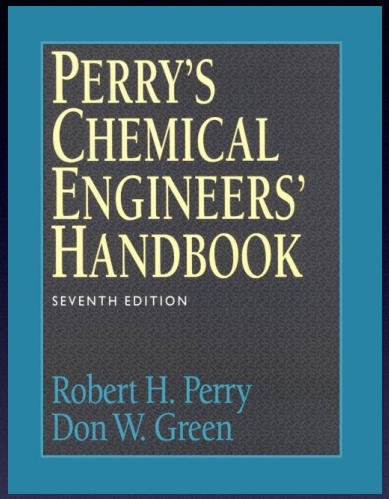
Lecture 2

Software Architecture

Today's Agenda

- Software Architecture
  - Introduction to Software Architecture
  - ADL
  - ArchJava

Example handbook



- Contents
  - Chemical and physical property data
  - Fundamentals (e.g. thermodynamics)
  - Processes (the bulk of the book)
    - heat transfer operations
    - distillation
    - kinetics
    - liquid-liquid
    - liquid-solid
    - etc.
  - Materials of construction
  - Waste management
  - Process safety

Slide from Rob DeLine @ Microsoft Research

Other Precedents

- Polya's *How to Solve It*
  - Catalogs techniques for solving mathematical (geometry) problems
  - Two categories: problems to prove, problems to find/construct
- Christopher Alexander's books, e.g. *A Pattern Language*
  - Saw building architecture/urban design as recurring patterns
  - Gives 253 patterns as: name; example; context; problem; solution
- **Pattern languages as engineering handbooks**
  - Hype aside, it's about recording known solutions to problems
  - Pattern languages exist for many problems, but we'll look at design
  - Best known: Gamma, Helm, Johnson, Vlissides ("Gang of four")
    Design Patterns: Elements of reusable object-oriented software
  - Notice the subtitle: here, design is about objects and their interactions

Slide from Rob DeLine @ Microsoft Research

Today's Lecture on Software Architecture

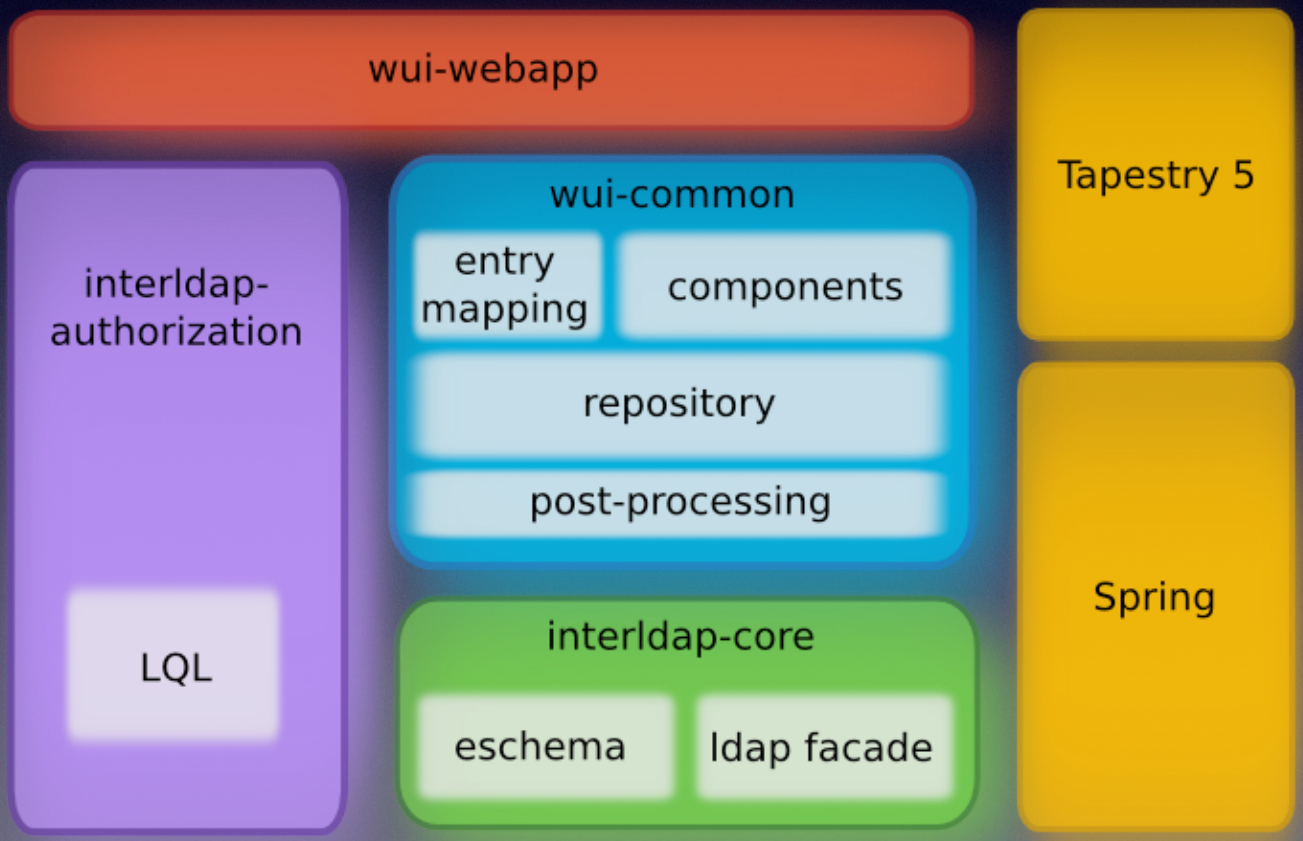
- We read the software architecture paper by David Garlan and Mary Shaw at CMU.
- Around the same time (or before), Alexander Wolf and Dewayne Perry (back then they were at Bell Lab) also wrote a paper on the idea of software architecture.
- Some of today's slides are borrowed from Rob DeLine at Microsoft Research, who did his Ph.D under the supervision of Mary Shaw at CMU.
- Some of today's slides are borrowed from Vibha Sazawal at UMD, who worked with Jonathan Aldrich at CMU, a creator of ArchJava.

