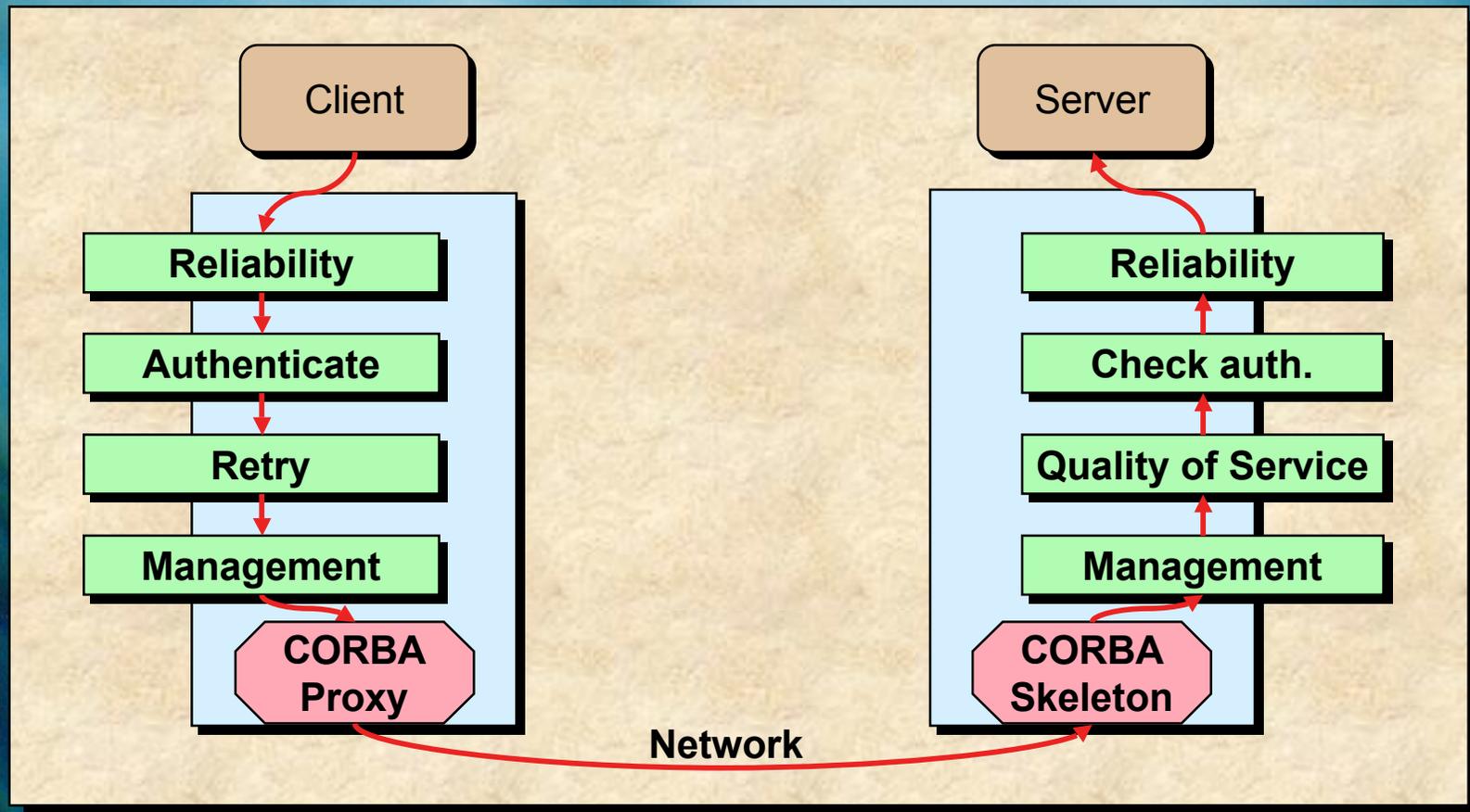
*OIF Challenge: Can we leverage the “proxy” design to address these missing characteristics?*



## *Key OIF Ideas*

- ❖ **Injecting behavior** on the communication paths between components
  - Injectors are discrete, uniform objects
  - Injectors are by object/method
  - Injectors are dynamically configurable
- ❖ **Annotated communications** allow injected services to pass parameters to service peers (e.g., message priority, user-id, tracing status)
- ❖ **Thread contexts** preserve annotations through calls
- ❖ **Pragma**: High-level specification language for describing desired injections

# Configurable Proxies



- ↪ OIF's injectors can operate in pairs (e.g., encrypt/decrypt; request authentication/authenticate) or singly (e.g., retry on failure; log results)
- ↪ Configuration is by proxy/method instance
- ↪ Configuration is dynamic

