Assuring Object-Oriented Architecture

Jonathan Aldrich
School of Computer Science
Carnegie Mellon University

Dahl-Nygaard Junior Prize Lecture
ECOOP 2007

August 3, 2007

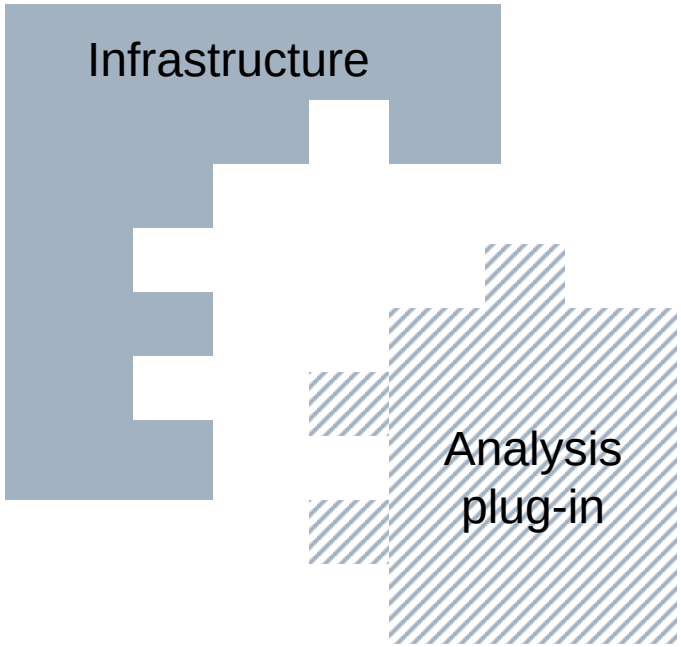
Assuring Object-Oriented Architecture

- What is Architecture?
  - High-level design intent
  - Ensures important program properties
- Examples of Architectural Assurance
  - High-level object structure
  - Protocols of behavior
- Requirements for Impact
  - Natural expression of architectural intent
  - Provide immediate value and maintain over time
  - Work with object-oriented designs
    - Inheritance, recursion, state, aliasing

1998: What I Learned from Vortex

- Task: adding analysis to Vortex compiler
  - Optimizing synchronization in Java
  - As student of Craig Chambers

- How should I do this?
  - Many sub-tasks
  - Need to understand where to plug in to infrastructure
  - Very difficult!



1999: I learned about Software Architecture

[Perry and Wolf 1992] [Garlan and Shaw 1993]

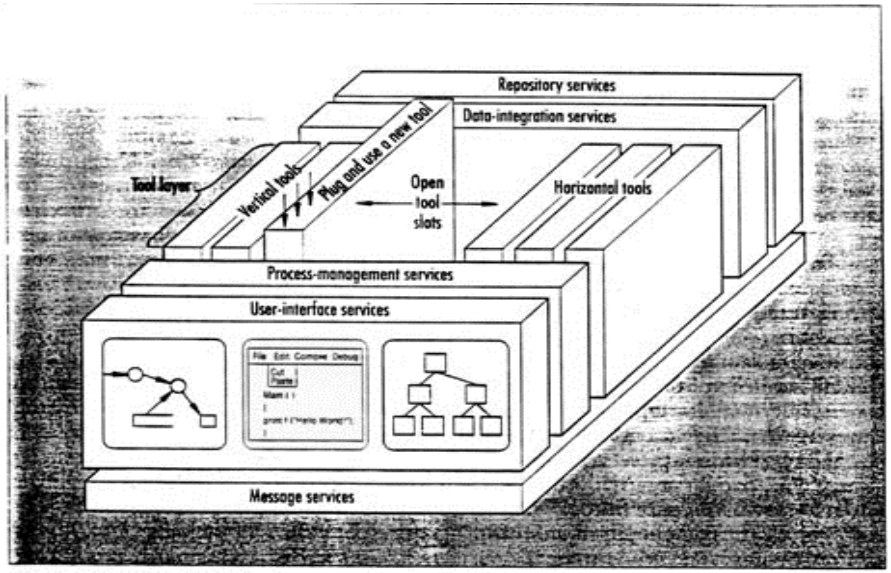
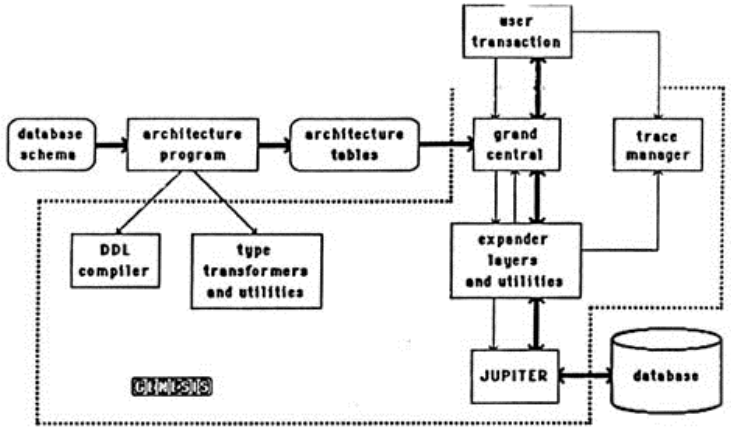


Figure 1. The NIST/ECMA reference model.



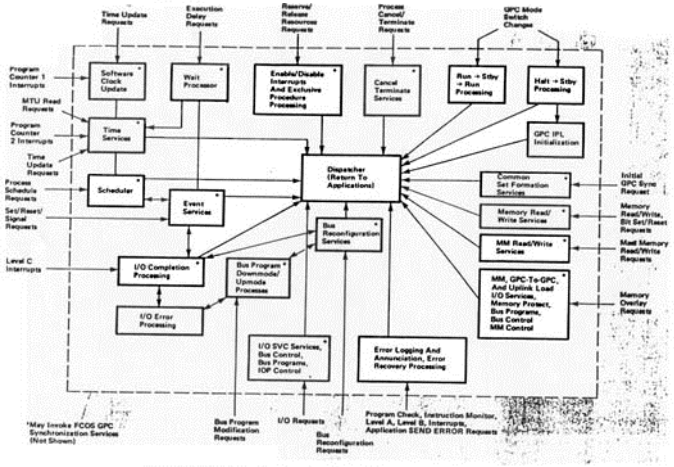


FIGURE 7. Flight Computer Operating System (The FCOS dispatcher coordinates and controls all work performed by the on-board computers.)

Communications of the ACM, "Architecture of the Space Shuttle Primary Avionics Software Systems," Gene D. Carlow, September 1984, Vol. 27, No. 9, P. 933

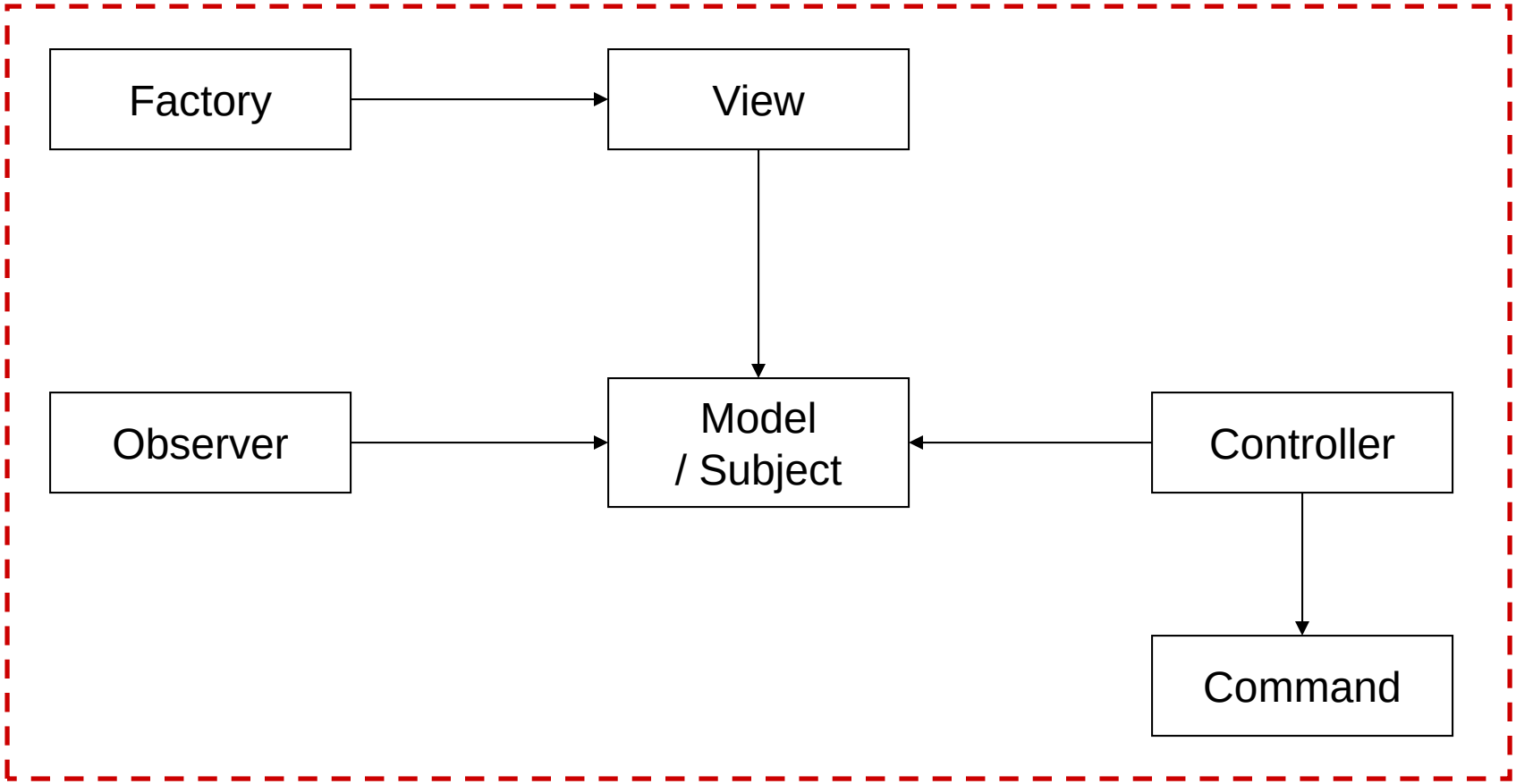
- Architecture is the fundamental organization of a system, embodied in its components, their relationships to each other and the environment, and the principles governing its design and evolution - ANSI/IEEE Std 1471-2000

What does ***Architecture*** mean for ***Objects***?