Why we care about Software Architecture?

- Recognize common paradigms so that high-level relationships among systems can be understood.
  => New systems can be built as variations
- Getting the architecture wrong can lead to disastrous results. (Really?)
- Help make principled choices among design alternatives.
- Essential to understanding the high-level properties of complex systems

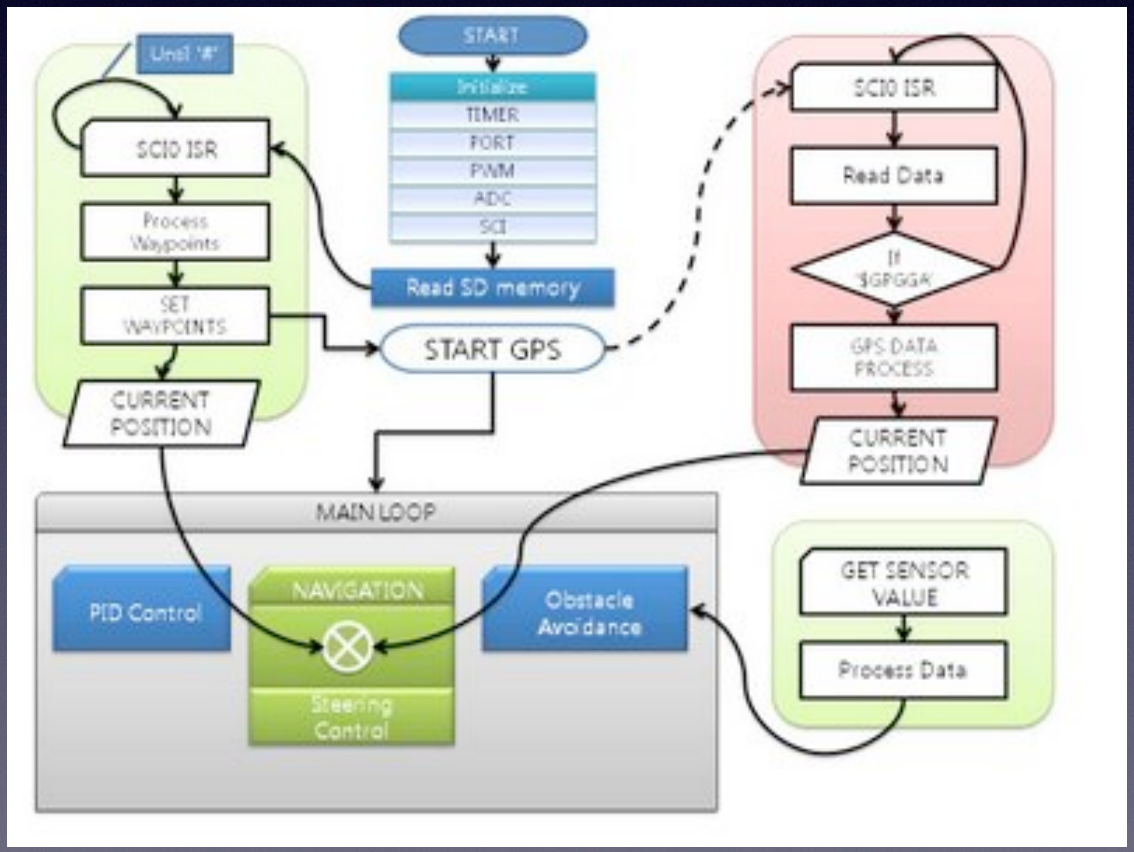
What is a software architecture?

