Software Architecture
an informal introduction

David Schmidt

Kansas State University

www.cis.ksu.edu/~schmidt

Outline

1. **Components and connectors**
2. **Software architectures**
3. **Architectural analysis**
4. **Architectural description languages**
5. **Domain-specific design**
6. **Product lines**
7. **Middleware**
8. **Model-driven architecture**
9. **Aspect-oriented programming**
10. **Closing remarks**

An apology...

Because of a shortage of time, I was unable to draw and typeset all the diagrams and text.

So, I downloaded the needed items, captured their images on the screen, and inserted the captured images into these notes. For each image, I have indicated its source.

I apologize for the bad quality of the some of the screen captures.

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

Single-component design

We learned first how to read and implement single-component designs – a single algorithm or a single data structure:

`isPrime(int x):boolean`

pre: `x > 1`

post: returns `true`, if `x` is prime; returns `false`, otherwise

datatype `Stack`

operations

$\mathrm{push} : \mathrm{Value} \times \mathrm{Stack} \rightarrow \mathrm{Stack}$

$\mathrm{pop} : \mathrm{Stack} \rightarrow \mathrm{Stack}$

$\mathrm{top} : \mathrm{Stack} \rightarrow \mathrm{Value}$

axioms $\mathrm{top}(\mathrm{push}(v, s)) = v$

$\mathrm{pop}(\mathrm{push}(v, s)) = s$

etc.

Multi-component design

It is more difficult to design a system of many components:

How do the system requirements suggest the design?

How do the users and their domain experts help formulate the design?

How is the design expressed so that it is understandable by the domain experts as well as the implementors?

How is the design mapped to software components?

How are the components *organized* (sequence, hierarchy, layers, star)?

How are the components *connected*? How do they synchronize/communicate?

How do we judge the success of the design at meeting its requirements?

Programming-in-the-large

was the name given in the 1970's to the work of designing multi-component systems. Innovations were

- the concept of *module* (a collection of data and related functions) and its implementation in languages like Modula-2 and Ada
- controlled visibility of a module's contents (via *import* and *export*)
- logical *invariant* properties of a module's contents
- *interface descriptions* for the modules that can be analyzed separately from the modules themselves (cf. Java `interface`s)

Reference: F. DeRemer and H. H. Kron. Programming-in-the-Large versus Programming-in-the-Small. *IEEE Transactions on Software Engineering*, June 1976.

Component reuse

By the 1980's, virtually all applications required multi-component design. Some practical techniques arose:

- *incremental development*: working systems were incremented and modified into new systems that met a similar demand
- *rapid prototyping*: interpreter-like generator systems were used to generate quick-and-inefficient implementations that could be tested and incrementally refined.
- *buy-versus-build*: "Commercial Off The Shelf" (COTS) modules were purchased and incorporated into new systems.

These techniques promoted *component reuse* — it is easier to reuse than to build-from-scratch. But, to reuse components successfully, one must have an *architecture* into which the components fit!

Motivation for software architecture

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

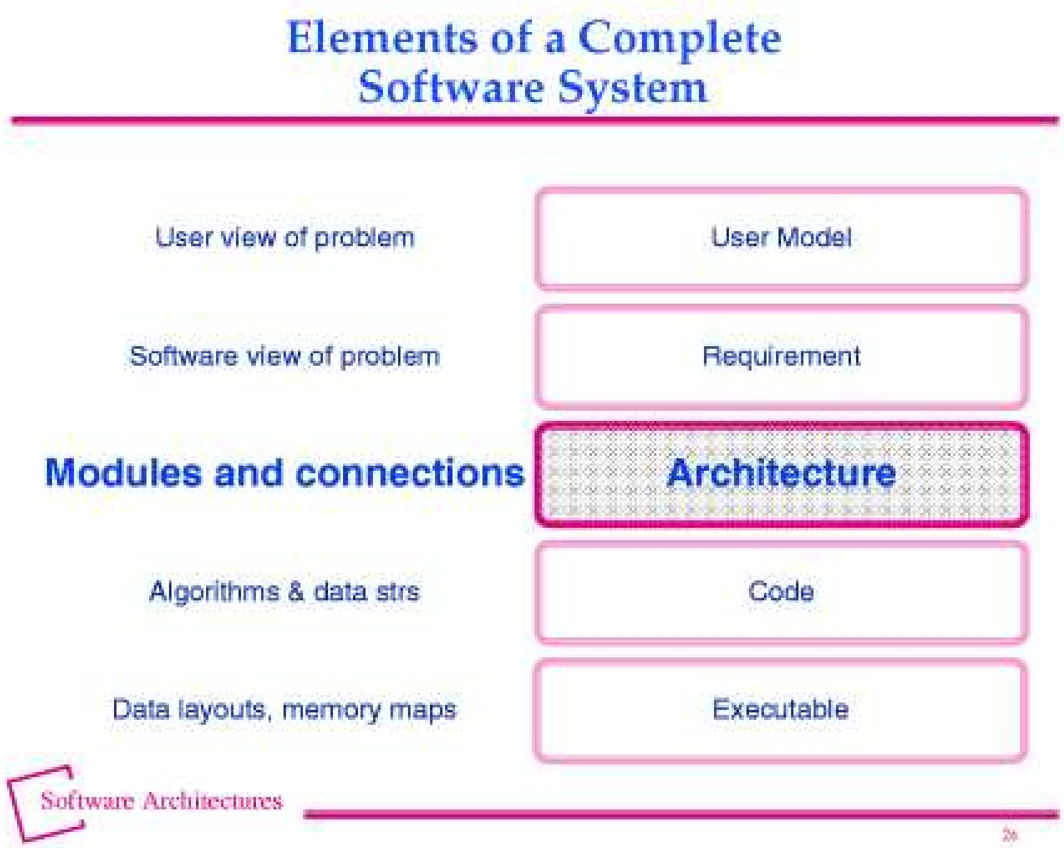
- “Camelot is based on the *client-server model* and uses remote procedure calls both locally and remotely to provide communication among applications and servers.” [Spector87]
- “The easiest way to make the canonical sequential compiler into a concurrent compiler is to *pipeline* the execution of the compiler phases over a number of processors.” [Seshadri88]
- “The ARC network follows the *general network architecture* specified by the ISO in the Open Systems Interconnection Reference Model.” [Paulk85]

Reference: David Garlan, *Architectures for Software Systems*, CMU, Spring 1998.

`http://www.cs.cmu.edu/afs/cs/project/tinker-arch/www/html/index.html`

Architectural description has a natural position in system design and implementation

A slide from one of David Garlan's lectures:



Reference: David Garlan, *Architectures for Software Systems*, CMU, Spring 1998.

`http://www.cs.cmu.edu/afs/cs/project/tinker-arch/www/html/index.html`

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.

Reference: D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, October 1992.

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-fbw, data-fbw, modular structure, behavioral requirements, ...

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

(iii) *engineering*: modules, fi lters, 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:

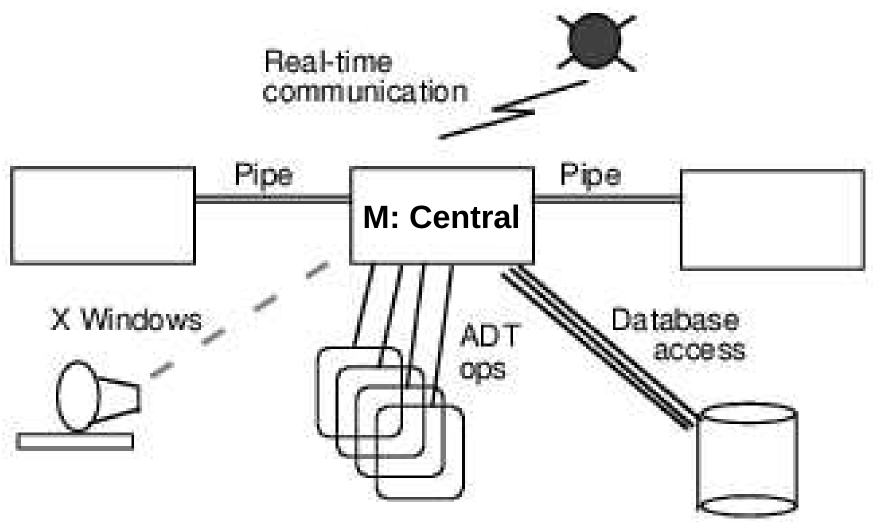


Figure 2: Revised architecture diagram with discrimination among connections

Reference: Mary Shaw, Procedure Calls are the Assembly Language of Software Interconnections: Connectors Deserve First-Class Status. Workshop on Studies of Software Design, 1993.

Shaw's observations

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*.

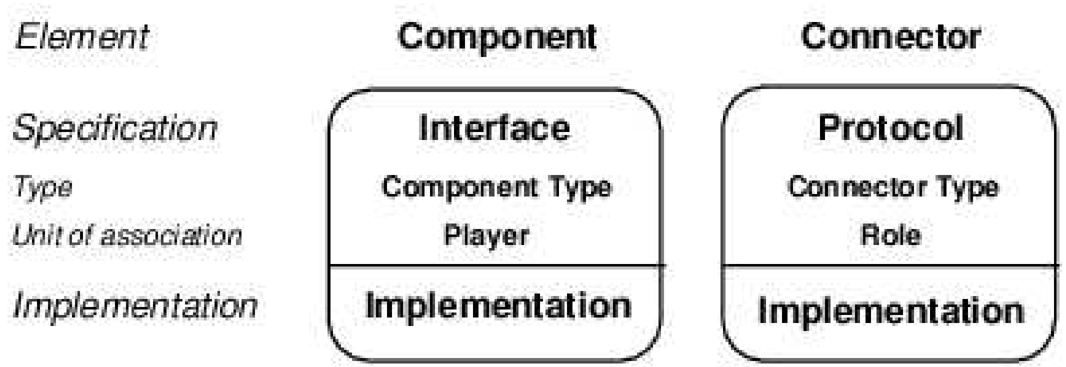


Figure 3: Gross structure of an architecture language

Example:

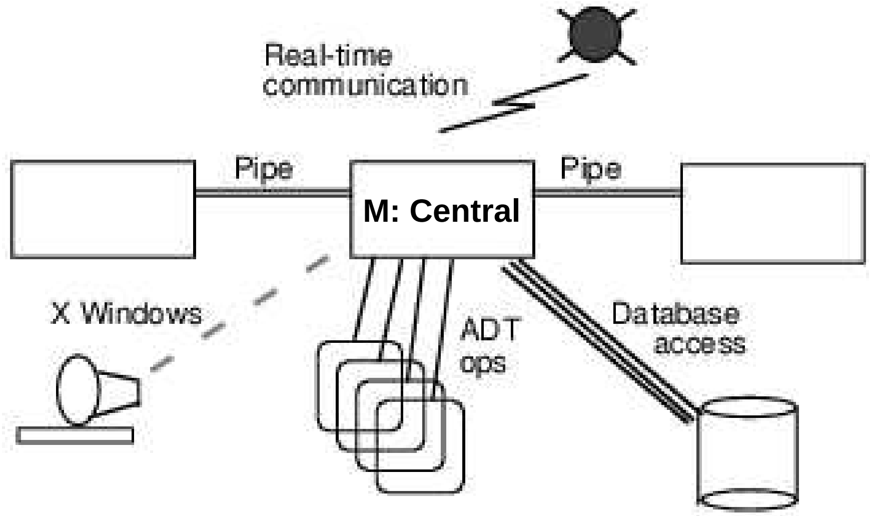


Figure 2: Revised architecture diagram with discrimination among connections

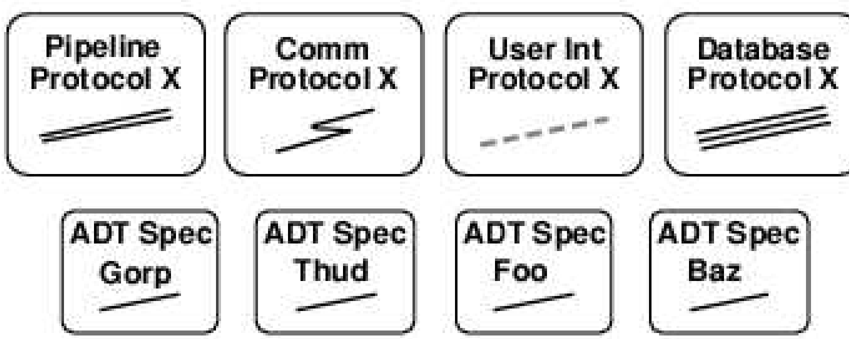


Figure 4: Constellation of protocol specifications required by exampl

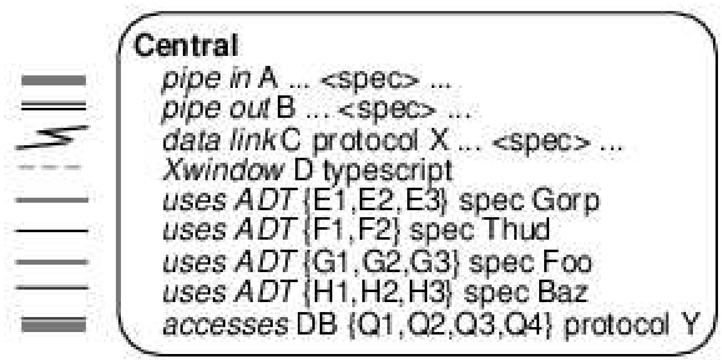


Figure 5: Interface specification of central component, referring to protocols

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.

The connector's *protocol* lists

(i) the types of component interfaces it can "mediate";

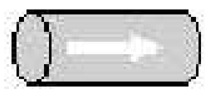
(ii) orderings and invariants of component interactions;

(iii) performance guarantees.

Example: Shaw's description of a unix pipe:

***Pipe* Connector**

Informal Description: The Unix abstraction for pipe, i.e. a bounded queue of bytes that are produced at a source and consumed at a sink. Also supports interactions between pipes and files, choosing the correct Unix implementation.



Icon: pipe section

Properties: *PipeType*, the kind of Unix pipe. Possible values Named, Unnamed

Roles: *Source*

Description: the source end of the pipe
Accepts player types: *StreamOut* of component Filter; *ReadNext* of component SeqFile
Properties: *MinConns*, minimum number of connections. Integer values, default 1
MaxConns, maximum number of connections. Integer values, default 1

Sink

Description: the sink end of the pipe
Accepts player types: *StreamIn* of component Filter; *WriteNext* of component SeqFile
Properties: *MinConns*, *MaxConns*, as for Source

Reference: M. Shaw, R. DeLine, and G. Zelesnik. Abstractions and Implementations for Architectural Connections. In 3d. Int. Conf. on Confi gurable Distributed Systems, Annapolis, Maryland, May 1996.

Connectors can act as

- *communicators:* transfer data between components (e.g., message passing, buffering)
- *mediators:* manage shared resource access between components (e.g., reader/writer policies, monitors, critical regions)
- *coordinators:* define control flow between components (e.g., synchronization (protocols) between clients and servers, event broadcast and delivery)
- *adaptors:* connect mismatched components (e.g., a pipe connects to a file rather than to a filter)

Perhaps you have written code for a bounded buffer or a monitor or a protocol or a shared, global variable — you have written a connector!

Connectors can facilitate

- *reuse:* components from one application are inserted into another, and they need not know about context in which they are connected

- *evolution:* components can be dynamically added and removed from connectors

- *heterogenity:* components that use different forms of communication can be connected together in the same system

A connector should have the ability to handle limited *mismatches* between connected components, via information reformatting, object-wrappers, and object-adaptors, such that the component is not rewritten — the connector does the reformatting, wrapping, adapting.

If connectors are crucial to systems building, why did we take so long to “discover” them? One answer is that components are “pre-packaged” to use certain connectors:

Component type	Common types of interaction
Module	Procedure call, data sharing
Object	Method invocation (dynamically bound procedure call)
Filter	Data flow
Process	Message passing, remote procedure call
	various communication protocols, synchronization
Data file	Read, write
Database	Schema, query language
Document	Shared representation assumptions

But “smart” connectors make components simpler, because the coding for interaction rests in the connectors — not the components.

The philosophy, ***system = components + connectors*** was a strong motivation for a theory of software architecture.

Reference: M. Shaw and D. Garlan. Formulations and Formalisms in Software Architecture. *Computer Science Today: Recent Trends and Developments* Jan van Leeuwen, ed., Springer-Verlag LNCS, 1996, pp. 307-323.

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.

Reference: D. E. Perry and A. L. Wolf. Foundations for the Study of Software Architectures. *ACM SIGSOFT Software Engineering Notes*, October 1992.

What is a software architecture? (Garlan)

[A software architecture states] the structure of the components of a program/system, their interrelationships, and principles and guidelines governing their design and evolution over time.

The architectural description

1. *describes the system* in terms of components and interactions between them
2. *shows correspondences* between requirements and implementation
3. *addresses properties* such as scale, capacity, throughput, consistency, and compatibility.

Mary Shaw calls the previous defi nitions

structural (constituent parts) models.

She notes that there are also

framework (whole entity) models,
dynamic (behavioral) models,
and *process* (implementational) models

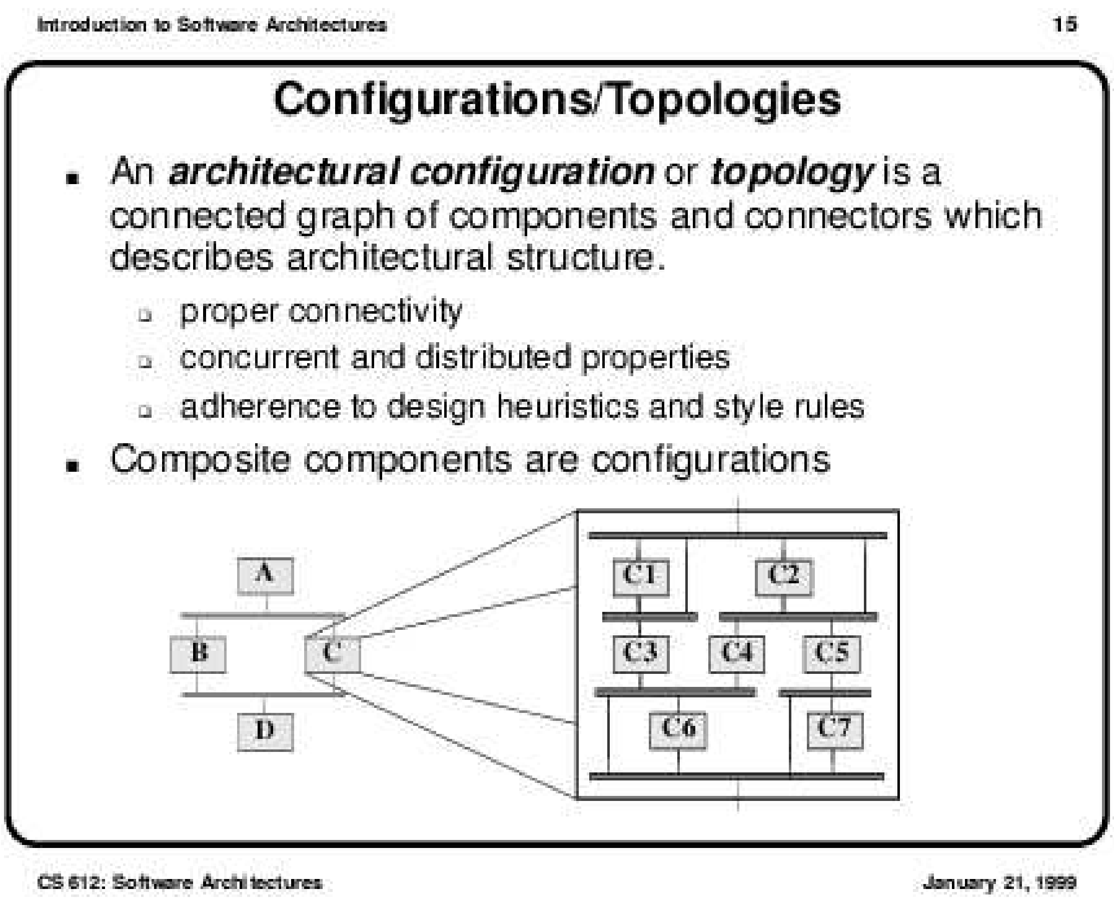
of software architecture.

- *Structural (constituent parts) models*: components, connectors, and "other stuff" (configuration, rationale, semantics, constraints, styles, analysis, properties, requirements, needs). Readily supports architectural description languages; underemphasizes dynamics.
- *Domain-specific (whole-entity/"framework") models*: a single structure well suited to a problem domain (e.g, telecommunications, avionics, client-server). The narrow focus allows one to give a detailed presentation of syntax, semantics, and pragmatics and tool support.
- *Dynamic (behavioral) models*: explains patterns of communications, how components are added and removed, how system evolves. (e.g., reactive systems, π-calculus, chemical abstract machines). Emphasizes dynamics over statics.
- *Process (implementational) models*: Construction steps for converting architecture into implementation. Disappearing.

We begin with the *structural (constituent parts)* model:

- *Components:* What are the building blocks? (e.g., filters, ADTs, databases, clients, servers)
- *Connectors:* How do the blocks interact? (e.g., call-return, event broadcast, pipes, shared data, client-server protocols)
- *Configuration:* What is the topology of the components and connectors?
- *Constraints:* How is the structure constrained? Requirements on function, behavior, performance, security, maintainability....

We have seen components and connectors, but what is a *configuration* ?



The slide is from Nenad Medvidovic's course on software architectures,

`http://sunset.usc.edu/classes/cs578_2002`

***Architectural Styles* (patterns)**

1. *Data-flow systems:* batch sequential, pipes and fi lters
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 specifi c instances of the architectures (e.g., a client-server architecture is a distributed system). But these notions are not fi rm!

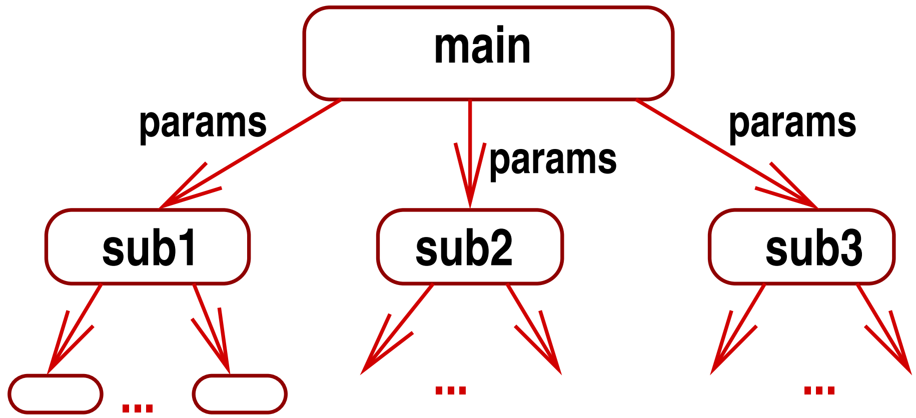
Data-flow systems: Batch-sequential and Pipe-and-filter

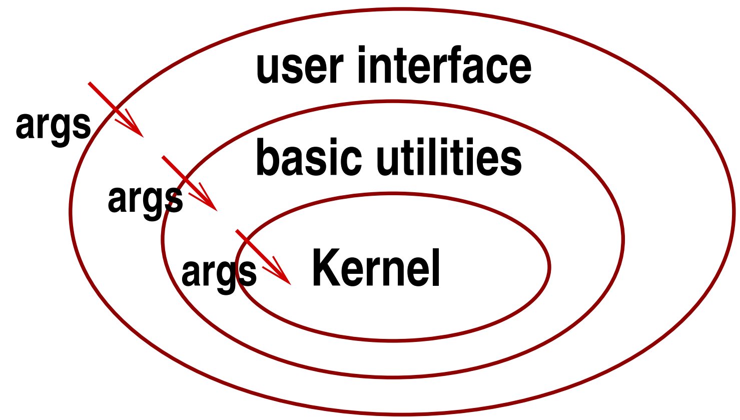
	Batch sequential	*Pipe and filter*
Components:	whole program	filter (function)
Connectors:	conventional input-output	pipe (data flow)
Constraints:	components execute to completion, consuming entire input, producing entire output	data arrives in increments to filters

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

Advantages: easy to unplug and replace fi lters; interactions between components easy to analyze. *Disadvantages:* interactivity with end-user severely limited; performs as quickly as slowest component.