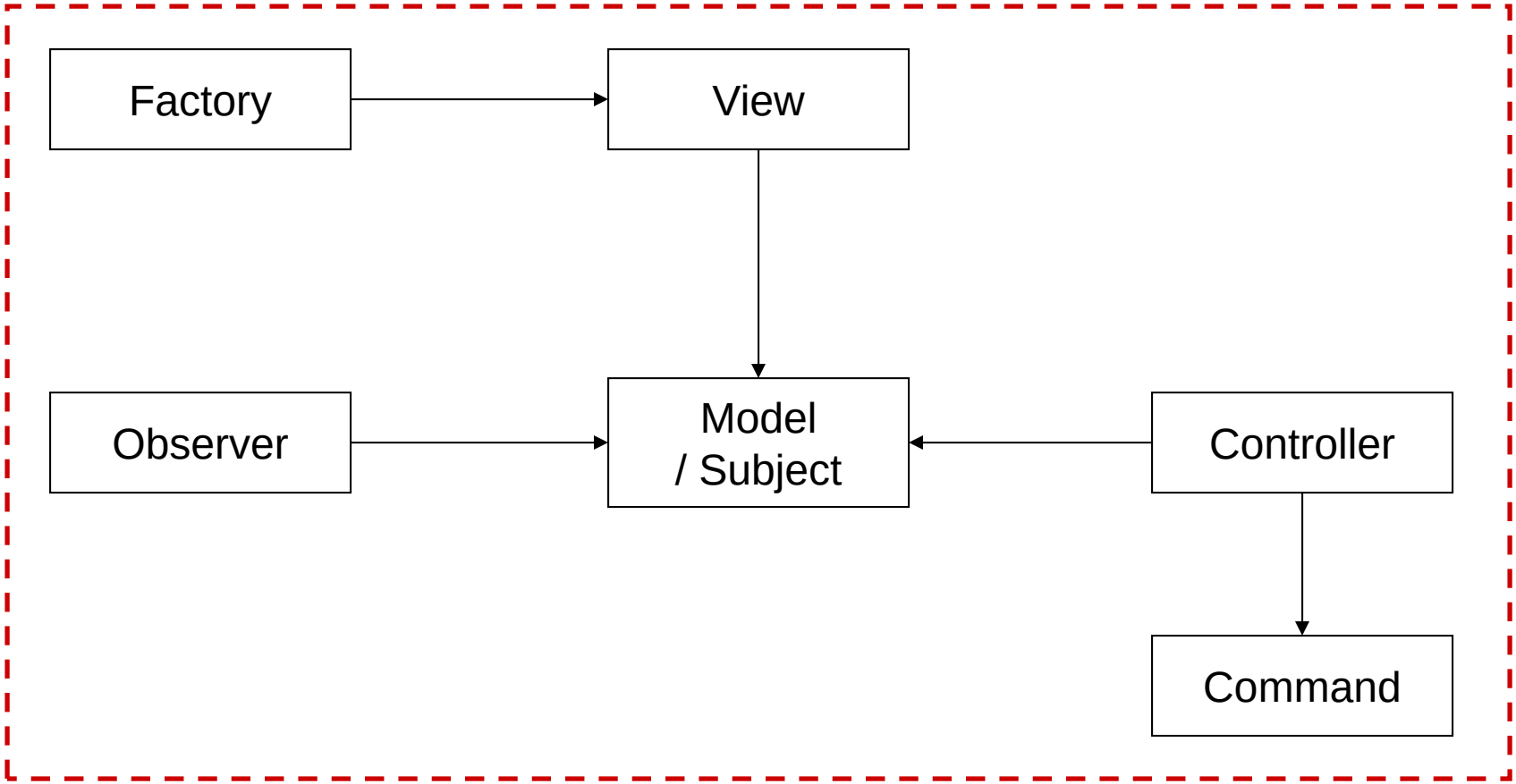
Objects [Dahl and Nygaard '67]

Model

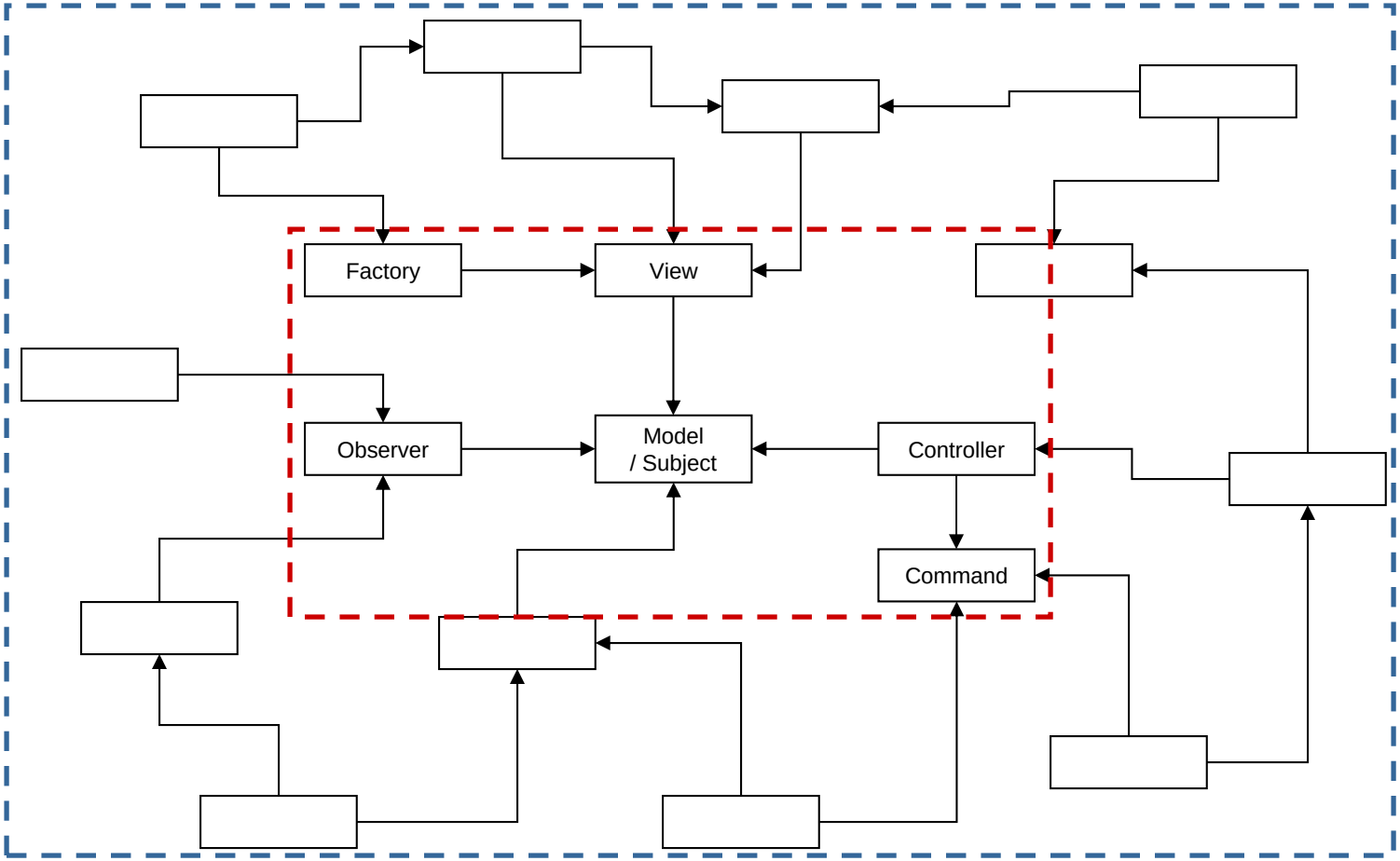
Patterns [Gamma, Helm, Johnson, and Vlissides '95]



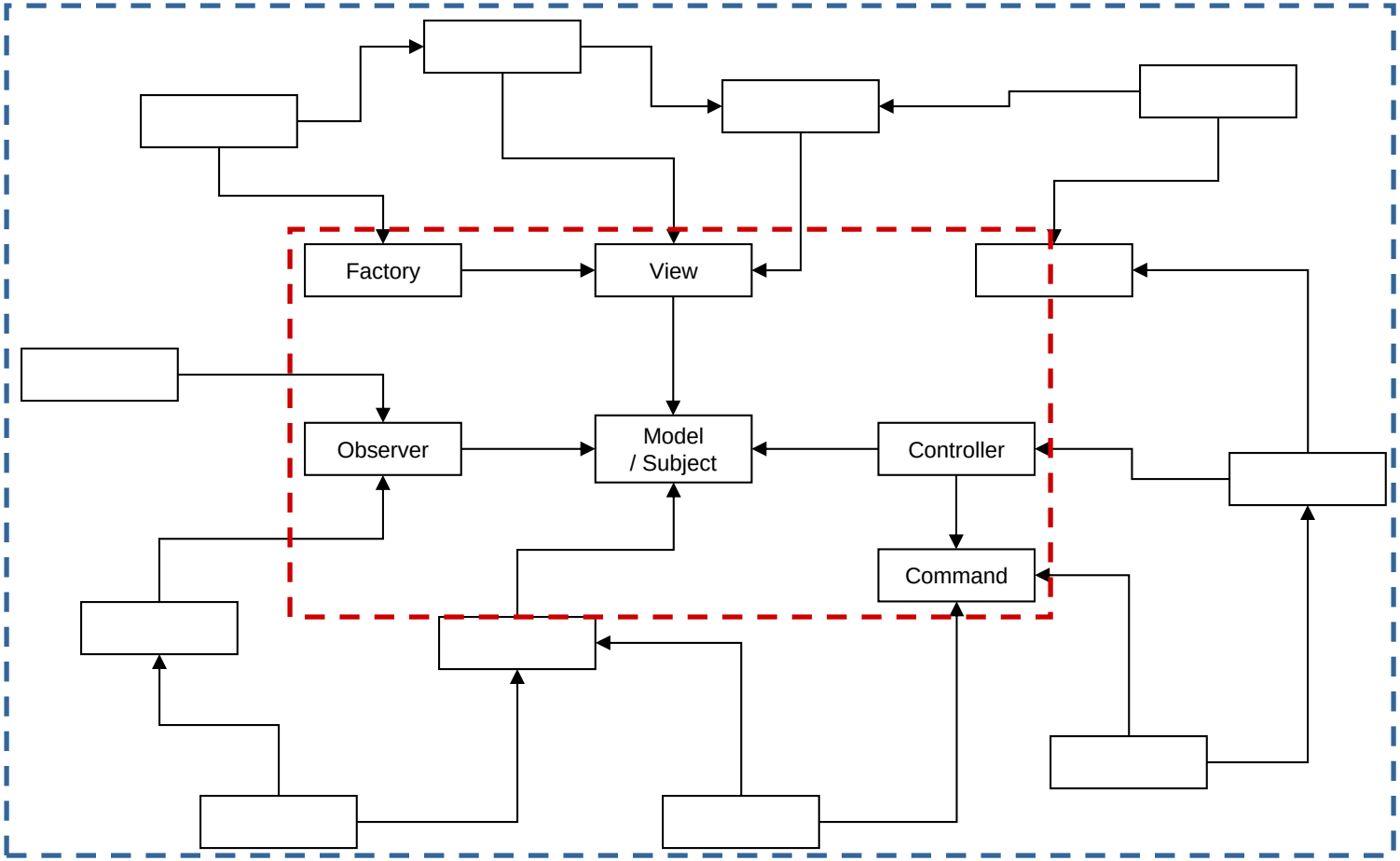
Patterns [Gamma, Helm, Johnson, and Vlissides '95]