- According to Google Images



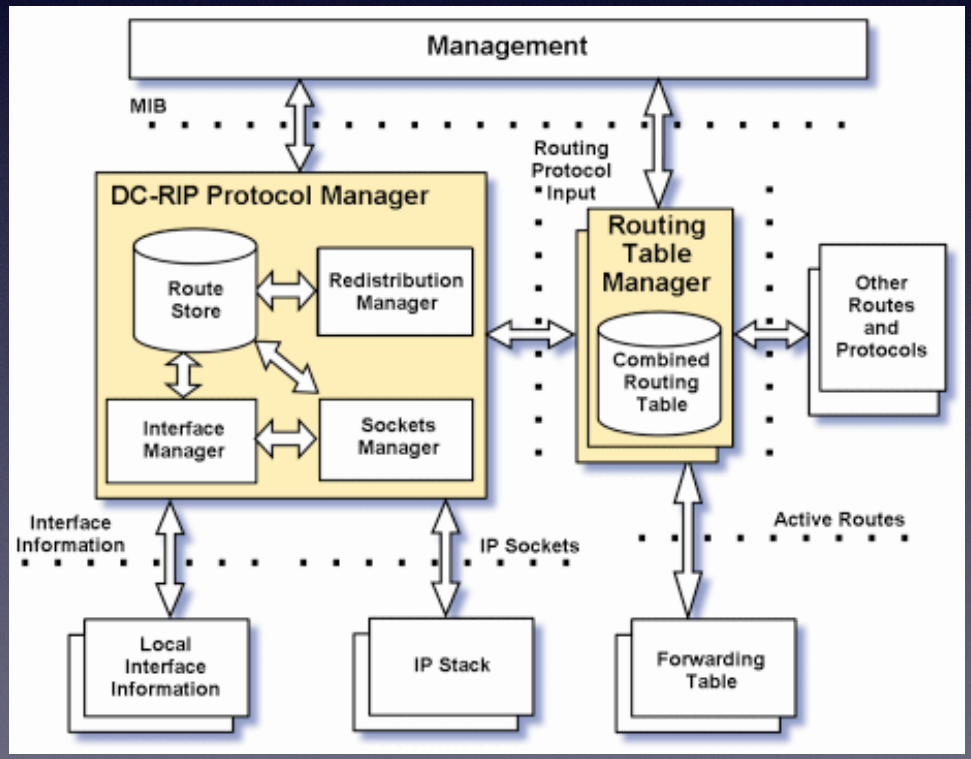
What is a software architecture?

- According to Google Images



What is a software architecture?

- According to Google Images



What is software architecture?

- CMU-SEI definition
  - software elements, the externally visible properties of those elements and the relationships among them

What do these figures mean?

- Boxes
- Lines
- Grouping

What do these figures mean?