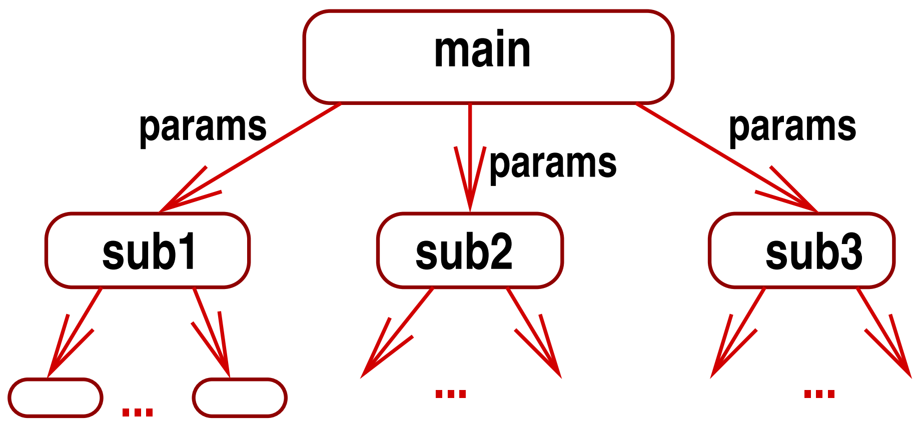
Call-and-return systems: subroutine and layered

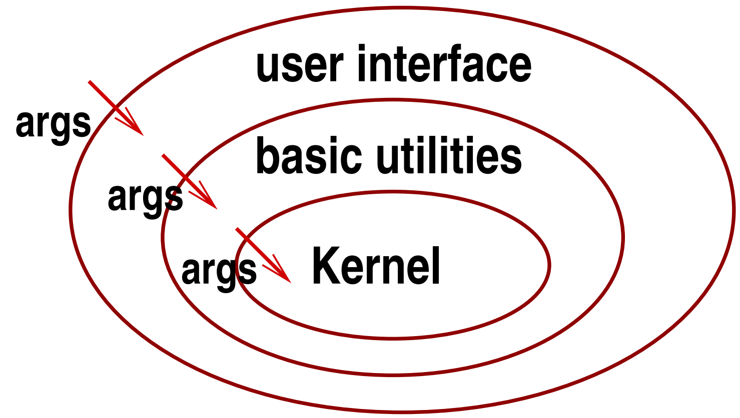




	Subroutine	*Layered*
Components:	subroutines (“servers”)	functions (“servers”)
Connectors:	parameter passing	protocols
Constraints:	hierarchical execution and encapsulation	functions within a layer invoke (API of) others at next lower layer

Examples: modular, object-oriented, N-tier systems (subroutine); communication protocols, operating systems (layered)

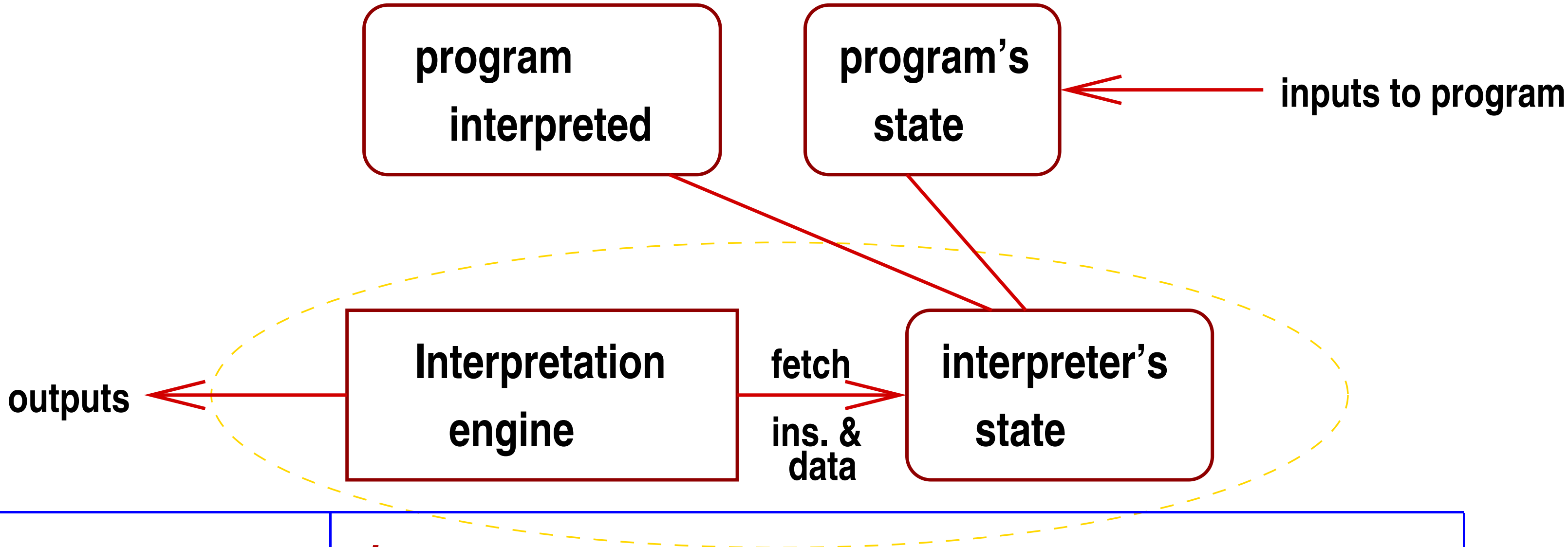




Advantages: hierarchical decomposition of solution; limits range of interactions between components, simplifying correctness reasoning; each layer defines a *virtual machine*; supports portability (by replacing lowest-level components).

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

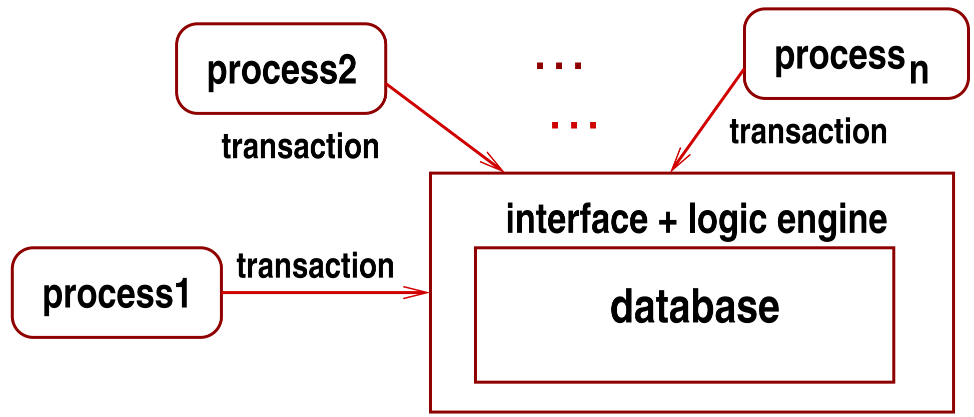


	Interpreter
Components:	"memories" and state-machine engine
Connectors:	fetch and store operations
Constraints:	engine's "execution cycle" controls the simulation of program's execution

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

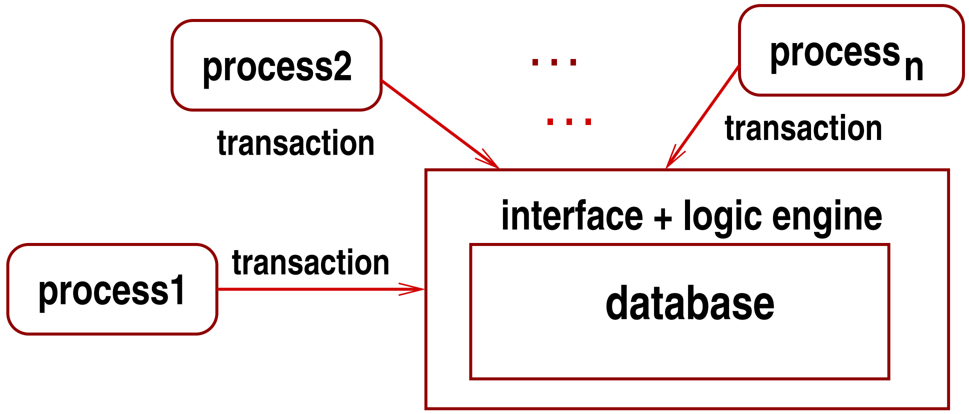
Advantages: rapid prototyping *Disadvantages:* ineffi cient.

Repositories: databases and blackboards



	Database	*Blackboard*
Components:	processes and database	knowledge sources and blackboard
Connectors:	queries and updates	notifi cations 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.

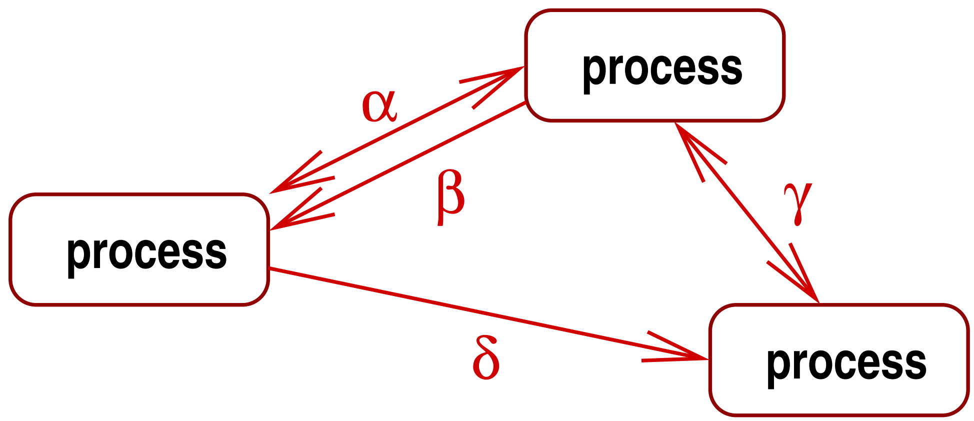
Examples: speech and pattern recognition (blackboard); syntax editors and compilers (parse tree and symbol table are repositories)



Advantages: easy to add new processes.

Disadvantages: alterations to repository affect all components.

Independent components: communicating processes

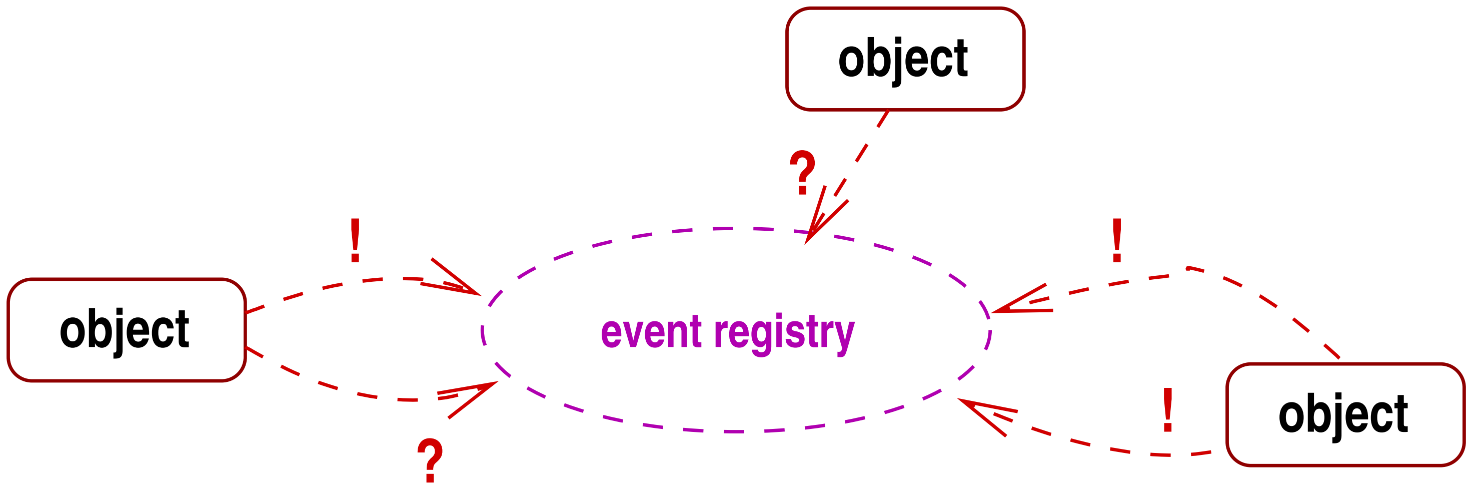


	Communicating processes
Components:	processes ("tasks")
Connectors:	ports or buffers or RPC
Constraints:	processes execute in parallel and send messages (synchronously or asynchronously)

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

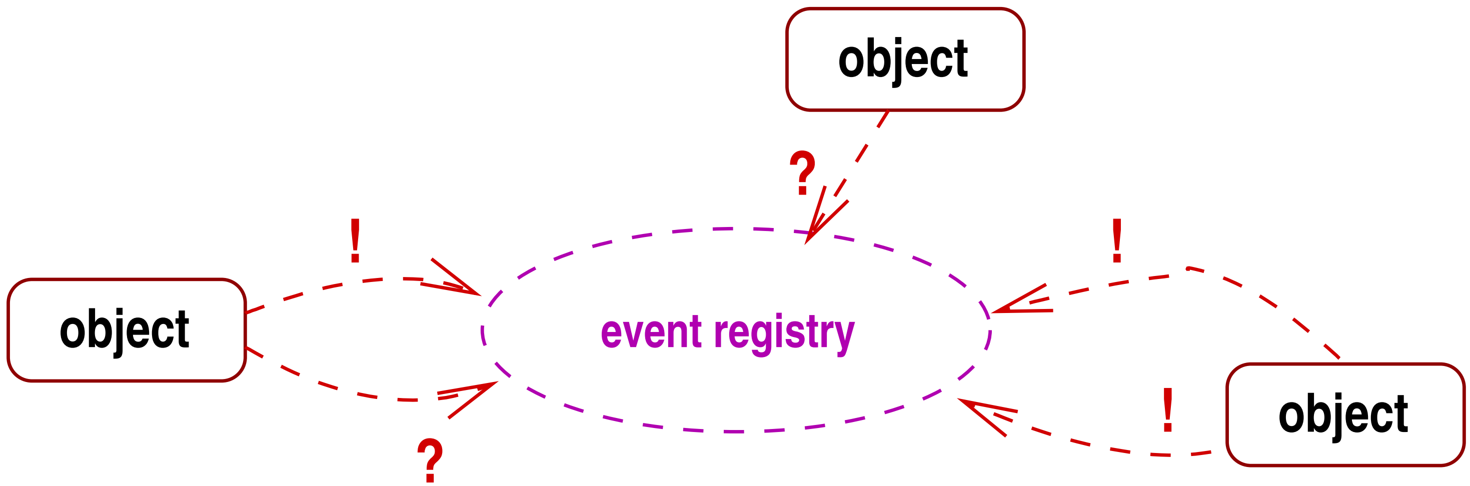
Advantages: easy to add and remove processes. *Disadvantages:* diffi cult to reason about control fbw.

Independent components: event systems



	Event systems
Components:	objects or processes (“threads”)
Connectors:	event broadcast and notification (implicit invocation)
Constraints:	components “register” to receive event notification; components signal events, environment notifies registered “listeners”

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.

Other forms of architecture

Process control system: Structured as a feedback loop where input from sensors trigger computation whose outputs adjust the physical environment. For controlling a physical environment, e.g., software for flight control.

State transition system: Structured as a finite automaton; for reactive systems, e.g., vending machines.

Domain-specific software architectures: architectures tailored to specific application areas. Requires a domain model, which lists domain-specific objects, operations, vocabulary. Requires a reference architecture, which is a generic depiction of the desired architecture. The architecture is then instantiated and refined into the desired software architecture.

Examples: Client-server models like CORBA, DCOM (in .NET), Enterprise Javabeans (in J2EE).

Three architectures for a compiler (Garlan and Shaw)

Figure 15: Traditional Compiler Model

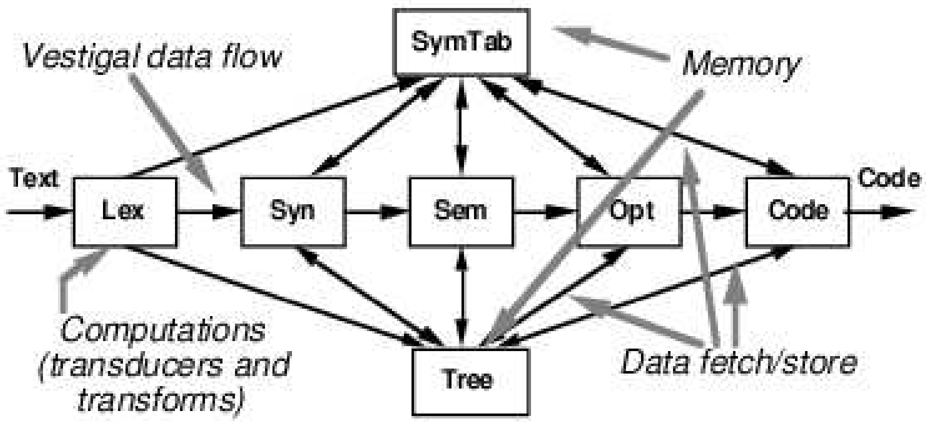


Figure 17: Modern Canonical Compiler

The symbol table and tree are "shared-data connectors"

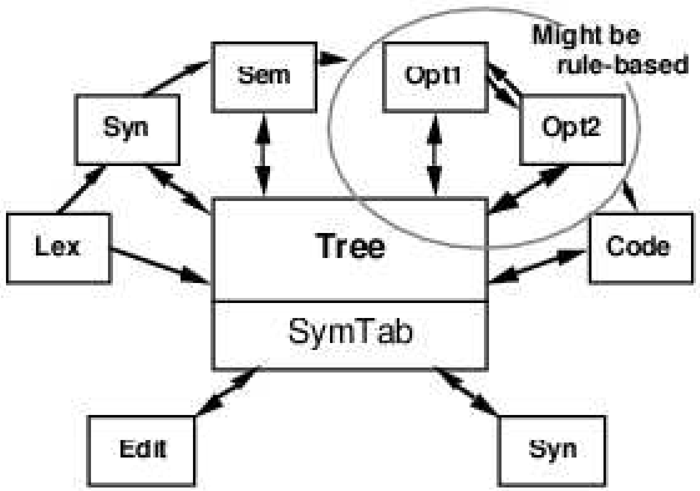


Figure 18: Canonical Compiler, Revisited

The blackboard triggers incremental checking and code generation

What do we gain from using a software architecture?

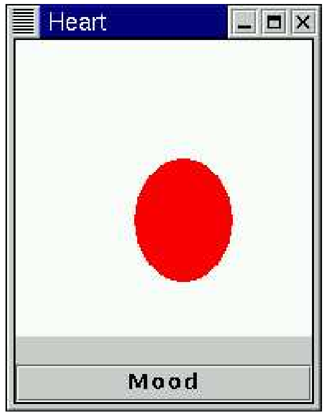
1. the architecture helps us *communicate* the system's design to the project's stakeholders (users, managers, implementors)
2. it helps us *analyze* design decisions
3. it helps us *reuse* concepts and components in future systems

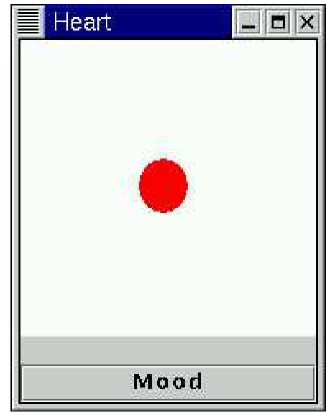
An example of an application and its software architecture

An architecture that is heavily used for single-user, GUI-based applications is the *Model-View-Controller (MVC)* architecture.