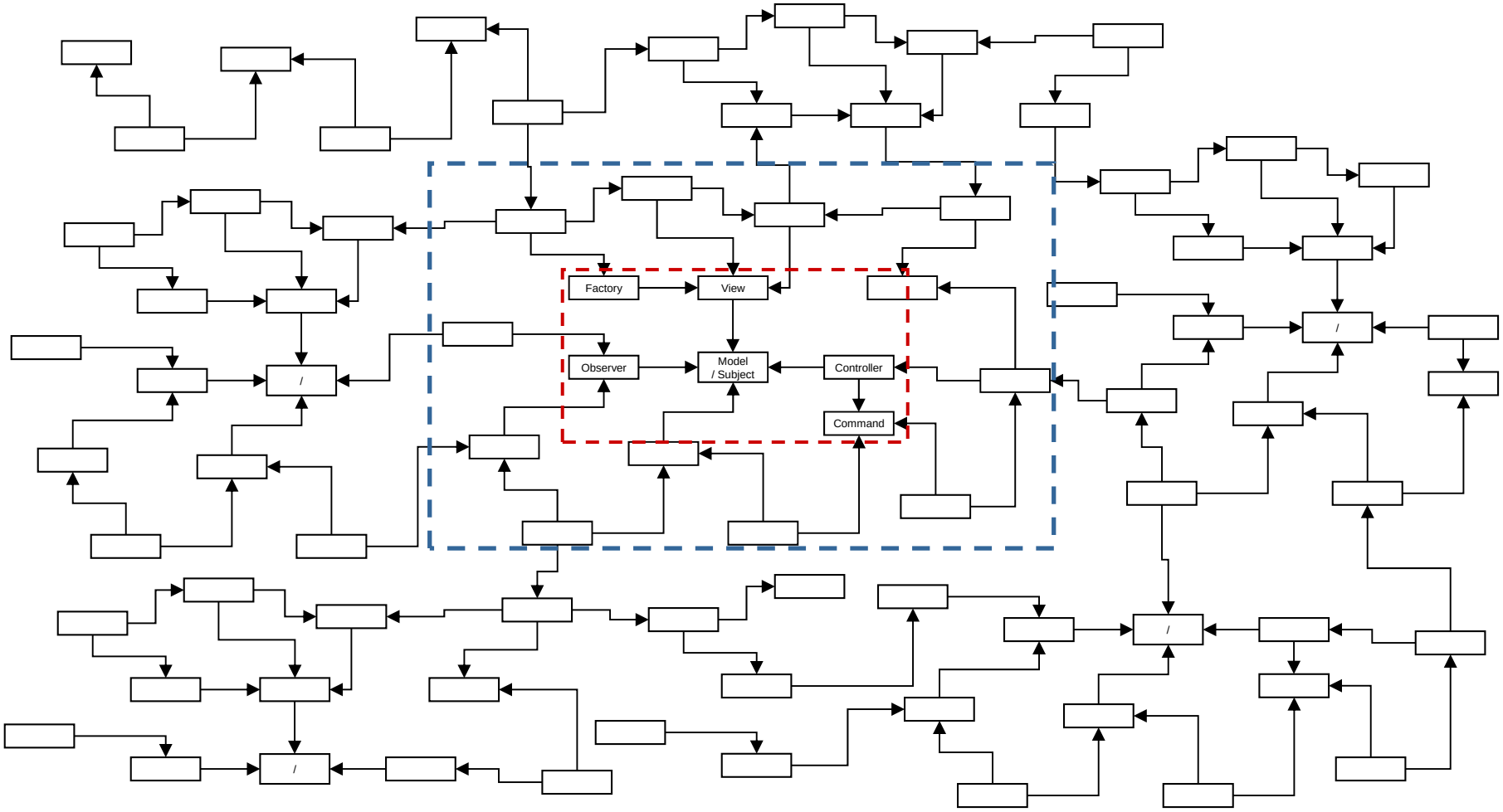
Architecture [Perry and Wolf 1992] [Garlan and Shaw 1993]



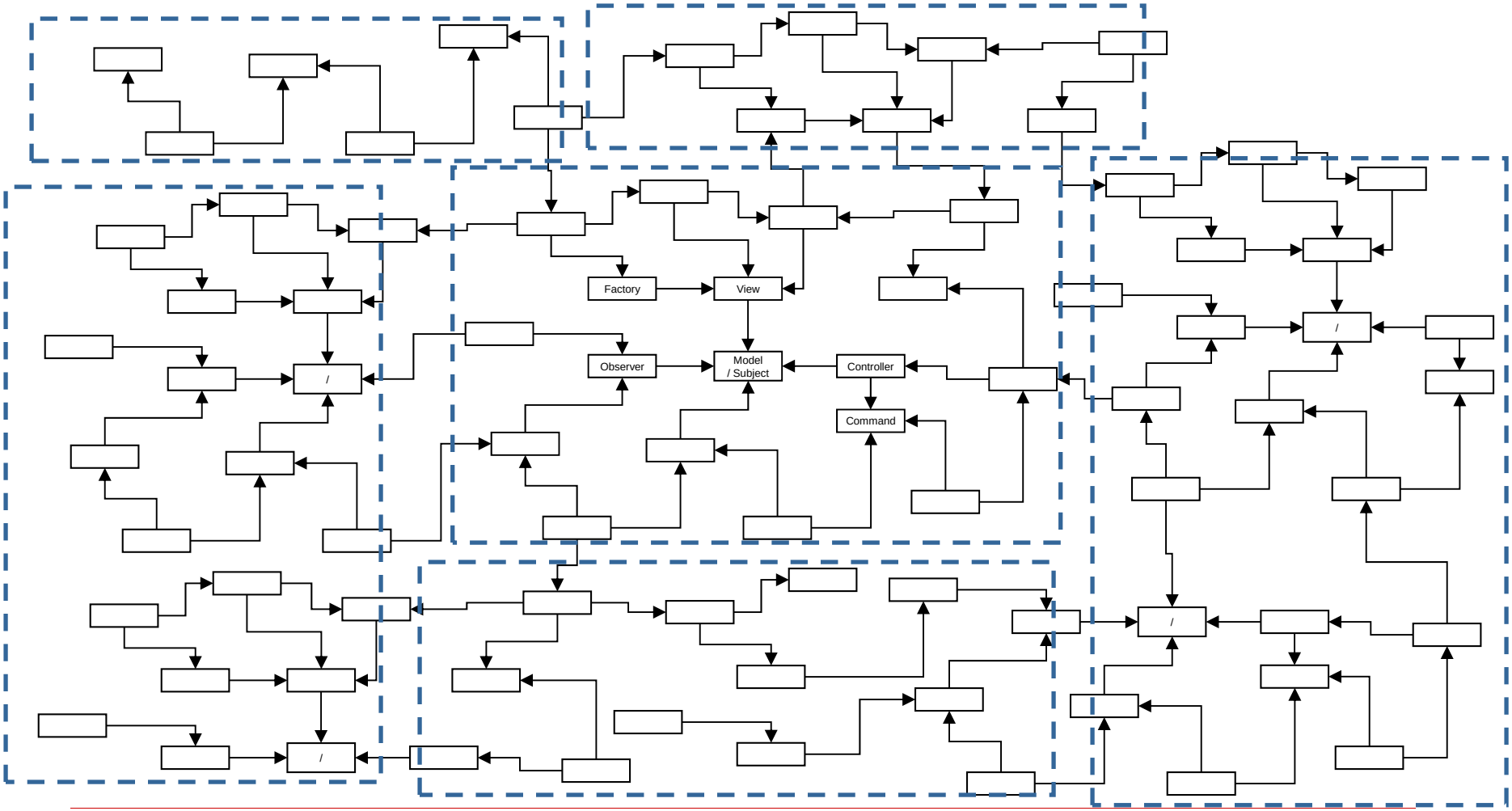
Object-Oriented Architecture: Patterns and Beyond



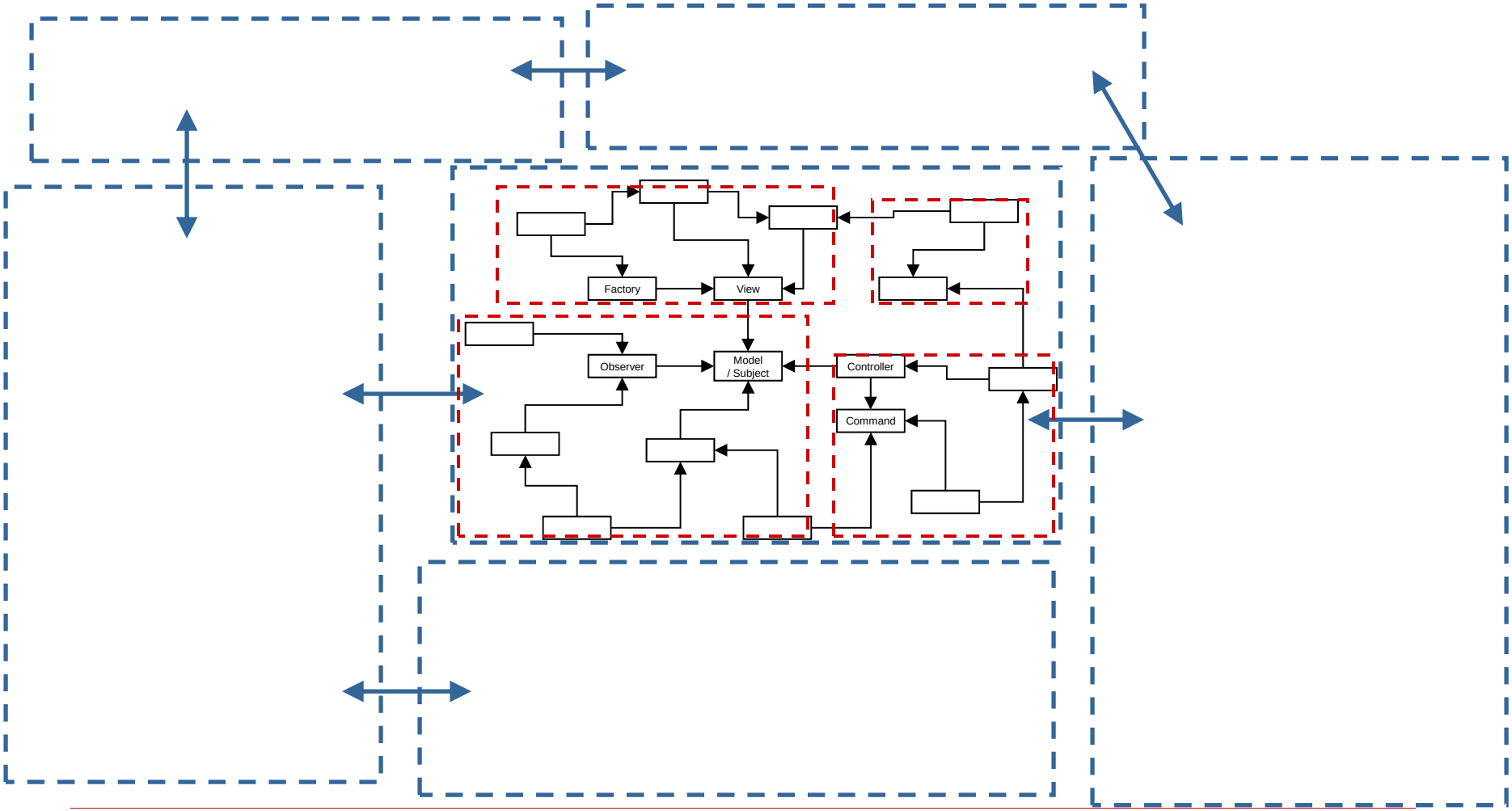
Object-Oriented Architecture: Patterns and Beyond