# Injector Invocation

## Client Side

Client

Log  
before after

Authenticate  
before after

Encrypt  
before after

Marshal

## Server Side

Server

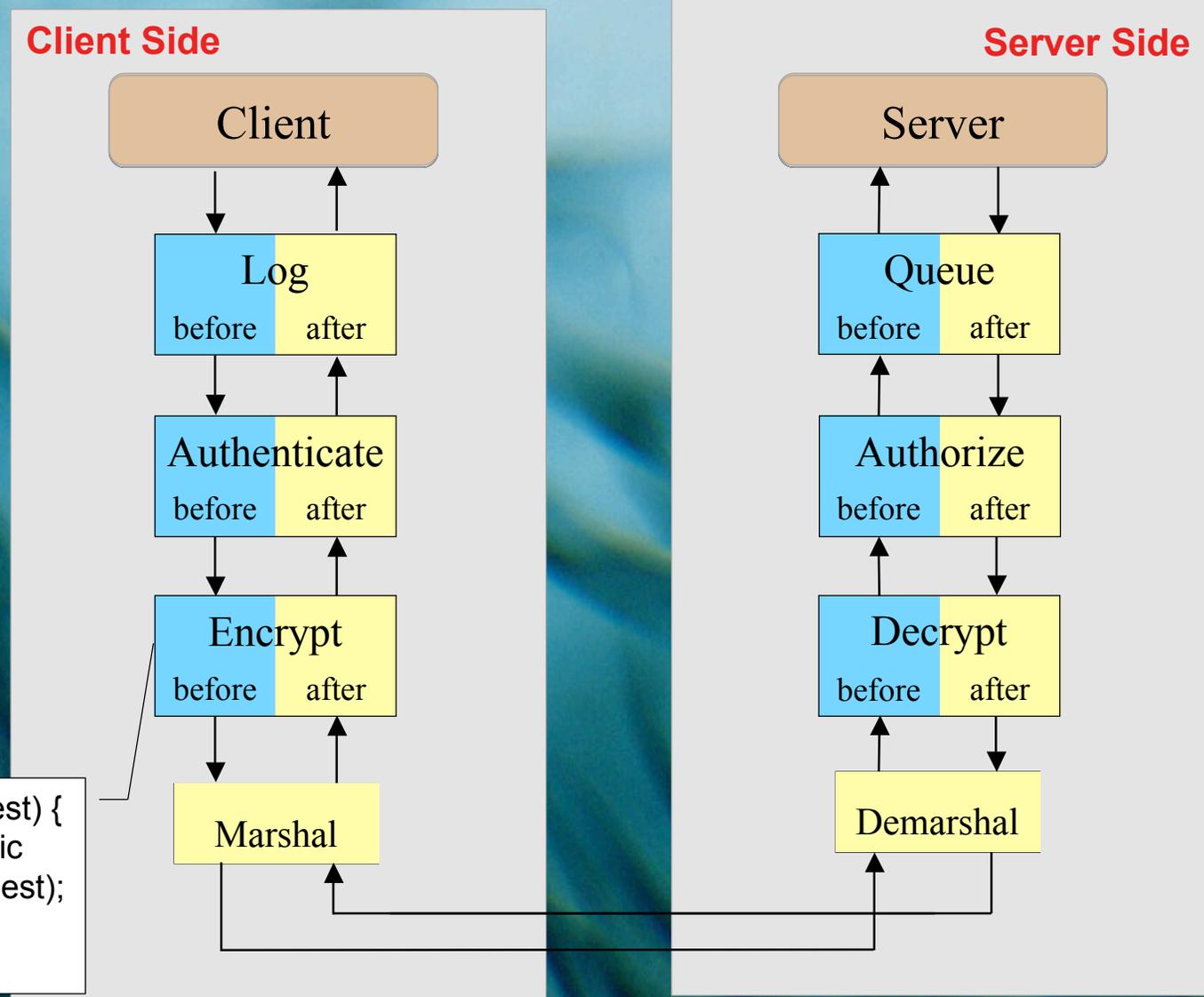
Queue  
before after

Authorize  
before after

Decrypt  
before after

Demarshal

```
injector.exec(request) {  
  ... before logic  
  doNext (request);  
  ... after logic  
  return;  
}
```



# *Annotations*

## ❖ **Problem:**

- Injectors need to communicate among themselves
  - e.g., passing the user authentication and session information

## ❖ **Solution:**

- Add additional meta-information, *annotations*, to communications
  - Reify communications
  - Annotations are name-value pairs
    - Names are strings
    - Values are “any”s
- Injectors can read and write annotations.

# *Thread Contexts*

## ❖ **Problem:**

- The application needs to communicate with the injectors
  - e.g., setting the priority of a request
- Don't want to change the client interface to the IDL
  - Can't add additional arguments to function calls

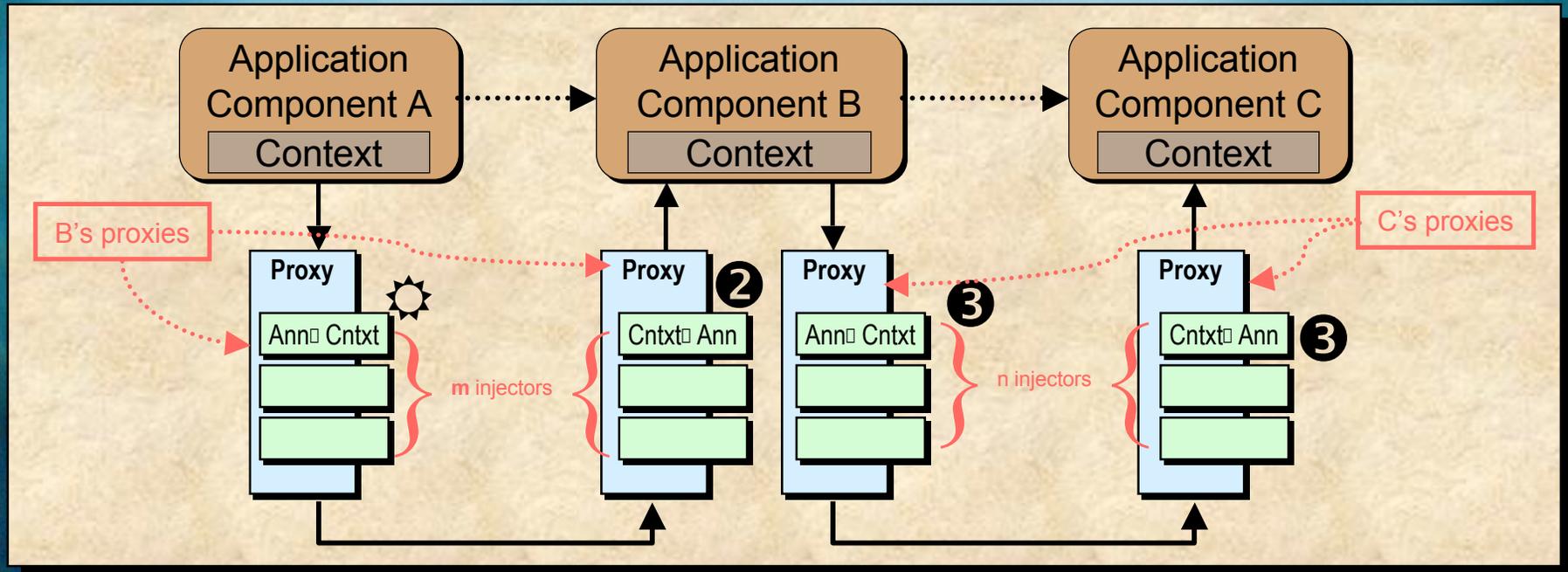
## ❖ **Solution:**

- Provide annotations for user threads---the *thread context*
- Copy information between request annotations and thread contexts
  - Declarations control which information is copied at each juncture

## ❖ **Benefit:**

- Contextual information propagates through a series of calls

# Propagating Annotations



1. When object *A* makes a call on method *m* in Object *B*, it's thread context is copied over into the annotation of the request.
2. After creating the thread to serve *A*'s request, the annotations of the request are copied to that thread's context
3. *B* calls method *n* on *C*. This process is repeated for *B*'s calls when handling that request. Thus, an annotation (e.g., *priority*) set in *A* is carried over through *B* to *C*.

# *Injector Features*

- ❖ **Ability to access/mutate method arguments and return value.**
- ❖ **Ability to pass meta-data (property sets) between themselves to coordinate their behavior.**
- ❖ **Access to CORBA's DII and DSI services**
  - **Access/modify function arguments, return value**
  - **Change "target" of request (load balancing, reliability)**
- ❖ **Fully capable code module.**
  - **Can be multi-threaded.**
  - **Can access other objects/services**
  - **Throw/catch exceptions**

# *Injector-enabled services*

- ❖ Caching of static object attributes reduces repetitious remote requests and enables “delayed call by value.”
- ❖ Serialization of arriving requests reduces cognitive load on application developer.
- ❖ Reified requests allow reasoning about priorities
- ❖ Targets allow application components to invoke logical destinations rather than a specific object (so injected services can do load-balancing, replication, transaction processing, redirection, etc.)
- ❖ Futures enable asynchronous interaction between application components while writing synchronous code.

