Software Architecture
an informal introduction for
CIS501

David Schmidt
Kansas State University

www.cis.ksu.edu/~schmidt
Outline

1. Components and connectors
2. Software architectures
3. Architectural description languages
4. Domain-specific design
5. Product lines
6. Middleware
7. Aspect-oriented programming
8. Closing remarks
1. Components and connectors
Programming has evolved (from the 1960s)

- Single programmer-projects have evolved into *development teams*
- Single-component applications are now *multi-component, distributed, and concurrent*
- One-of-a-kind-systems are replaced by *system families*, specialized to a problem *domain* and solution *framework*
- Built-from-scratch systems are replaced by systems composed from *Commerical-Off-The-Shelf (COTS) components* and components *reused* from previous projects
Motivation for software architecture

We use already architectural idioms for describing the structure of complex software systems:

♦ “Camelot is based on the \textit{client-server model} and uses remote procedure calls both locally and remotely to provide communication among applications and servers.” [Spector87]

♦ “The easiest way to make the canonical sequential compiler into a concurrent compiler is to \textit{pipeline} the execution of the compiler phases over a number of processors.” [Seshadri88]

♦ “The ARC network follows the \textit{general network architecture} specified by the ISO in the Open Systems Interconnection Reference Model.” [Paulk85]

Hardware architecture

There are standardized descriptions of computer hardware architectures:

- **RISC** (reduced instruction set computer)
- **pipelined architectures**
- **multi-processor architectures**

These descriptions are well understood and successful because

(i) there are a relatively small number of design components

(ii) large-scale design is achieved by replication of design elements

In contrast, software systems use a huge number of design components and scale upwards, not by replication of existing structure, but by adding more distinct design components.

Network architecture

Again, there are standardized descriptions:

♦ **star** networks

♦ **ring** networks

♦ **manhattan street** (grid) networks

The architectures are described in terms of *nodes* and *connections*. There are only a few standard topologies.

In contrast, software systems use a wide variety of topologies.
Classical architecture

The architecture of a building is described by

♦ *multiple views*: exterior, floor plans, plumbing/wiring, ...

♦ *architectural styles*: romanesque, gothic, ...

♦ *style and engineering*: how the choice of style influences the physical design of the building

♦ *style and materials*: how the choice of style influences the materials used to construct (implement) the building.

These concepts also appear in software systems: there are

(i) *views*: control-flow, data-flow, modular structure, behavioral requirements, ...

(ii) *styles*: pipe-and-filter, object-oriented, procedural, ...

(iii) *engineering*: modules, filters, messages, events, ...

(iv) *materials*: control structures, data structures, ...
A crucial motivating concept: *connectors*

The emergence of networks, client-server systems, and OO-based GUI applications led to the conclusion that components can be *connected* in various ways.

Mary Shaw stressed this point:

*Reference:* Mary Shaw, Procedure Calls are the Assembly Language of Software Interconnections: Connectors Deserve First-Class Status. Workshop on Studies of Software Design, 1993.
Connectors are forgotten because (it appears that) there are no codes for them.

But this is because the connectors must be coded in the same language as the components, which confuses the two forms.

Different forms of low-level connection (synchronous, asynchronous, peer-to-peer, event broadcast) are fundamentally different yet are all represented as procedure (system) calls in programming language.

Connectors can (and should?) be coded in languages different from the languages in which components are coded (e.g., unix pipes).
Shaw’s philosophy

**Components** — compilation units (module, data structure, filter) — are specified by *interfaces*.

**Connectors** — “hookers-up” (RPC (Remote Procedure Call), event, pipe) — mediate communications between components and are specified by *protocols*. 

*Figure 3: Gross structure of an architecture language*
Example:

Interface **Central** is different from a Java-interface; it lists the “players” — inA, outB, linkC, Gorp, Thud, ... (connection points/ports/method invocations) — that use connectors.
2. Software Architecture
What is a software architecture? (Perry and Wolf)

A software architecture consists of

1. **elements**: processing elements (“functions”), **connectors** (“glue” — procedure calls, messages, events, shared storage cells), **data elements** (what “flows” between the processing elements)

2. **form**: properties (constraints on elements and system) and relationship (configuration, topology)

3. **rationale**: philosophy and pragmatics of the system: requirements, economics, reliability, performance

There can be “views” of the architecture from the perspective of the process elements, the data, or the connectors. The views might show static and dynamic structure.