- Boxes => Component
- Lines => Connections
- Grouping, backgrounds, fences => Composition

Components (boxes)

- Places where computation takes place
- Places where data is stored
- Box shapes distinguish component types

Connections (lines, arrows)

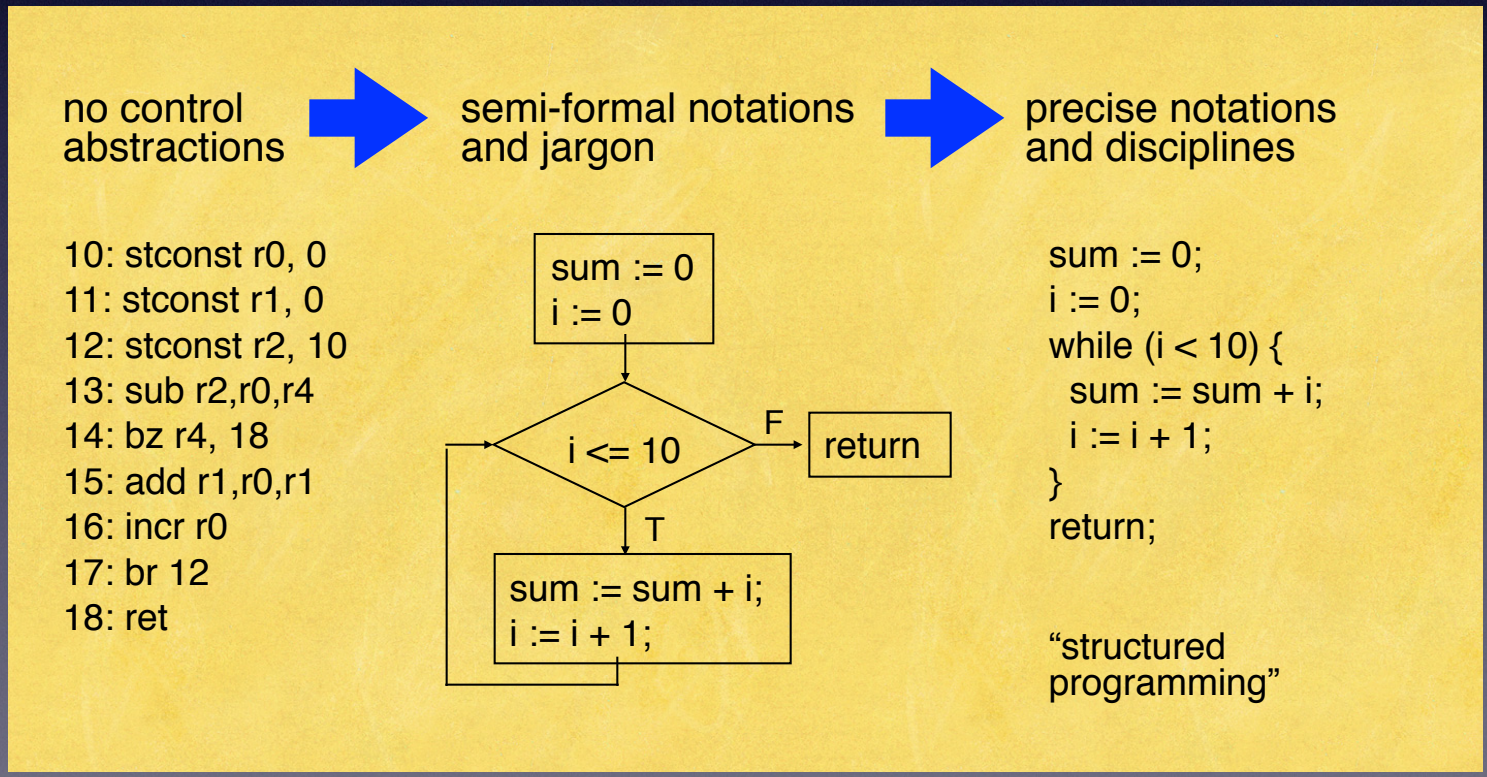
- Some kind of interaction among components
- Often binary, sometimes n-ary
- Line attributes distinguish connection types

Composition (grouping, backgrounds, fences)