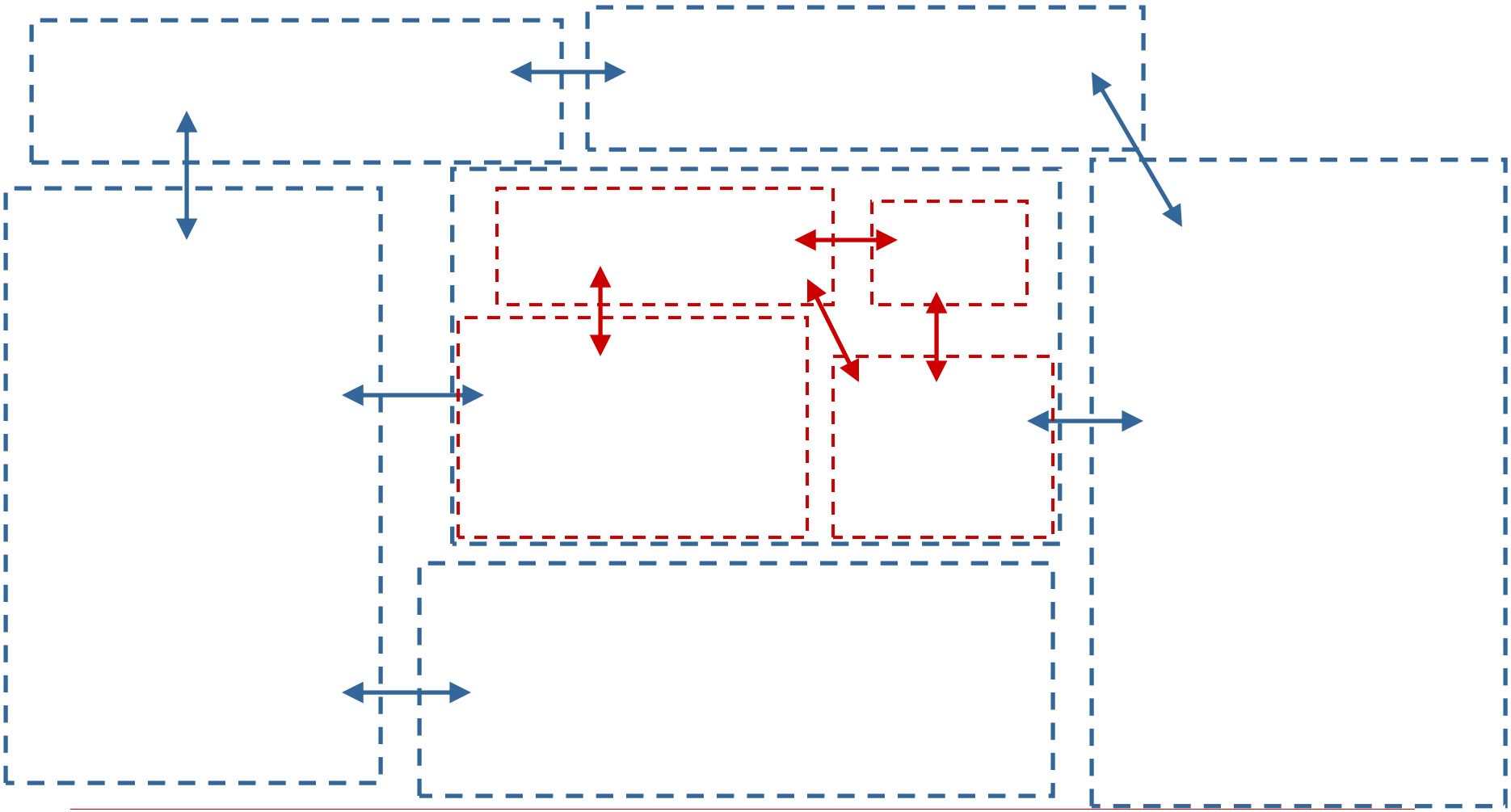
Object-Oriented Architecture: Patterns and Beyond



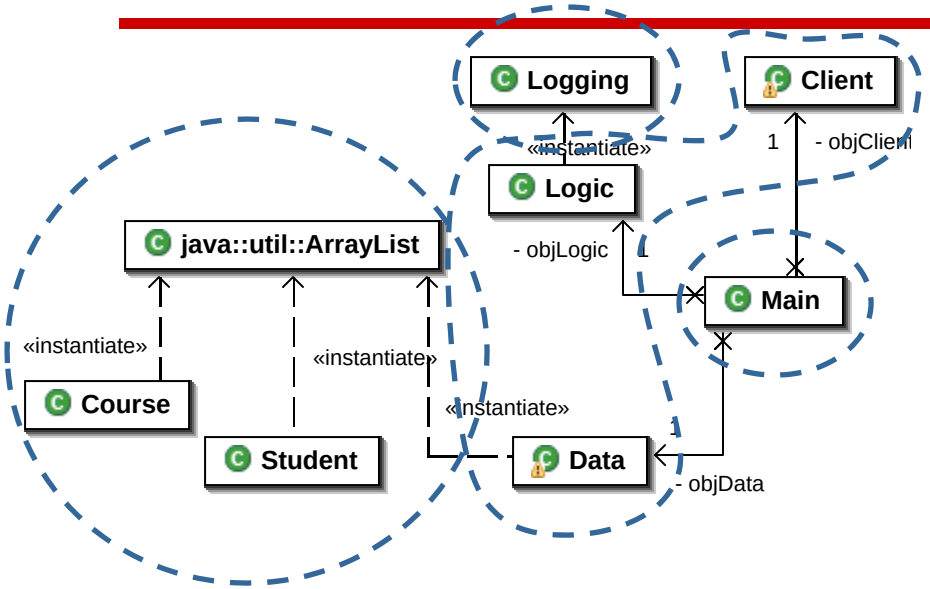
Object-Oriented Architecture: Patterns and Beyond

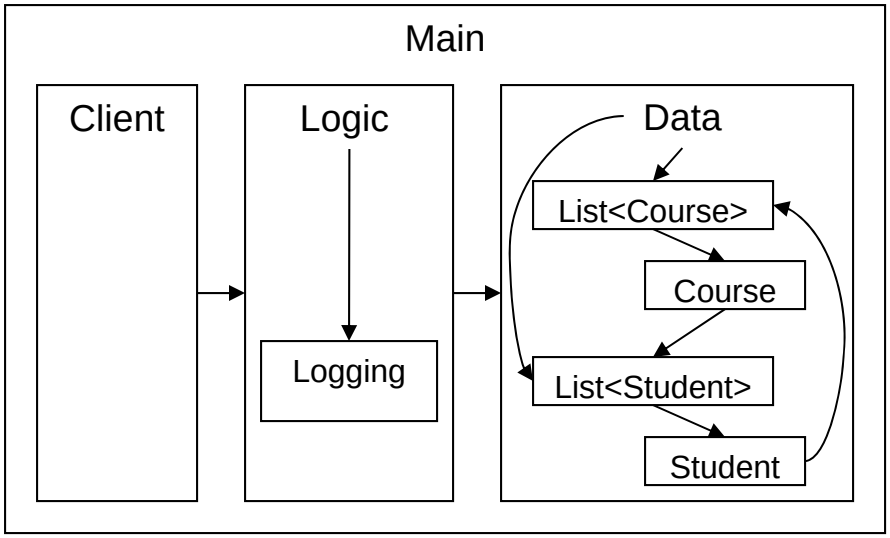


Object-Oriented Architecture: Patterns and Beyond



Code vs. Execution Structure





Code structure

- Shows classes
- Shows relationships
  - instantiation
  - calls
  - references
- Flat graph
  - Doesn't scale to large applications
  - Doesn't show what's important

Execution structure

- Shows object hierarchy
  - Most important objects near root
  - Grouped according to logical containment
    - NOT complete encapsulation
    - E.g. the Logic tier accesses courses and students
- Also shows relationships

Object-Oriented Architecture

- High-level design constraints on the structure and interaction of groups of objects
  - Shows object structure
  - Scales from patterns to applications
  - Assures quality attributes

- Examples
  - Components will have their own thread and will communicate through pipes with no shared data [c.f. Erlang]
    - Benefits: ease of modification, safe concurrency
  - Plugins will follow lifecycle constraints on calling framework methods [c.f. EJB, Eclipse]
    - Benefits: safe reuse of framework code

Assuring Architecture

- Even the best architecture may ***degrade over time***
  - Implementers may be unaware of architectural constraints
    - Need architectural knowledge specific to coding context
  - Ad-hoc workarounds for architectural limitations
    - Need mechanism for triggering review/revision or architecture
- My research: static verification of OO architectural ***structure*** and ***behavior***
  - ***Document architectural constraints in implementation***
    - Ensures awareness of architecture within a specific context
  - ***Verify implementation conforms to architecture***
    - Ensures architecture evolves as it is needed
- Doing this right for objects is challenging!
  - ***Natural expression*** of architectural constraints
  - Provide ***immediate value***
  - ***Support OO designs***, including inheritance, recursion, state, aliasing