Architectural Styles (patterns)

1. **Data-flow systems**: batch sequential, pipes and filters
2. **Call-and-return systems**: main program and subroutines, hierarchical layers, object-oriented systems
3. **Virtual machines**: interpreters, rule-based systems
4. **Independent components**: communicating systems, event systems, distributed systems
5. **Repositories (data-centered systems)**: databases, blackboards
6. and there are many others, including **hybrid** architectures

The italicized terms are the styles (e.g., *independent components*); the roman terms are architectures (e.g., communicating system). There are specific instances of the architectures (e.g., a **client-server architecture** is a distributed system). But these notions are not firm!
**Data-flow systems:** Batch-sequential and Pipe-and-filter

<table>
<thead>
<tr>
<th></th>
<th><strong>Batch sequential</strong></th>
<th><strong>Pipe and filter</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Components:</strong></td>
<td>whole program</td>
<td>filter (function)</td>
</tr>
<tr>
<td><strong>Connectors:</strong></td>
<td>conventional input-output</td>
<td>pipe (data flow)</td>
</tr>
<tr>
<td><strong>Constraints:</strong></td>
<td>components execute to completion, consuming entire input, producing entire output</td>
<td>data arrives in increments to filters</td>
</tr>
</tbody>
</table>

**Examples:** Unix shells, signal processing, multi-pass compilers

**Advantages:** easy to unplug and replace filters; interactions between components easy to analyze. **Disadvantages:** interactivity with end-user severely limited; performs as quickly as slowest component.
**Call-and-return systems:** subroutine and layered

<table>
<thead>
<tr>
<th></th>
<th><strong>Subroutine</strong></th>
<th><strong>Layered</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Components:</strong></td>
<td>subroutines (&quot;servers&quot;)</td>
<td>functions (&quot;servers&quot;)</td>
</tr>
<tr>
<td><strong>Connectors:</strong></td>
<td>parameter passing</td>
<td>protocols</td>
</tr>
<tr>
<td><strong>Constraints:</strong></td>
<td>hierarchical execution and encapsulation</td>
<td>functions within a layer invoke (API of) others at next lower layer</td>
</tr>
</tbody>
</table>

**Examples:** modular, object-oriented, N-tier systems (subroutine); communication protocols, operating systems (layered)
Advantages: hierarchical decomposition of solution; limits range of interactions between components, simplifying correctness reasoning; each layer defines a virtual machine; supports portability (by replacing lowest-level components).

Disadvantages: components must know the identities of other components to connect to them; side effects complicate correctness reasoning (e.g., A uses C, B uses and changes C, the result is an unexpected side effect from A’s perspective; components sensitive to performance at lower levels/layers.)
Virtual machine: interpreter

**Components:** “memories” and state-machine engine

**Connectors:** fetch and store operations

**Constraints:** engine’s “execution cycle” controls the simulation of program’s execution

**Examples:** high-level programming-language interpreters, byte-code machines, virtual machines

**Advantages:** rapid prototyping **Disadvantages:** inefficient.
Repositories: databases and blackboards

Components:
- processes and database
- knowledge sources and blackboard

Connectors:
- queries and updates
- notifications and updates

Constraints:
- transactions (queries and updates) drive computation
- knowledge sources respond when enabled by the state of the blackboard. Problem is solved by cooperative computation on blackboard.

Examples: speech and pattern recognition (blackboard); syntax editors and compilers (parse tree and symbol table are repositories)
Advantages: easy to add new processes.

Disadvantages: alterations to repository affect all components.
**Independent components:** communicating processes

| Components: | processes ("tasks") |
| Connectors: | ports or buffers or RPC |
| Constraints: | processes execute in parallel and send messages (synchronously or asynchronously) |

**Example:** client-server and peer-to-peer architectures

**Advantages:** easy to add and remove processes. **Disadvantages:** difficult to reason about control flow.
**Independent components:** event systems

**Event systems**

<table>
<thead>
<tr>
<th>Components:</th>
<th>objects or processes (“threads”)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Connectors:</td>
<td>event broadcast and notification (implicit invocation)</td>
</tr>
<tr>
<td>Constraints:</td>
<td>components “register” to receive event notification; components signal events, environment notifies registered “listeners”</td>
</tr>
</tbody>
</table>

**Examples:** GUI-based systems, debuggers, syntax-directed editors, database consistency checkers
**Advantages:** easy for new listeners to register and unregister dynamically; component reuse.

**Disadvantages:** difficult to reason about control flow and to formulate system-wide invariants of correct behavior.
Three architectures for a compiler (Garlan and Shaw)

The symbol table and tree are “shared-data connectors”

The blackboard triggers incremental checking and code generation
What do we gain from using a software architecture?