- Show commonality and boundaries

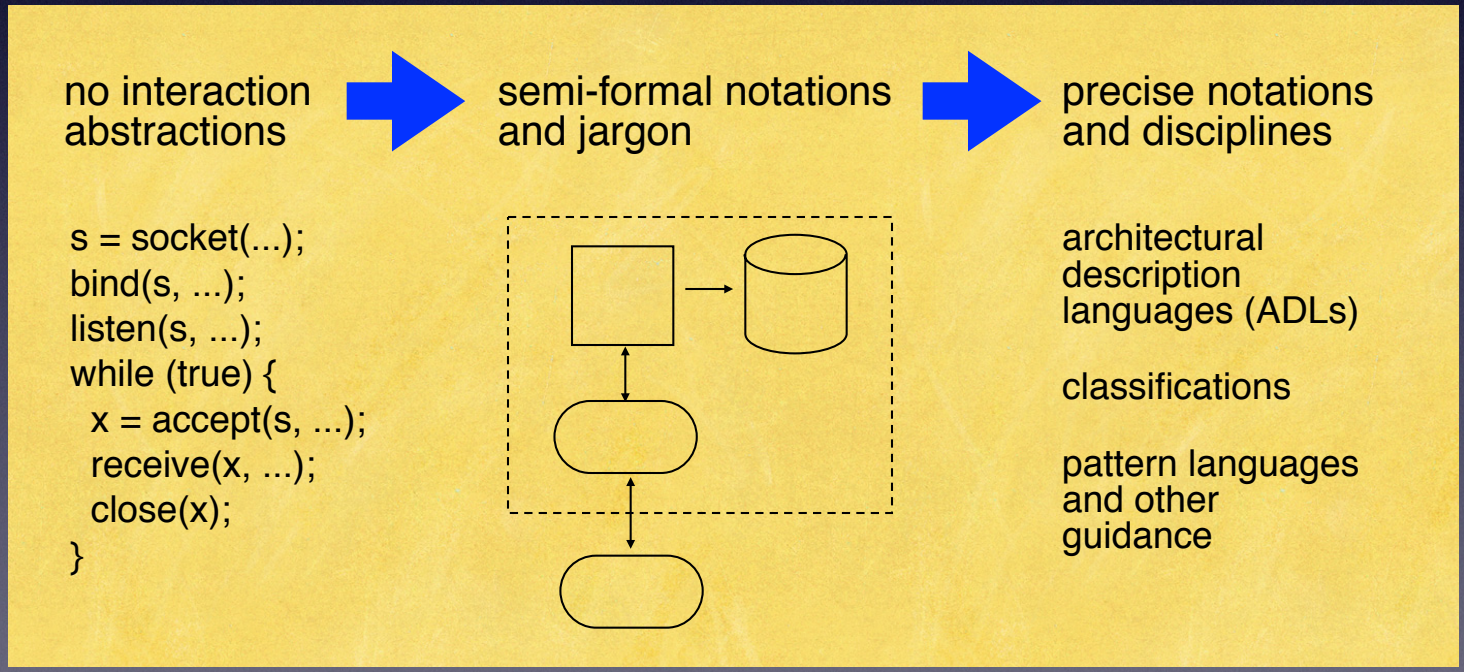
Carving out a new level of abstraction

- In the early age of programming languages...



Architecture as a new abstraction

- Researchers are carving out a higher-level abstraction



Think-Pair-Share

- What are limitations of such de-facto, state-of-the practice architecture description method?

- What are limitations of such de-facto, state-of-the practice architecture description method?

- Lack of traceability -> Less helpful debugging
- Abstraction is a double edge sword— not knowing implementation underneath
- Lack of details, not concrete enough to guide me with further development
- lines / interaction semantics are blurry.

Software Architecture Styles

- Pipe and filter
- Client and server
- Object oriented
- Publish and subscribe
- Layers
- Microkernel
- Web services

Example Architectural Styles

- 1. Pipe and Filter
- 2. Object Oriented
- 3. Event-based Implicit Invocation
- 4. Layered Systems

Style Example 1: Pipe and Filter

- A filter reads streams of data on its inputs and produces streams of data on its outputs by applying a local transformation.
- Component (Filter)
- Connector (Pipe)

What are the constraints on Pipe and Filter?

What are the constraints on Pipe and Filter?