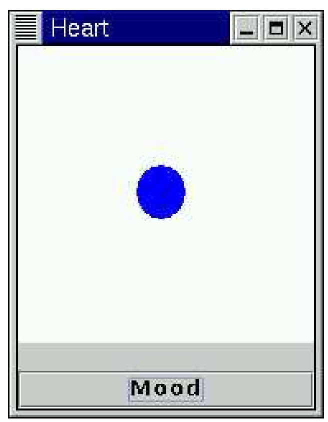
A demonstration example: Heart Animation

When started, a view appears of an animated, beating heart:

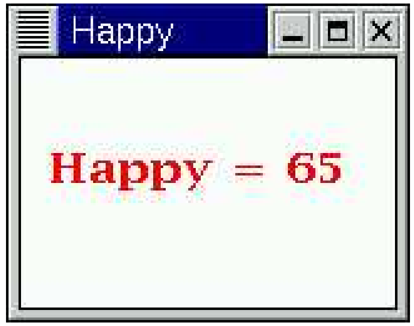


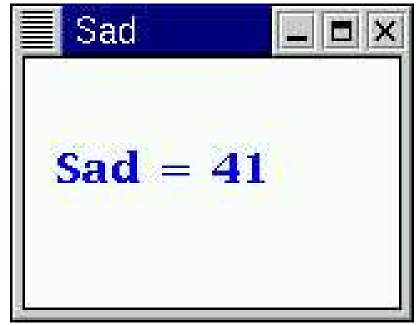


When the “Mood” button is pressed, the heart changes from its “happy” color to its “sad” color:



But there is another view of the heart—two additional windows display the state of the heart in terms of its history of happy and sad beats:

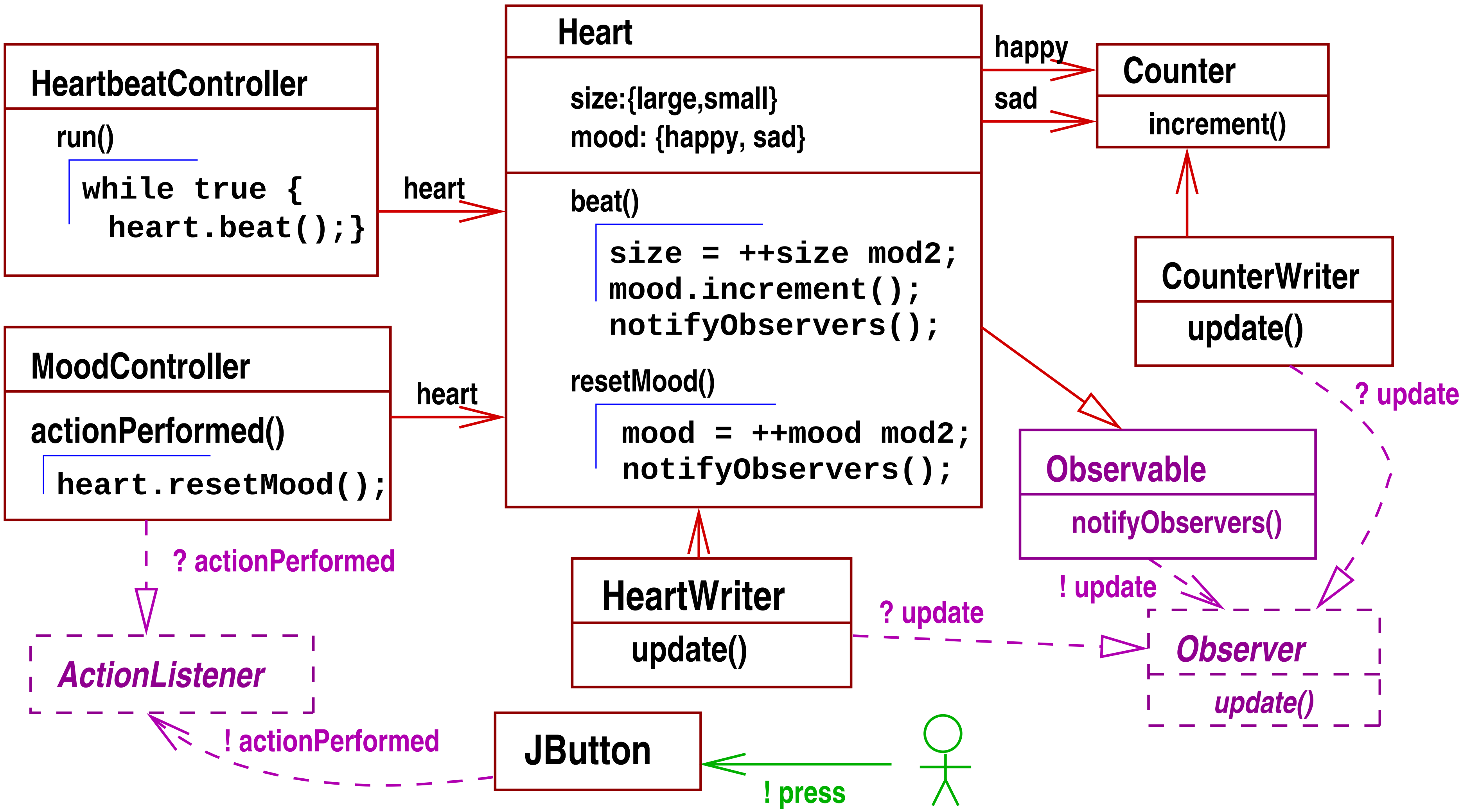




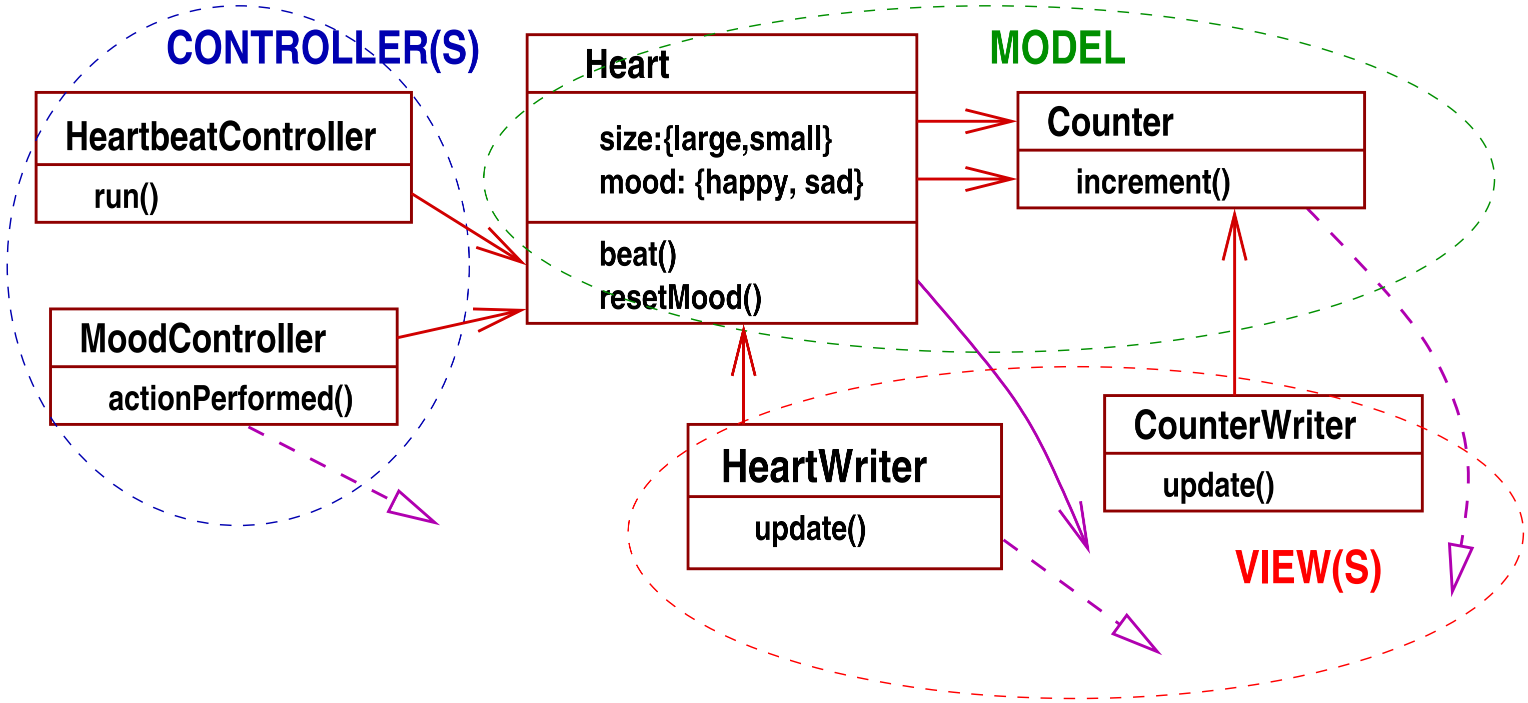
The heart is *modelled* within the animation and is *viewed* in two different ways (by color and counts). It is *controlled* by a "clock" and a Mood button.

The source code is available at `www.cis.ksu.edu/santos/schmidt/ppdp01/Heart`

MVC Architecture of the Heart Animation



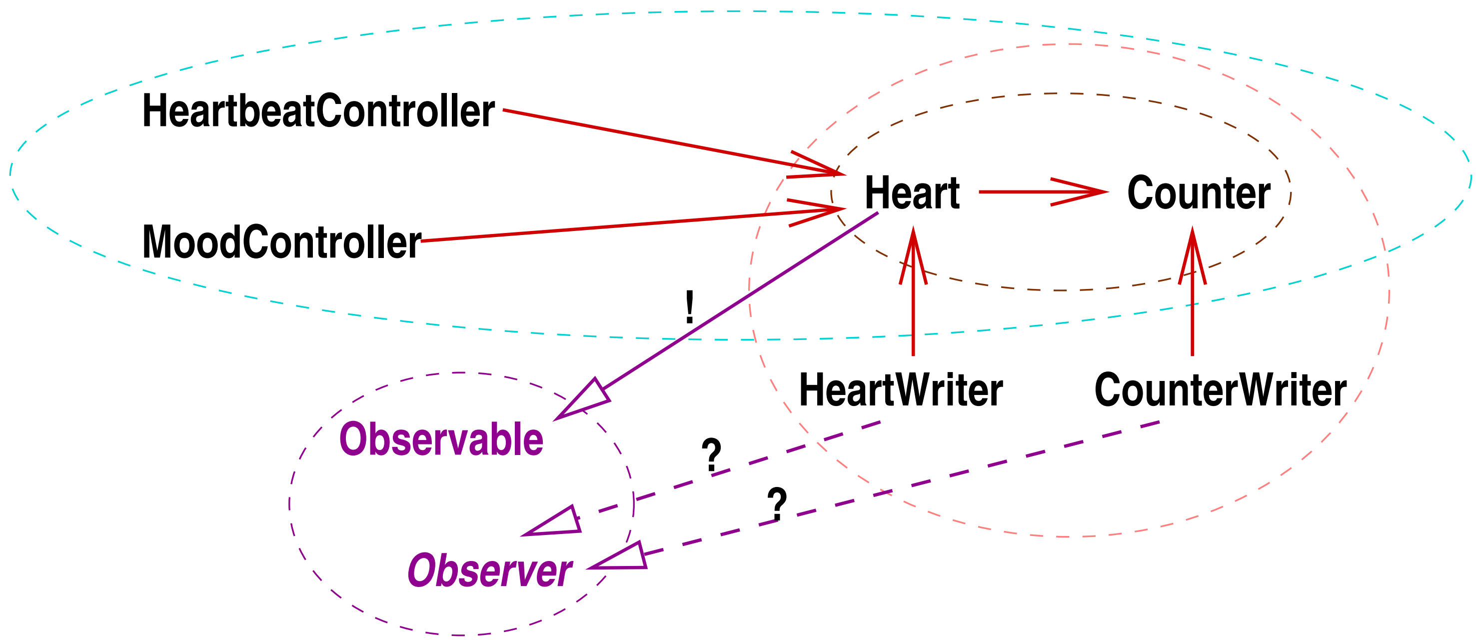
MVC is a *hybrid* architecture: the subassemblies are object-oriented and are connected as an event system. (The `java.util` and `javax.swing` packages implement the event registries.)



	MVC
Components:	classes and interfaces (to event registries)
Connectors:	call-return message passing, event broadcast
Properties:	Architecture is divided into Model, View, and Controller subassemblies. Controller updates Model's state; when updated, Model signals View(s) to revise presentation.

Analyzing the architecture: *Couplings*

Consider the dependency structure of the heart animation, where self-contained subassemblies are circled; these can be extracted for reuse:

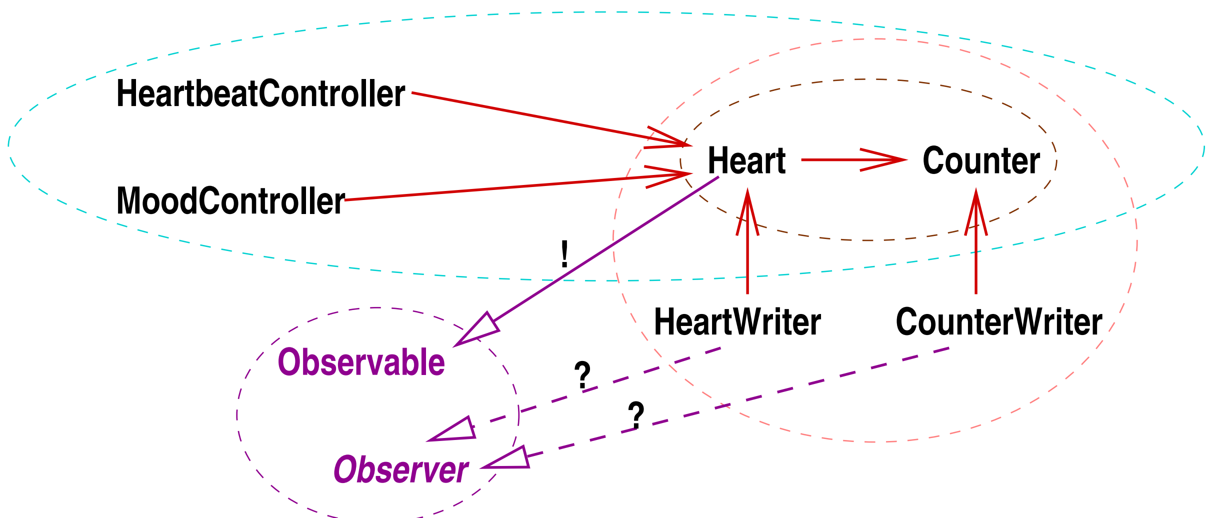


Couplings can be studied: `A` is *coupled* to `B` if modifications to `B`'s signature imply modifications to `A`'s implementation. (Normally, dependency implies coupling, and we will treat it as such here.)

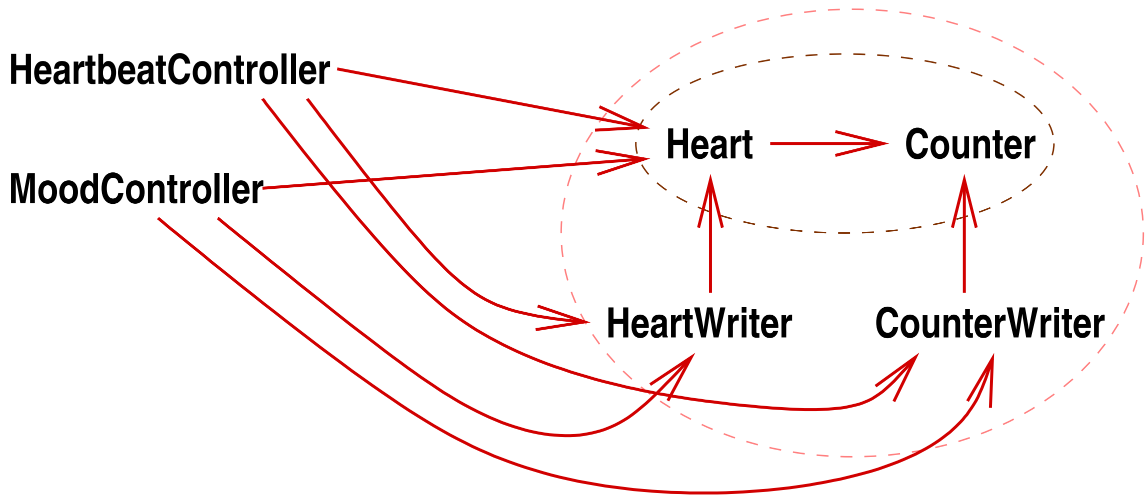
As a general rule, a system should have *weak coupling* — changes to a component imply minimal changes to the rest of the system.

(But data-centered systems, like a database, have *strong coupling* —all user processes are coupled to the database, making changes to the database expensive!)

In the example, the `Observer/Observable` event registry decouples the animation's controllers from its views and ensures that the model is decoupled from all other subassemblies:

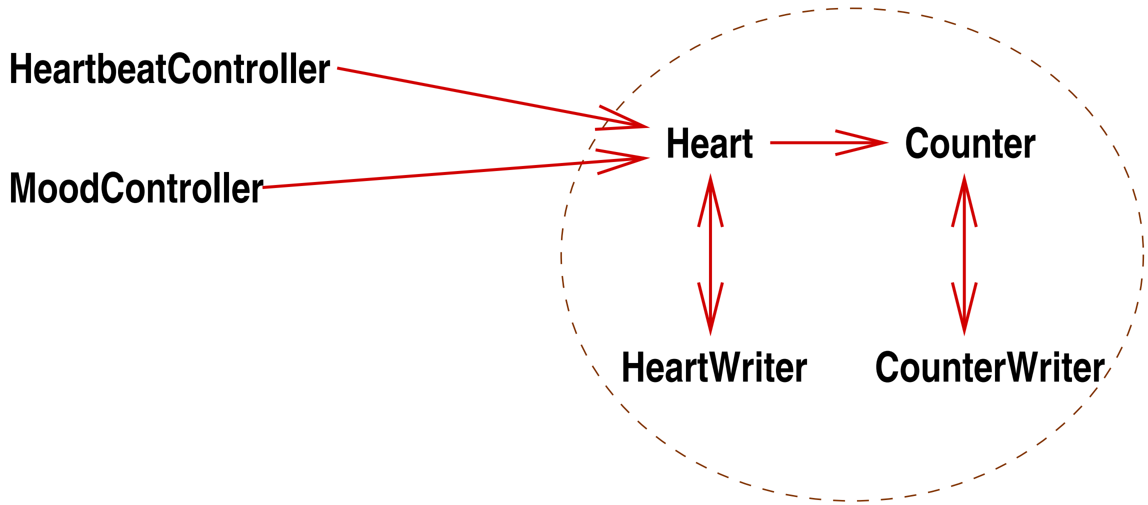


Without the `Observer` event registry, we might design the animation like this, where the controllers tell the model to update and tell the views to refresh:



The structure is hierarchical, coupling the controllers to all subassemblies; unfortunately, the controllers operate only with fixed views.

An alternative is to demand that the model contact all views whenever it is updated:



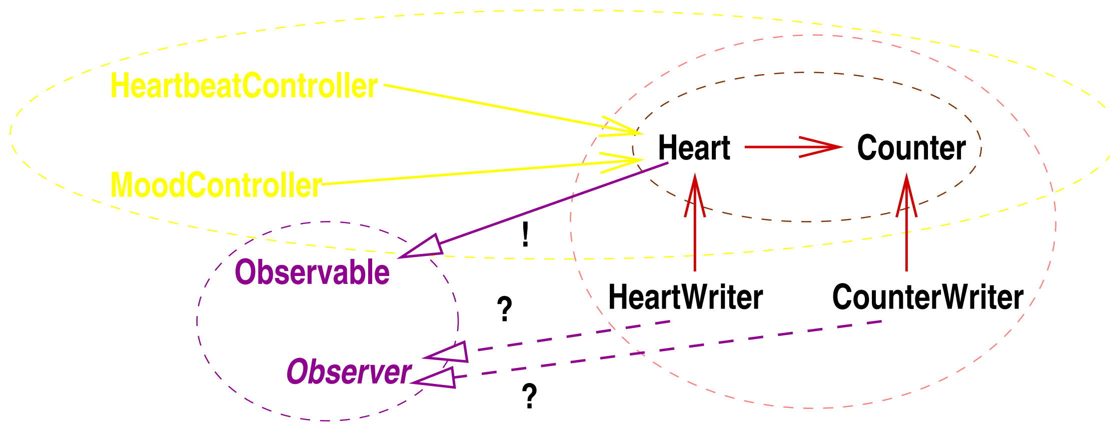
This looks clean, but the model controls the views! And it operates only with fixed views.

Both of the latter two architectures will be difficult to maintain as the system evolves. Subassembly reuse is unlikely.

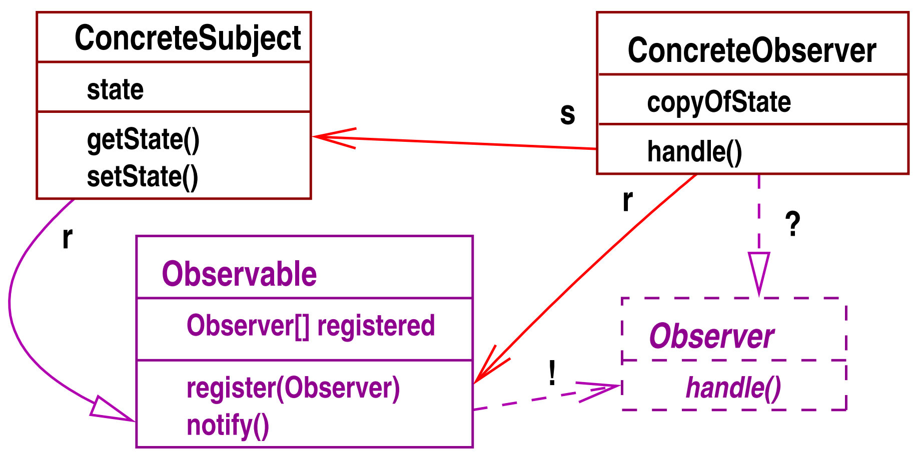
The first architecture is the best; indeed, it uses the *observer design pattern*.

Design patterns

When an architectural (sub)design proves successful in multiple projects, it defines a *design pattern* that can be used in future designs. The model and view subassemblies of the animation,



are assembled according to the *observer* design pattern:



A ***design pattern*** is a solution scheme to a common architectural problem that arises in a specific context. It is presented by

- stating the problem and the context in which it arises
- stating the solution in terms of an architectural structure (syntax)
- describing the behavior (semantics) of the structure
- assessing the pragmatics

Varieties:

1. *Creational:* patterns for constructing components
2. *Structural:* patterns for connecting components
3. *Behavioral:* patterns for communicating between components

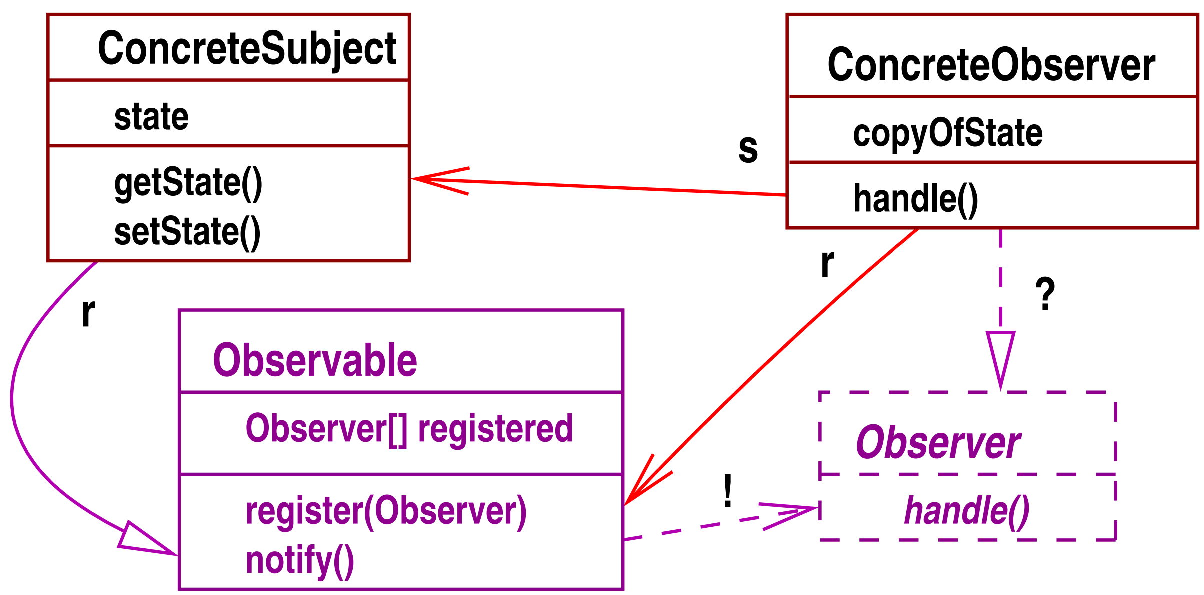
Reference: E. Gamma, et al., *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

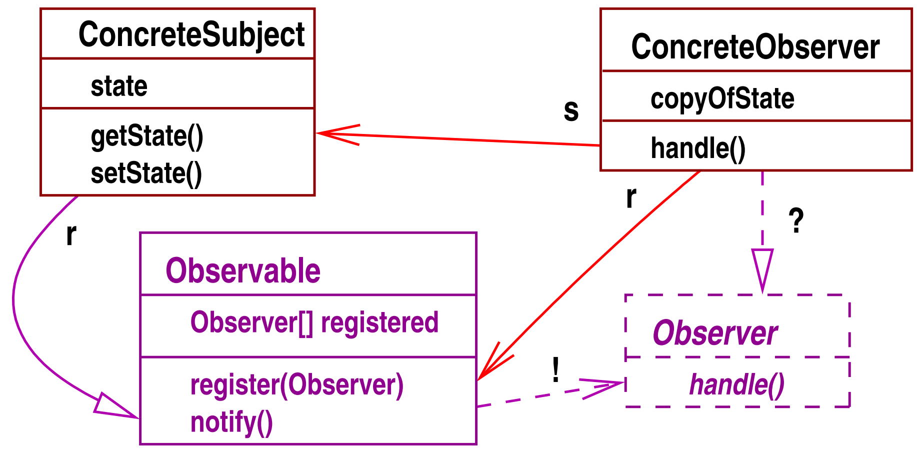
A behavioral pattern: *observer*

Problem Context: Maintain consistency of copies of state among multiple objects, where one object's state must be "mirrored" by all the others.