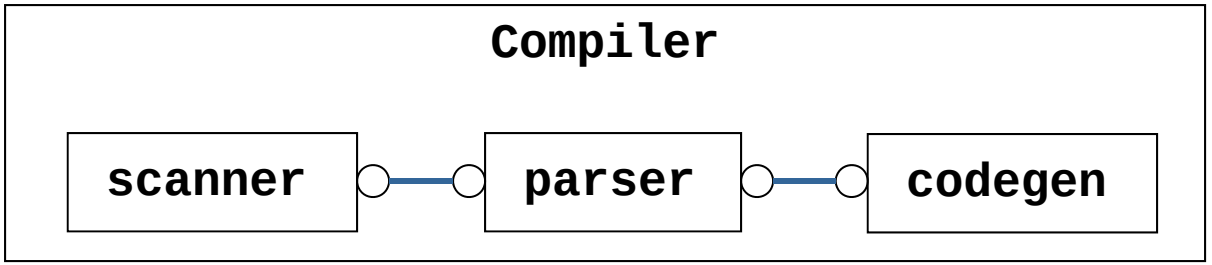
Assuring Architectural Structure

ArchJava: Structural Conformance Checking

with Craig Chambers and David Notkin

- *Specifies* architecture within implementation
  - Identifies certain objects as components
  - Explicitly declares connections between components
  - Describe what data is part of which component
- *Enforces* conformance with types
  - Can't pass component references across architecture
  - Private data must be confined to its component

ArchJava Example



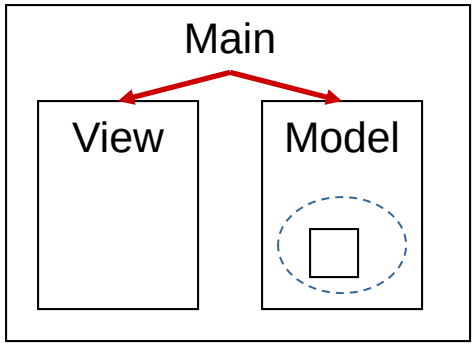

```
public component class Scanner {
  public port tokens { provides Token next(); }
}

public component class Compiler {
  private final owned Scanner scanner = new Scanner();
  private final owned Parser parser = new Parser();
  private final owned CodeGen codegen = new CodeGen();

  connect scanner.tokens, parser.tokens;
  connect parser.ast, codegen.ast;
}
```

Architectural Conformance in ArchJava

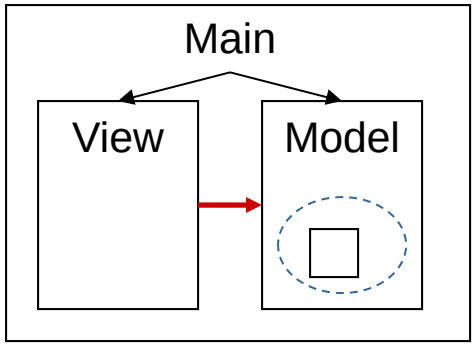
- Communication Integrity [Moriconi et al., Luckham et al.]
  - Components in the implementation communicate only as specified in the architecture



- Control Flow Integrity [ECOOP '02]
  - All inter-component control flow falls into one of the following categories:
    - A parent component invokes a method on one of its children

Architectural Conformance in ArchJava

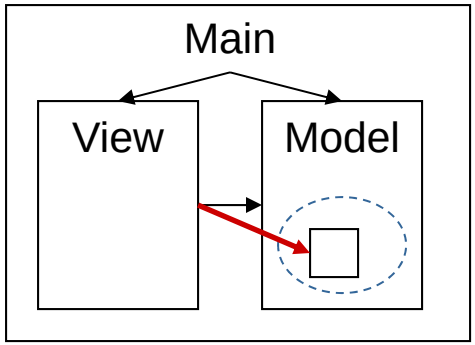
- Communication Integrity [Moriconi et al., Luckham et al.]
  - Components in the implementation communicate only as specified in the architecture



- Control Flow Integrity [ECOOP '02]
  - All inter-component control flow falls into one of the following categories:
    - A parent component invokes a method on one of its children
    - A component invokes a method through a connection

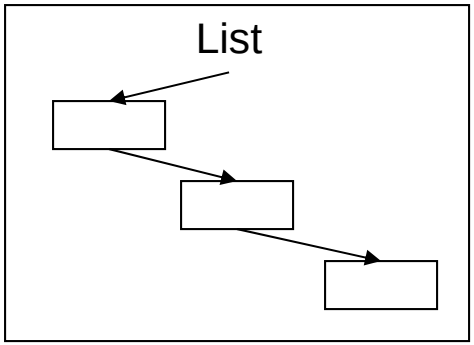
Architectural Conformance in ArchJava

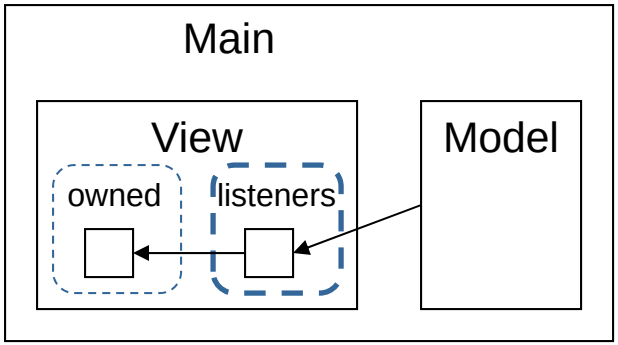
- Communication Integrity [Moriconi et al., Luckham et al.]
  - Components in the implementation communicate only as specified in the architecture



- Control ***and Data*** Flow Integrity [dissertation, 2003]
  - All inter-component control ***and data*** flow falls into one of the following categories:
    - A parent component invokes a method on one of its children
    - A component invokes a method through a connection
    - A component writes to another component's data, along a path documented in the architecture
    - [and two others based on lent/unique annotations]

What makes it work?



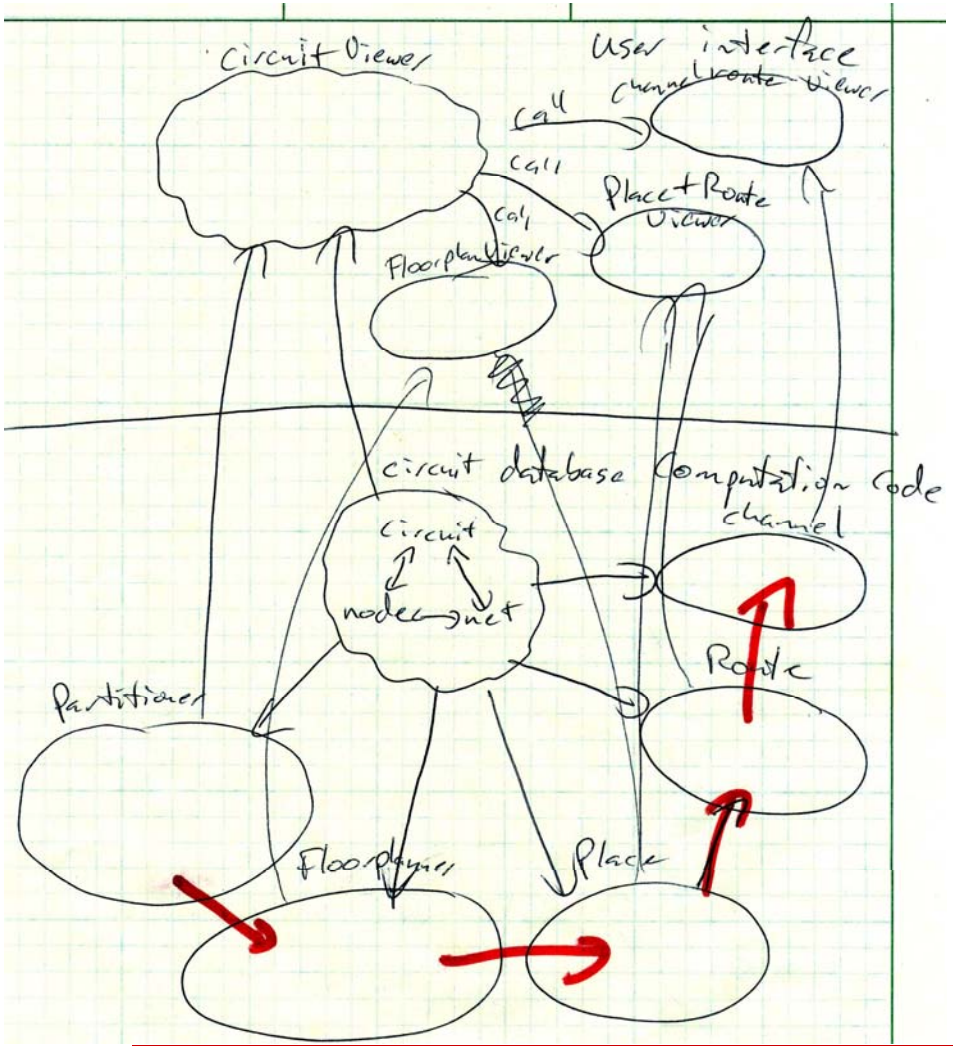


- Ownership [Noble, Vitek, Potter '98]
  - Shows which objects are part of the representation of others
  - Many contributors: Clarke, Boyland, Boyapati, Drossopoulou, Smith, Müller, Dietl, Wrigstad, Lu, …
- Challenges:
  - Structure within objects
  - Certain design patterns

- New idea: Ownership Domains [ECOOP '04, PLDI '05]
  - Declare multiple ***domains*** per object [Clarke '01]
    - Represents a group of objects
    - Adds flexibility, expressiveness
    - Maintains strong architectural reasoning
  - Public domains (listeners)
    - Hold interface objects
    - Accessible from outside
  - Private domains (owned)
    - Encapsulated
    - Accessible from public domain

Case Study: Aphyds

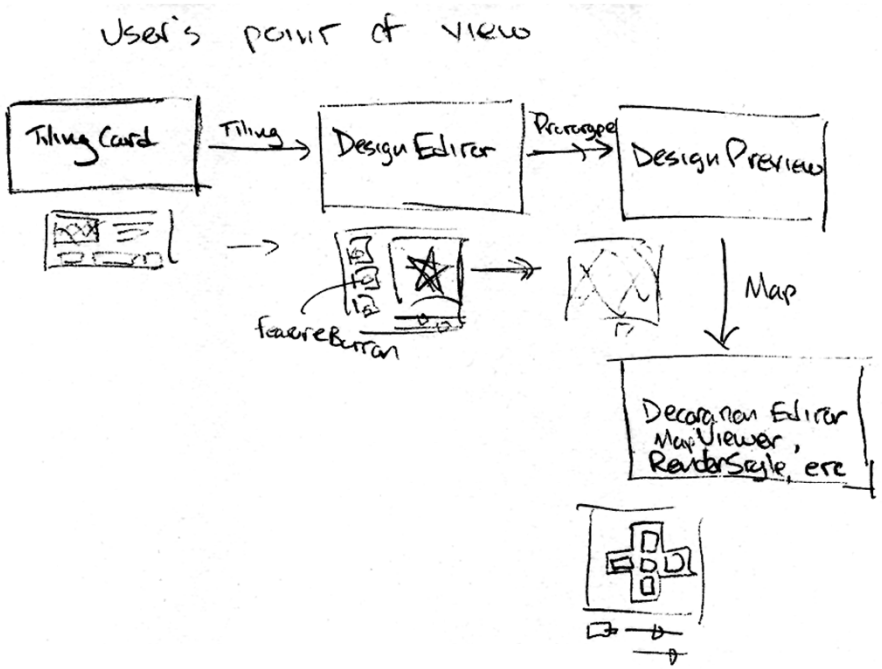
with Craig Chambers and David Notkin [ICSE '02]