1. the architecture helps us *communicate* the system’s design to the project’s stakeholders (users, managers, implementors)

2. it helps us *analyze* design decisions

3. it helps us *reuse* concepts and components in future systems
4. Architecture Description

Languages
A language for connectors: UniCon

Shaw developed a language, UniCon (Universal Connector Language), for describing connectors and components.

**Components** are specified by *interfaces*, which include

(i) type;

(ii) attributes with values that specialize the type;

(iii) players, which are the component’s connection points. Each player is itself typed.

**Connectors** are specified by *protocols*; they have

(i) type;

(ii) specific properties that specialize the type;

(iii) roles that the connector uses to mediate (make) communication between components.
Graphical depiction of an assembly of three components and four connectors:

A development tool helps the designer draw the configuration and map it to coding.

uses statements instantiate the parts composed
connect statements state how players satisfy roles
bind statements map the external interface to the internal configuration
Connectors described in UniCon:

♦ data-flow connectors (pipe)

♦ procedural connectors (procedure call, remote procedure call): pass control

♦ data-sharing connectors (data access): export and import data

♦ resource-contention connectors (RT scheduler): competition for resources

♦ aggregate connectors (PL bundler): compound connections
ArchJava: Java extended with Unicon features

♦ Each component (class) has its own interfaces (ports) that list which methods it requires and provides

♦ Connectors are coded as classes, too, and extend the basic classes, Connector, Port, Method, etc.

♦ The ArchJava run-time platform includes a run-time type checker that enforces correctness of run-time connections (e.g., RPC, TCP)

♦ There is an open-source implementation and Eclipse plug-in

♦ www.archjava.org
package pos;

public component class POS {
    private final Sales sales = new Sales();
    private final UserInterface userInterface = new UserInterface();

    connect pattern Sales.model, UserInterface.view;
    connect pattern Sales.client, Inventory.server
    with TCPConnector {
        connect(Sales sender) throws Exception {
            return connect(sender.client, Inventory.server)
            with new TCPConnector(connection, InetAddress.getByName(JDBC_SERVER_ADDRESS),
                                   JDBC_SERVER_PORT, JDBC_SERVER_NAME);
        }
    }

    public POS() {
        connect(sales.model, userInterface.view);
    }

    public void run() {
        sales.setData("Software Architecture in Practice, 2nd Edition");
    }

    public static void main (String[] args) {
        (new POS{}).run();
    }
}
package pos;

public component class Sales {
    private String data;

    public port model {
        provides String getData();
        provides void setData(String data);
        requires void updated();
    }

    public port interface client {
        requires connect() throws Exception;
        requires String executeUpdate(String statement);
    }

    public String getData() {
        return data;
    }

    public void setData(String data) {
        try {
            this.data = (new client()).executeUpdate(data);
        } catch (Exception e) {
            e.printStackTrace();
        }
        model.updated();
    }
}
package pos;
...

public component class Inventory {
    public port interface server {
        provides String executeUpdate(String statement);
    }

    public String executeUpdate(String statement) {
        return statement + " (validated)";
    }

    public Inventory () {
        try {
            TCPConnector.registerObject(this, POS.JDBC_SERVER_PORT,
                                       POS.JDBC_SERVER_NAME);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }

    public static void main(String[] args) {
        new Inventory();
    }
}

From K. M. Hansen, www.daimi.dk/~marius/teaching/ATiSA2005
public class TCPConnector extends Connector {
    // data members
    protected TCPEndpoint endpoint;
    // public interface
    public TCPConnector(InetAddress host, int port, String objName) throws IOException {
        endpoint = new TCPEndpoint(this, host, port, objName);
    }
    public Object invoke(Call call) throws Throwable {
        Method meth = call.getMethod();
        return endpoint.sendMessage(meth.getName(), meth.getParameterTypes(), call.getArguments());
    }
    public static void registerObject(Object o, int port, String objName) throws IOException {
        TCPDaemon.createDaemon(port).register(objName, o);
    }
    // interface used by TCPDaemon
    TCPConnector(TCPEndpoint endpoint, Object receiver, String portName) {
        super(new Object[] { receiver }, new String[] { portName });
        this.endpoint = endpoint;
        endpoint.setConnector(this);
    }
    Object invokeLocalMethod(String name, Type parameterTypes[], Object arguments[]) throws Throwable {
        // find method with parameters that match parameterTypes
        Method meth = findMethod(name, parameterTypes);
        return meth.invoke(arguments);
    }
}
A summary of some ADLs