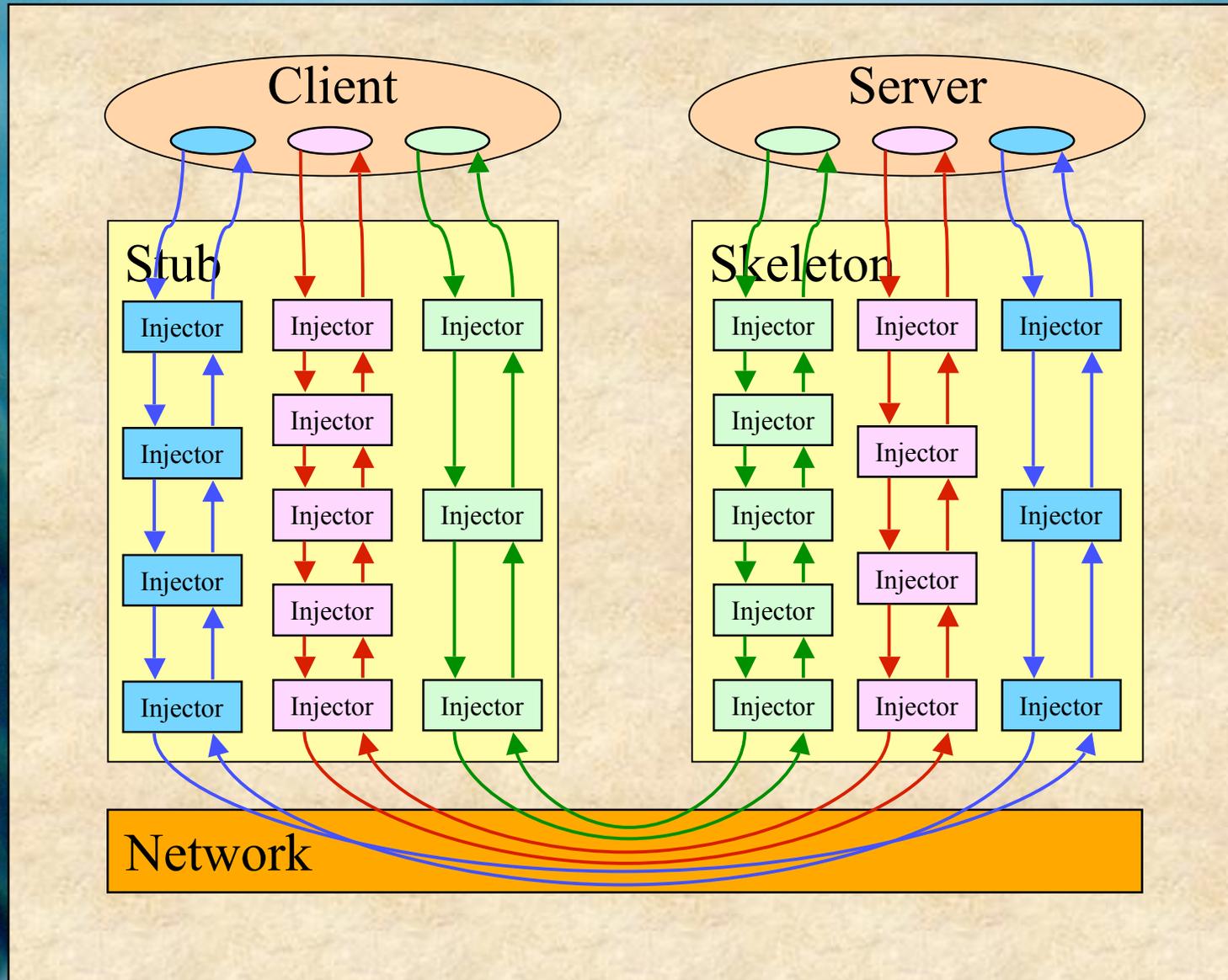
# *Reality*

**Illities must be grounded in the reality of invoking actual services**

**Saying you want security doesn't cause security to happen.**

- ❖ **Rather, you have to decide that you've got security if you**
  - **Encrypt all communications using { 64|128|3 } bit { DES | RSA | ROT-13 }**
  - **Check the user's { password | fingerprints | DNA } for { every | occasional } access to { all | only sensitive } methods**
  - **Recognize intrusions that { come from strange sites | try a series of passwords | ask too many questions }**
  - **Keep track of privileges by { proximity | job function | dynamic agreements }**
- ❖ **Need to have (implementations) of the algorithms**
- ❖ **Need to know where which algorithms are to be applied in which circumstances**

# Injecting Services



# *Pragma*

## OIF's Quantification Language

### ❖ Problem:

#### – Locally:

- Arranging for the appropriate injectors to be on the appropriate methods in the right order for each proxy
- Precluding incompatible injectors

#### – Globally:

- Achieving ilities

### ❖ Solution:

#### – Pragma: A high-level, declarative specification language for defining

- Ilities
- Ways to achieve ilities (i.e. which injectors + parameters to run to get that ility)
- The mapping for each ility to the methods of the application objects

#### – Pragma compiler:

- Takes declarative specification and compiles Java injector initializations

# *Pragma Concepts*

- ❖ **Ility:** Qualities an application is to have
  - (reliability, security, ...)
- ❖ **Action:** A way to achieve an ility.
  - “HighSecurity” achieves security by encrypting on the client last, decrypting on the server first, access control on both, ...
    - Can be required or allowed
- ❖ **Location:** Where an action is applied
  - On all methods of class C, on all methods named foo, ...
- ❖ **Context variable:** declares annotations
  - Priority, user-identification, due-date, electronic-wallet, ...
- ❖ **Command:** Commands tie ilities to locations and actions.
  - Use “HighSecurity” in class C
- ❖ **Policy:** A collection of Pragma statements
  - Mechanisms for successive refinement of policies through an organization

# Pragma Example

This is policy vendoom.

These are namespace imports.

Vendoom uses five ilities. For each ility, class and method, there (may) be more than one way to achieve that ility.

Var declares annotations, their types, default values and when they're copied to thread contexts.

For each ility, we can declare a mapping from a location (on *method* in *class*) to how that ility is to be achieved.

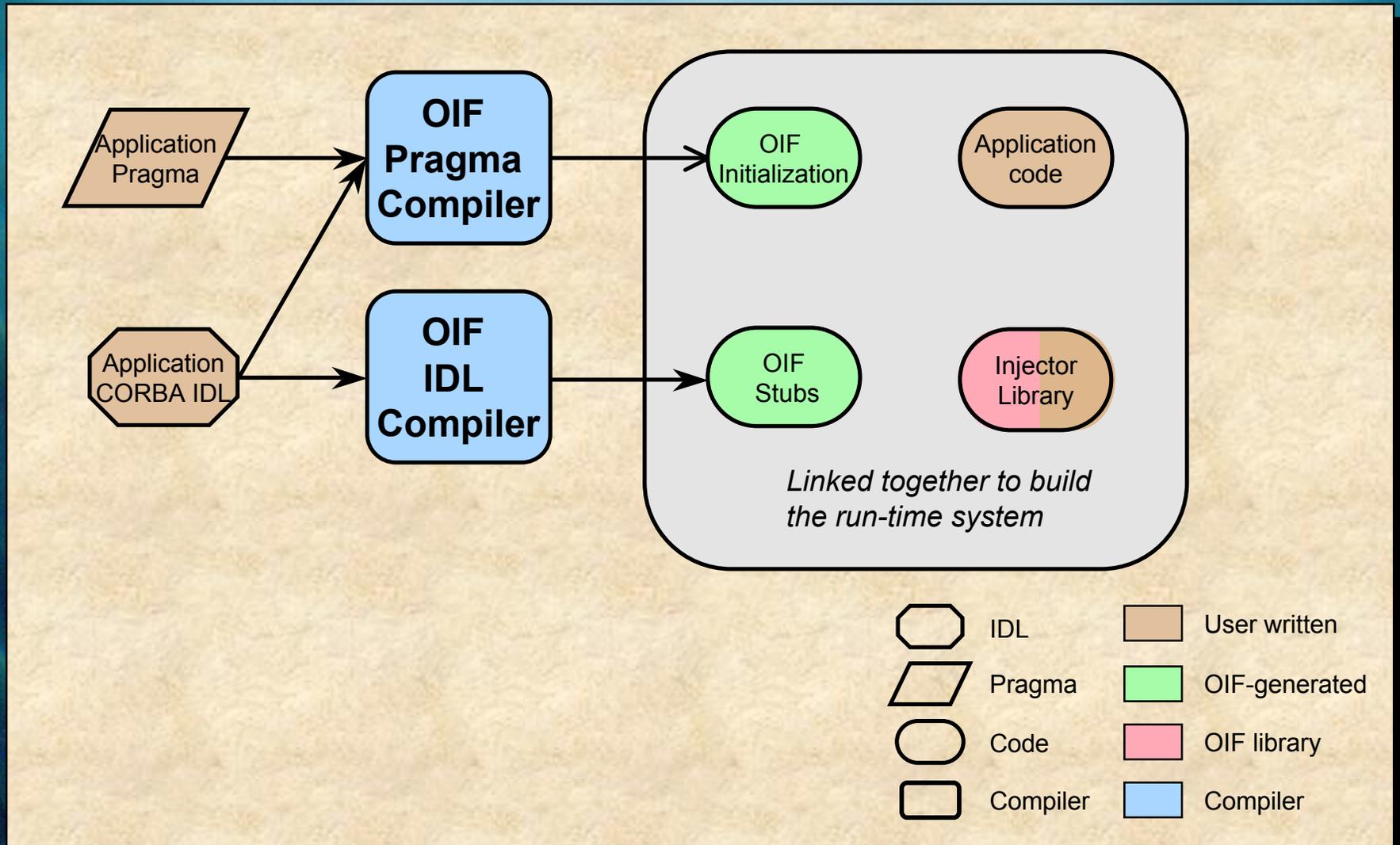
Here we define the mapping from the achieve names to injector factories. The "do" clauses specify a partial ordering on the injectors.

```
policy vendoom is
import vendoom;
import injectors;
ility Context, Security, QualityOfService,
    Reliability, Efficiency;
var priority : int = {1};
var retries : int = {5} only from client;
for Context do copyContext;
for Security on request in Controller do iButton;
for QualityOfService on call in ByPriorityController do
    queueing;
group cachedStuff on identifier, on description, on
    valueTo;
for Efficiency within cachedStuff do caching;
for Reliability do retry ;
define copyContext for Context as
    client ContextInjectorFactory do first,
    server ContextInjectorFactory do last;
define iButton for Security do last as
    client server injectors.
    AccessControlPkg.AccessControlInjectorFactory,
    client server injectors.
    IdentificationPkg.IButtonIdentificationInjectorFactory;
define queueing for QualityOfService as
    server injectors.
    QManager.QueueManagerInjectorFactory;
define caching for Efficiency as
    client CacheInjectorFactory do after copyContext;
define retry for Reliability as
    client ErrorRetryInjectorFactory ( retries = {"5"} ) do
    last;
end;
```