- Circuit layout app.
  - 12 kLOC
- Expressed in ArchJava
  - 3 days effort
- Benefits observed
  - Discourages coupling
  - Aids understanding
  - Aids defect repair

Case Study: Taprats

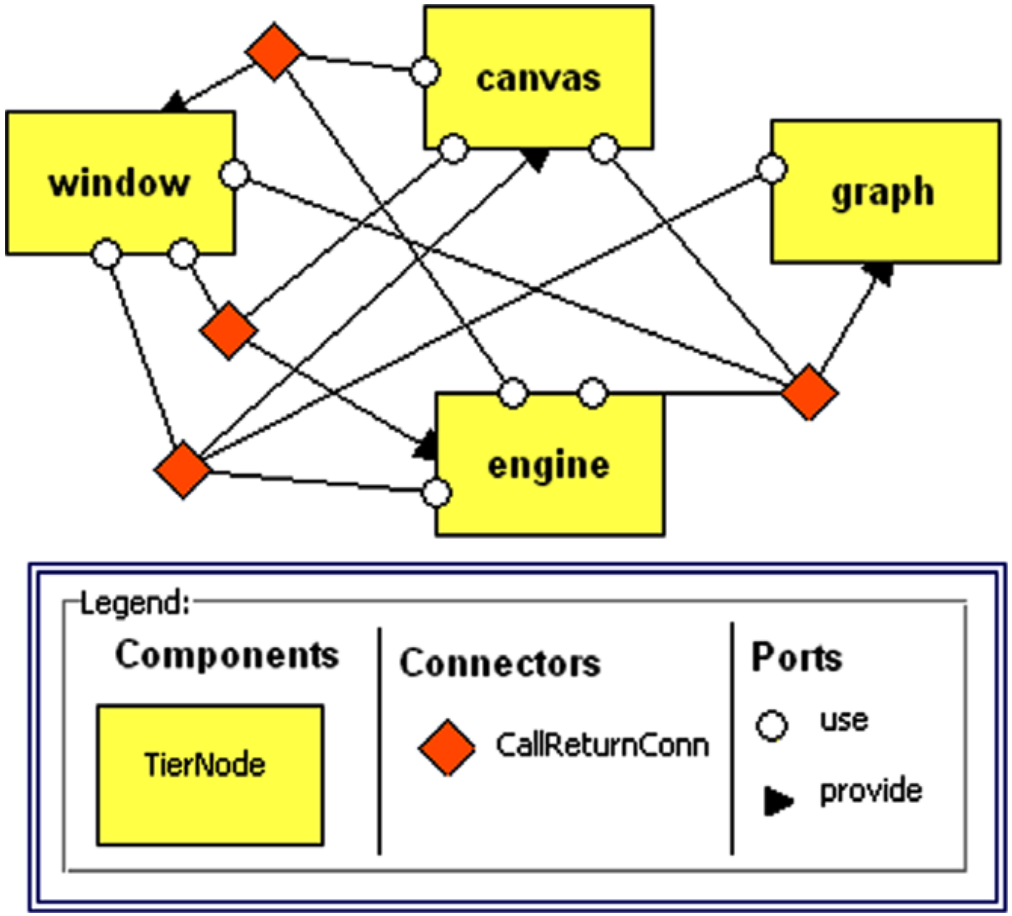
with Craig Chambers and David Notkin [ECOOP '02]



- Islamic tile design app.
  - 12 kLOC
- Expressed in ArchJava
  - Control architecture only
  - 5 hours effort
- Observations
  - Reduced coupling
  - Expressed dynamic structure

Case Study: HillClimber

with Marwan Abi-Antoun and Wesley Coelho [JSS '07]



- Optimization app.
  - 16 kLOC
- Expressed in ArchJava
- Suggestions for improvements in ArchJava

ArchJava Summary

- Advantages
  - Naturally express architecture within implementation
    - Write code = specify architecture
  - Immediate benefits
    - Improve code structure, better understanding, facilitates evolution
  - Maintain architectural consistency over time
    - Facilitates engineer ⇔ architect communication
  - Works with OO designs
    - Flexible model allows interface object aliasing
  - Validated in theory and practice
    - Proof of conformance property
    - 3 case studies show benefits
- Limitations
  - Some dynamic idioms awkward
  - New language and toolset
  - Cost for converting legacy applications
- Papers, open source implementation
  - http://www.archjava.org/

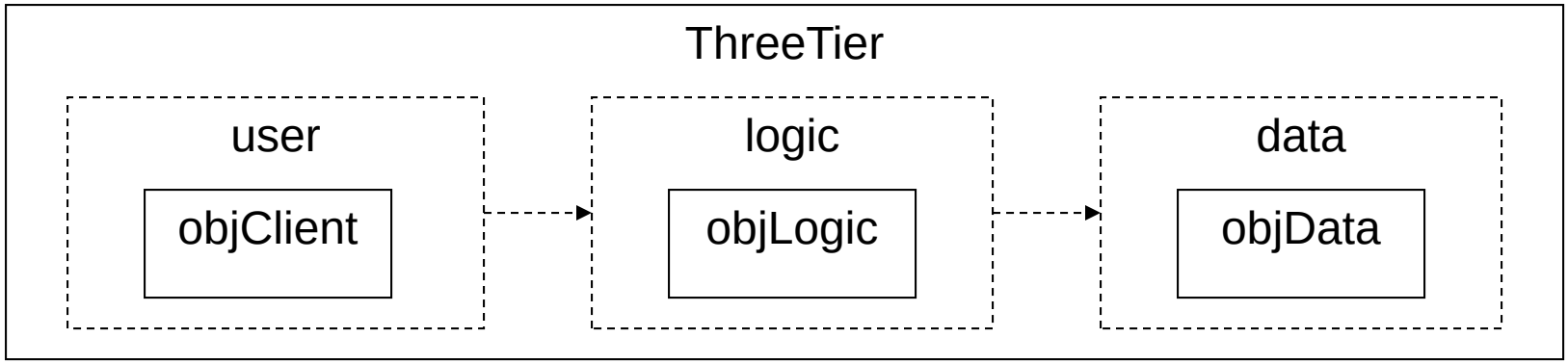
Ownership Object Graphs

with Marwan Abi-Antoun

- ArchJava, v2.0
  - No language change: just annotate code
  - Support more object-oriented designs
  - Reduce cost

- 5-year vision: a big picture you can trust
  - Summarize million LOC systems in 1 page
  - Zoom in to any level of detail
    - Look inside a top-level component
  - Assure that the 1-page summary is complete
    - Communication integrity

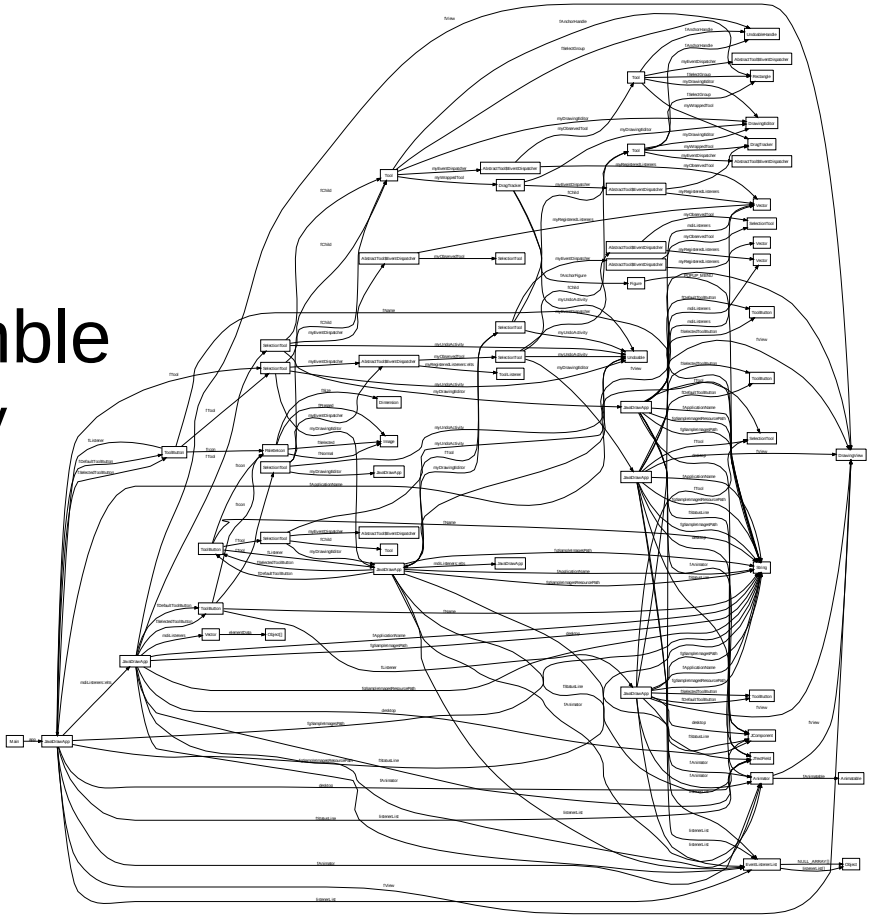
Architectural Ownership Declarations




```
@Domains ({"user", "logic", "data"})
@DomainLinks ({"user->logic", "logic->data"})
public class ThreeTier {
    private @Domain("data") Data objData;
    private @Domain("logic") Logic objLogic;
    private @Domain("user “) Client objClient;
    . . .
}
```