From K. M. Hansen, www.daimi.dk/~marius/teaching/ATiSA2005
<table>
<thead>
<tr>
<th>Features</th>
<th>Active Specification</th>
<th>Multiple Views</th>
<th>Analysis</th>
<th>Refinement</th>
<th>Implementation Generation</th>
<th>Dynamism</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACME</td>
<td>none</td>
<td>textual; &quot;weblets&quot; in ACME-Web; architecture views in terms of high-level (template), as well as basic constructs</td>
<td>parser</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
<tr>
<td>Aesop</td>
<td>syntax-directed editor for components; visualization classes invoke specialized external editors.</td>
<td>textual and graphical; style-specific visualizations; component and connector types distinguished iconically</td>
<td>parser; style-specific compiler; type checker; cycle checker; checker for resource conflicts and scheduling feasibility</td>
<td>none</td>
<td>build tool constructs system glue code in C for pipe-and-filter style</td>
<td>none</td>
</tr>
<tr>
<td>C2</td>
<td>proactive &quot;architecting&quot; process in DRADEL; reactive, non-intrusive type checker; design critics and to-do lists in Argo.</td>
<td>textual and graphical; view of development process</td>
<td>parser; style rule checker; type checker</td>
<td>generates application skeletons which can be completed by reusing OTS components</td>
<td>class framework enables generation of C/C++, Ada, and Java code; DRADEL generates application skeletons</td>
<td>ArchStudio allows unanticipated dynamic manipulation of architectures</td>
</tr>
<tr>
<td>Darwin</td>
<td>automated addition of ports to communicating components; propagation of changes across bound ports; dialogs to specify component properties</td>
<td>textual, graphical, and hierarchical system view</td>
<td>parser; compiler; &quot;what if&quot; scenarios by instantiating parameters and dynamic components</td>
<td>compiler; primitive components are implemented in a traditional programming language</td>
<td>compiler generates C++ code</td>
<td>compilation and runtime support for constrained dynamic change of architectures (replication and conditional configuration)</td>
</tr>
<tr>
<td>MetaH</td>
<td>graphical editor requires error correction once architecture changes are applied; constrains the choice of component properties via menus</td>
<td>textual and graphical; component types distinguished iconically</td>
<td>parser; compiler; schedulability, reliability, and security analysis</td>
<td>compiler; primitive components are implemented in a traditional programming language</td>
<td>DSSA approach; compiler generates Ada code</td>
<td>none</td>
</tr>
<tr>
<td>Rapide</td>
<td>none</td>
<td>textual and graphical; visualization of execution behavior by animating simulations</td>
<td>parser; compiler; analysis via event filtering and animation; constraint checker to ensure valid mappings</td>
<td>executable simulation construction in Rapide's executable sublanguage</td>
<td>executable simulation construction in Rapide's executable sublanguage</td>
<td>compilation and runtime support for constrained dynamic change of architectures (conditional configuration)</td>
</tr>
<tr>
<td>UniCon</td>
<td>graphical editor prevents errors during design by invoking language checker</td>
<td>textual and graphical; component and connector types distinguished iconically</td>
<td>parser; compiler; schedulability analysis</td>
<td>compiler; primitive components are implemented in a traditional programming language</td>
<td>compiler generates C code</td>
<td>none</td>
</tr>
<tr>
<td>Wright</td>
<td>none</td>
<td>textual only; model checker provides a textual equivalent of CSP symbols</td>
<td>parser; model checker for type conformance of ports to roles; analysis of individual connectors for deadlock</td>
<td>none</td>
<td>none</td>
<td>none</td>
</tr>
</tbody>
</table>
So, what is an architectural description language?

It is a notation (linear or graphical) for specifying an architecture.

It should specify

♦ **structure**: components (interfaces), connectors (protocols), configuration (both static and dynamic structure)

♦ **behavior**: semantical properties of individual components and connectors, patterns of acceptable communication, global invariants,

♦ **design patterns**: global constraints that support correctness-reasoning techniques, design- and run-time tool support, and implementation.

But it is difficult to design a general-purpose architectural description language that is **elegant, expressive, and useful**.
5. Domain-specific design
Domain-specific design

If the problem domain is a standard one (e.g., flight-control or telecommunications or banking), then there are precedents to follow.

A Domain-Specific Software Architecture has

♦ a **domain**: defines the problem area domain concepts and terminology; customer requirements; scenarios; configuration models (entity-relationship, data flow, etc.)