# *Language Semantics*

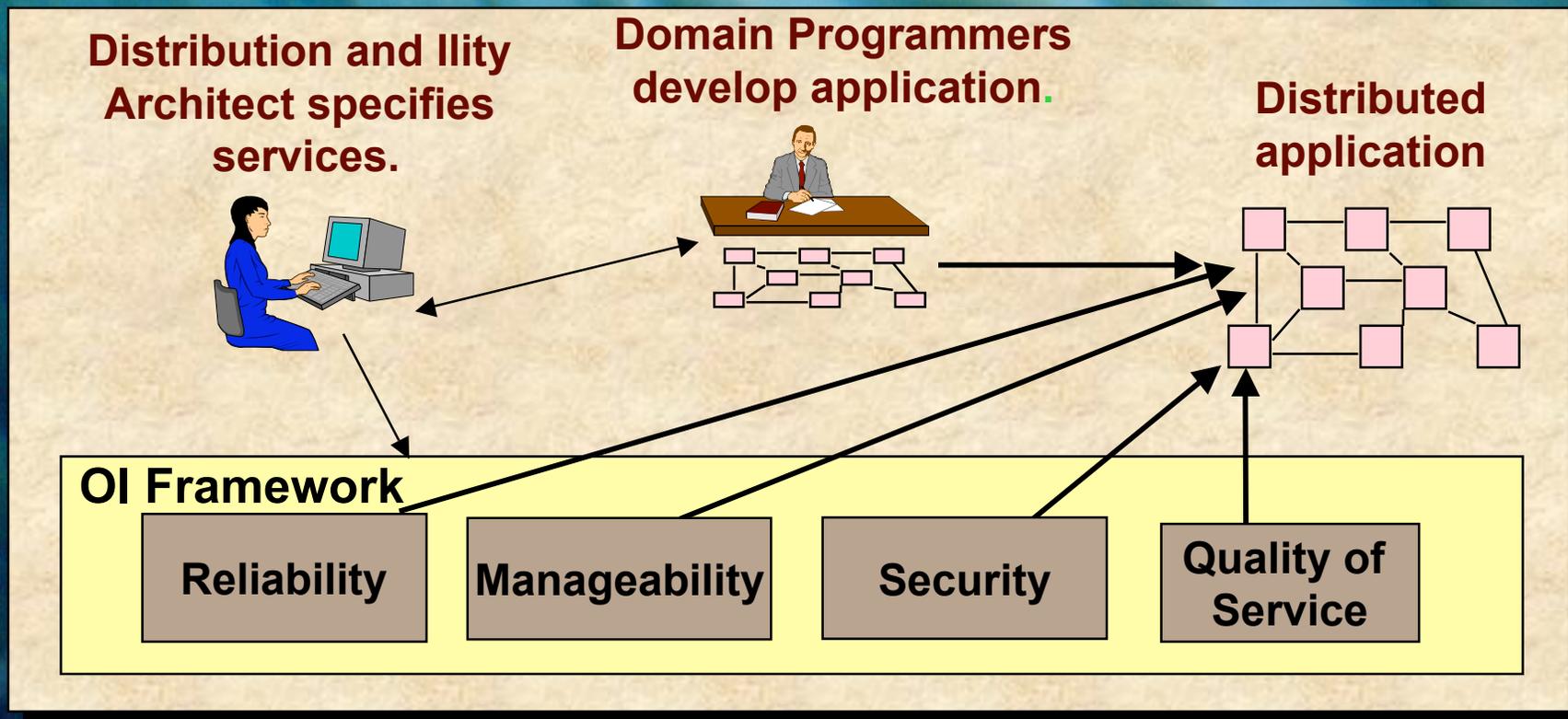
- ❖ **Top-level goal: for each interface/method, determine the appropriate default injector initialization sequence for that interface method.**
  - **Additional input: CORBA IDL**
- ❖ **Semantics:**
  - **For each method, class, ility, select the “most specific” way of doing that ility on that method, class**
    - **Actions inherit through the conventional interface hierarchy**
    - **An action on a class and method is more specific than one on just a class or a method; just a class or a method is more specific than an action done everywhere**
    - **Actions on subclasses are more specific**
  - **Order the actions on a method, class on the basis of their sequencers**
  - **Output the results**
- ❖ **Additional actions:**
  - **Declarations of annotations**
  - **Declarations of all classes/methods**
  - **Semantic compatibility checks (NYI)**

# Compilation Process



# OIF Process

- ❖ Map organizational *policies* to implementation
  - E.g.,
    - define a security policy
    - ensure that policy is followed by all distributed components



# *Aspect-Oriented Programming*

- ❖ **Aspect-Oriented Programming (AOP) is centered on**
  - **Separate expression of crosscutting concerns**
  - **Mechanisms to weave the separate expressions into a unified system**
- ❖ **OIF is an AOP mechanism**

# *Separation of concerns*

- ❖ A fundamental engineering principle is that of **separation of concerns**
  - Realizing different system concepts as separate, weakly linked elements
  - Distribution of expertise
- ❖ Separation of concerns promises better
  - Maintainability
  - Evolvability
  - Reusability
  - Adaptivity
- ❖ Concerns occur at both the
  - User/requirements level
  - Design/implementation level
- ❖ Concerns crosscut
  - Apply to different modules in a variety of places
- ❖ Concerns must be composed to build running systems