- Constraints
  - filters must be independent => no shared states among filters
  - filters do not know the identity of other filters
  - outputs are the same regardless of ordering of filters

Common Specializations

- pipeline: linear sequences of filters
- bounded pipes: restrict the amount of data that can reside on a pipe
- typed pipes: the data passed between two filters have a well-defined type

Example: Pipe and Filter

- Advantages:
  - programmers can understand the overall input and output behavior as a simple composition of filters
  - reuse: any two filters can be hooked together
  - different types of filters can be easily added or deleted
- Disadvantages:
  - not good for interactive applications as each filter provides a complete transformation of input data to output data
  - each filter has to parse and unparse the data

Pipeline Architecture

- a linear sequence of filters
- e.g. a compiler architecture

Style Example 2. Object-Oriented Organization

- Objects are components, responsible for preserving the integrity of a resource
- Objects interact through function invocations.
- Advantages: Hide its representation from clients
- Disadvantages:
  - must know the identity of that other object (e.g., need to change the import list when an object is renamed.)
  - When A uses B, C uses B, C's effect on B look like unexpected side effects to A.

Style Example 3: Event-based, Implicit Invocation

- Instead of invoking a procedure directly, a component can announce or broadcast one or more events.
- Other components in the system can register an interest in an event by associating a procedure with the event.
- e.g. Java Swing GUI
- Component: modules whose interfaces provide both a collection of procedures and a set of events
- Connector: traditional procedure calls as well as bindings between event announcements and procedure calls

Example: Event-based, Implicit Invocation

- Constraints
  - Announcers of events do not know which components will be affected by those events
  - Components cannot make assumptions about order of processing
- Advantages
  - Any components can be introduced into a system by registering for the events
- Disadvantages
  - Component relinquish control over the computation performed by the system
  - Ordering is difficult to understand, difficult to expect when finished
  - Shared event data

Style Example 4. Layered Systems

- Each layer providing service to the layer above it and serving as a client to the layer below
- e.g. Operating systems.

- Advantages:
  - design based on increasing levels of abstraction
  - support enhancement— changes to the function of one layer affect at most two layers
  - support reuse — different implementations of the same layer can be used interchangeably

- Disadvantage
  - Not all systems are easily structured in a layered fashion
  - consideration of performance may bypass layers, resulting on closer coupling

Piazza Questions

Architecture Description Languages (ADL)

- In the 90s, researchers created many architectural notations.
  - grew out of module interconnection languages (1975)
  - focus on recording system structure (typically static structure)
  - different goals, but many shared concepts

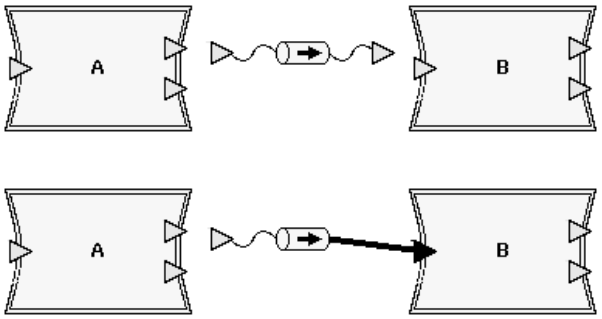
Common Concepts in ADL

- Components (computation)
- Connectors (common disagreement: aren't these just components?)
- Compositions (combinations of elements to form new elements)
- Architectural Styles (constraints on elements and their composition)

UniCon

Focus on encapsulating complex construction rules

- Editor lets you drag-and-drop elements and hook them up



- Given a system description, UniCon's compiler produces
  - low-level interaction code
  - build instructions (makefile) that invokes needed tools

Shaw, DeLine, Klein, Ross, Young and Zelesnik, "Abstractions for software architectures and tools to support them", Trans. on Soft. Eng. 21(4):314-335.

Wright

- Focus on making interaction formal
  - components interact through ports
  - connectors interact through roles
  - attachments are made by binding ports to roles
  - ports and roles are formally defined as CSP (communicating sequential processes).
  - i.e., a ***process*** description language for defining connector types as a protocol of interaction of components
  - what is a process? a "thing" that engages in communication/ interaction events in a sequence. an event can have associated data.

Allen & Garlan, "Formalizing architectural connection", ICSE 1994

Wright Component Description Example

```
component Split =
    port In = read?x -> In [] read-eof → close → ✓
    port Left, Right = write!x → Out ┌┐ close → ✓
    comp spec =
        let Close = In.close → Left.close → Right.close → ✓
        in      Close []
            In.read?x → Left.write!x →
                (Close [] In.read?x → Right.write!x → computation)
```