The pattern designates one *subject* object to hold the state; *observer* objects hold the copies and are notified by indirect event broadcast when the subject's state changes. The observers then query the subject and copy the state changes.

Syntax:





Semantics:

I. The `ConcreteSubject` owns an event registry, `r. Observable`.

II. Each `ConcreteObserver` invokes `r.register(this)`, registering itself.

III. When the `ConcreteSubject`'s `setState` method is invoked, the method updates state and signals `r.notify()`, which broadcasts events to all `registered[i]`, starting these objects' `handle` methods.

IV. Each `handle` method invokes `s.getState()` and updates its local state.

Pragmatics:

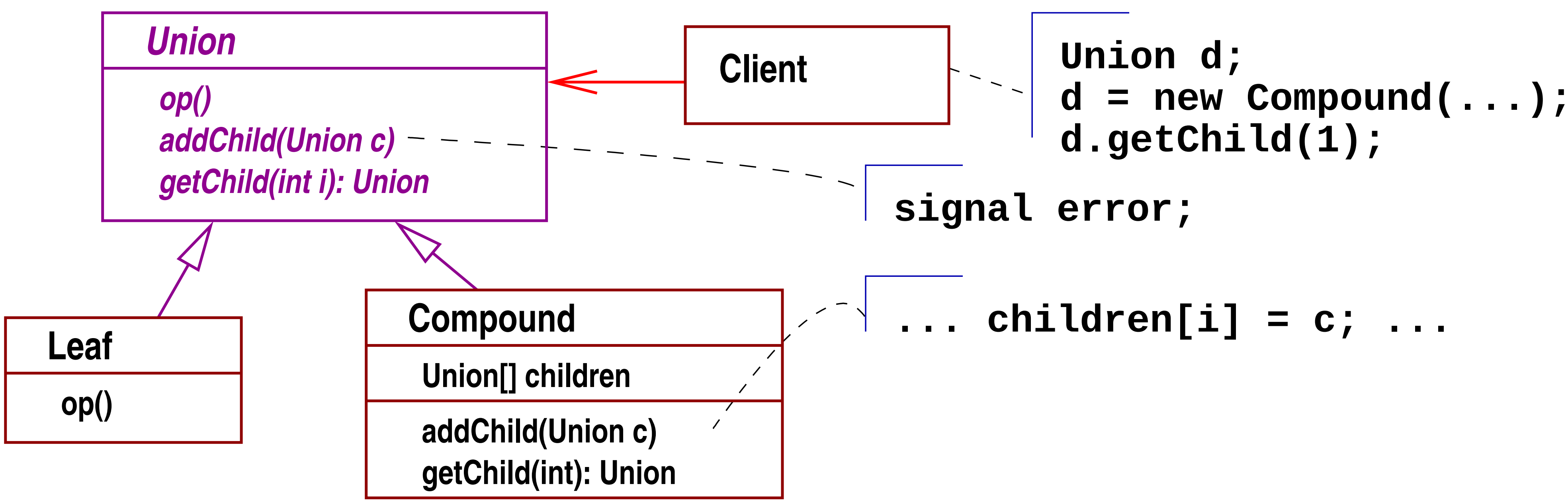
- ✔ weak coupling: the subject knows nothing about its observers
- ✔ observers are readily added, modifi ed, and detached
- ✘ a minor state update signals *all* observers

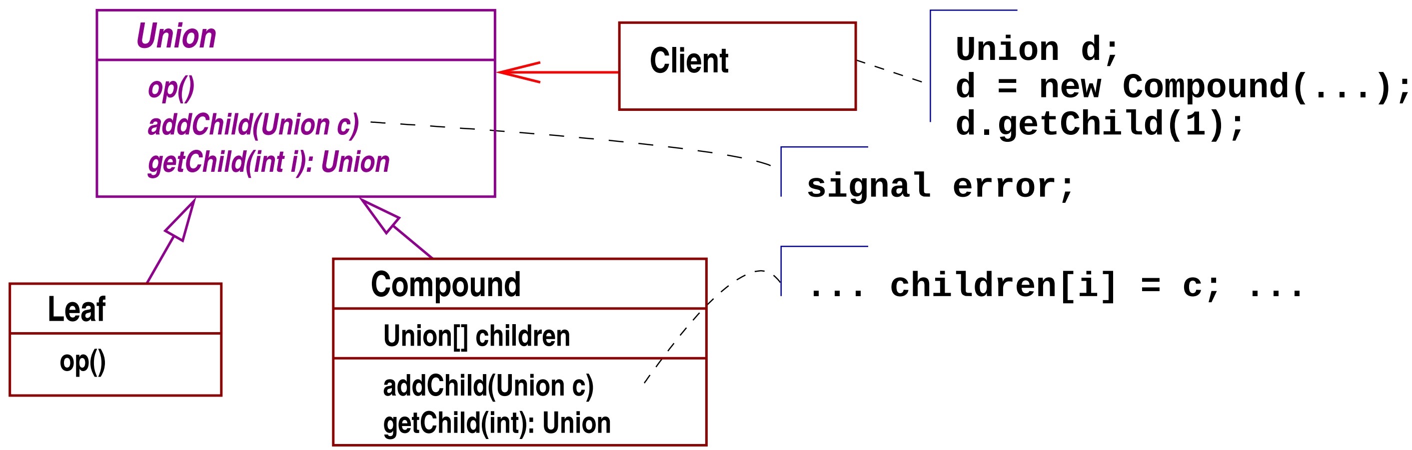
A structural pattern: *composite*

Problem Context: Compound data structures, constructed from "leaves" and "compound" classes, must be manipulated by a client, which treats all structures uniformly.

The pattern adds an abstract class to name the (disjoint) union of the data classes and hold default methods for all operations on the data classes. The client treats all objects as having the union type.

Syntax:





Semantics:

I. `Union` holds default codings for all operations of all data classes. Each subclass overrides some of the defaults.

II. The `Client` treats all data as having type `Union` and invokes its methods without employing down-casts.

Pragmatics:

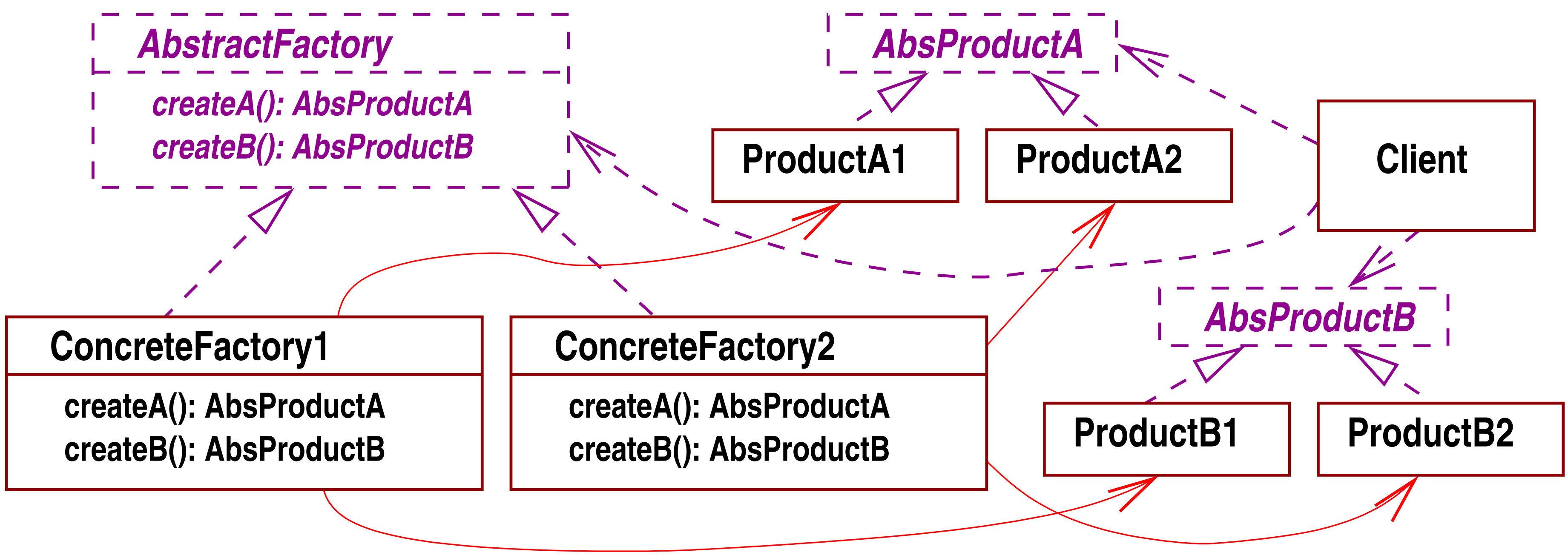
- ✔ client can process the data structures recursively without down-casts
- ✔ easy to add new data classes to `Union`
- ✘ diffi cult to restruct the classes that may be `children` of `Compound`

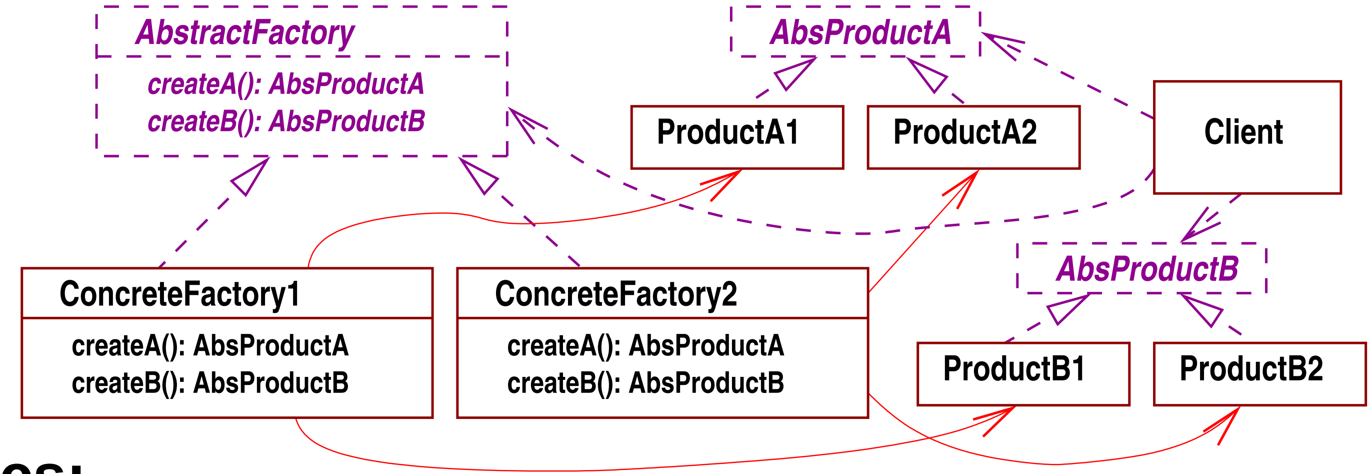
A creational pattern: *abstract factory*

Problem Context: A client uses a “product family” (e.g., widgets — windows, scroll bars, menus), constructed on demand. The client must be separate from the family so that the family can be easily changed (e.g., a different “look and feel”).

The pattern uses an interface to list the constructors for the products, and each family implements the interface.

Syntax:





Semantics:

I. The `AbstractFactory` interface is implemented by one of `ConcreteFamily1` or `ConcreteFamily2`, and interfaces `AbsProductA` and `AbsProductB` are implemented by the respective concrete products.

II. The `Client` invokes the methods in `AbstractFactory` to receive objects of type `AbsProduct1` and `AbsProduct2` —it does not know the identities of the concrete products.

Pragmatics:

- ✔ `Client` is decoupled from the products it uses
- ✔ interface `AbstractFactory` forces all product families to be consistent
- ✘ it is diffi cult to add new products to just one factory

Of course, the abstract factory pattern is a compensation for the lack of a polymorphic class — but it does indicate a context when the “polymorphism” can be profitably applied.

And the composite pattern is a compensation for the lack of a disjoint union type — but it does indicate a context when disjoint union can be profitably applied.

In this sense, design patterns are universal across programming paradigms, although each programming paradigm will support some design patterns more simply than others.

3. Architectural analysis

How do we classify architectural styles?

1. Forms of components and connectors. See earlier slides.
2. Control-flow: how control is transferred, allocated, and shared.
   topology: geometric shape of control —linear, hierarchical, hub-and-spoke. Static or dynamic. *synchronicity*: lockstep, synchronous, asynchronous. *binding time*: when the partner of a transfer of control is established: compile-, link-, or run-time.
3. Data-flow: how data is communicated through the system.
   topology: geometric shape of the data flow; *continuity*: continuous, sporadic, high-volume, low-volume flow; *mode*: how data is transferred: passed, shared, copy-in-copy-out (from shared structure), broadcast, or multicast.
4. Control/data interaction. *shape*: are control/data topologies similar? *directionality*: do data and control travel in the same direction?
5. Which form of reasoning is compatible with the style? *state machine theory/process algebra* (for independent components); *function composition* (for pipe-and-filter);*inductive/compositional* (for hierarchical).

Table 1: Specializations of the dataflow network style

Style	Constituent parts		Control issues			Data issues				Ctrl/data interaction	
	Components	Connectors	Topology	Synchronicity	Binding time	Topology	Continuity	Mode	Binding time	Isomorphic shapes	Flow directions
Data flow styles: Styles dominated by motion of data through the system, with no "upstream" content control by recipient											
Dataflow network [B+88]	transducers	data stream	arbitrary	asynch	i, r	arbitrary	cont lvol or hvol	passed	i, r	yes	same
• Acyclic [A+95]			acyclic			acyclic					
• Fanout [A+95]			hierarchy			hierarchy					
• Pipeline [DG90, Se88, A+95]			linear			linear					
-Unix pipes and filters [Ba86a]		ascii stream			i				i		
Key to column entries											
Synchronicity	asynch (asynchronous)										
Binding time	i (invocation-time), r (run-time)										
Continuity	cont (continuous), hvol (high-volume), lvol (low-volume)										

(The pipe-and-fi lter example seen earlier is called*pipeline* here.)

Reference: M. Shaw and P. Clements. A Field Guide to Boxology: Preliminary Classifi cation of Architectural Styles for Software Systems. Proc. COMPSAC'97, 21st Int'l Computer Software and Applications Conference, August 1997, pp. 6-13.

Andrew’s classifications of communicating-process architectures:

- one-way data flow
- client-server-style request and reply
- back-and-forth (heartbeat) interaction between neighboring processes
- probes and echoes from a process to its successors
- message broadcast
- token passing (for control/access privileges)
- coordination between replicated servers
- decentralized workers

Table 2: Specializations of the interacting processes style

Style	Constituent parts		Control issues			Data issues				Ctrl/data interaction	
	Components	Connectors	Topology	Synchronicity	Binding time	Topology	Continuity	Mode	Binding time	Isomorphic shapes	Flow directions
Interacting process styles: Styles dominated by communication patterns among independent, usually concurrent, processes											
Communicating processes [An91, Pa85]	processes	message protocols	arb	any but seq	w, c, r	arb	spor lvol	any	w, c, r	possibly	if isomorphic either
One-way data flow, networks of filters			linear	asynch		linear		passed	w, c	yes	same
Client/server request/reply			star	synch		star		passed		yes	opposite
Heartbeat			hier	ls/par		hier or star		passed shared ci/co		no	same
Probe/echo			incomplete graph	asynch		incomplete graph		passed		yes	same
Broadcast			arb	asynch		star		bdcast		no	same
Token passing Decentralized servers			arb	asynch		arb.		passed		yes	same
Replicated workers			hier	synch		hier		passed shared		yes	yes
Key to column entries											
Topology	hier (hierarchical), arb (arbitrary), star, linear (one-way)										
Synchronicity	seq (sequential, one thread of control), ls/par (lockstep parallel), synch (synchronous), asynch (asynchronous), opp (opportunistic)										
Binding time	w (write-time--that is, in source code), c (compile-time), i (invocation-time), r (run-time)										
Continuity	spor (sporadic), lvol (low-volume)										
Mode	shared, passed, bdcast (broadcast), mcast (multicast), ci/co (copy-in/copy-out)										

A two-slide table of architectural styles:

Table 1: A feature-based classification of architectural styles

Style	Constituent parts		Control issues			Data issues				Control/data interaction		Type of reasoning
	Components	Connectors	Topology	Synchronicity	Binding time	Topology	Continuity	Mode	Binding time	Isomorphic shapes	Flow directions	
Data flow styles: Styles dominated by motion of data through the system, with no "upstream" content control by recipient												Functional composition
Batch sequential [Be90]	stand-alone programs	batch data	linear	seq	r	linear	spor hvol	passed, shared	r	yes	same	
Dataflow network [B+88]	transducers	data stream	arb	asynch	i, r	arb	cont lvol or hvol	passed	i, r	yes	same	
•Sub-styles	*See Section 4.1*											
Closed loop control [Sh95]	embedded process, function	continuous refresh	fixed	asynch	w	fixed cyclic[1]	cont lvol	passed, shared	w	no	n/a	
Call-and-return styles: Styles dominated by order of computation, usually with single thread of control												Hierarchy (local reasoning)
Main program/subroutines [Pa72, Bo86]	procedures, data	procedure calls	hier	seq	w, c	arb	spor lvol	passed, shared	w, c, r	no	n/a	
Information hiding systems [Pa72]	managers	procedure calls	arb	seq	w, c, r	arb	spor lvol	passed	w, c, r	yes	same	
•Abstract data types [Sh81]	managers	static procedure calls	arb	seq	w, c	arb	spor lvol	passed	w, c, r	yes	same	
•Classical[2] objects [Bo86]	managers (objects)	dynamic procedure calls	arb	seq	w, c, r	arb	spor lvol	passed	w, c, r	yes	same	
•Naive[3] client/server	programs	procedure calls or RPCs	star	synch	w, c, r	star	spor lvol	passed	w, c, r	yes	opposite	
Interacting process styles: Styles dominated by communication patterns among independent, usually concurrent, processes												Nondeterminism
Communicating processes [An91, Pa85]	processes	message protocols	arb	Any but seq	w, c, r	arb	spor lvol	any	w, c, r	possibly	if isomorphic, either	
•Lightweight processes	lightweight processes	threads, (shared data[4])	arb	ls/par, synch	w, c	arb	spor (th), cont (da)	passed, shared	w, c	no	n/a	
•Distributed objects	managers	remote proc calls	arb	ls/par, synch	w, c, r	arb	spor lvol	passed	w, c, r	no	n/a	
•Process-based naive client/server[3]	processes	request/reply messages	star	synch	w, c, r	star	spor lvol	passed	w, c, r	yes	opposite	
•Other sub-styles	*See Section 4.2*											
Event systems [Ba86b, G+92, Ge89, HN86, He69, KP88, Re90]	processes	implicit invocation	arb	asynch, opp	i, r	arb	spor lvol	bdcast	i, r	no	n/a	

Table 1: A feature-based classification of architectural styles

Style	Constituent parts		Control issues			Data issues				Control/data interaction		Type of reasoning
	Components	Connectors	Topology	Synchronicity	Binding time	Topology	Continuity	Mode	Binding time	Isomorphic shapes	Flow directions	
Data-centered repository styles: Styles dominated by a complex central data store, manipulated by independent computations												Data integrity
Transactional database [Be90, Sp87]	memory, computations	trans. streams (queries)	star	asynch, opp	w	star	spor lvol	shared, passed	w	possibly	if isomorphic, opposite	ACID[5] properties
•Client/server	managers, computations	transaction opns with history[3]	star	asynch.	w, c, r	star	spor lvol	passed	w, c, r	yes	opposite	
Blackboard [Ni86]	memory, computations	direct access	star	asynch, opp	w	star	spor lvol	shared, mcast	w	no	n/a	convergence
Modem compiler [SG96]	memory, computations	procedure call	star	seq	w	star	spor lvol	shared	w	no	n/a	invariants on parse tree
Data-sharing styles: Styles dominated by direct sharing of data among components												
Compound document	editable documents	shared representation				hier	cont	shared	r			
Hypertext	documents	internal refs.	n/a	n/a	n/a	arb	cont	shared	w, c, r	n/a	n/a	Representation
Fortran common, Jovial Compool	data structures	sharing (aliasing)				arb	cont	shared	w, c			
Lightweight processes[4]	*See interacting processes style group. This style hybridizes processes and shared data, with emphasis on process*											
Hierarchical styles: Styles dominated by reduced coupling, with resulting partition of the system into subsystems with limited interaction												
Layered [Fr85, LS79]	various	various	hier	any	any	hier	spor lvol, cont	any	w, c, i, r	often	same or opp	Levels of service
•Interpreter (Virtual machine) [HR85]	memory, state machine	direct data access	fixed hier	seq	w, c	hier	cont	shared	w, c	no	n/a	

Key to column entries	
Topology	hier (hierarchical), arb (arbitrary), star, linear (one-way), fixed (determined by style)
Synchronicity	seq (sequential, one thread of control), ls/par (lockstep parallel), synch (synchronous), asynch (asynchronous), opp (opportunistic)
Binding time	w (write-time--that is, in source code), c (compile-time), i (invocation-time), r (run-time)
Continuity	spor (sporadic), cont (continuous), hvol (high-volume), lvol (low-volume)
Mode	shared, passed, bdcast (broadcast), mcast (multicast), ci/co (copy-in/copy-out)

Notes:

1. Closed loop control establishes a controlling relation between an embedded process and a control function that responds to perturbations.
2. By "classical object" we mean objects as they originally emerged: non-concurrent, interacting via procedure-like methods. Objects are now often defined much more broadly, especially in their types of interactions.
3. True client/server systems maintain context that captures the current state of an ongoing series of actions. "Client/server" is sometimes used to describe systems that ignore this requirement and simply use components that call and define procedures or send request/reply messages amonge processes. We call the latter "naive client/server systems."
4. Lightweight processes may take advantage of the shared name space; they become a hybrid of communicating processes and shared data.
5. The ACID properties are atomicity, consistency, isolation, and durability.

How do we select a style of software architecture?

Shaw gives this simple *checklist* from A Field Guide to Boxology, COMPSAC'97:

(1) If the problem can be decomposed into *sequential stages*, consider a *data-flow architecture*: batch sequential or pipeline.

In addition, if each stage is incremental, so that later stages can begin before earlier stages finish, consider a pipeline architecture.

(2) If the problem involves *transformations on continuous streams* of data (or on very long streams), consider a *pipeline architecture*.

But the problem passes “rich” data representations, avoid pipelines restricted to ASCII.

(3) If the central issues are *understanding the data* of the application, its *management*, and *representation*, consider a *repository or abstract-data-type architecture*. If the data is long-lived, focus on repositories.

If the representation of data is likely to change over the lifetime of the program, than abstract data types can confine the changes to particular components.

If you are considering repositories and the input data has a low signal-to-noise ratio and the execution order cannot be predetermined, consider a blackboard.

If you are considering repositories and the execution order is determined by a stream of incoming requests and the data is highly structured, consider a database management system.

(4) If your system involves controlling *continuing action*, is embedded in a *physical system*, and is subject to *unpredictable external pertubation* so that preset algorithms go wrong, consider a *closed-loop control architecture*.

(5) If you have designed a computation but *have no machine* on which you can execute it, consider an *interpreter architecture*.

(6) If your task requires a *high degree of flexibility/configurability*, loose coupling between tasks, and reactive tasks, consider *interacting processes*.

If you have reason not to bind the recipients of signals from their originators, consider an event architecture.

If the tasks are of a hierarchical nature, consider a replicated worker or heartbeat style.

If the tasks are divided between producers and consumers, consider client/server.

If it makes sense for all of the tasks to communicate with each other in a fully connected graph, consider a token-passing style.