♦ **reference requirements**: features that restrict solutions to fit the domain. (“Features” are studied shortly.) Also: platform, language, user interface, security, performance

♦ a **reference architecture**

♦ a **supporting environment/infrastructure**: tools for modelling, design, implementation, evaluation; run-time platform

♦ a **process** or **methodology** to implement the reference architecture and evaluate it.
Avionics DSSA

- **Avionics Domain Application Generation Environment**
- Layered reference architecture
  - subsystems decomposed into primitive components with standardized interfaces
  - over 40 different realms with over 350 distinct components
  - realm \( \equiv \{ x : \text{component} | ( \forall i,j)(x_i.\text{interface} = x_j.\text{interface}) \} \)
- ADAGE reference architecture model:
  - reference architecture is defined by component realms and domain-specific composition constraints
  - even simple avionics systems often require over 50 distinct components stacked 15 layers deep

from Medvidovic's course, [http://sunset.usc.edu/classes/cs578_2002](http://sunset.usc.edu/classes/cs578_2002)
Domain-specific (modelling) language (DSL)

is a modelling language specialized to a specific problem domain, e.g., telecommunications, banking, transportation.

We use a DSL to describe a problem and its solution in concepts familiar to people who work in the domain.

It might define (entity-relationship) models, ontologies (class hierarchies), scenarios, architectures, and implementations.

Example: a DSL for sensor-alarm networks: domains: sites (building, floor, hallway, room), devices (alarm, movement detector, camera, badge), people (employee, guard, police, intruder). Domain elements have features/attributes and operations. Actions can be by initiated by events — “when a movement detector detects an intruder in a room, it generates a movement-event for a camera and sends a message to a guard....”

When a DSL can generate computer implementations, it is a domain-specific programming language.
In the Unix world, these are “little languages” or “mini-languages,” designed to solve a specific class of problems. Examples are awk, make, lex, yacc, ps, and Glade (for GUI-building in X).

Other examples are Excel, HTML, XML, SQL, regular-expression notation and BNF. These are called top-down DSLs, because they are designed to implement domain concepts and nothing more. Non-programmers can use a top-down DSL to write solutions.

The bottom-up approach, called embedded or in-language DSL, starts with a dynamic-data-structure language, like Scheme or Perl or Python, and adds libraries (modules) of functions that encode domain-concepts-as-code, thus “building the language upwards towards the problem to be solved.” Experienced programmers use bottom-up DSLs to program solutions.
Tradeoffs in using (top-down) DSLs

✔ non-programmers can discuss and use the DSL
✔ the DSL supports patterns of design, implementation, and optimization
✔ fast development
✘ staff must be trained to use the DSL
✘ interaction of DSL-generated software with other software components can be difficult
✘ there is high cost in developing and maintaining a DSL

6. Software product lines
A software product line is also called a *software system family* — a collection of software products that share an architecture and components, constructed by a product line. They are inspired by the products produced by industrial assembly lines, e.g., automobiles.

The CMU Software Engineering Institute definition:

**A product line** is a set of software intensive systems that (i) share a common set of features, (ii) satisfy the needs of a particular mission, and (iii) are developed from a set of core assets in a prescribed way.

**Key issues:**

*variability:* Can we state precisely the products’ variations (*features*)?  
*guidance:* Is there a precise recipe that guides feature selection and product assembly?

Reference: www.softwareproductlines.com
produces diesel engines for trucks and heavy machinery. An engine controller has 100K-200K lines-of-code. At level of 12 engine “builds,” company switched to a product line approach:

1. defined engine controller domain
2. defined a reference architecture
3. built reusable components
4. required all teams to follow product line approach

Cummins now produces 20 basic “builds” — 1000 products total; development time dropped from 250 person/months to < 10. A new controller consists of 75% reused software.

Features and feature diagrams

are a development tool for domain-specific architectures and product lines. They help define a domain’s reference requirements and guide implementations of instances of the reference architecture.

A feature is merely a property of the domain. (Example: the features/options/choices of an automobile that you order from the factory.)

A feature diagram displays the features and guides a user in choosing features for the solution to a domain problem.

It is a form of decision tree with and-or-xor branching, and its hierarchy reflects dependencies of features as well as modification costs.
Feature diagram for assembling automobiles

Filled circles label required features; unfilled circles label optional ones. Filled arcs label xor-choices; unfilled arcs label or-choices (where at least one choice is selected).

Here is one possible outcome of “executing” the feature diagram:
Feature diagrams work well for configuring generic data structures:

Compare the diagram to the typical class-library representation of a generic list structure.

An advantage of a feature-diagram construction of a list structure over a class-library construction is that the former can generate a smaller, more efficient list structure, customized to exactly the choices of the client.
Feature diagrams are useful for both _constraining_ as well as _generating_ an architecture: the feature requirements are displayed in a feature diagram, which guides the user to generating the desired instance of the reference architecture.

Feature diagrams are an attempt at making software assembly appear similar to assembly of mass-produced products like automobiles.

In particular, feature diagrams encourage the use of _standardized, parameterized, reusable software components_.