Component type is described as a component-specs plus a set of ports

Wright Connector Description Example

connector Pipe =
role Writer = write!x → Writer ┌┐ close → ✓
role Reader = **let** ExitOnly = close → ✓
in let DoRead = (read?x → Reader [] read-eof → ExitOnly)
in DoRead ExitOnly
glue = **let** ReadOnly = Reader.read!y → ReadOnly
[] Reader.read-eof → Reader.close → ✓ [] Reader.close → ✓
in let WriteOnly = Writer.write?x → WriteOnly [] Writer.close → ✓
in Writer.write?x → **glue** [] Reader.read!y → **glue**
[] Writer.close → ReadOnly [] Reader.close → WriteOnly
spec ∀ Reader.read_i!y . ∃ Writer.write_j?x . i=j ∨ x=y
∧ Reader.read-eof ⇒ (Writer.close ∧ #Reader.read = #Writer.write)

Roles: obligation of each participating component.

A glue spec: protocol description (coordination among roles)

Connector type is described as a set of roles and a glue specification.

Wright Connector Description Example

connector Pipe =
role Writer = write!x → Writer ┌┐ close → ✓
role Reader = **let** ExitOnly = close → ✓
in let DoRead = (read?x → Reader [] read-eof → ExitOnly)
in DoRead ExitOnly
glue = **let** ReadOnly = Reader.read!y → ReadOnly
[] Reader.read-eof → Reader.close → ✓ [] Reader.close → ✓
in let WriteOnly = Writer.write?x → WriteOnly [] Writer.close → ✓
in Writer.write?x → **glue** [] Reader.read!y → **glue**
[] Writer.close → ReadOnly [] Reader.close → WriteOnly
spec ∀ Reader.read_i!y . ∃ Writer.write_j?x . i=j ∨ x=y
∧ Reader.read-eof ⇒ (Writer.close ∧ #Reader.read = #Writer.write)

Roles specify possible behaviors (the steps that can make up a protocol and possible ordering). Glue describes how behaviors are combined across roles.

Wright System Description

A system composes components and connectors

```
system Capitalize
   component Split = ...
   connector Pipe = ...
   ...
instances
   split: Split; p1, p2: Pipe;
attachments
   split.Left as p1.Writer;
   upper.In as p1.Reader;
   split.Right as p2.Writer;
   lower.In as p2.Reader;
   ...
end Capitalize.
```

Any Comments on ADL?

Take away message

- Software Architecture is a high-level abstraction of software design.
- A software architecture is usually specified by its components, connections, and composition mechanism.
- Active research in architecture description languages, architectural styles, and enforcing architecture at an implementation level.

- Read ArchJava paper for the next lecture.

ADL to ArchJava

- Existing ADLs decouple implementation code from architecture, allowing inconsistencies, causing confusion, violating architectural properties, and inhibiting software evolution.

ArchJava

- ArchJava is an extension to Java that seamlessly unifies software architecture with implementation.
- It also ensures that the implementation conforms to architectural constraints
- It ensures traceability between architecture and code and support the co-evolution of architecture and implementation

ArchJava: Connecting Software Architecture to Implementation, Jonathan Aldrich, Craig Chambers and David Notkin [ICSE 2002]

ArchJava Component Example

```
public component class Parser {
  public port in {
    provides void setInfo(Token symbol,
                          SymTabEntry e);
    requires Token nextToken()
                    throws ScanException;
  }
  public port out {
    provides SymTabEntry getInfo(Token t);
    requires void compile(AST ast);
  }

  void parse(String file) {
    Token tok = in.nextToken();
    AST ast = parseFile(tok);
    out.compile(ast);
  }

  AST parseFile(Token lookahead) { ... }
  void setInfo(Token t, SymTabEntry e) {...}
  SymTabEntry getInfo(Token t) { ... }
  ...
}
```

ArchJava Component Example

```
public component class Parser {
  public port in {
    provides void setInfo(Token symbol,
                          SymTabEntry e);
    requires Token nextToken()
                    throws ScanException;
  }
  public port out {
    provides SymTabEntry getInfo(Token t);
    requires void compile(AST ast);
  }

  void parse(String file) {
    Token tok = in.nextToken();
    AST ast = parseFile(tok);
    out.compile(ast);
  }

  AST parseFile(Token lookahead) { ... }
  void setInfo(Token t, SymTabEntry e) {...}
  SymTabEntry getInfo(Token t) { ... }
  ...
}
```

A ***component*** can only communicate with other components through explicitly declared ports; regular method calls between components are not allowed.