Architectural views: stating and satisfying requirements

A building is too complex to be described in just one way — multiple *views* are presented. An architect might draw these views:

- floor plans
- elevation drawings
- electrical and plumbing diagrams
- traffic patterns
- sunlight and solar views

The views help show how the building's requirements are satisfied by the architecture.

But the views also direct the implementation: Some of the views are "aspects" that might be "woven" into the construction; others are "properties" of the construction (that should be monitored or enforced).

Process-driven design: *4+1 view* model (Kruchten)

A software architecture might be “viewed” four different ways:

1. *logical*: behavior requirements — function, key abstractions, domain elements, data flow
2. *process*: distribution, concurrency, coordination, synchronization
3. *development*: organization of software modules
4. *physical*: deployment onto hardware — performance, reliability, scalability

Finally, *scenarios* direct the design and show how the views “operate” and “work together” (scenarios generate an “execution view”)

Some of the views present aspects that can be woven into the system; others present properties (that should be monitored or enforced).

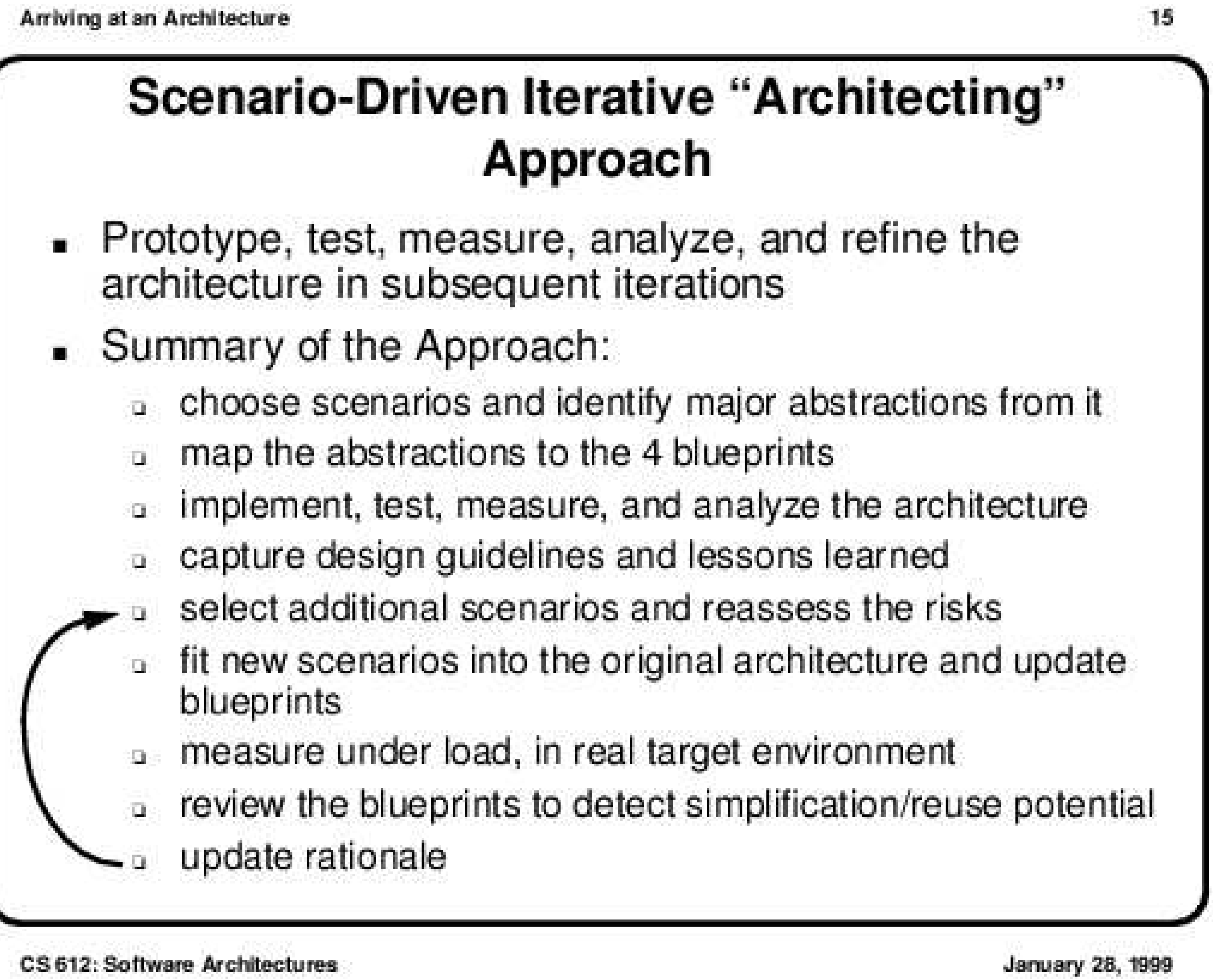


Diagram is from Medvidovic's course, `http://sunset.usc.edu/classes/cs578_2002`

You are not limited to just Kruchten's "4+1" views.

The different formats of *UML diagrams* can be used to present different views of an architecture:

- use-case diagrams logical/scenarios
- class diagrams logical/development
- package diagrams development/process
- sequence diagrams process/physical
- collaboration diagrams process/physical
- state-transition diagrams process

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.

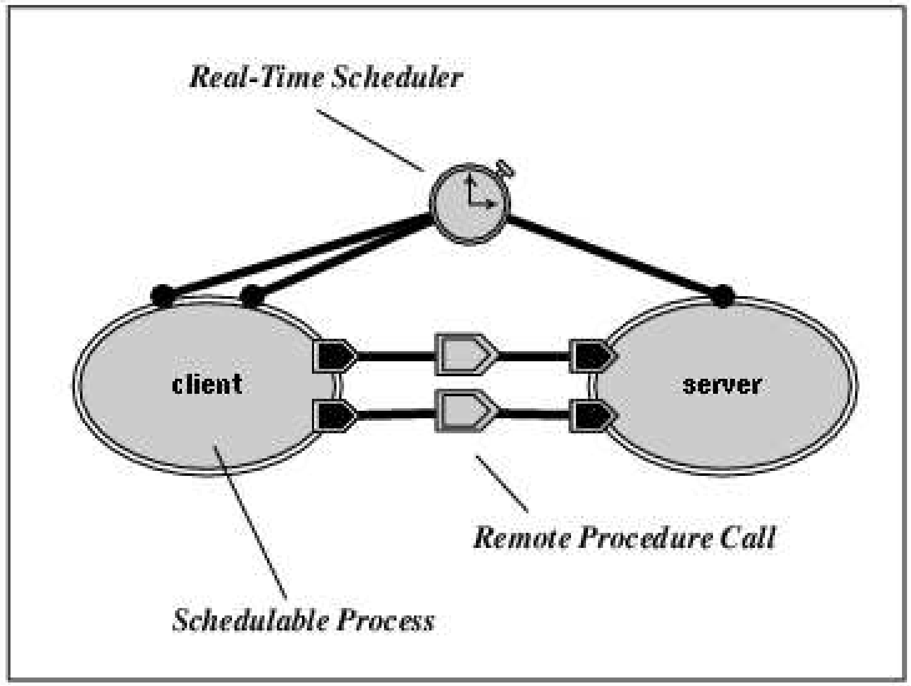
Connectors **are specified by** ***protocols*****; they have**

(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.

Reference: M. Shaw, R. DeLine, and G. Zelesnik, Abstractions and Implementations for Architectural Connections. In 3d Int. Conf. Confi gurable Distributed Systems, Annapolis, Maryland, May 1996.

```
component Real_Time_System
  interface is
    type General
  end interface

  implementation is
    uses client interface rtclient
      PRIORITY (10)
      ENTRYPOINT (client)
      end client

    uses server interface rtserver
      PRIORITY (9)
      RPCTYPEDEF (new_type; struct; 12)
      RPCTYPESIN ("unicon.h")
      end server

    establish RTM-realtime-sched with
      client.application1 as load
      client.application2 as load
      server.services as load
      ALGORITHM (rate_monotonic)
      PROCESSOR ("TESTBED.XX.EDU")
      TRACE (client.application1.
                 external_interrupt1;
        client.application1.work_block1;
        server.services.work_block1;
        client.application1.work_block2;
        server.services.work_block2;
        client.application1.work_block3)
      TRACE (client.application2.
                 external_interrupt2;
        client.application2.work_block1;
        server.services.work_block1;
        client.application2.work_block2;
        server.services.work_block2;
        client.application2.work_block3)
      end RTM-realtime-sched

    establish RTM-remote-proc-call with
      client.timeget as caller
      server.timeget as definer
      IDLTYPE(Mach)
      end RTM-remote-proc-call

    establish RTM-remote-proc-call with
      client.timeshow as caller
      server.timeshow as definer
      IDLTYPE(Mach)
      end RTM-remote-proc-call
  end implementation
end Real_Time_System
```

```
component RTClient
  interface is
    type SchedProcess
    PROCESSOR ("TESTBED.XX.EDU")
    TRIGGERDEF (external_interrupt1; 1.0)
    TRIGGERDEF (external_interrupt2; 0.5)
    SEGMENTDEF (work_block1; 0.02)
    SEGMENTDEF (work_block2; 0.03)
    SEGMENTDEF (work_block3; 0.05)
    player application1 is RTLoad
      TRIGGER (external_interrupt1)
      SEGMENTSET (work_block1,
        work_block2, work_block3)
      end application1
    player application2 is RTLoad
      TRIGGER (external_interrupt2)
      SEGMENTSET (work_block1,
        work_block2, work_block3)
      end application2
    player timeget is RPCCall
      SIGNATURE ("new_type *"; "void")
      end timeget
    player timeshow is RPCCall
      SIGNATURE ("void"; "void")
      end timeshow
  end interface

connector RTM-realtime-sched
  protocol is
    type RTScheduler
    role load is load
  end protocol

  implementation is
    builtin
  end implementation
end RTM-realtime-sched

connector RTM-remote-proc-call
  protocol is
    type RemoteProcCall
    role definer is definer
    role caller is caller
  end protocol

  implementation is
    builtin
  end implementation
end RTM-remote-proc-call
```

uses statements instantiate the parts composed

connect statements state how players satisfy roles

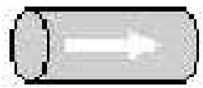
bind statements map the external interface to the internal configuration

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

Pipe Connector

Informal Description: The Unix abstraction for pipe, i.e. a bounded queue of bytes that are produced at a source and consumed at a sink. Also supports interactions between pipes and files, choosing the correct Unix implementation.

Icon: pipe section 

Properties: *PipeType*, the kind of Unix pipe. Possible values Named, Unnamed

Roles: *Source*

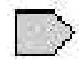
Description: the source end of the pipe
Accepts player types: *StreamOut* of component Filter; *ReadNext* of component SeqFile
Properties: *MinConns*, minimum number of connections. Integer values, default 1
MaxConns, maximum number of connections. Integer values, default 1

Sink

Description: the sink end of the pipe
Accepts player types: *StreamIn* of component Filter; *WriteNext* of component SeqFile
Properties: *MinConns*, *MaxConns*, as for Source

ProcedureCall Connector

Informal Description: The architectural abstraction corresponding to the procedure call of standard programming languages. Requires signatures (eventually pre/post conditions) in the RoutineDef and RoutineCall players to match; if they don't, requests remediation. Supports renaming.

Icon: blunt arrowhead 

Roles: *Definer*

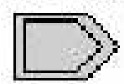
Description: role played by the procedure definition
Accepts player types: *RoutineDef* of component Computation or Module
Properties: *MinConns*, minimum number of definitions allowed. Integer, must be 1
MaxConns, maximum number of definitions allowed. Integer, must be 1

Caller

Description: the role played by the procedure call
Accepts player types: *RoutineCall* of component Computation or Module
Properties: *MinConns*, minimum number of callers allowed. Integer, default 1
MaxConns, maximum number of callers allowed. Integer, default many

RemoteProcCall Connector

Informal Description: The abstraction for the remote procedure call facility supplied by the operating system. Requires signatures and eventually pre/post conditions in the RPCDef and RPCCall players to match. RemoteProcCall connectors require much more UniCon support than ProcedureCall connectors, as they must establish communication paths between processes.

Icon: bordered blunt arrowhead 

Roles: *Definer*

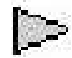
Description: role played by the procedure definition
Accepts player types: *RPCDef* of component Process or SchedProcess
Properties: *MinConns*, *MaxConns*, as for ProcedureCall

Caller

Description: the role played by the procedure call
Accepts player types: *RPCCall* of component Process or SchedProcess
Properties: *MinConns*, *MaxConns*, as for ProcedureCall

DataAccess Connector

Informal Description: The architectural abstraction corresponding to imported and exported data of conventional programming languages.

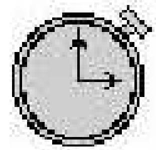


Icon: triangle

Roles: *Definer*, essentially similar to *Definer* of *ProcedureCall*
User, essentially similar to *Caller* of *ProcedureCall*

RTScheduler Connector

Informal Description: Mediates competition for processor resources among a set of real-time processes (requires an operating system with appropriate real-time capabilities).



Icon: stopwatch

Properties: *Algorithm*, the scheduling discipline. Possible values: RateMonotonic, TimeSharing, EarliestDeadline, DeadlineMonotonic, RoundRobinFixPriority, FIFOFixPriority
Processor, the name of the processor on which this set of processes will run
Trace, a path through the real-time code and the trigger that invokes it

Roles: *Load*

Description: the role played by a real-time load on a processor
Accepts player types: *RTLoad* of component SchedProcess
Properties: *MinConns*, minimum number of competing processes. Integer, default 2
MaxConns, maximum number of competing processes. Integer, default many

PLBundler Connector

Informal Description: A composite abstraction for matching definitions and uses of a collection of procedures and data. It allows multiple procedure and data definitions and uses to be matched with a single abstraction. Supports renaming.

Icon: chain links 

Properties: *Match,* the correspondences between individual definitions in the bundles. Values are sets of pairs of names.

Roles: *Participant*

Description: a set of definitions and uses to take part in the linkage
Accepts player types: *PLBundle* of component Computation, Module, or SharedData
Properties: *MinConns*, minimum number of bundles to match. Integer, default 2
MaxConns, maximum number of bundles to match. Integer, default many

Wright: Unicon + CSP

Garlan and Allen developed Wright to specify protocols. Here is a single-client/single-server example:

```
System SimpleExample
    component Server =
        port provide [provide protocol]
        spec [Server specification]
    component Client =
        port request [request protocol]
        spec [Client specification]
    connector C-S-connector =
        role client [client protocol]
        role server [server protocol]
        glue [glue protocol]
Instances
    s: Server
    c: Client
    cs: C-S-connector
Attachments
    s.provide as cs.server;
    c.request as cs.client
end SimpleExample.
```

The *protocols* are specified with Hoare's CSP (Communicating Sequential Processes) algebra.

$$
\begin{aligned}
&\textbf{connector}\ \text{C-S-connector} = \\
&\quad \textbf{role}\ \text{Client} = (\text{request!x} \rightarrow \text{result?y} \rightarrow \text{Client}) \sqcap \S \\
&\quad \textbf{role}\ \text{Server} = (\text{invoke?x} \rightarrow \text{return!y} \rightarrow \text{Server}) \Box \S \\
&\quad \textbf{glue} = (\text{Client.request?x} \rightarrow \text{Server.invoke!x} \rightarrow \text{Server.return?y} \\
&\qquad\qquad \rightarrow \text{Client.result!y} \rightarrow \textbf{glue}) \ \Box \S
\end{aligned}
$$

The *glue* protocol synchronizes the Client and Server roles:

$$
\begin{aligned}
& \texttt{Client} \parallel \texttt{Server} \parallel \texttt{glue} \\
\Rightarrow\ & \texttt{result?y} \rightarrow \texttt{Client} \parallel \texttt{Server} \parallel \texttt{Server.invoke!x} \rightarrow \texttt{...} \\
\Rightarrow\ & \texttt{result?y} \rightarrow \texttt{Client} \parallel \texttt{return!y} \rightarrow \texttt{Server} \parallel \\
& \texttt{Server.return?y} \rightarrow \texttt{...} \\
\Rightarrow\ & \texttt{...} \ \Rightarrow \texttt{Client} \parallel \texttt{Server} \parallel \texttt{glue}
\end{aligned}
$$

Forms of CSP processes:

- prefixing: $e \rightarrow P$

  $\mathit{plusOne?x} \rightarrow \mathit{return!x} + 1 \rightarrow \cdots \,\|\, \mathit{plusOne!2} \rightarrow \mathit{return?y} \rightarrow \cdots$

  $\Rightarrow\ \mathit{plusOne!3} \rightarrow \cdots \,\|\, \mathit{plusOne?y} \rightarrow \cdots$

- external choice: $P \,\Box\, Q$

  $\mathit{plusOne?x} \rightarrow \cdots \ \Box\ \mathit{plusTwo?x} \rightarrow \cdots x + 2 \cdots \,\|\, \mathit{plusTwo!5} \rightarrow \cdots$

  $\Rightarrow\ \cdots 7 \cdots \,\|\, \cdots$

- internal choice: $P \sqcap Q$

  $\mathit{plusOne?x} \rightarrow \cdots \,\|\, \mathit{plusOne!5} \rightarrow \cdots \sqcap \mathit{plusTwo!5} \rightarrow \cdots$

  $\Rightarrow\ \mathit{plusOne?x} \rightarrow \cdots \,\|\, \mathit{plusTwo!5} \rightarrow \cdots$

- parallel composition: $P \| Q$

- halt: $\S$

- (tail) recursion: $p = \cdots p$ (More formally, $\mu z.P$, where z may occur free in P.)

A pipe protocol in Wright

```
connector Pipe =
    role Writer = write→Writer ⊓ close→ §
    role Reader = let ExitOnly = close→ §
                  in let DoRead = (read→Reader [] read-eof→ExitOnly)
                  in DoRead ⊓ ExitOnly
    glue = let ReadOnly = Reader.read→ReadOnly
                          [] Reader.read-eof →Reader.close → §
                          [] Reader.close→ §
        in let WriteOnly = Writer.write→WriteOnly [] Writer.close→ §
        in Writer.write→glue [] Reader.read→glue
          [] Writer.close→ReadOnly [] Reader.close→WriteOnly
```

Reference: R. Allen and D. Garlan. A formal basis for architectural connection. *ACM TOSEM 1997.*

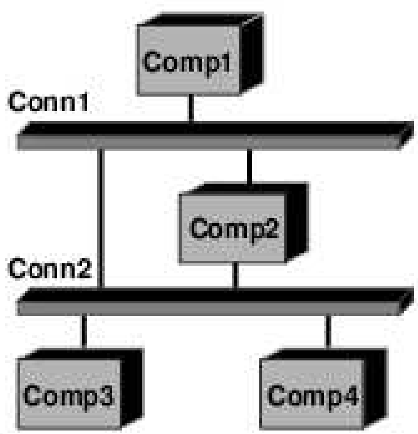
C2: an N-tier framework and language

Developed at Univ. of California, Irvine, Institute of Software Research: `http://www.isr.uci.edu/architecture/c2.html`

Examples of Domain- and Style-Specific Architectures 12

C2

- A component- and message-based style
  - for highly distributed software systems
- Generalized from GUI intensive systems' architectures
- C2 architectures are networks of concurrent components hooked together by connectors



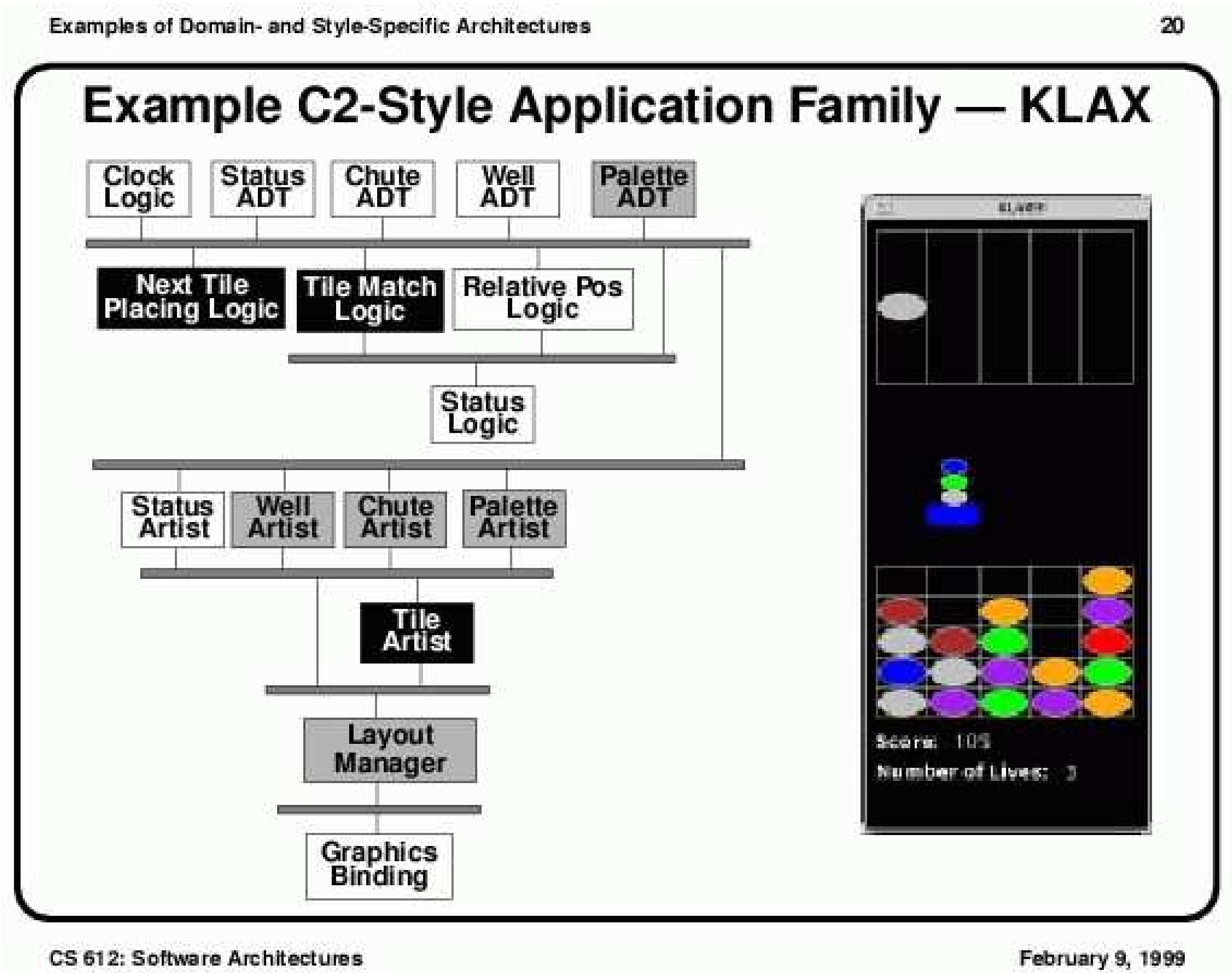
  - no component-to-component links
  - "one up, one down" rule for components
  - connector-to-connector links are allowed
  - "many up, many down" rule for connectors
  - all communication by exchanging messages
  - *substrate independence*

CS 612: Software Architectures February 9, 1999

Diagrams are from Medvidovic's course,

`http://sunset.usc.edu/classes/cs578_2002`

Example architecture in C2: video game

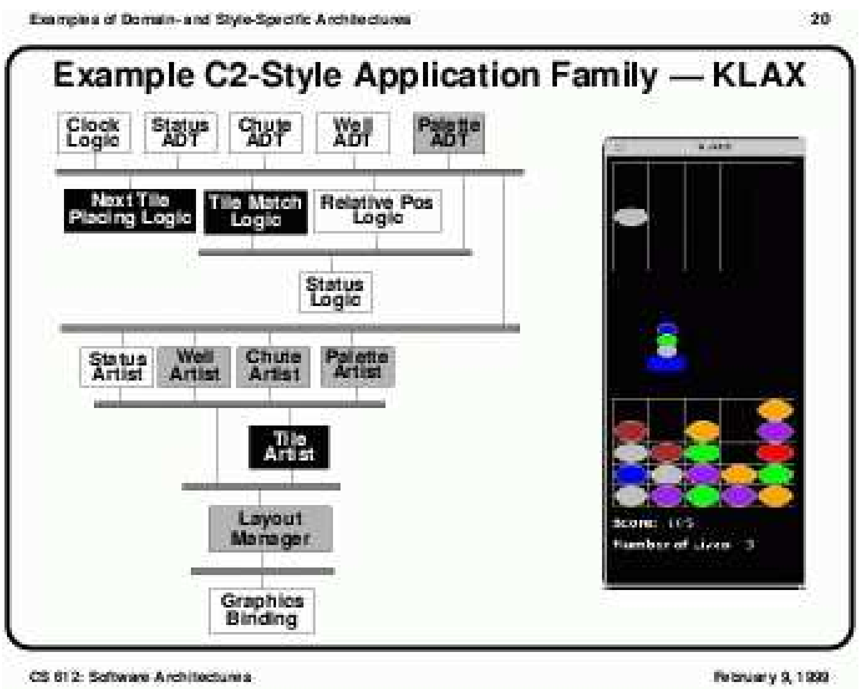


Here is a *C2SADEL* description of the video game's "Well" component:

```
component WellADT is subtype Matrix (beh) {
   state {
      capacity : Integer;
      num_tiles : Integer;
      well_at : Integer -> GSColor;
   }
   invariant {
      (num_tiles \eqgreater 0) \and (num_tiles \eqless capacity);
   }
   interface {
      prov gt1: GetTile (location : Integer) : Color;
      prov gt2: GetTile (i : Natural) : GSColor;
   }
   operations {
      prov tileget: {
         let pos : Integer;
         pre (pos \greater 0) \and (pos \eqless num_tiles);
         post \result = well_at(pos) \and ~num_tiles = num_tiles - 1;
      }
   }
   map {
      gt1 -> tileget (location -> pos);
      gt2 -> tileget (i -> pos);
   }
}
```

Reference: N. Medvidovic, et al. A Language and Environment for Architecture-Based Software Development and Evolution. 21st Int. Conf. on Software Engineering, Los Angeles, May 1999.