Feature diagrams might be implemented by a tool that selects components according to feature selection. Or, they might be implemented within the structure of a _domain-specific programming language_ whose programs select and assemble features.

7. Middleware
**Middleware: a popular form of domain-specific software architecture**

*Middleware* lies between hardware and software in the design of independent-component (e.g., client-server) architectures. Middleware is also called a *distributed component platform*. It gives

- **standards** for writing the APIs (and code) for components (and connectors) so that they can connect, communicate, and be reused. The standards are independent of any particular programming language, allowing *heterogeneous* (different styles of) components to be used together.

- **prebuilt components, connectors, and interfaces**, along with a *development environment*, for assembling an architecture.

Middleware provides “smart” connectors that hide the details behind communication. The user writes components that conform to the middleware’s standards/APIs.
CORBA: Common Object Request Broker Architecture

CORBA is middleware for building distributed, object-based, client-server architectures; developed by the Object Management Group (OMG).

Components communicate through a centralized service, the Object Request Broker (ORB).

An object can be a client or a server (or both).

To use the ORB, a server component must implement an API (interface) that lets it connect to an object adapter, which itself connects to the ORB. (Object adapters contain code for object registration with a global naming service, reference generation, and server activation).
Object adapters are available in Java, C++, Perl, etc.; components are written in these languages and communicate via procedure calls.

The physical locations of objects are hidden — references, held in a naming service, are used instead.

The implementations of objects are hidden.

The communications protocols (TCP/IP, RPC, ...) are hidden.

*Only the interfaces are known.*

How connectors work

A client knows the API of the server it wishes to use.

The client uses the naming service to obtain a reference to a server; the reference is used to obtain a local copy of the server object, a “proxy,” called a stub. To send a request, the client invokes a method of the stub. The stub encodes (marshalls) the request and forwards it to the ORB, which transmits it to the true server object.

The request is received by the server’s skeleton, which decodes (unmarshalls) the request and invokes the appropriate method of the server.

The result is returned along the same “path.”
Let’s look at that one up close...

- Local ORB
- Remote ORB
- Request Marshaling
- Request Unmarshaling
- Server Skeleton
- Method Call
- Server Method Implementations
- IIO over TCP/IP
- Method Call
- Client Code
- Client Stub
From the client's perspective, a send connection looks like a method invocation:

Language mappings usually map operation invocation to the equivalent of a function call in the programming language. For example, given a Factory object reference in C++, the client code to issue a request looks like this:

```cpp
// C++
Factory_var factory_objref;

// Initialize factory_objref using Naming or Trading Service (not shown), then issue request
Object_var objref = factory_objref->create();
```

This code makes the invocation of the create operation on the target object appear as a regular C++ member function call. However, what this call is really doing is invoking a stub. Because the stub essentially is a stand-in within the local process for the actual (possibly remote) target object, stubs are sometimes called surrogates or proxies. The stub works directly with the client ORB to marshal the request.

CORBA has become popular because it is a *standard* that is *supported* by many programming languages. Its architecture is useful because it allows *heterogenous components* that communicate by *implementing interfaces*: the ORB interfaces, the object-adapter interfaces, the stub and skeleton interfaces.

But CORBA has some disadvantages, too:

♦ the architecture is difficult to optimize

♦ there is no deadlock detection nor garbage collection (in the middleware)

♦ *all objects are treated as potentially remote*

♦ all object’s references are stored in a global database
DCOM: Microsoft’s Distributed Object Component Model (now in .NET)

has similar objectives and structure as CORBA but tries to address some of CORBA’s deficiencies:

supports reference-counting garbage collection (uses “pinging” to detect inactive clients)

batches together multiple method calls (and pings) to minimize network “round trips”

exploits locality: thread-local and machine-local method calls are implemented more efficiently than RPCs. Uses a virtual table to standardize method call lookup and hide the differences between implementations

makes it easier to program proxy objects and implement dynamic load balancing

allows a component to learn dynamically the interface of another.

But it uses a different IDL and interfaces than CORBA’s)}
Java beans: middleware for Java

A Java bean is a reusable (Java-coded) component, that can be manipulated (its attributes set and its methods executed) both at design-time and run-time.

For this reason, a bean has a design-time interface and a separate run-time interface — this is the key architectural concept for beans. The design-time interface almost always includes a GUI that is displayed by the builder tool. The run-time interface lists properties (attributes), methods, and events that the bean possesses.

The interfaces are more general than usual: they include “properties” (attributes – local state), methods, and event broadcast-listening. The interfaces need not be written by the programmer; they can be extracted from the bean by a development tool.