In conventional programming, the code for different concerns often becomes mixed-together (*tangled*)

# *Examples of Software Concerns*

- ❖ **Security**
  - Always call the security check before allowing database access
- ❖ **Accounting**
  - Always debit the user's account on each access to a service of objects in the class...
- ❖ **Synchronization**
  - Don't let multiple users call any of methods *f*, *g*, or *h* on a single object simultaneously
  - The effects of these actions should be transactional
- ❖ **Quality of service**
  - Queue up the waiting calls handling them by priority
- ❖ **Reliability**
  - Provide replicants of this object
- ❖ **Performance enhancements**
  - Cache the results of calls to elements in this class
  - Display routines should show the results of changes, except display routines called in the scope of other display routines should buffer their changes for display all at once

# *Aspect-Oriented Programming (AOP)*

- ❖ **Software engineering technology for separately expressing systematic properties while nevertheless producing running systems that embody these properties**
- ❖ **Need to express**
  - **Base program**
  - **Separate concerns**
  - **How the separate concerns map to the base program**
    - **Or, if you prefer, just a jumble of program elements that must be combined.**

## *OIF as AOP*

- ❖ **OIF is an instance of an AOP system**
  - We separate concerns into injectors, and provide a mechanism to integrate the injector mechanism into the running system

## *Real AOP Value*

- ❖ **We don't have to define all these policies before building the system**
- ❖ **Developers of tools, services, and repositories can remain (almost) completely ignorant of these issues**
- ❖ **We can change policies without reprogramming the system**
- ❖ **We can change policies of a running system**

# *Other AOP approaches*

- ❖ **Wrapping technologies**
  - Composition filters
  - JAC
- ❖ **Frameworks**
  - Aspect-Moderator Framework
- ❖ **Compilation technologies**
  - AspectJ
  - HyperJ
- ❖ **Post-processing strategies**
  - JOIE
  - JMangler
- ❖ **Traversals**
  - DJ
- ❖ **Event-based**
  - EAOP
- ❖ **Meta-level strategies**
  - Bouraqadi et al.
  - Sullivan
  - QSOUL/Logic Meta-Programming

# *Traditional Separation of Concerns*

- ❖ **Subprograms (procedures, functions, methods)**
- ❖ **Inheritance**
- ❖ **Do a good job of concern separation, but**
  - The programmer has to explicitly invoke the desired behavior
  - The programmer has to always be aware of when to invoke what behavior
  - Changing a policy (that's not already embodied in a subprogram) requires finding all the places that need modification and changing them
- ❖ **AOP is an alternative to this regime**

# *Quantification and Implicit Invocation*

- ❖ The unifying element of these approaches (and the characterizing definition of AOP) is the ability to state universally quantified programmatic assertions (quantification) on programs that have not been explicitly prepared to receive these assertions (implicit invocation, obliviousness).
  - Quantification: A given assertion can have effect in many places in the system
  - Implicit invocation: One can't tell for examining the local program source that the aspect will be invoked.
    - Surgery

# *The space of AOP language design*

*In programs P, whenever condition C arises, perform action A.*

- ❖ **Dimensions of concern for the designer and implementer of an AOP system:**
  - **Quantification:** What kinds of conditions C can be specified.
  - **Interaction:** What is the interface of the actions A. That is, how do they interact with base programs and each other.
  - **Weaving:** How will the system arrange to intermix the execution of the base actions of P with the actions A.

# *Quantification*

- ❖ **Over which events can one quantify**
  - **Static quantification refers to events recognizable in the source code**
  - **Dynamic quantification refers to the pattern of dynamic execution events**

# *Interaction*

- ❖ **The structure of the aspect code**
- ❖ **Interactions among aspects**
  - Including which runs first and how conflicts are recognized and resolved
  - Ordering
- ❖ **How aspects communicate with each other and the base code**
  - Visibility
- ❖ **Aspect parameterizations**

# *Weaving*

- ❖ **How does the system arrange to intermix the aspect and base behaviors**
  - **Compilers**
  - **Link-level wrapping**
  - **IDL compilers**
  - **Object-code modifiers**
  - **Meta-interpreters**

# *Extreme Experiment*

- ❖ **Over What can one Quantify?**
  - Static structure of the program
  - The events that happen in the dynamic execution of a system
- ❖ **The extreme of expressiveness in quantification is to be able to quantify over all the history of events in a program execution**
- ❖ **Events are with respect to the abstract interpreter of a language**
  - Software dark matter
- ❖ **Unfortunately, language definitions don't define their abstract interpreters.**

# *Events and Event Loci*

<b>Event</b>	<b>Syntactic locus</b>
Accessing the value of a variable or field	References to that variable
Modifying the value of a variable or field	Assignments to that variable
Invoking a subprogram	Subprogram calls
Cycling through a loop	Loop statements
Branching on a conditional	The conditional statement
Initializing an instance	The constructors for that object
Throwing an exception	Throw statements
Catching an exception	Catch statements

## *More Events and Loci*

<b>Event</b>	<b>Syntactic locus</b>
Resuming after a lock wait	Other's notify and end of synchronizations
Testing a predicate on several fields	Every modification of any of those fields
Changing a value on the path to another	Control and data flow analysis over statements (slices)
Swapping the running thread	Not reliably accessible, but atomization may be possible
Being below on the stack	Subprogram calls
Freeing storage	Not reliably accessible, but can try using built-in primitives
Throwing an error	Not reliably accessible; could