And here is a description of a connector and part of the configuration:




```
connector BroadcastConn is {
   message_filter no_filtering;
}

architectural_topology {
   component_instances {
      Well : WellADT;
      WellArt : WellArtist;
      MatchLogic : TileMatchLogic;
   }
   connector_instances {
      ADTConn : BroadcastConn;
      ArtConn : BroadcastConn;
   }
   connections {
      connector ADTConn {
         top Well;
         bottom MatchLogic, ArtConn;
      }
      connector ArtConn {
         top ADTConn;
         bottom WellArt;
      }
   }
}
```

So then, 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-specifi c design

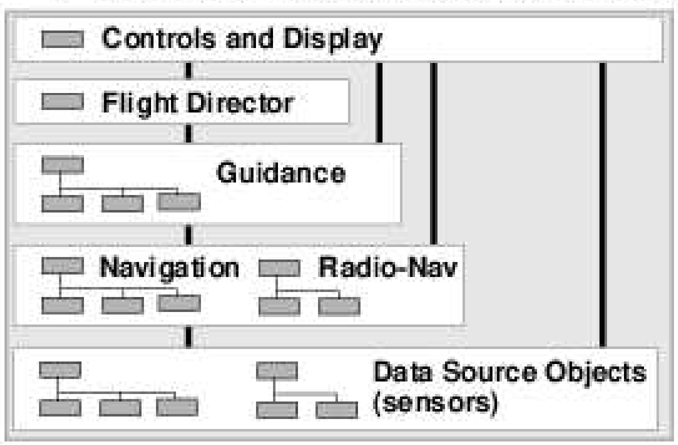
If the problem domain is a standard one (e.g., flight-control or telecommunications or banking), then there are precedents to follow.

A *Domain-Specific Software Architecture* has

- a *domain*: defines the problem area domain concepts and terminology; customer requirements; scenarios; confi guration models (entity-relationship, data fbw, 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

- **A**vionics **D**omain **A**pplication **G**eneration **E**nvironment
- Layered reference architecture
  - subsystems decomposed into primitive components with standardized interfaces
  - over 40 different realms with over 350 distinct components
    - realm $\equiv$ { x : 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`

Domain-specific language (DSL)

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

A DSL is useful for describing a problem and its solution in concepts familiar to people who work in the domain.

It might be used to define (entity-relationship) models, architectures, and implementations.

In effect, defining the *domain* and *reference requirements* (features) of a Domain Specific Software Architecture is defining (most of) a DSL.

When a DSL is designed explicitly for describing a computer implementation, it is a *domain-specific programming language*.

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 HTML, XML, SQL, and even regular-expression notation (as embedded in, say, Perl or Python)

These are good examples of *top-down* domain-specific programming, because they give high-level, direct expression of domain concepts.

The *bottom-up* approach to domain-specific programming, sometimes called *in-language DSL*, uses a language like Scheme or Smalltalk or Python to write many little functions that encode domain-concepts-as-code, thus “building the language upwards towards the problem to be solved.”

From DSLs to product lines (Steve Cook, Microsoft)

A *model* is a representation, written in a DSL, whose elements correspond to domain elements/concepts. It helps stakeholders (users, managers, implementors) communicate about the system.

A *framework* is a collection of components that implement the domain's aspects/features. (Example: GUI frameworks)

The model should show how to build upon or extend the framework to generate an application.

A *pattern* is a "model with holes" with rules for filling the holes.

A *value chain* is a manufacturing process where each participant takes inputs (goods or information) from suppliers, adds "value," and passes the output to the successors in the chain.

A *product line* is a software value chain, based on domain-specific models, patterns, and frameworks:
requirements engineer → architect → developer → tester → user

6. Product lines

Product lines

are also known as *software system families*. They are software products that share an architecture and components. They are inspired by industrial assembly lines, e.g., for manufacturing 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 the products' variations (*features*) be precisely stated?
guidance: Is there a reference architecture, parameterized on the variations, that guides us in generating the products?

An example product line: Cummins Corporation

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.

Reference: S. Cohen. Product line practice state of the art report. CMU/SEI-2002-TN-017.

Features and feature diagrams

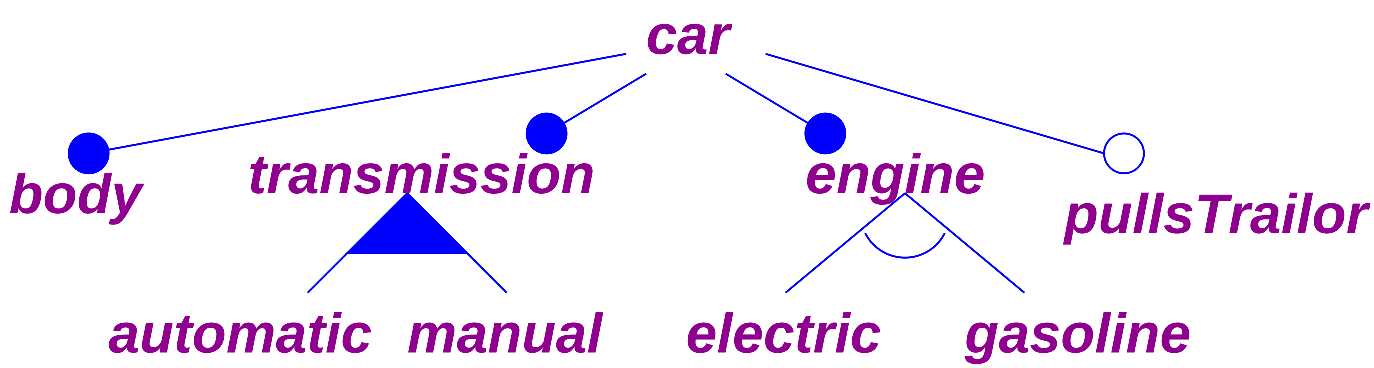
are a development tool for domain-specific architectures and product lines. They help define a domain's reference requirements and guide implementions of instances of the reference architecture.

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.

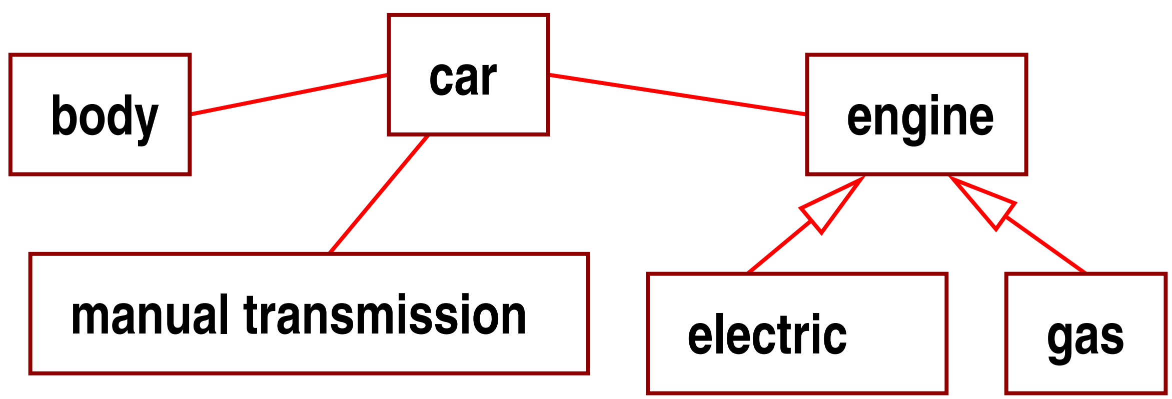
It is a form of decision tree with *and-or-xor* branching, and its hierarchy reflects dependencies of features as well as modification costs.

Feature diagram for assembling automobiles

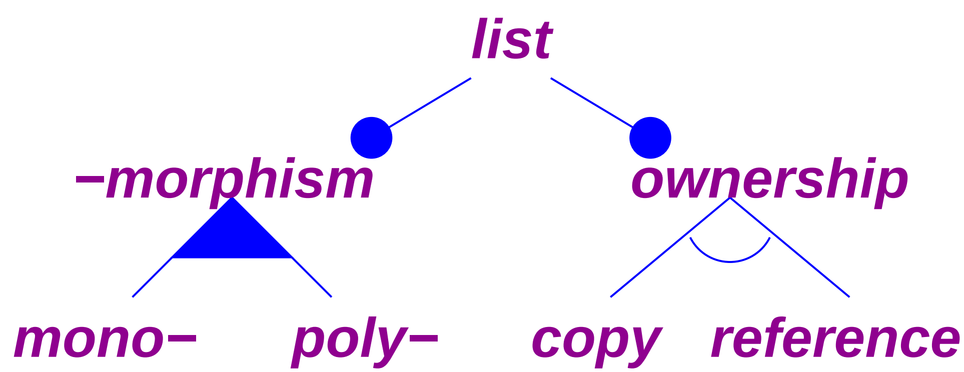


Filled circles label required features; unfilled circles label optional ones. Filled arcs label xor-choices; unfilled arcs label or-choices (where at least one choice is selected).

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.

Reference: K. Czarnecki and U. Eisenecker. *Generative Programming.* Addison-Wesley 2000.

Feature generation is implemented by

Explored Variability Mechanisms

IESE

	Advantages	Disadvantages
Naive Impl.	wide-spread, simple	unscaleable, unmaintainable
Conditional Compilation	wide-spread, no space/ performance loss fine-grained	no direct language support, unscalable
Subtype Polymorphism	rather wide-spread, dynamic	space/performance loss, OR-variability difficult to express
Parametric Polymorphism	no performance loss, all 3 kinds of variability expressible, built-in	less wide-spread, less supported
Ad-Hoc Polymorphism	rather wide-spread	less important than universal polymorphism
Collaborations	good support for PL Implementation	not wide-spread, might require tools
Aspect-Orientation	support for PL Implementation	often requires tools, market caution
Frame Technology	good support for PL Implementation	not wide-spread, additional tool

Reference: D. Muthig, Software product lines and reengineering. Fraunhofer Inst. 2002

Generative programming

is the name given to the application of programs that generate other programs (cf. “automatic programming” in the 1950s). A compiler is of course a generating program, but so are feature-diagram-driven frameworks, partial evaluators, and some development environments (e.g., for Java beans).

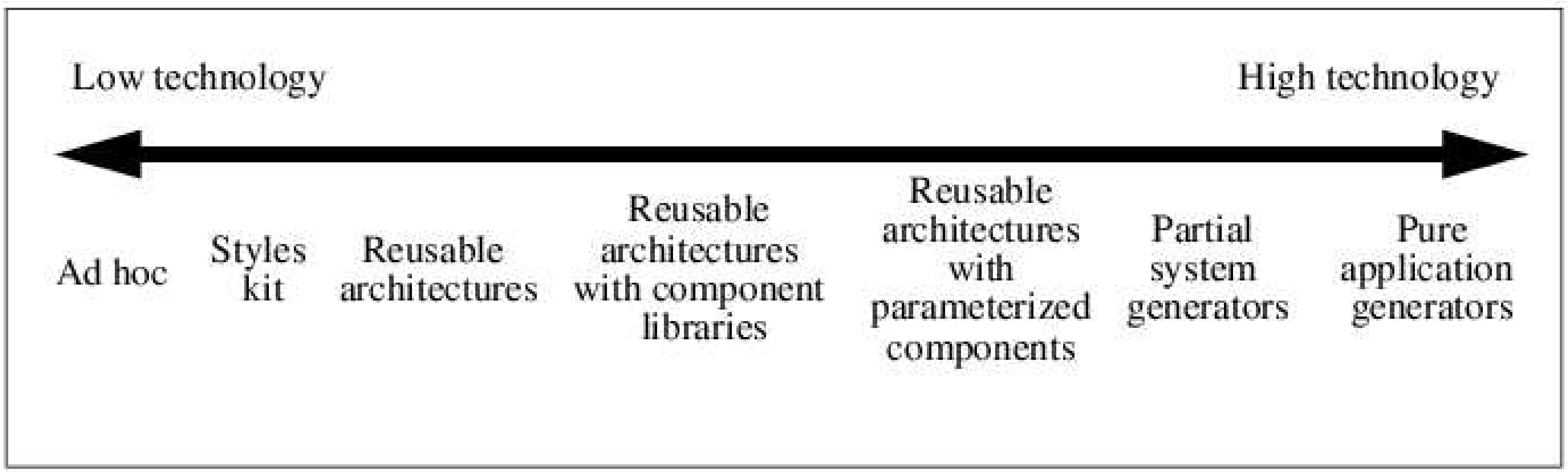
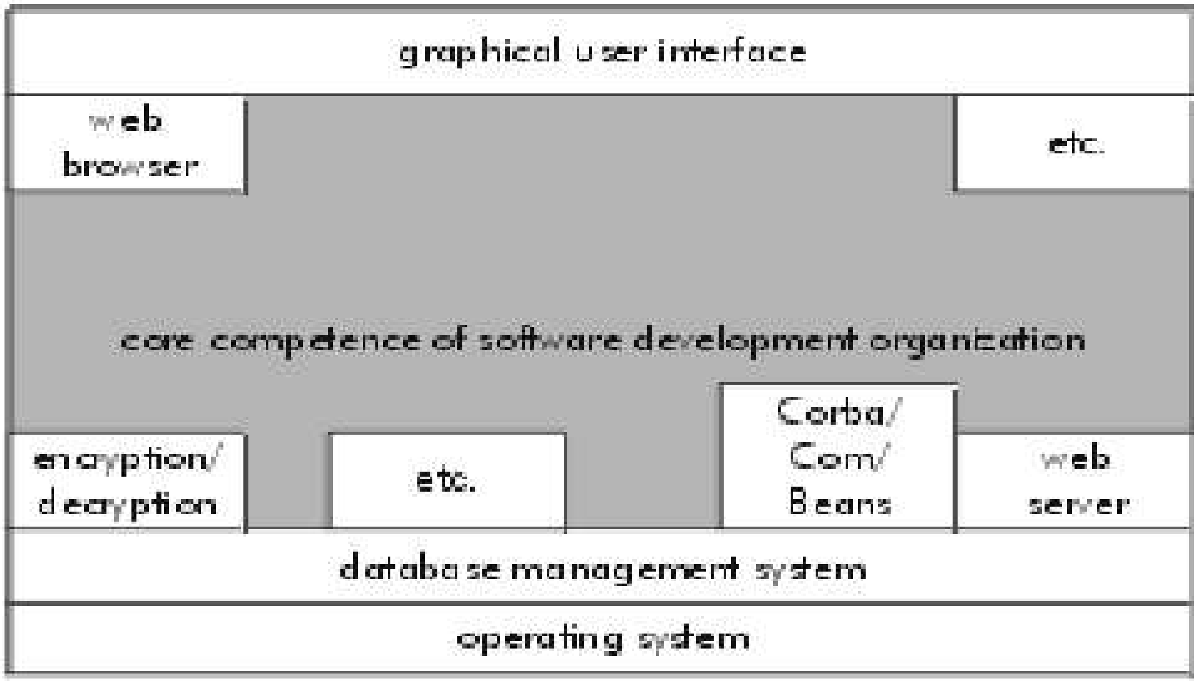


Figure 1: Technology spectrum for architecture selection and creation

Reference: Coming attractions in software architecture, P. Clements. CMU/SEI-96-TR-008.

Generative programming is motivated by the belief that conventional software production methods (even those based on “object-oriented” methodologies) will never support component reuse:



software product

Reference: Jan Bosch. *Design and Use of Software Architectures*. Addison-Wesley, 2000.

One solution is to understand a software system as a customized product, produced by generative programming, from a product line.

Reference: K. Czarnecki and U. Eisenecker. *Generative Programming.* Addison-Wesley 2000.

Software factories (Microsoft)

A *software factory* is a “meta-software-product line”: it combines *DSLs, patterns, models, frameworks, tools,* and *guidance* to “accelerate life-cycle tasks for a type of software application” [Steve Cook, Microsoft].

That is, it is a kind of “product line” for assembling the correct language, architecture, and software components of a software product line — a kind of software-industrial engineering.

DSLs and XML provide the language for assembling and using the software factory.

The goal is complete automation of sofware development — no more coding (except in DSLs (-:   )

Reference: J. Greenfi eld, et al. *Software Factories*, Wiley, 2004. See also Microsoft Visual Studio Team 2005.

7. Middleware

Middleware: a popular form of domain-specific software architecture

Middleware lies between hardware and software in the design of independent-component (and distributed) 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.

Middleware typically demands these hardware services:

- remote communication protocols
- global naming services
- security services
- data transmission services

Warning! The term, “middleware,” is overused and abused — almost any tool that provides a run-time platform is called “middleware” these days.

CORBA: Common Object Request Broker Architecture

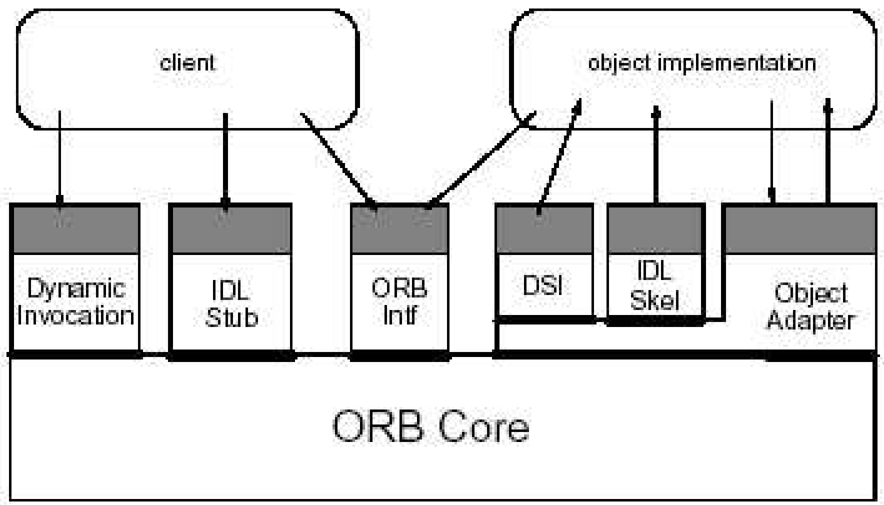
CORBA is middleware for building distributed, object-based, client-server architectures; developed by the Object Management Group (OMG).

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.

Diagram is from: S. Vinoski. CORBA: Integrating Diverse Applications Within Distributed Heterogeneous Environments. *IEEE Communications*, Feb. 1997.