A development tool (the bean box) uses a bean’s design-time
interface to help an application builder position a bean in the application, customize its appearance, and select its run-time behaviors (methods).

Java beans were originally tailored towards GUI-building applications — buttons, text fields, and sliders are obvious candidates for beans — but the concept also works for data structures and algorithms.

**Examples:**

♦ insert a sorting-algorithm-bean into a spreadsheet bean

♦ insert a spreadsheet bean into a table bean

♦ insert a table bean into a web-page bean
A calculator and its assembly via beans:

Examples are from http://www.tcs.tifr.res.in/man/javaTutorial/beans
Java beans communicate by Java-style event broadcast; a bean can be an *event source* or an *event listener* or both.

Beans execute within a run-time environment, a form of middleware. The environment broadcasts and delivers events; it rests on top of the Java Virtual Machine.

Because it is complex to construct the design-time and run-time interfaces, beans have an *introspection* facility, based on a Java interface Property, which the development tool uses to extract the bean’s interfaces.

The extraction is done in a primitive way: the bean must use standard naming conventions for its attributes, methods, and events (e.g., addListener, removeListener, get, set). Better, the programmer can write a class BeanInfo whose methods surrender the property-method-event interfaces.
The BeanBox is a very simple test container. It allows you to try out both the BDK example beans and your own newly created beans. The BeanBox allows you to:

- drop beans onto a composition window
- resize and move beans around
- edit the exported properties of a bean
- run a customizer to configure a bean
- connect a bean event source to an event handler method
- connect together bound properties on different beans
- save and restore sets of beans
- make applets from beans
- get an introspection report on a bean
- add new beans from JAR files

Note that the BeanBox is intended as a test container and as a reference base, but not as a serious development tool.

Slide is from http://sunset.usc.edu/classes/cs578_2002
Beans and remote access

Beans can serve as front-ends to the network (e.g. CORBA)
There are three primary network access mechanisms:

- Java RMI (Remote Method Invocation) bridges client and server components. RMI allows component interfaces to be designed as regular Java interfaces. RMI automatically handles the network communication.

- Java IDL (Interface Definition Language) implements the OMG CORBA Distributed Object Model. Interfaces are designed in CORBA IDL and Java Stubs can be generated from them.

- JDBC (Java Database Connectivity) for access to SQL databases.
Servlets: beans as proxies
Enterprise Java Beans (EJB) (now in J2EE)

are a variant of Java beans (and not truly compatible with them), oriented to client-server applications.

An EJB is a servlet-like object that is remotely constructed by a client, using methods in the server’s home interface. The EJB is placed in a container (an “adaptor” or “wrapper”) that receives the client’s transaction, decodes it, and gives it to the EJB. Such an EJB is called a session bean.

(An entity bean is an EJB that is shared by multiple clients; it has no internal state.)

The EJB implements methods in the remote interface, which are the method names invoked by the client to request transactions.

The client uses methods in the home interface to remove the session bean.
9. Aspect-oriented programming
Recall Kruchten’s 4 views of software:

1. *logical*: behavioral and functional requirements
2. *process*: concurrency, coordination, and synchronization
3. *development*: organization of software modules
4. *physical*: deployment onto hardware

Each view tells us how to code part of the software.

Kiczales at Xerox PARC said that software contains *aspects*:

- functional behavior (what the software “does”)
- synchronization and security control
- error handling
- persistency and memory management
- monitoring and logging

Each aspect tells us how to code part of the software.

But the aspect’s codings “cross cut” the functional components and are “scattered” throughout the program.
Example: a synchronized stack in Java: functional code in black, synchronization code in red, error-handling in blue:

```java
public class Stack
{
    private int top;
    private Object[] elements;

    public Stack(int size) {
        elements = new Object[size];
top = 0;
    }

    public synchronized void push(Object element) {
        while (top == elements.length) {
            try {
                wait();
            } catch (InterruptedException e) {
                ...}
        }
        elements[top] = element;
top++;
        if (top == 1) {
            notifyAll();
        } // signal that stack is nonempty
    }

    public synchronized Object pop() {
        while (top == 0) {
            try {
                wait();
            } catch (InterruptedException e) {
                ...}
        }
top--;
Object return_val = elements[top];
if (top < elements.length) {
    notifyAll();
} // stack not full
return return_val;
}
```
The synchronized stack example is not so elegant:

- The various aspects are “tangled” (intertwined) in the code, and it is difficult to see which lines of code compute which aspect.

- One aspect is divided (“scattered”) across many components; if there is a change in the aspect, many components must be rewritten.