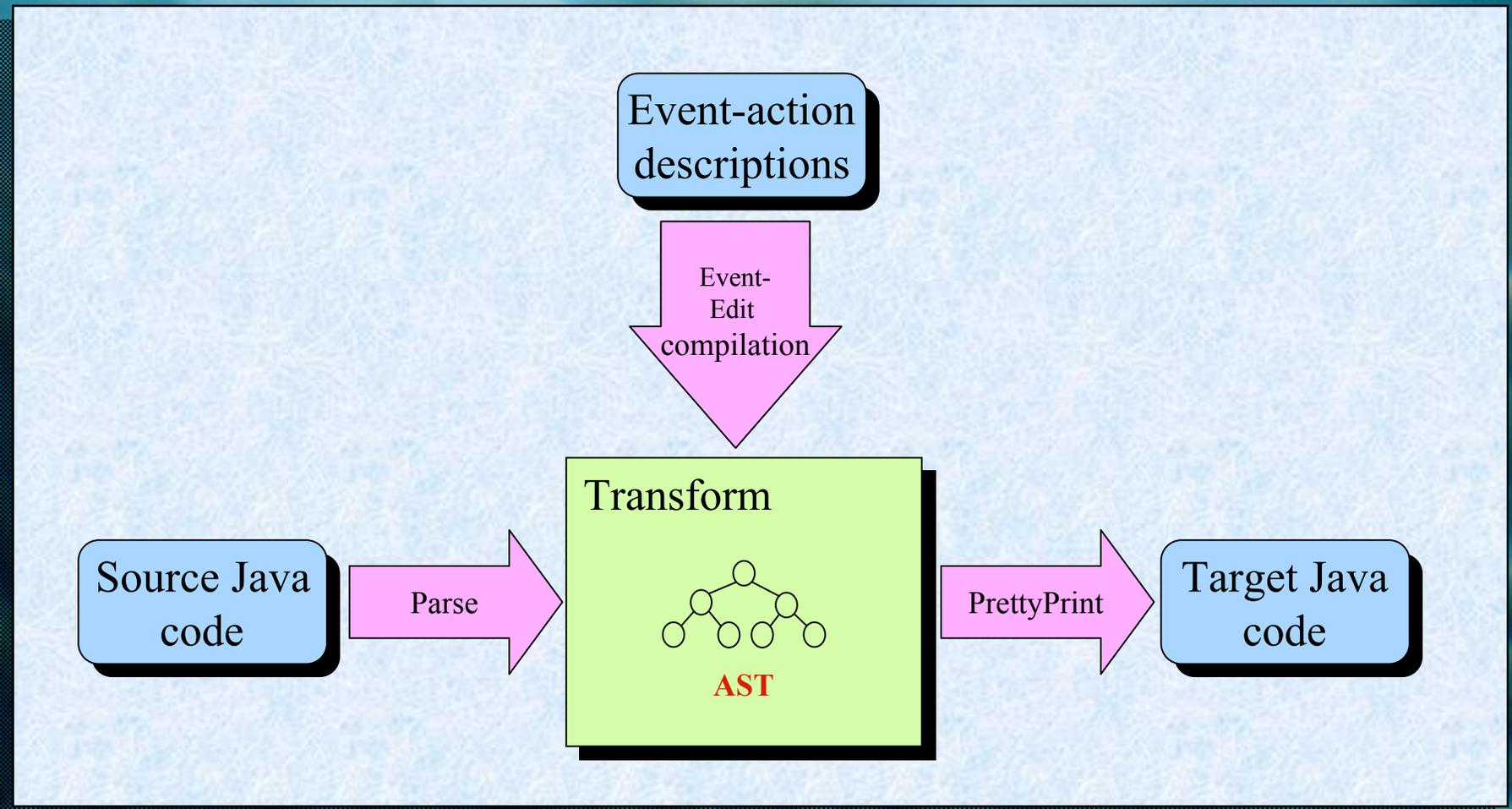
## *Research regime*

- ❖ Define a language of events and actions on those events.
- ❖ Determine how each event is reflected (or can be made visible) in source code.
- ❖ Create a system to transform programs with respect to these events and actions.
- ❖ Developing an environment for experimenting with AOP languages (DSL for AOP)

# *Transformational Alternatives*

- ❖ **For Java, can transform at**
  - **The source-code level**
  - **The byte-code level**

# *Architectural View*



# *Applications*

- ❖ Applying AOP to debugging and validating concurrent programs.
- ❖ Applying AOP to monitor programs during operation, so that actions can be initiated in case bad things happen.
- ❖ Applying AOP as a general programming paradigm.

# *Program Debugging*

- ❖ **Detect multi-threading problems caused by access to shared resources by competing threads.**
- ❖ **Validate trace executions against user requirements.**
- ❖ **Validate multithreaded programs by exploring schedule interleavings.**

# *Detect*

## *Multi-threading Problems*

- ❖ Deadlocks: Observe in what order locks are taken and released and infer potential deadlocks from cycles.
- ❖ Data Races: Observe what locks threads own when they access variables and infer potential data races from empty overlaps.

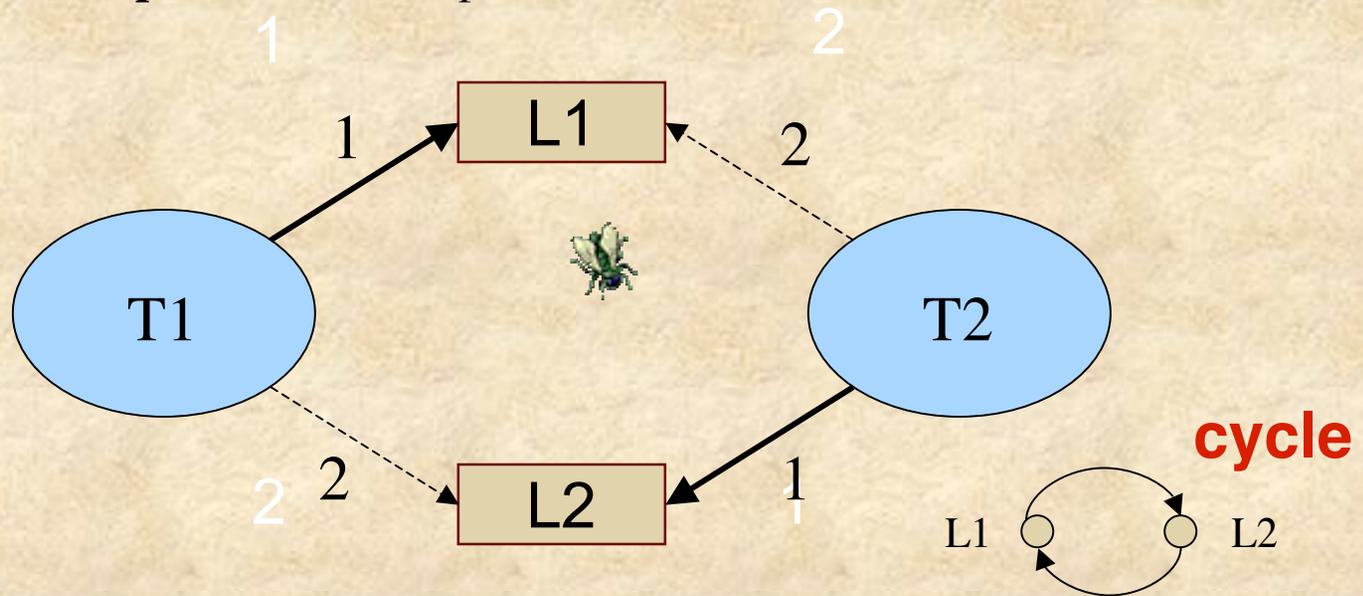
# Deadlocks

A **deadlock** can occur when threads access and lock shared resources, and lock these in different order.

**Example Solution:** Impose order on locks:  $L1 < L2$

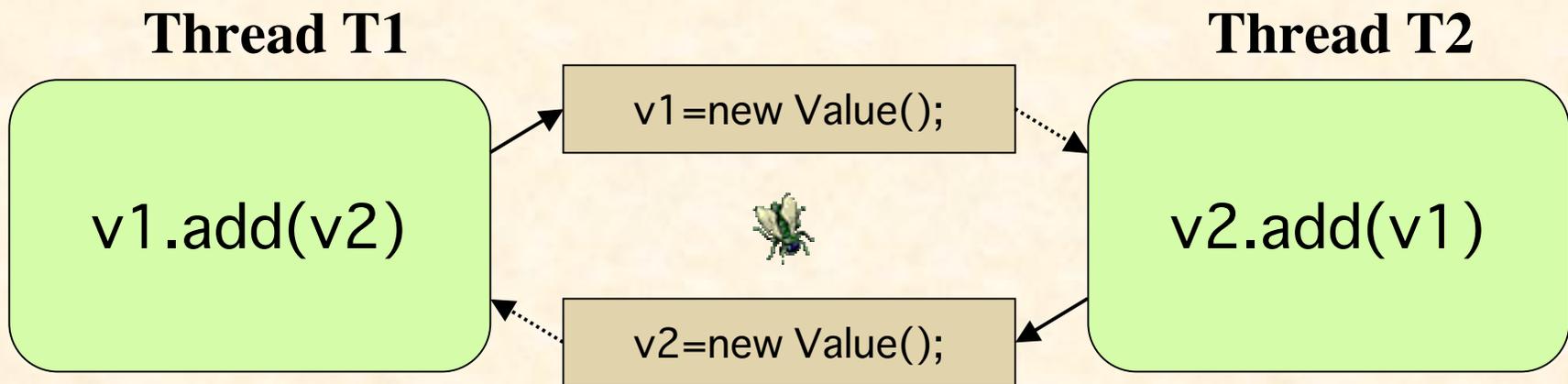
**Problem:**

T1 locks L1 first  
T2 locks L2 first



# Java Program with Deadlock

```
class Value{  
    int x = 1;  
    synchronized void add(Value v){x = x + v.get();}  
    synchronized int get(){return x;}  
}
```