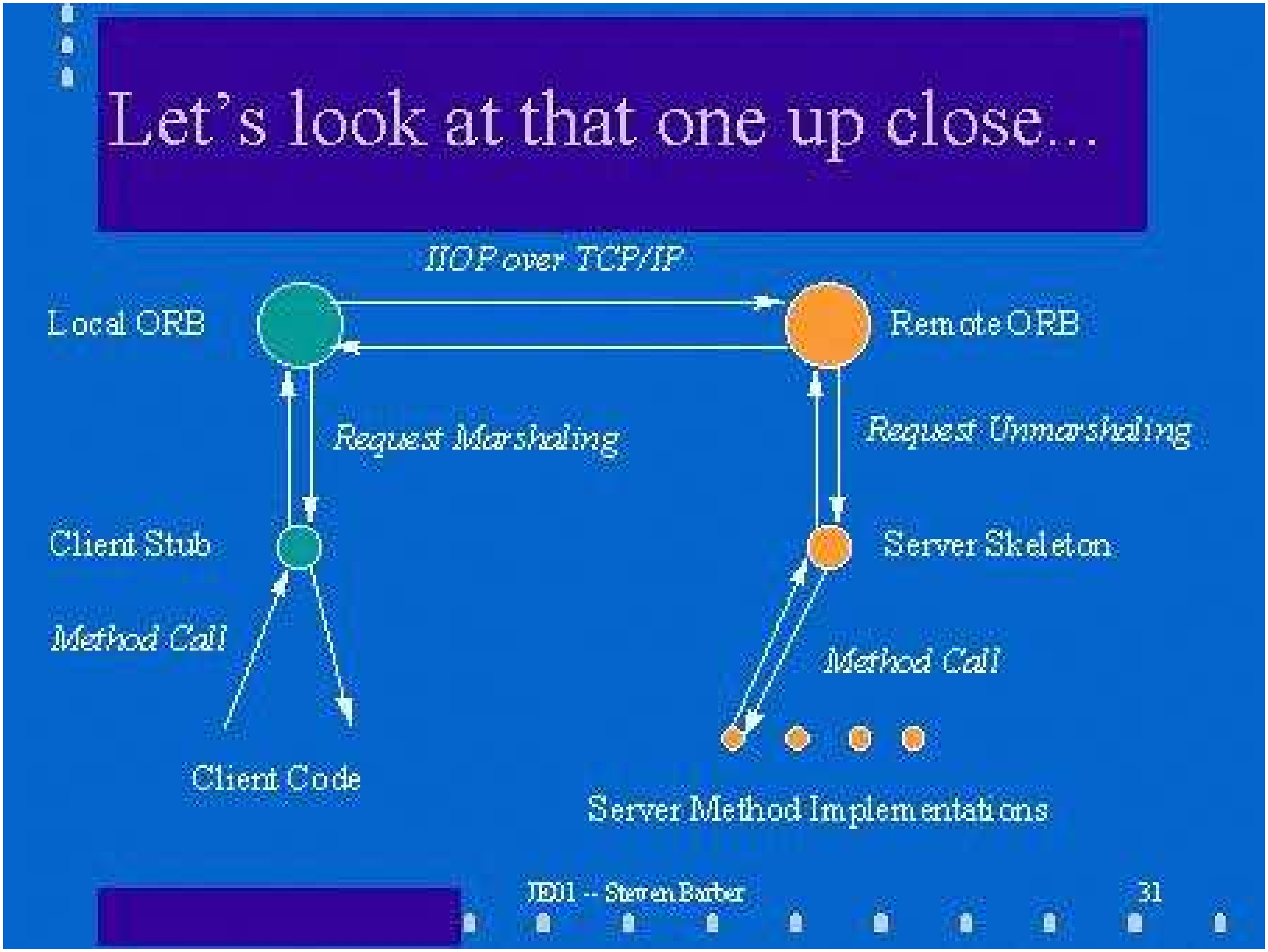
How connectors work

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...
IIOP over TCP/IP
Local ORB
Remote ORB
Request Marshaling
Request Unmarshaling
Client Stub
Server Skeleton
Method Call
Method Call
Client Code
Server Method Implementations
JE01 -- Steven Barber
31

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:

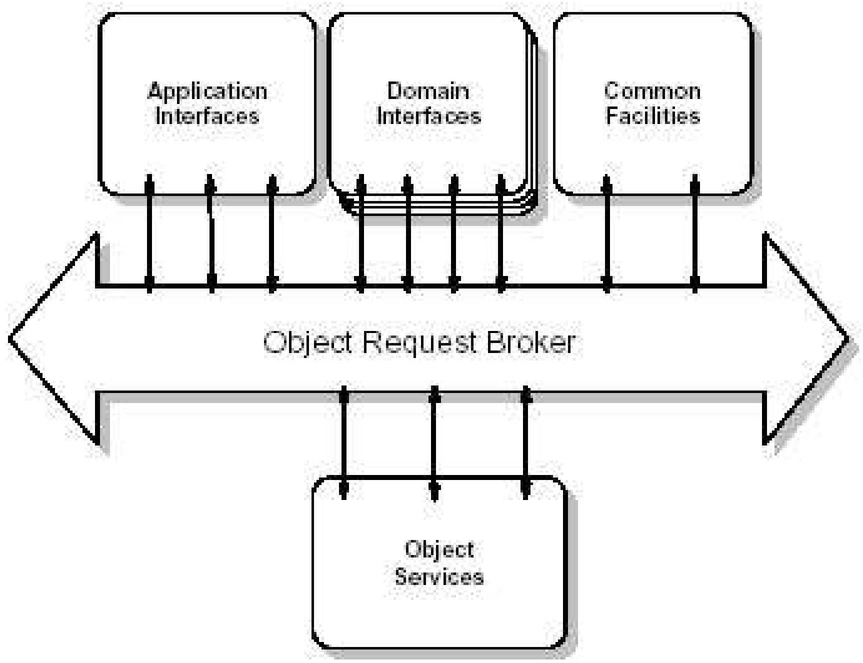
```
// C++
Factory_var factory_objref;

// 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.

Reference: S. Vinoski. CORBA. *IEEE Communications*, Feb. 1997.

Structure of the *Reference Model* for CORBA

An implementation of CORBA must support this structure:



Object Services are interfaces for the ORB, providing transmission, security, and server lookup by naming and “trading” (property).

Domain Interfaces are the object interfaces for the problem area (telecommunication, financial, medical).

Application Interfaces are object interfaces for the application; written by the software architect.

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)-:

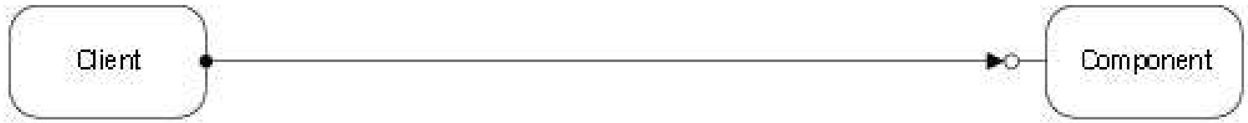


Figure 1 - COM Components in the same process

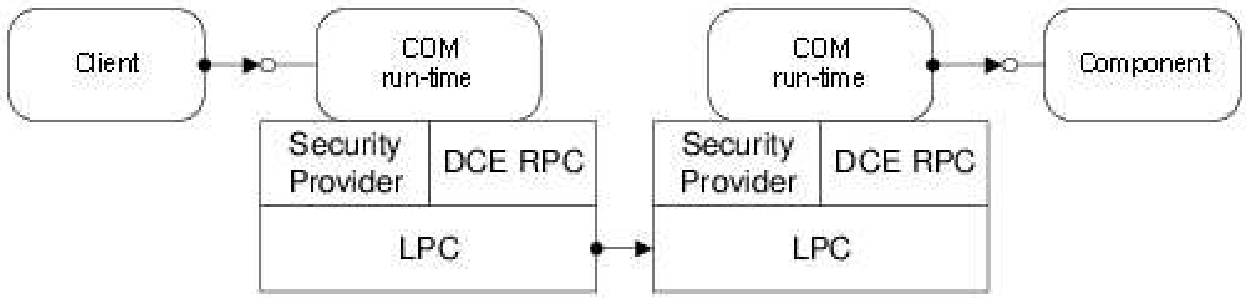


Figure 2 - COM Components in different processes

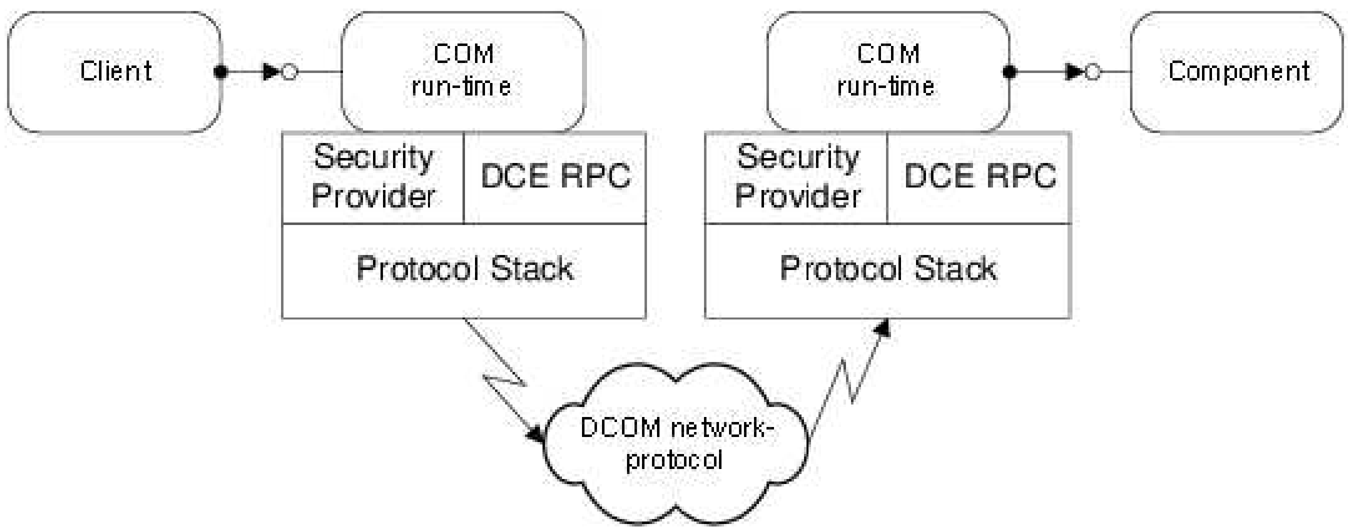
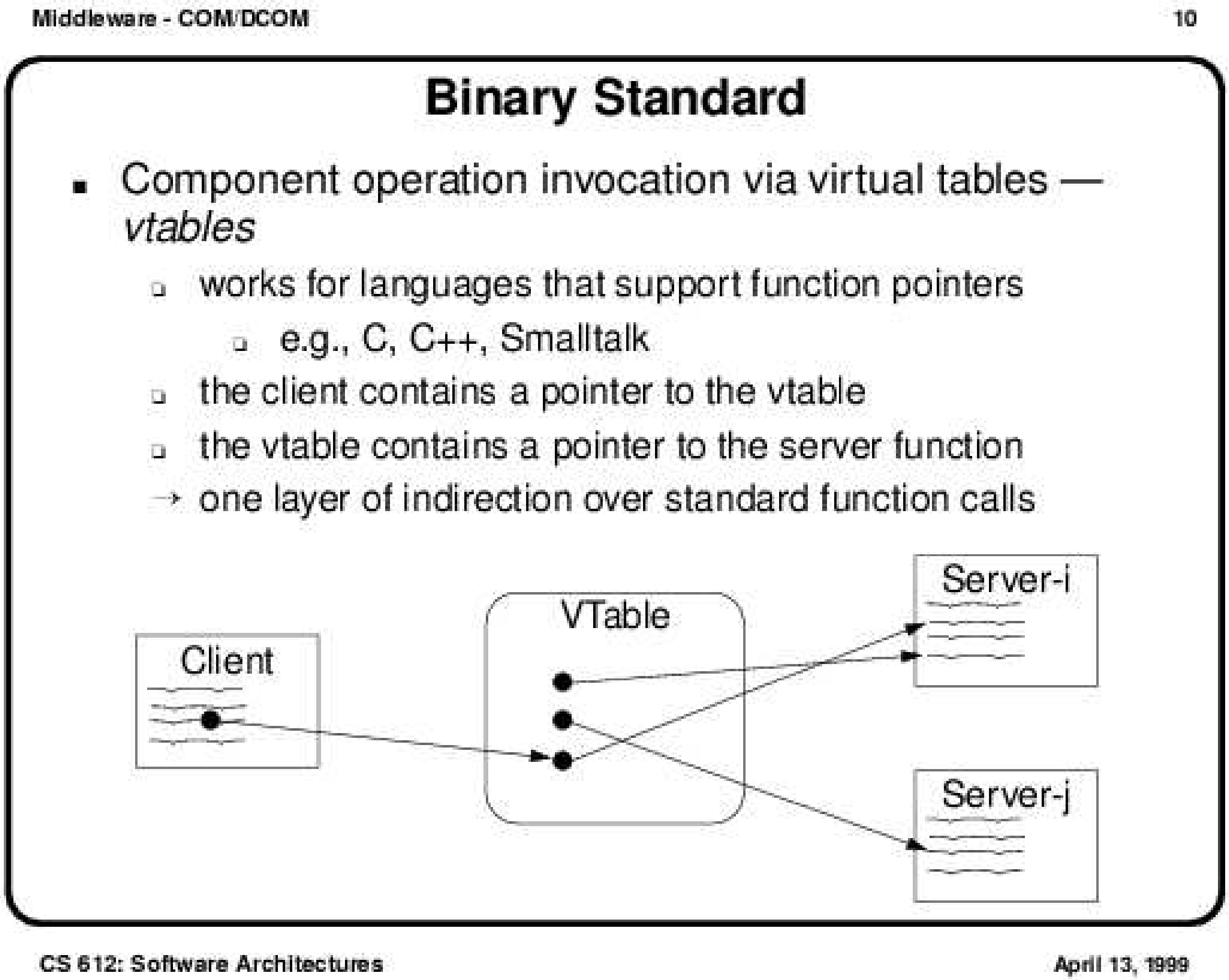


Figure 3 - DCOM: COM Components on different machines

Reference: DCOM Technical Overview. Microsoft Windows/NT white paper, 1996.

DCOM uses a *virtual table* to implement communication, as function call, as efficiently as possible:



Reference: `http://sunset.usc.edu/classes/cs578_2002`

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 that uses the bean’s “introspection” methods.

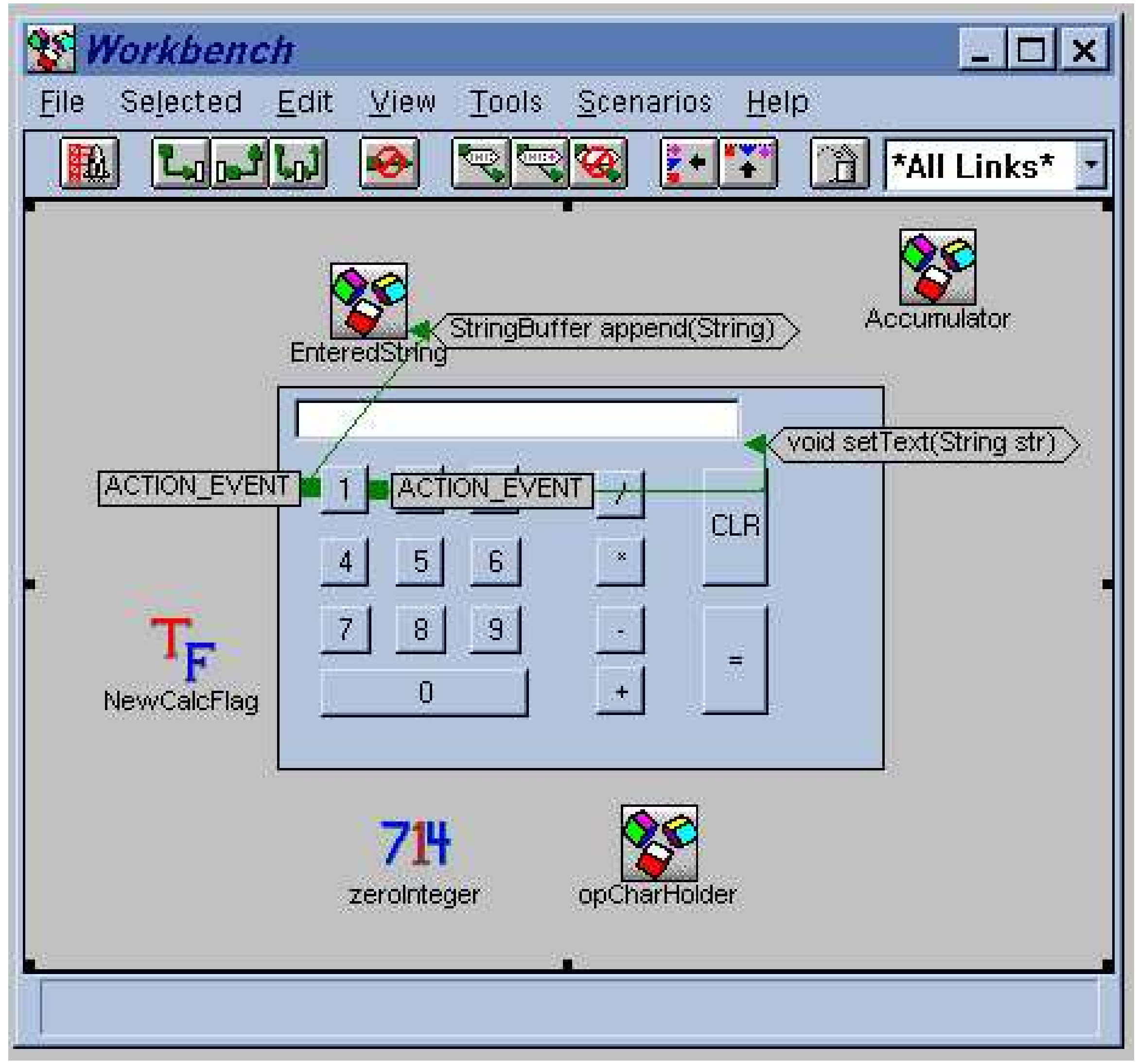
A development tool (the *bean box*) uses a bean's design-time interface to help an application builder position a bean in the application, customize its appearance, and select its run-time behaviors (methods).

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:



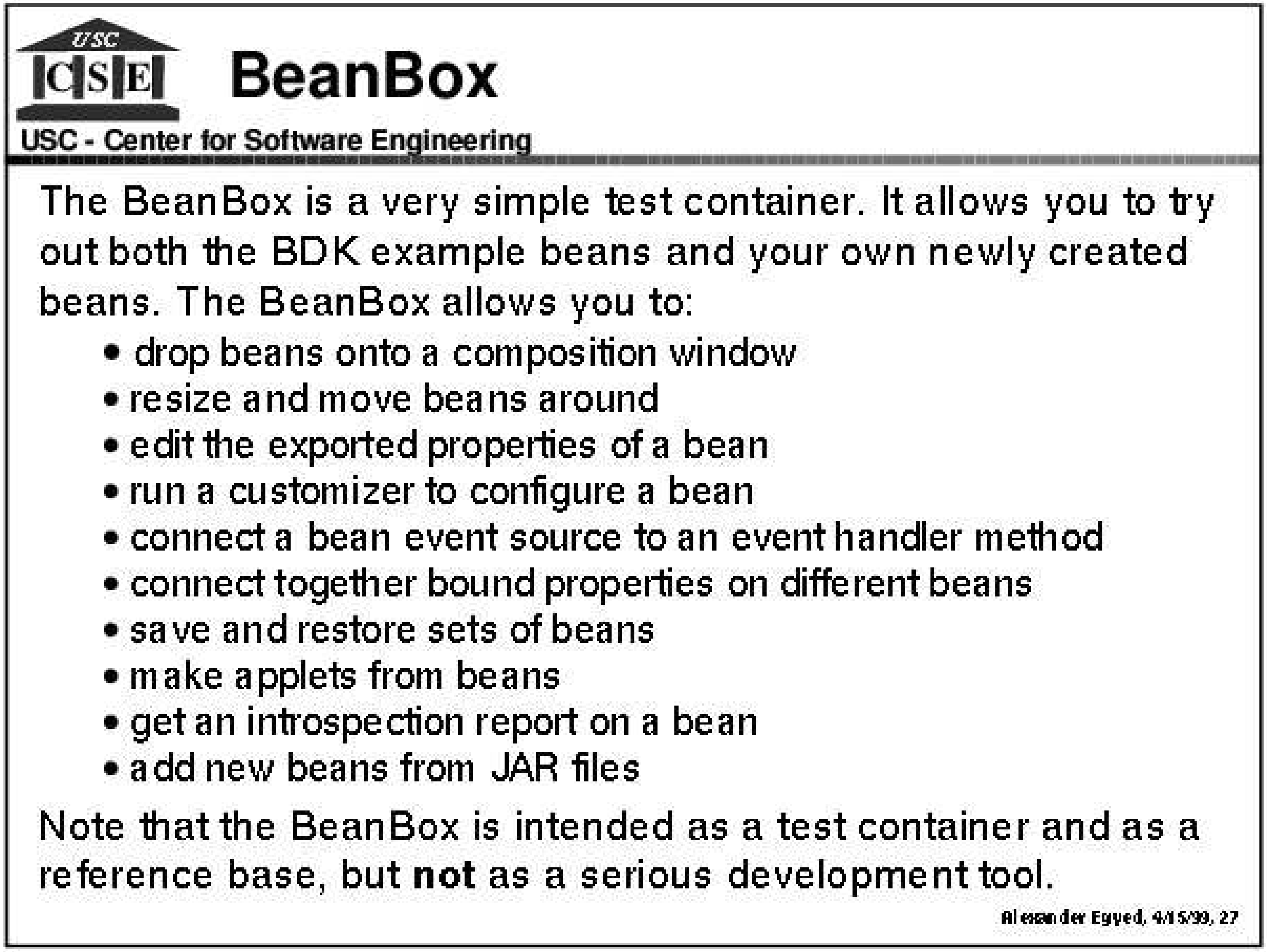
Examples are from `http://www.tcs.tifr.res.in/man/javaTutorial/beans`

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. Better, the programmer can write a `class BeanInfo` whose methods surrender the property-method-event interfaces.

The Java Bean Box: a simple development tool



Slide is from `http://sunset.usc.edu/classes/cs578_2002`

Beans and remote access

Beans and Remote Access

USC - Center for Software Engineering

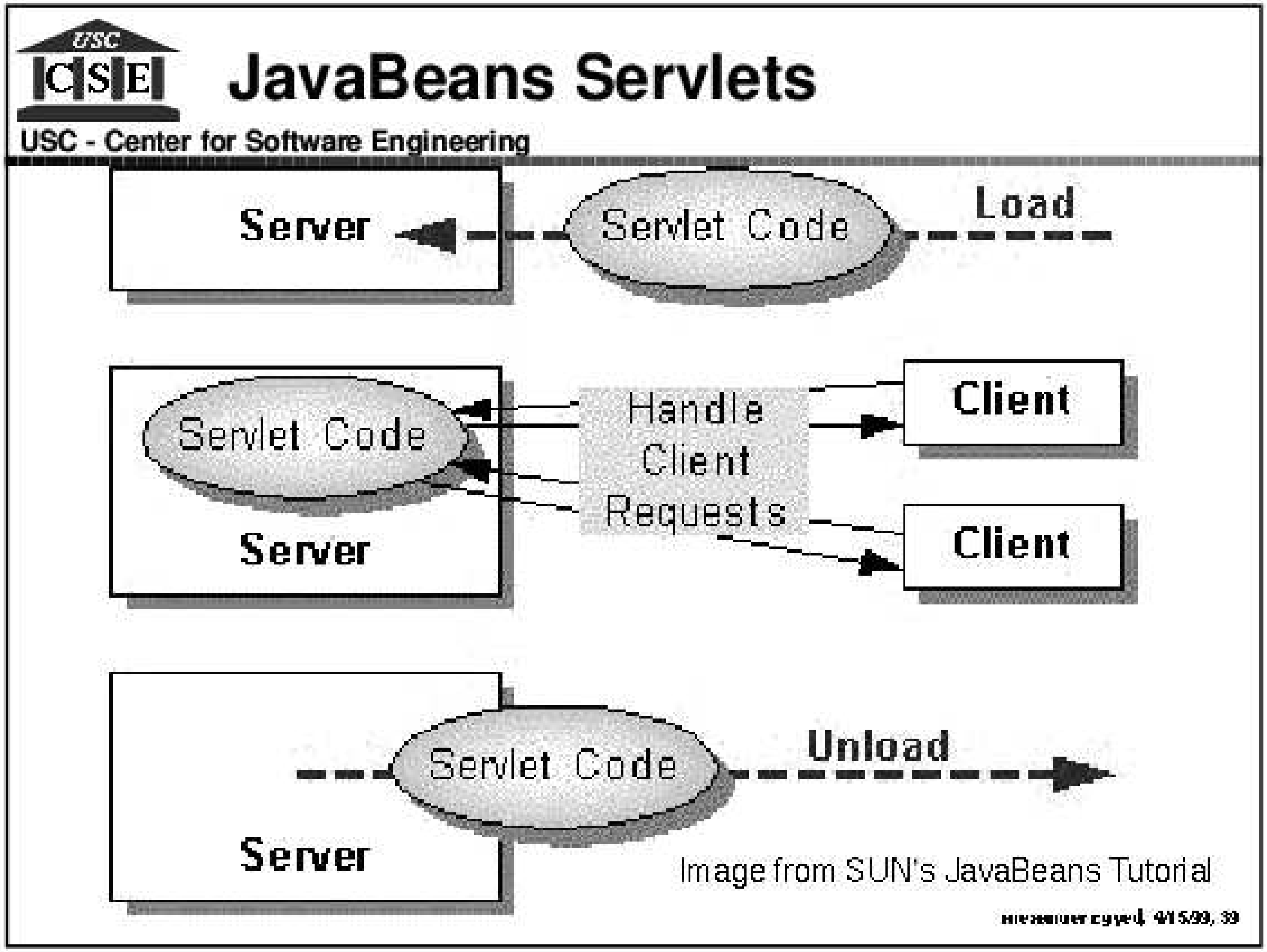
Beans can serve as front-ends to the network (e.g. CORBA)

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.

Alexander Egyed, 4/15/99, 35

Servlets: beans as proxies



Enterprise Java Beans (EJB) (now in J2EE)

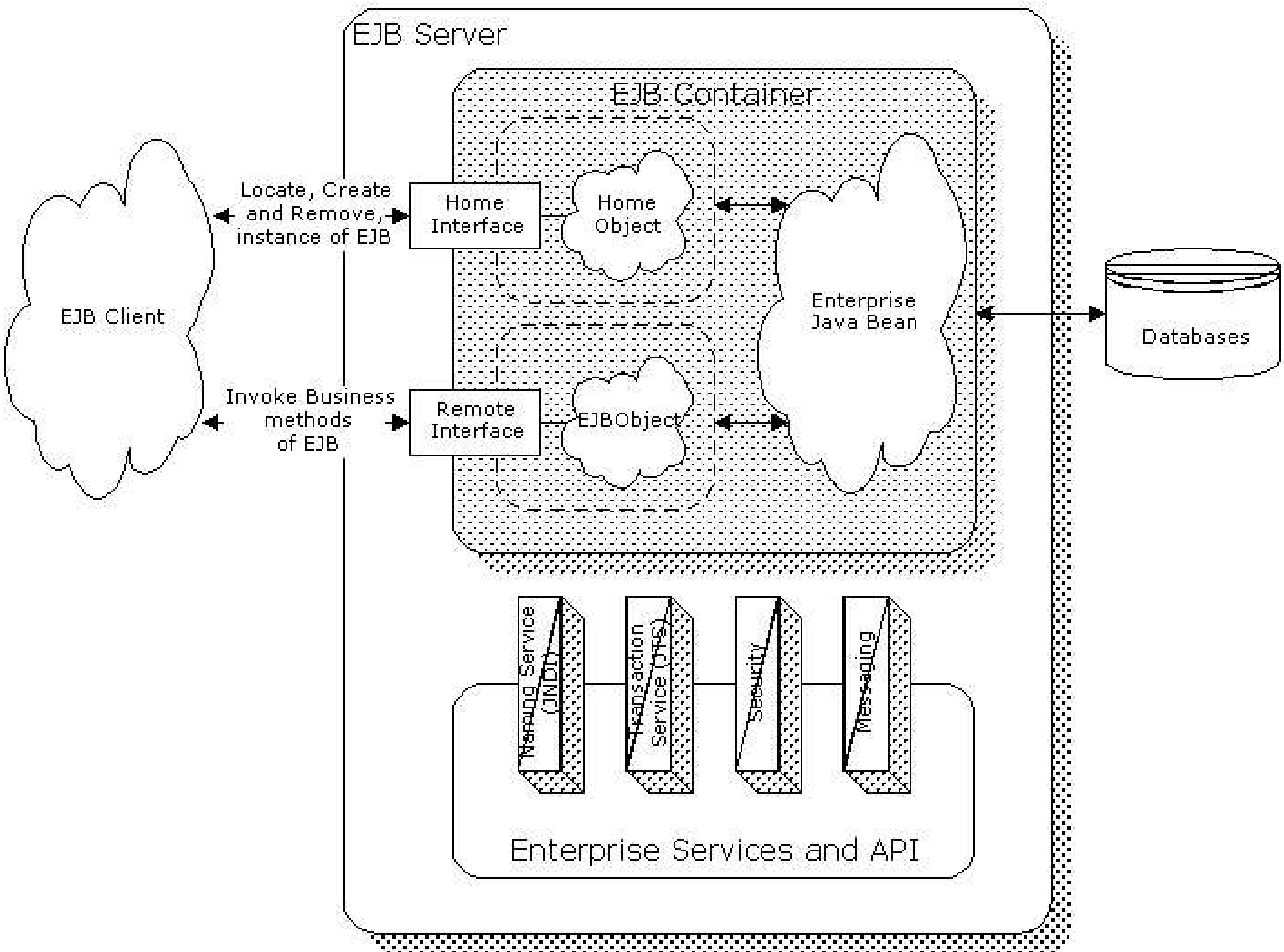
are a variant of Java beans (and not truly compatible with them), oriented to client-server applications.

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.


EJB Server
EJB Container
Locate, Create and Remove, instance of EJB
Home Interface
Home Object
EJB Client
Enterprise Java Bean
Databases
Invoke Business methods of EJB
Remote Interface
EJBObject
Naming Service (JNDI)
Transaction Service (JTS)
Security
Messaging
Enterprise Services and API

8. Model-driven architecture

An imprecise description: a *model-driven architecture* is software (architecture) development based on a model written in a modelling language. (Example: using UML to describe and suggest implementation of a system.)

A slightly more precise description: a model-driven architecture is a two-stage software architecture development:

1. starting at the "business level," define a *platform-independent model* (PIM) of the system,
2. now at the "architectural level," map the PIM to a *platform specific model* (PSM) at the "technology level."
3. implement the required PSM interfaces

But **the most precise description** comes from the OMG's response to the CORBA/COM/EJB competition....

The OMG's MDA methodology

CORBA, EJB (now, J2EE), DCOM (now, .NET) are competing frameworks for building client-server architectures. There are even interchange languages for mapping between their IDLs.

The OMG defined a "meta-model" (the PIM) of client-server and mappings from the PIM to PSMs for CORBA, J2EE, etc.

The PIM is to be written in *UML2*, which is UML extended to write PIMs. (UML2 includes concepts from SPL, a telecommunications design language.)

The mapping from PIM to PSM maps architecture, data forms, and IDL to the PSM's. A mapping from the client-server PIM to J2EE is well underway.

Advantages: hides multiplicities of programming languages, IDLs, etc.; supports upgrades of the PSMs. *Disadvantages:* requires two more meta- languages, MOF and XMI; relies heavily on UML2; unclear it will map to non-J2EE PSMs

Domain Specific Modelling

is a backlash to the UML-based MDA:

It is *MDA using DSLs* (instead of UML).

Each level of architecture is coded in a DSL, and translators map each level of domain-specific program to a (doman-specific) program at the next lower-level (and finally to assembly code).

Reference: `www.dsmforum.org`

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*:

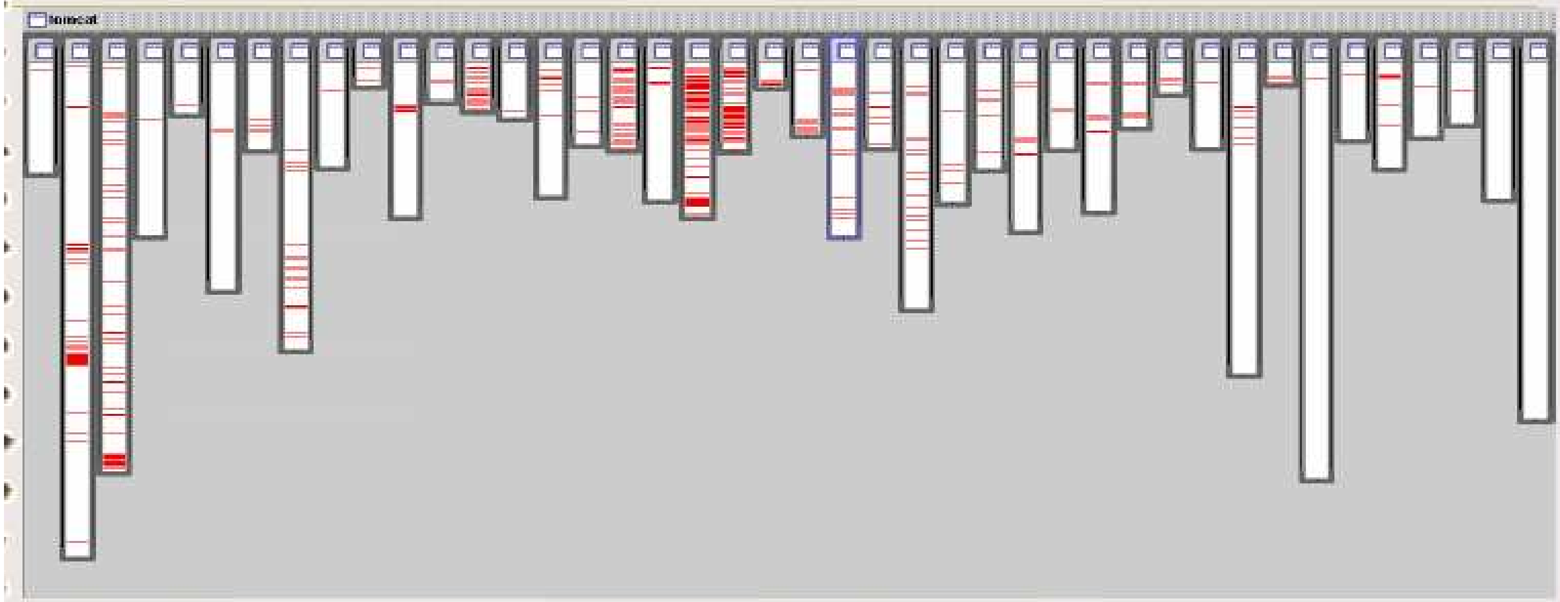
```
public class Stack
{ private int top;    private Object[] elements;