Declarations are simplified

Case Study: JHotDraw



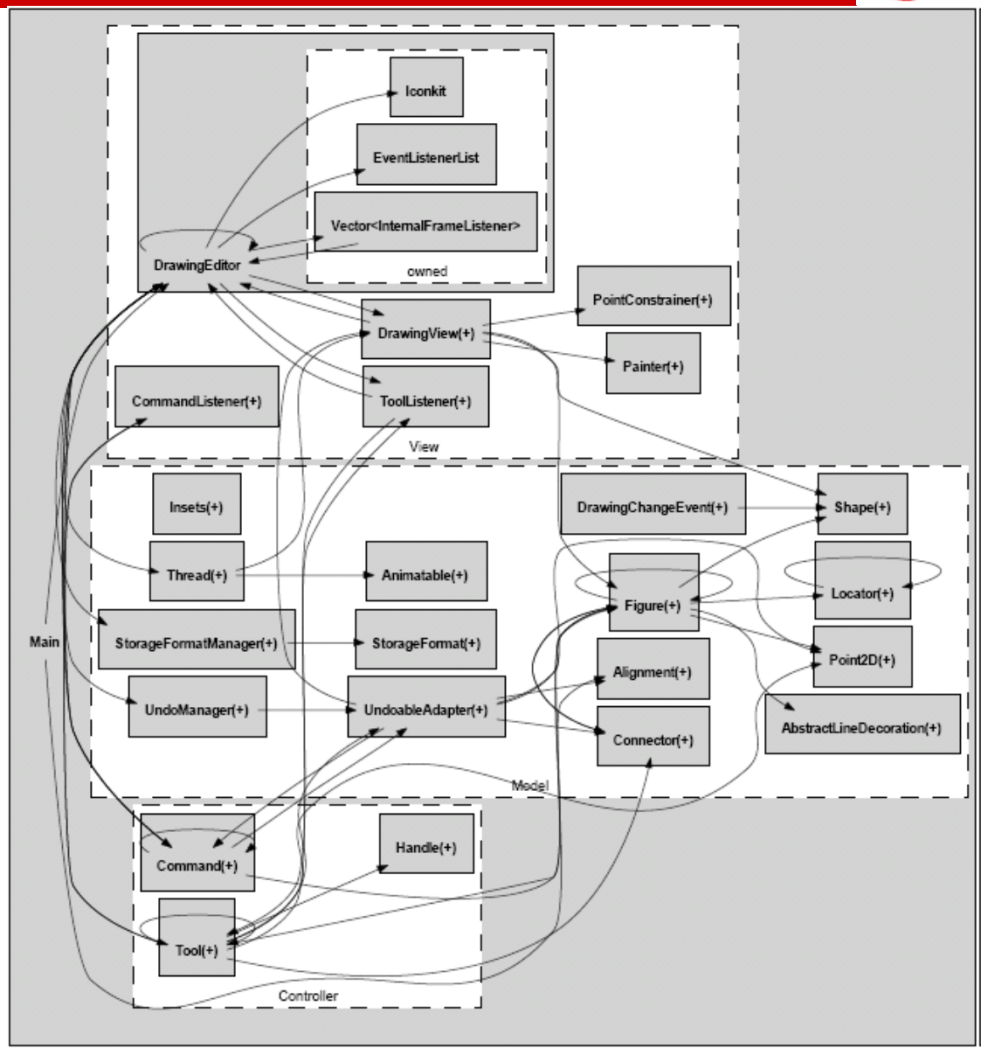
- JHotDraw v. 5.3
  - 195 classes
  - 15,000 LOC
- Object graph from Womble
  - Shows details effectively
  - Does not scale to high-level view
  - Unsound

Output of Womble (Jackson and Waingold [JW01])

Case Study: JHotDraw

- Live Demonstration

Case Study: JHotDraw



- Architectural structure
  - Shows object relationships
  - Displays Model-View-Controller design intent
  - Fits on a page
- Characteristics
  - Hierarchical
    - Top level objects are summaries
    - Can zoom in to details
  - Sound
    - Represents all objects
    - Shows all field links

Ongoing Work

- Increasing expressiveness
  - Existential types [Clarke ’01] [Krishnaswami & Aldrich ’05] [Lu and Potter ’06] [Dietl et al. ’07]
- Extract richer architectural information
  - More edge types
  - Node arities
- Reducing cost with inference
  - Builds on recent work [Aldrich et al. ’03] [Liu & Milanova ’07] [Ma & Foster ’07] [Dietl & Müller ’07]
- Prove soundness of architectural extraction
- Case studies to show benefits in practice

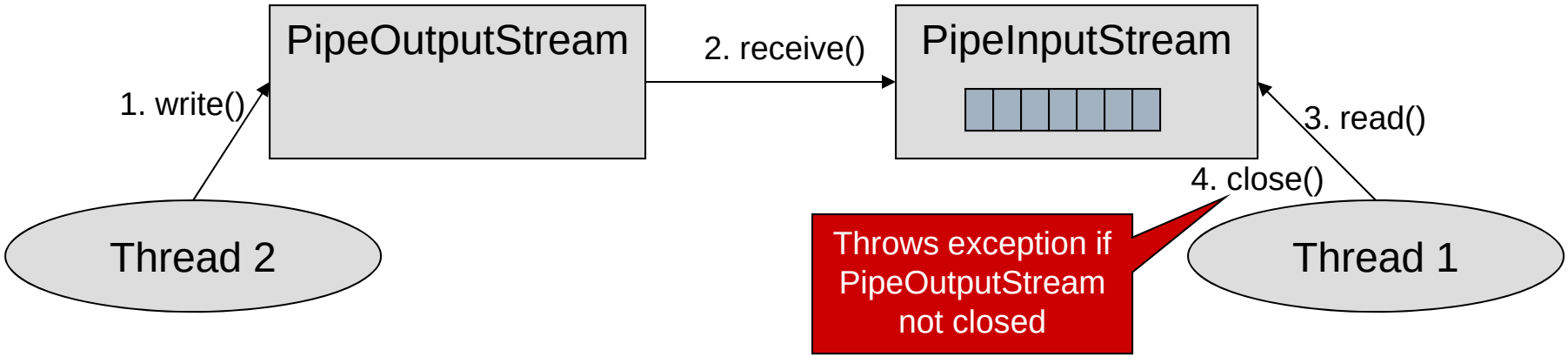
Assuring Architectural Behavior

Architectural Behavior

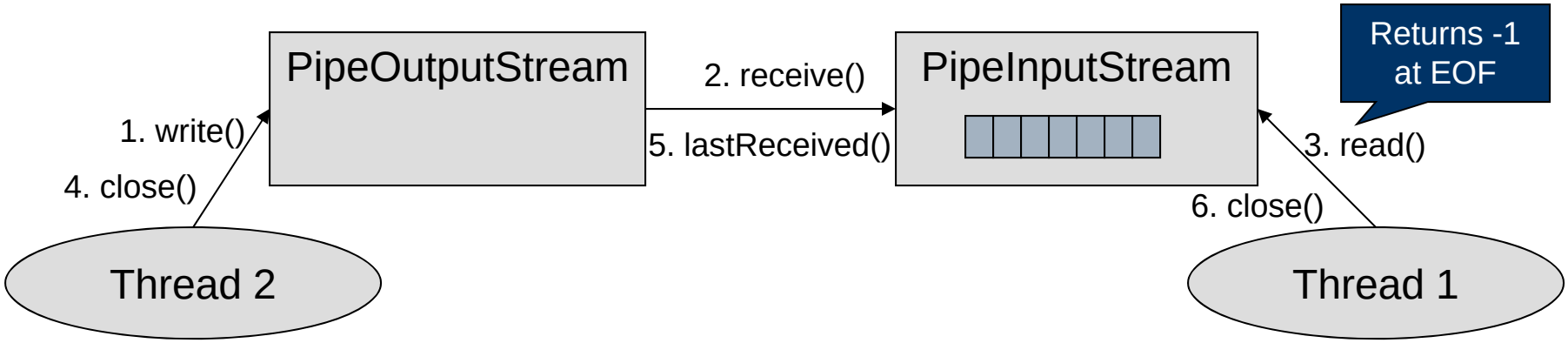
with Kevin Bierhoff

- Source of many defects
  - Incorrect assumptions about how a component is used
- Pioneering work
  - Wright [Allen and Garlan, '97]
    - Formalize component protocols using CSP
    - Model-check for consistency
    - Found defects in simulation and JavaBeans specs
  - Model checking protocols in C
    - examples: [Chaki et al. '03], [Giannakopoulou et al '04]
  - Typestates for Objects
    - [DeLine et al.][Fink et al.][Lam et al.]
- Open problems
  - Tracking protocols of aliased objects
  - Protocols for reentrant methods
  - Supporting object-oriented idioms
- Examples: Pipes and Streams

Pipes in Java

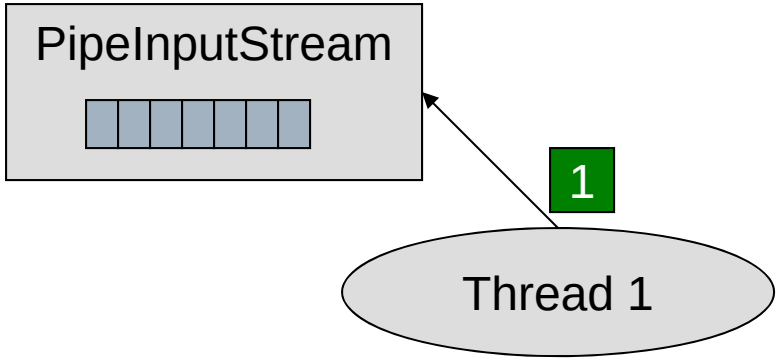


Pipes in Java



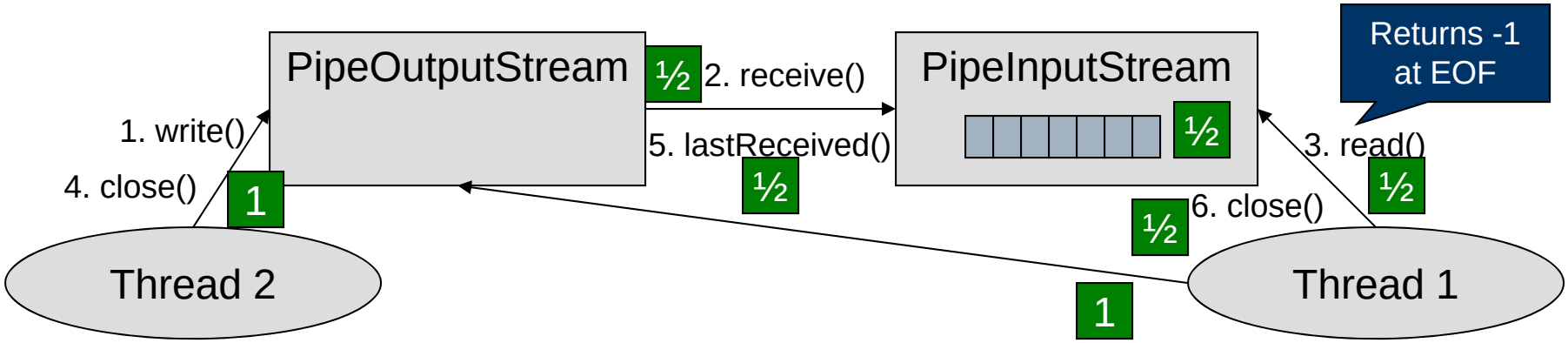
- Key intuition
  - Thread 1 and Thread 2 share the pipe
  - Thread 1 can't close until Thread 2 gives Thread 1 the permission to do so
  - This occurs through close() -> lastReceived() -> read() returning -1
- Idea: apply fractional permissions [Boyland '03]
  - Can split permission for an object in half, giving part to each thread
  - A half permission allows reads/writes, but not closes
  - Permissions can later be recombined through the lastReceived() call, allowing the close