# *Applying AOP*

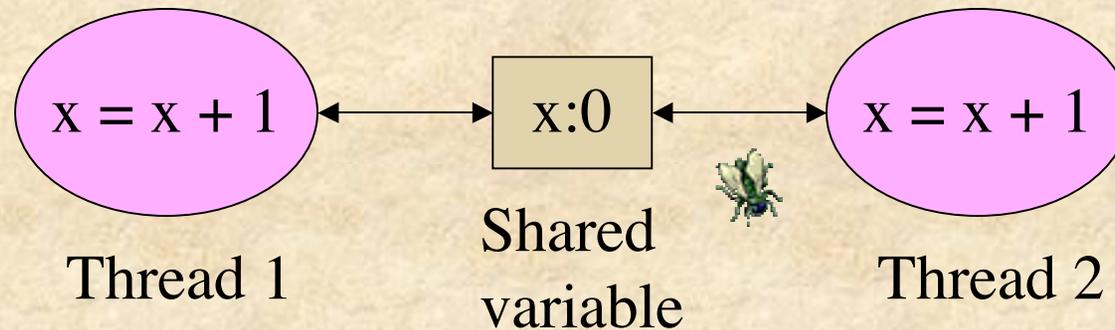
```
aspect DeadlockDetection{
  when synchronize(obj){
    Thread curr = Thread.currentThread();
    Set locks = Threads.getLocks(curr);
    Graph.addEdges(locks,obj);
    Graph.findCycles();
    Threads.addLock(curr,obj);
  }
  when endof synchronize(obj){
    Threads.remove(curr,obj);
  }
}
```

# Data Races

A **data race** occurs when two threads

- Access a shared variable,
- At least one access is a write, and
- No mechanism is used to prevent simultaneous access.

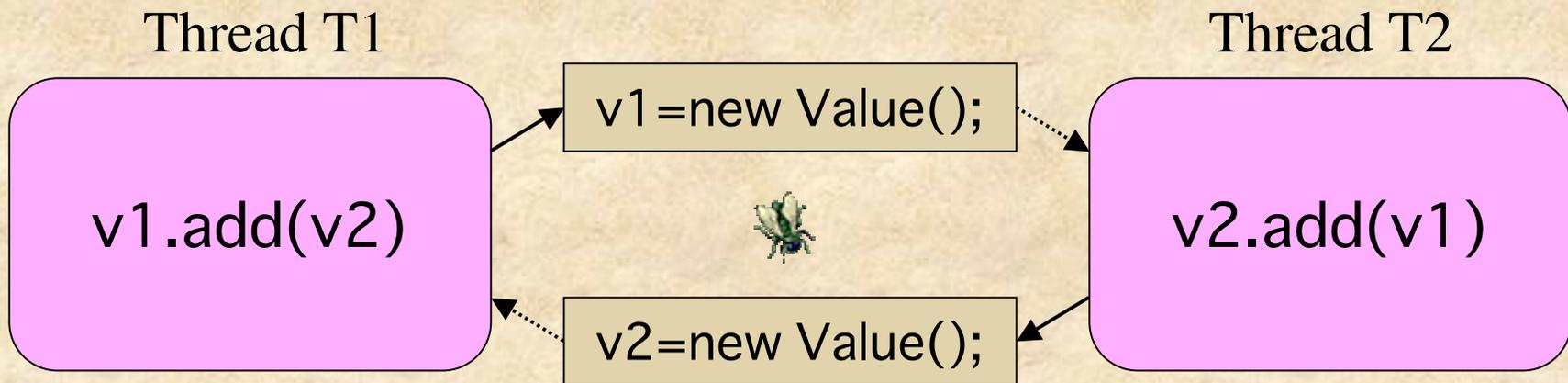
**Example Solutions:** monitors, semaphores, ...



Result after both updates : 2 ... or maybe 1

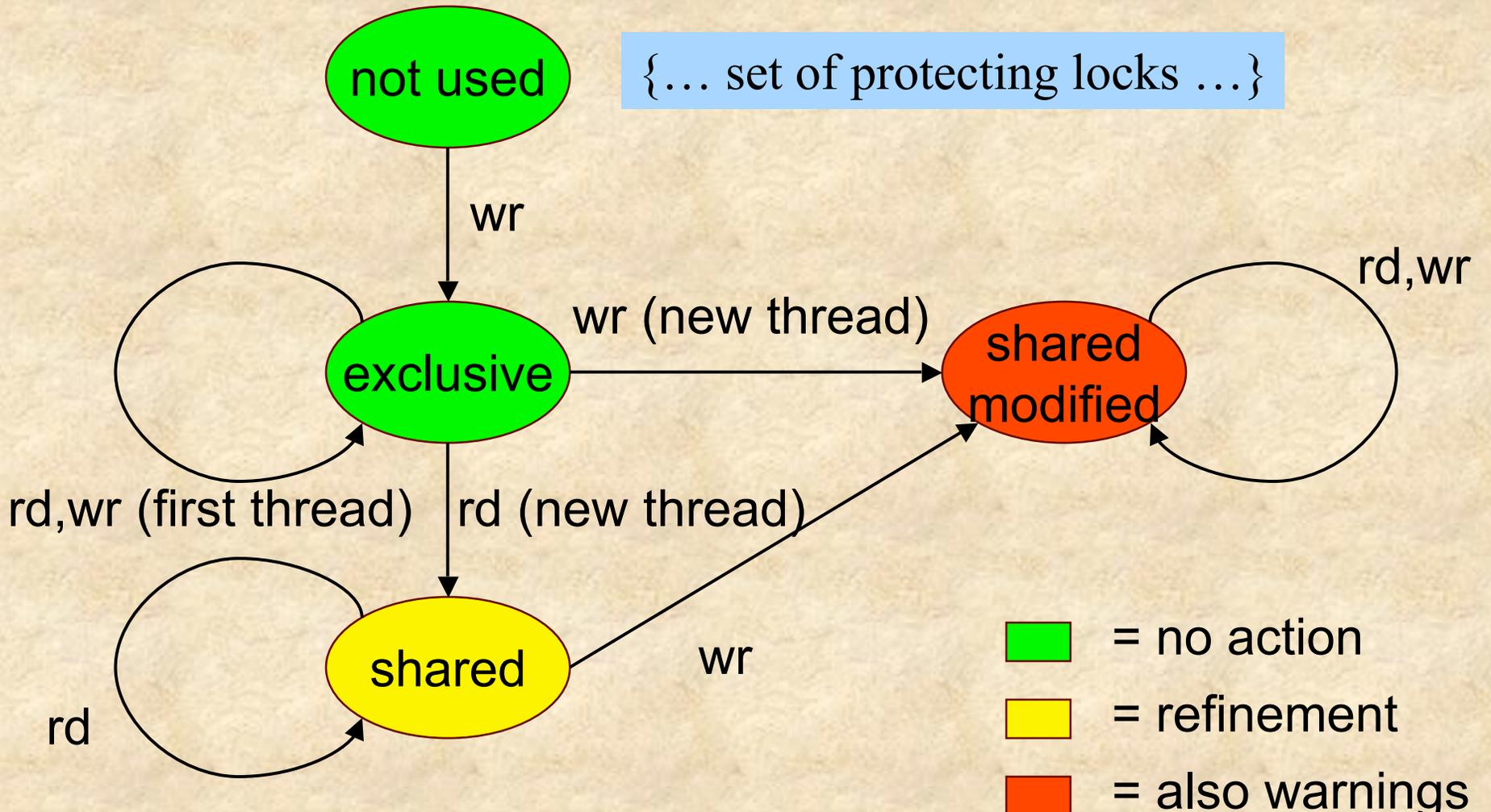
# *Java Program with Datarace*

```
class Value{  
    int x = 1;  
    void add(Value v){x = x + v.get();}  
    int get(){return x;}  
}
```