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

PLI 2003 9

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.

Wrappers implement simple aspects

When join points are method definitions, where an aspect merely adds code before method entry and after exit, then we can mimick weaving with a *wrapper*.

Example: pre-condition error checking via a subclass-wrapper:

```
public class NumericalOperator {
   public double square_root(double d) { ...  } }

public class NumericalWrapper extends NumericalOperator {
   public double square_root(double m) { // check that m>= 0 :
      double answer;
      if (m >= 0) { answer = super.square_root(m); }
         else { throw new RuntimeException( ...  )  }
      return answer; }
}
```

The technique is simple but inelegant — it changes the name of `class NumericalOperator`. Also, one quickly obtains too many layers of wrappers.

Composition filters: “smart wrappers”

Filters integrate “local” as well as “global” aspects, in both “horizontal” and “vertical” composition:

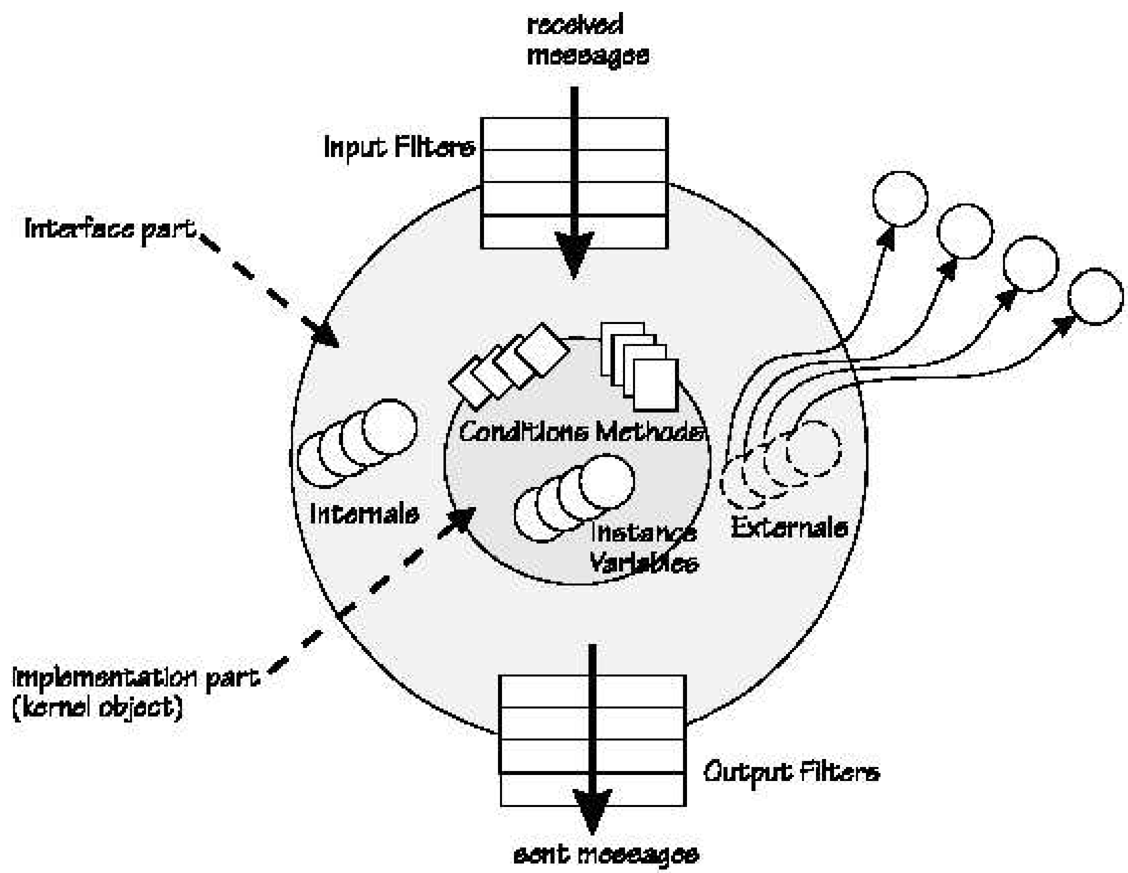


Figure 1.1 The components of the composition-filters model.

L. Bergmans, *The composition filters object model*, Computer Science, Univ. Twente, 1994.

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:

```
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!

Weaving automata: Colcombet and Fradet

Program and aspect might be represented as automata and woven into a product automaton (enforces policies for error handling, synchronization):

$P \equiv$
```
manager();
if(...) accountant();
if(...) {critical();
         manager()  ;}
accountant();
critical();
```

$E \equiv$
```
manager(*)     ->   m
accountant(*)  ->   a
critical(*)    ->   c
```

$$T = ((a^+m \mid m^+a)(a \mid m)^*c)^*$$

$Trans[P,T] \equiv$
```
state = 0;
manager();
if(...) {state=1;
         accountant();}
if(...) {if (state == 0) {abort;}
         critical();
         manager();}
accountant();
critical();
```

Figure 1: A small example of property enforcement

From T. Colcombet and P. Fradet. *Enforcing trace properties by program transformation*, ACM POPL 2000.

The policy, program, and product automaton:

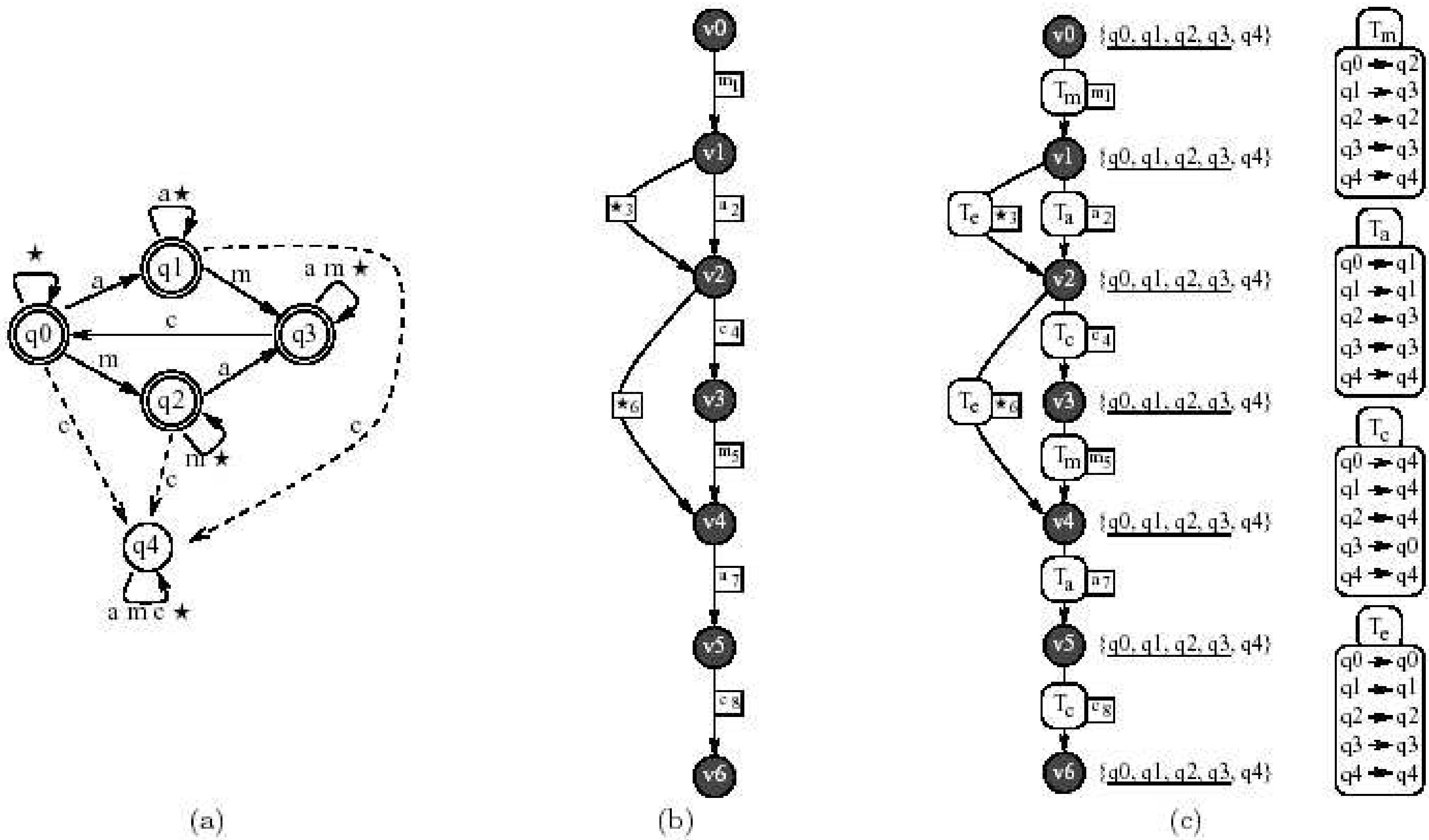
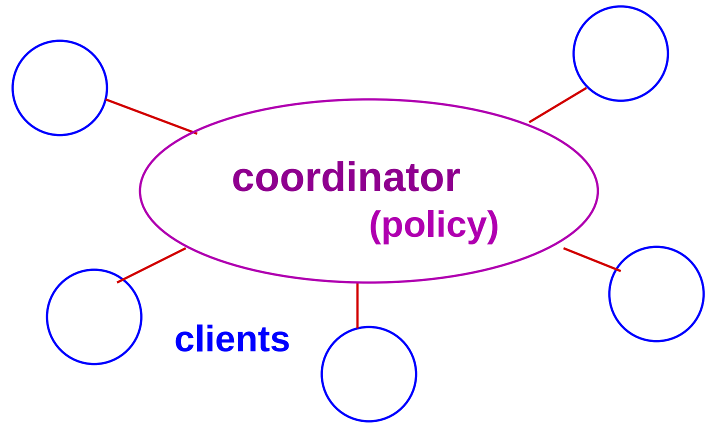


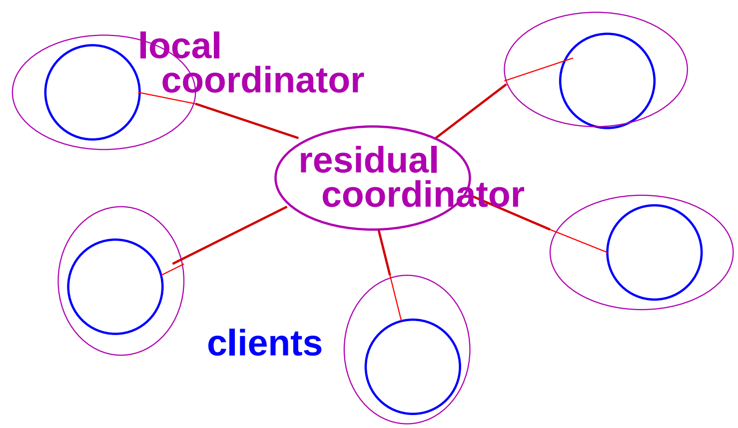
Figure 2: Automaton (a), control-flow graph (b) and direct instrumentation (c)

Aspects as coordinators

An aspect is sometimes specified as a global “coordinator” that enforces a synchonization or security policy:



The coordinator is coded separately, and the weaver distributes the coordinator’s code into the clients, giving distributed coordination. (*Partial evaluators* do this weaving.) The result looks like CORBA:



Subject-oriented programming

IBM (Harrison and Ossher): a *subject* is an aspect of a data structure.

Example: a book viewed in two different ways

```
// as a literary subject:
LiteraryBook {
   title
   topic
   abstract
   getAbstract()
      { return abstract }
}
```

```
// as a subject of production:
ProductionBook {
   book_title
   kind_of_paper
   kind_of_binding
   kind_of_cover
   printTheCover()
      { println(book_title, abs()) }
}
```

These look like multiple interfaces or abstract classes (*c.f.* Java beans); the `Book` class is assembled from the subjects, which are “unioned” (a kind of tensor product) using *correspondence rules*.

```
// as a literary subject:
LiteraryBook {
   title
   topic
   abstract
   getAbstract()
      { return abstract } }
```

```
// as a subject of production:
ProductionBook {
   book_title
   kind_of_paper
   kind_of_binding
   kind_of_cover
   printTheCover()
      { println(book_title, abs()) } }
```

The join points are class, attribute, and method names, as used in the *correspondence rules*:

```
ByNameMerge(Book, (LiteraryBook, ProductionBook)

Equate(attribute Book.title (LiteraryBook.title,
                             ProductionBook.book_title))

Equate(operation Book.abs (LiteraryBook.getAbstract,
                           ProductionBook.abs))
```

We are moving towards programming-language support for these formats of interface, connection, and implementation. Examples:

- *Jiazzi*: `www.cs.utah.edu/plt/jiazzi/`
- *GenVoca/AHEAD*: `www.cs.utexas.edu/users/schwartz`
- *composition filters*: `http://trese.ewi.utwente.nl/oldhtml/composition_filters`
- *subject-oriented programming*: `www.research.ibm.com/sop`
- *COOL/RIDL*: Lopes, C. *A Language Framework for Distributed Programming*. PhD thesis, Northeastern Univ., 1998.
- *AspectJ*: `www.parc.com/research/csl/projects/aspectj`

These are the "modern-day" architectural description languages! See `www.generative-programming.org` for an overview.

10. Final Remarks

TABLE 1. Academic versus industrial view on software architecture

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

Reference: Jan Bosch. *Design and Use of Software Architectures*. Addison-Wesley, 2000.

TABLE 2. **Academic versus industrial view on reusable components**

Academia	Industry
• Reusable components are black-box entities.	• 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.
• Components have narrow interface through a single point of access.	• The component interface is provided through entities, e.g., classes in the component. These interface entities have no explicit differences to non-interface entities.
• Components have few and explicitly defined variation points that are configured during instantiation.	• 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
• Components implement standardized interfaces and can be traded on component markets.	• Components are primarily developed internally. Externally developed components go through considerable (source code) adaptation to match the product-line architecture requirements.
• Focus is on component functionality and on the formal verification of functionality.	• Functionality and quality attributes, e.g. performance, reliability, code size, reusability and maintainability, have equal importance.

Selected textbook references

F. Buschmann, et al. *Pattern-Oriented Software Architecture.* Wiley 1996.

P. Clements and L. Northrup. *Software Product Lines*. Addison-Wesley 2002.

P. Clements, et al. *Documenting Software Architectures: Views and Beyond*. Addison Wesley, 2002.

K. Czarnecki and U. Eisenecker. *Generative Programming.* Addison-Wesley 2000.

E. Gamma, et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison Wesley, 1994.

M. Shaw and D. Garlan. *Software Architecture.* Prentice Hall 1996.