A ***port*** represents a logical communication channel between a component and other components that it is connected to.

ArchJava Component Example

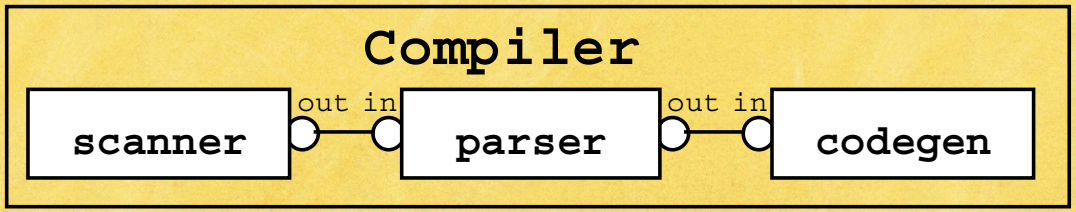
```
public component class Parser {
  public port in {
    provides void setInfo(Token symbol,
                          SymTabEntry e);
    requires Token nextToken()
                  throws ScanException;
  }
  public port out {
    provides SymTabEntry getInfo(Token t);
    requires void compile(AST ast);
  }

  void parse(String file) {
    Token tok = in.nextToken();
    AST ast = parseFile(tok);
    out.compile(ast);
  }

  AST parseFile(Token lookahead) { ... }
  void setInfo(Token t, SymTabEntry e) {...}
  SymTabEntry getInfo(Token t) { ... }
  ...
}
```

provides: a provided method is implemented by the component and is available to be called by other components connected to this port.

requires: each required method is provided by some other component connected to this port.

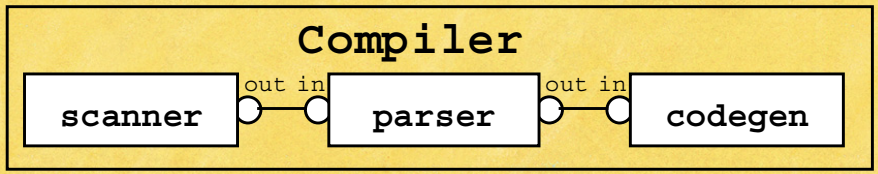
broadcasts: the same as required except that they can be connected to any number of implementations and must return void.

ArchJava Connector Example




```
public component class Compiler {
  private final Scanner scanner = ...;
  private final Parser parser = ...;
  private final CodeGen codegen = ...;

  connect scanner.out, parser.in;
  connect parser.out, codegen.in;

  public static void main(String args[]) {
    new Compiler().compile(args);
  }

  public void compile(String args[]) {
    // for each file in args do:
    ...parser.parse(file);...
  }
}
```

ArchJava Connector Example




```
public component class Compiler {
  private final Scanner scanner = ...;
  private final Parser parser = ...;
  private final CodeGen codegen = ...;

  connect scanner.out, parser.in;
  connect parser.out, codegen.in;

  public static void main(String args[]) {
    new Compiler().compile(args);
  }

  public void compile(String args[]) {
    // for each file in args do:
    ...parser.parse(file);...
  }
}
```

connect: this primitive connects two or more ports together, binding each required method to a provided method with the same name and signature. Connection consistency checks are performed to ensure that each required method is bound to a unique provided method.

ArchJava: Connector TypeChecking

- ArchJava is integrated with Java
- ArchJava makes dependencies explicit, reduces coupling, and promotes understanding of components in isolation
- ArchJava gives you a mechanism for expressing and checking connections but those connections are modeled as individual method calls

Case Study

- 10K
- Aphyd
- Jonathan — an observer/ experimenter refactored Aphyd
- Developer — who wrote Aphid
- program comprehension / refactoring / bug fix

Case Study

- Comprehension
  - Deviation from conceptual architecture
- Refactoring
  - Convert instance variables into ports and invoking methods on ports instead of instance variables
  - Communication integrity rules focused us to refactor problematic code

Take away message

- Software Architecture is a high-level abstraction of software design.
- A software architecture is usually specified by its components, connections, and composition mechanism.
- Active research in architecture description languages, architectural styles, and enforcing architecture at an implementation level.

Next Lecture: Design Patterns

- No assigned reading
- I will do a whirlwind presentation of a few design patterns that I cover in my undergraduate class.