# *For Each Variable: A Lockset and a Statemachine*

{... set of protecting locks ...}



Eraser algorithm (Compaq)

- = no action
- = refinement
- = also warnings

# *Applying AOP*

```
aspect DataraceDetection{
  when synchronize(obj){
    Thread curr = Thread.currentThread();
    Threads.addLock(curr,obj);
  }
  when endof synchronize(obj){
    Thread curr = Thread.currentThread();
    Threads.remove(curr,obj);
  }
  when accessto(var,isWrite){
    Thread curr = Thread.currentThread();
    Statemachine.update(curr,var,isWrite);
    Statemachine.checkEmptyness(var);
  }
}
```

# *Validating Execution Traces Against User Requirements*

```
aspect CheckRequirements{  
  when(CLOSED and not previously DO_CLOSE){  
    Report( "System closed by itself" );  
  }  
  whennot(DO_CLOSE implies  
    eventually(20)CLOSED){  
    Report( "System did not close" );  
    CloseSystem(); // repair  
  }  
}
```

# *Explore Scheduling*

- ❖ **Simple example: assume that all variable accesses are protected with locks.**
- ❖ **Insert a call of a randomized yield statement in front of all synchronization statements and calls of synchronized methods.**
- ❖ **This will cause the scheduler to randomly make a context switch whenever a lock is taken. This may be used, for example, to reveal deadlocks.**

# Readings

## ❖ OIF

- Robert E. Filman, Stu Barrett, Diana D. Lee, and Ted Linden. Inserting Ilities by Controlling Communications. *Communications of the ACM*, Vol. 45, No. 1, January, 2002, pp. 116-122.
- <http://ic.arc.nasa.gov/~filman/text/oif/cacm-oif.pdf>

## ❖ AOP Is

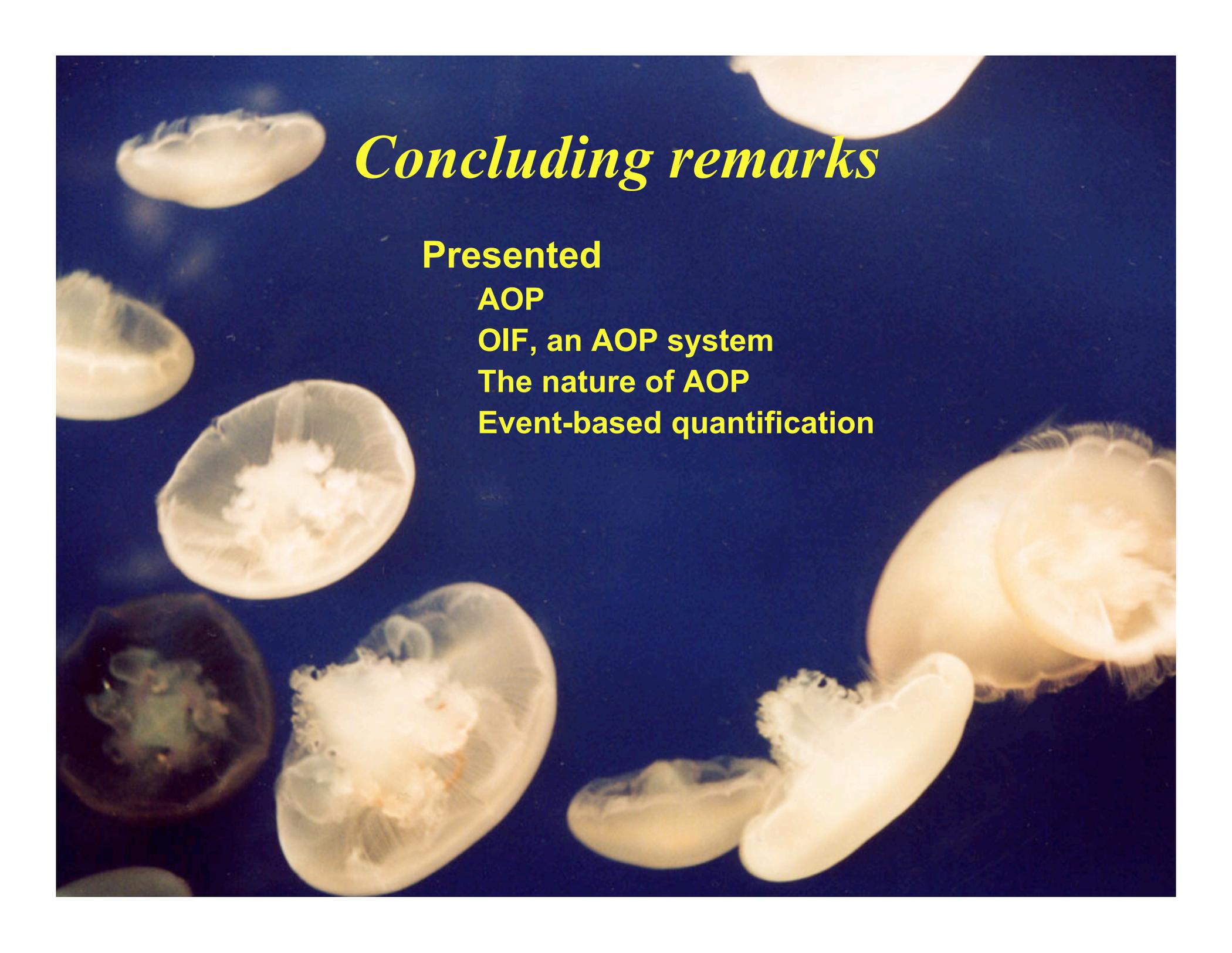
- Robert E. Filman and Daniel P. Friedman. Aspect-Oriented Programming is Quantification and Obliviousness. Workshop on Advanced Separation of Concerns, OOPSLA 2000, Minneapolis, October 2000.
- <http://ic.arc.nasa.gov/~filman/text/oif/aop-is.pdf>

## ❖ Event-based AOP

- Robert E. Filman and Klaus Havelund. Source-Code Instrumentation and Quantification of Events. AOSD 2002 Workshop on Foundations Of Aspect-Oriented Languages (FOAL), Twente, Netherlands, April 2002.
- <http://ic.arc.nasa.gov/~filman/text/oif/aop-events.pdf>

## *Related work on transformations*

- ❖ **De Volder et al. metaprogramming**
- ❖ **AOP through program transformation**
  - Colcumbet, Fradet and Sudholt
  - Schonger et al. XML transformation
  - Skipper ku
- ❖ **Nelson et al. concern-level foundational composition operators: correspondence, behavioral semantics and binding**
- ❖ **Walker and Murphy on events as join points**



# *Concluding remarks*

## **Presented**

**AOP**

**OIF, an AOP system**

**The nature of AOP**

**Event-based quantification**