Pipes in Java



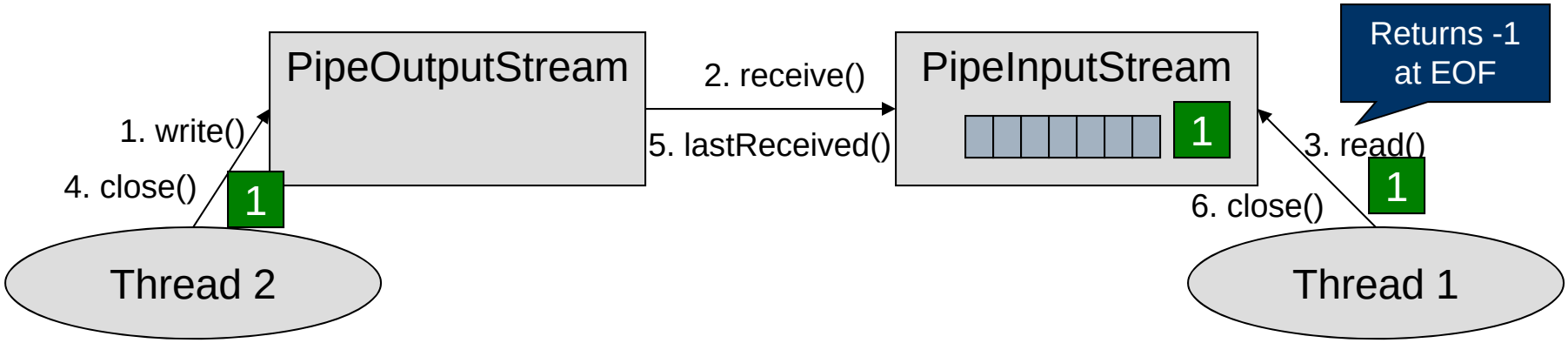
- Key intuition
  - Thread 1 and Thread 2 share the pipe
  - Thread 1 can't close until Thread 2 gives Thread 1 the permission to do so
  - This occurs through close() -> lastReceived() -> read() returning -1
- Idea: apply fractional permissions [Boyland '03]
  - Can split permission for an object in half, giving part to each thread
  - A half permission allows reads/writes, but not closes
  - Permissions can later be recombined through the lastReceived() call, allowing the close

Pipes in Java



- Key intuition
  - Thread 1 and Thread 2 share the pipe
  - Thread 1 can't close until Thread 2 gives Thread 1 the permission to do so
  - This occurs through close() -> lastReceived() -> read() returning -1
- Idea: apply fractional permissions [Boyland '03]
  - Can split permission for an object in half, giving part to each thread
  - A half permission allows reads/writes, but not closes
  - Permissions can later be recombined through the lastReceived() call , allowing the close

Pipes in Java



- Key intuition
  - Thread 1 and Thread 2 share the pipe
  - Thread 1 can't close until Thread 2 gives Thread 1 the permission to do so
  - This occurs through close() -> lastReceived() -> read() returning -1
- Idea: apply fractional permissions [Boyland '03]
  - Can split permission for an object in half, giving part to each thread
  - A half permission allows reads/writes, but not closes
  - Permissions can later be recombined through the lastReceived() call, allowing the close
- Contribution: better support for OO designs
  - Allow writes through two pointers
    - e.g. Thread 1 and Thread 2 are still changing PipeInputStream's state, but they can't close it until the permissions are combined

Pipes Analysis

- Verification challenges
  - Aliases to the pipe in 2 threads
  - Data mutated through both aliases

- Cooperative Permissions
  - Naturally ***express design intent***
    - Who's allowed to do what
    - How permissions are transferred
  - ***Immediate benefits*** and assurance as code evolves
    - Catch incorrect use of architectural pipe connector
  - Support ***OO designs***
    - Allowing shared mutable data

BufferedInputStream Defect

- Live Demonstration

BufferedInputStream Defect

```
int read() {
        if (pos >= count) {
                fill();
                if (pos >= count) {
                        return -1;
        }
        return buf[pos++];
}

void fill() {
        pos = 0;
        count = pos;
        int n = in.read(buf, pos, buf.length - pos);
        if (n > 0)
                count = n + pos;
}
```

1. Initially pos = count = 0

2. Call fill(…)

6. pos < count here, so if count = 8192, we should not get an array bounds exception

What does the debugger say?

3. Sets pos = count = 0

4. Reads n = 8192 chars

5. What is count? 8192, right?

BufferedInputStream Defect

```
int read() {
    if (pos >= count) {
        fill();
        if (pos >= count) {
            return -1;
    }
    return buf[pos++];
}

void fill() {
    pos = 0;
    count = pos;
    int n = in.read(buf, pos, buf.length - pos);
    if (n > 0)
        count = n + pos;
}
```

1. Initially pos = count = 0

4(a). pos = count = 0, still

2. Call fill(…)

4(b). Call fill(…) recursively

4(f). Sets pos = 1

Eventually pos = 8192: Array out of bounds!

3. Sets pos = count = 0

4(c). Sets pos = count = 0

4. Calls in.read(…)
Results in recursive call!

4(d). Reads n = 8192 chars

4(g). Reads n = 8192 chars

4(e). Sets count = 8192

5. Now pos = 1 so count = 8193!

BufferedInputStream Invariant

```
int read() {
        if (pos >= count) {
                fill();
                if (pos >= count) {
                        return -1;
        }
        return buf[pos++];
}

void fill() {
        pos = 0;
        count = pos;
        int n = in.read(buf, pos, buf.length - pos);
        if (n > 0)
                count = n + pos;
}
```

Invariant: pos should not change in read(…) call

BufferedInputStream Analyzed

- Is it a major defect?
  - No; this is unlikely to occur in practice
  - But documentation does not forbid recursive use
  - Furthermore, a library should not throw an internal exception
    - Anyone here from Sun? Read our defect report!
- ***Don't ignore recursion!*** ***[Lu et al., ECOOP '07]***
  - Ubiquitous in object-oriented code
  - Potential source of very subtle defects
- How to verify?
  - Forbidding recursion
    - Exclusive-write permission in our system or [DeLine et al. '04]
    - Omit the stream from the modifies clause in Spec#
    - Use ownership [Müller '02][Lu et al. ECOOP '07]
  - Allowing recursion
    - Our system: use shared permissions, re-set the necessary fields

Verifying BufferedInputStream

```
/*  requires full permission on this */
int read() {
        if (pos >= count) {
                fill();
                if (pos >= count) {
                        return -1;
        }
        return buf[pos++];
}

/*  requires full permission on this */
void fill() {
        pos = 0;
        count = pos;

        int n = in.read(buf, pos, buf.length - pos);

        if (n > 0)
                count = n + pos;
}
```

We have full permission, so no one else can modify pos. The recursive call is forbidden.

Technically there's a pack-unpack pair surrounding the call.

Verifying BufferedInputStream

```
/*  requires shared permission on this */
int read() {
        if (pos >= count) {
                fill();
                if (pos >= count) {
                        return -1;
        }
        return buf[pos++];
}

/*  requires shared permission on this */
void fill() {
        pos = 0;
        count = pos;
        int oldpos = pos;
        int n = in.read(buf, pos, buf.length - pos);
        pos = oldpos;
        if (n > 0)
                count = n + pos;
}
```

With a shared permission, anyone else could modify the fields. So we need to reset pos to the value we need. The recursive call is allowed.

Related Paper

- Tracking Linear and Affine Resources with Java(X)
  - Tracks object typestate
  - Relaxes previous work that required typestate uniqueness

- Today at 14:00, by Markus Degen, Peter Thiemann, and Stefan Wehr

Scaling Cooperative Permissions Up

- No implementation yet
  - Examples worked by hand suggest the approach can handle new constructs
  - Still challenges with decidability, concurrency, practical design
- Ultimate goal: scale to architecture
  - First step: leverage structure to check behavior
    - Specify protocols in ArchJava ports
    - Interprocedural analysis checks code against protocol
    - Model checker verifies protocols in architecture are compatible
      - Following [Allen and Garlan, '97]
    - Technical advances in modular checking and handling recursion
      - But limited to static architectures
    - Bierhoff et al., April 2006 tech report (work ongoing)
  - Cooperative Permissions could be used to check architectural protocols without the limitations of ArchJava

Cooperative Permissions - Summary

- Lightweight verification of architectural protocols
  - Naturally express coordination among pointers
    - Follows engineering intuition
  - Immediate benefits and consistency over time
    - Assure correct usage of libraries, frameworks; find defects
  - Works with OO designs
    - Supports aliasing, recursion, and inheritance
  - Proved sound, validated on small but real examples
    - For more, come to OOPSLA 2007!

Assuring Object-Oriented Architecture

- High-level design constraints on the structure and interaction of groups of objects
  - Abstraction scales beyond design patterns
- Emerging assurance approaches
  - Architectural structure
  - Behavioral protocols
- Nearing major impact on practice
  - Natural expression of architectural intent
  - Provide immediate value and maintain over time
  - Work with object-oriented designs
    - Inheritance, recursion, state, aliasing

Thanks

- Dahl-Nygaard Prize Committee
- My advisors
  - Craig Chambers
  - David Notkin
- My students
  - Donna Malayeri
  - Neel Krishnaswami
  - Marwan Abi-Antoun
  - Kevin Bierhoff
  - Nels Beckman
  - Ciera Jaspan
- Mentors (a few of many!)
  - My parents
  - Ron Tennison
  - Rusty Whitney
  - Lougie Anderson
  - Mani Chandy
  - Susan Eggers
  - Doug Lea
  - Gary Leavens
  - Gail Murphy
  - David Garlan
  - Bill Scherlis
  - Frank Pfenning

Questions?

Object-Oriented Architecture Principles

- Expressing
  - Instances matter
  - Hierarchy is important for scale
  - Capture engineering intuition
- Checking
  - Allow aliasing
  - Allow recursion
  - Support object-oriented abstractions

We are approaching a revolution in our ability to express and verify the architectural properties on which object-oriented software depends