- It is difficult to study and code an aspect separately.
Example of scattering

[Kiczales 2001]

logging in org.apache.tomcat
- each bar shows one module
- red shows lines of code that handle logging
- not in just one place, not even in a small number of places

From M. Wand, invited talk, ICFP 2003: www.ccs.neu.edu/home/wand
How do we code and integrate an aspect?

Kiczales proposed that each aspect be coded separately and the aspects be *woven* together by a tool called a *weaver*. The weaver inserts code at connection points, called *join points*.

A standard join point is a method call; another is (the entry and exit points of) a method’s definition. Join points can be field declarations or even references to variable names (e.g., for monitoring).

The aspects should be

- **noninvasive**: one aspect should not be written specially to allow it to be “woven into” by another
- **orthogonal**: one aspect does not interfere with the local, logical properties of another
- **minimal coupling**: aspects can be unconnected and reused

Normally, other aspects are woven into the functional aspect.
Lopes developed **COOL**: A language dedicated to synchronization aspects

// In a separate Java file, write the functional component:
public class Stack {
    private int top; private Object[] elements;
    public Stack(int size) { elements = new Object[size]; top = 0; }
    public void push(Object element) { elements[top] = element; top++; }
    public Object pop() { top--; return elements[top]; }
}

// In a separate Cool file, state the synchronization policy:
coordinator Stack {
    selfex push, pop; // self exclusive methods
    mutex { push, pop }; // mutually exclusive methods
    condition full = false; condition empty = true;
    guard push: requires !full;
        onexit { if (empty) empty = false; }
    guard pop: requires !empty;
        onexit { if (full) full = false;
            if (top == 0) empty = true; }
}
When the two classes are woven, the result is the synchronized stack:

```java
public class Stack {
    private int top;
    private Object[] elements;
    private boolean empty;
    private boolean full;

    public Stack(int size) {
        elements = new Object[size]; top = 0;
        full = false; empty = true;
    }

    public synchronized void push(Object element) {
        while (full) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        elements[top] = element; top++;
        if (empty) {
            empty = false; notifyAll();
        }
    }

    public synchronized Object pop() {
        while (empty) {
            try {
                wait();
            } catch (InterruptedException e) {
            }
        }
        top--; Object return_val = elements[top];
        if (top == 0) empty = true;
        if (full) {
            full = false; notifyAll();
        }
        return return_val;
    }
}
```
The COOL language looks somewhat like a language for writing connectors!

Indeed, when join points are method calls or method definitions, then weaving two aspects is weaving the connector code into the component code!
10. Final Remarks
### TABLE 1. Academic versus industrial view on software architecture

<table>
<thead>
<tr>
<th>Academia</th>
<th>Industry</th>
</tr>
</thead>
<tbody>
<tr>
<td>• Architecture is explicitly defined.</td>
<td>• Mostly conceptual understanding of architecture. Minimal explicit definition, often through notations.</td>
</tr>
<tr>
<td>• Architecture consists of components and first-class connectors.</td>
<td>• No explicit first-class connectors (sometimes ad-hoc solutions for run-time binding and glue code for adaptation between assets).</td>
</tr>
<tr>
<td>• Architectural description languages (ADLs) explicitly describe architectures and are used to automatically generate applications.</td>
<td>• Programming languages (e.g., C++) and script languages (e.g., Make) used to describe the configuration of the complete system.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Academia</strong></th>
<th><strong>Industry</strong></th>
</tr>
</thead>
<tbody>
<tr>
<td>- Reusable components are black-box entities.</td>
<td>- Components are large pieces of software (sometimes more than 80 KLOC) with a complex internal structure and no enforced encapsulation boundary, e.g., object-oriented frameworks.</td>
</tr>
<tr>
<td>- Components have narrow interface through a single point of access.</td>
<td>- The component interface is provided through entities, e.g., classes in the component. These interface entities have no explicit differences to non-interface entities.</td>
</tr>
<tr>
<td>- Components have few and explicitly defined variation points that are configured during instantiation.</td>
<td>- Variation is implemented through configuration and specialization or replacement of entities in the component. Sometimes multiple implementations (versions) of components exist to cover variation requirements.</td>
</tr>
<tr>
<td>- Components implement standardized interfaces and can be traded on component markets.</td>
<td>- Components are primarily developed internally. Externally developed components go through considerable (source code) adaptation to match the product-line architecture requirements.</td>
</tr>
<tr>
<td>- Focus is on component functionality and on the formal verification of functionality.</td>
<td>- Functionality and quality attributes, e.g. performance, reliability, code size, reusability and maintainability, have equal importance.</td>
</tr>
</tbody>
</table>
Selected textbook references


