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 and views
4. **Architectural description languages**
5. **Domain-specific design**
6. **Product lines**
7. Middleware
8. Model-driven architecture
9. Aspect-oriented programming
10. **Closing remarks**

1. Components and connectors

Motivation for software architecture

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

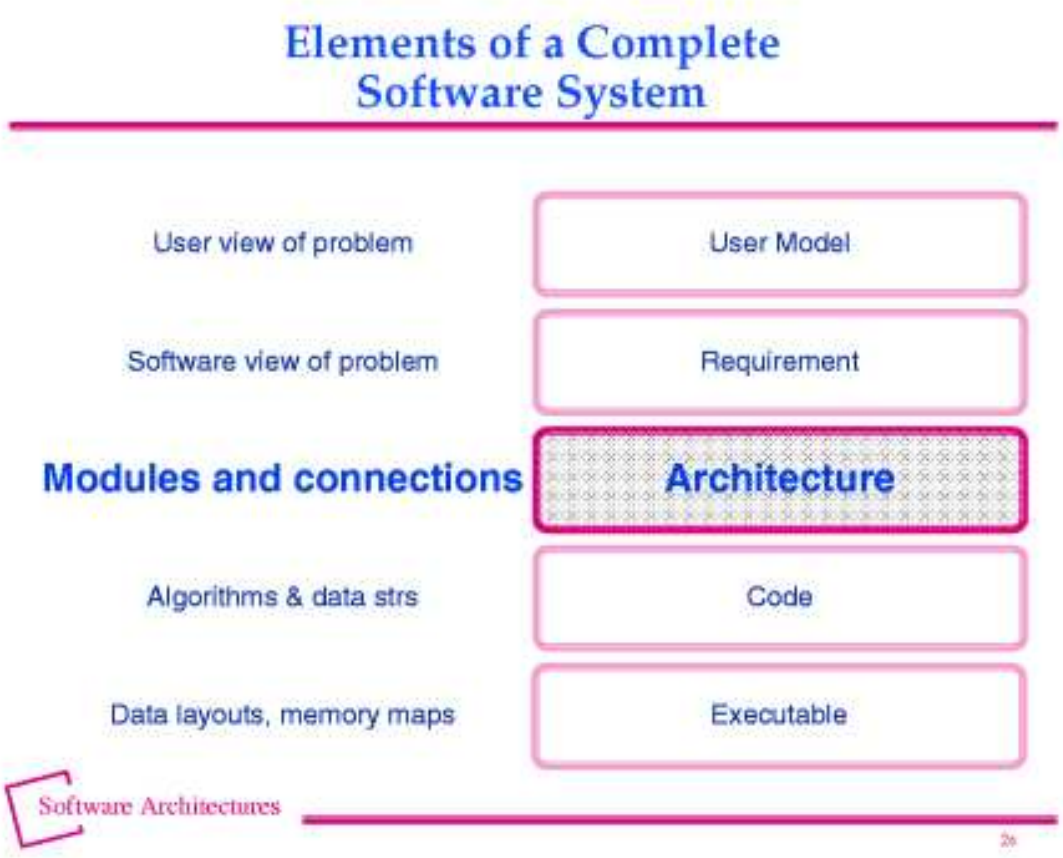
- "Camelot is based on the *client-server model* and uses remote procedure calls both locally and remotely to provide communication among applications and servers." [Spector87]
- "The easiest way to make the canonical sequential compiler into a concurrent compiler is to *pipeline* the execution of the 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-flow, data-flow, modular structure, behavioral requirements, ...

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

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

(iv) *materials*: control structures, data structures, ...

A crucial motivating concept: *connectors*

The emergence of networks, client-server systems, and OO-based GUI applications led to the conclusion that

components can be *connected* in various ways

Mary Shaw stressed this point:

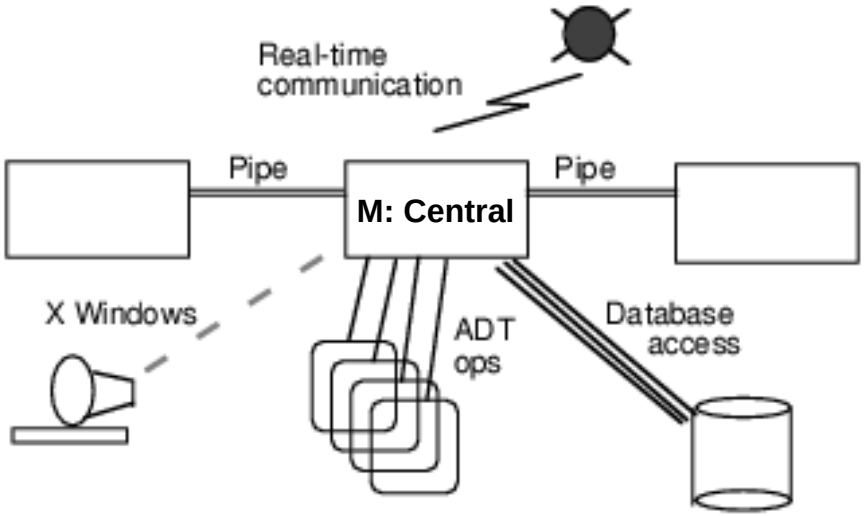


Figure 2: Revised architecture diagram with discrimination among connections

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 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*.

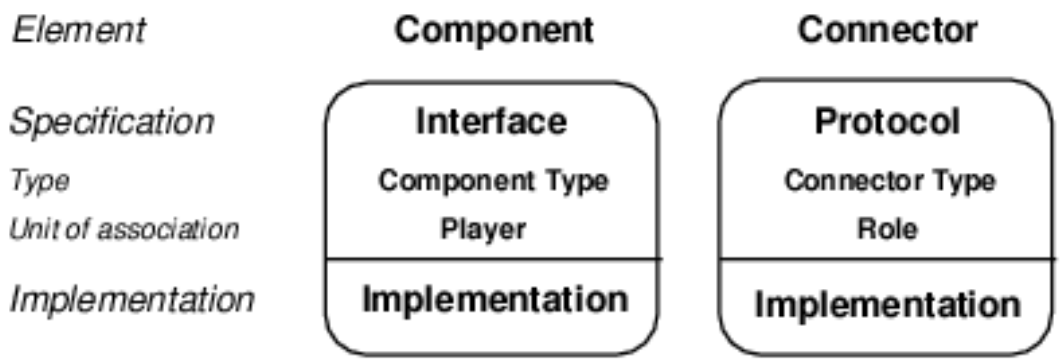


Figure 3: Gross structure of an architecture language

Example:

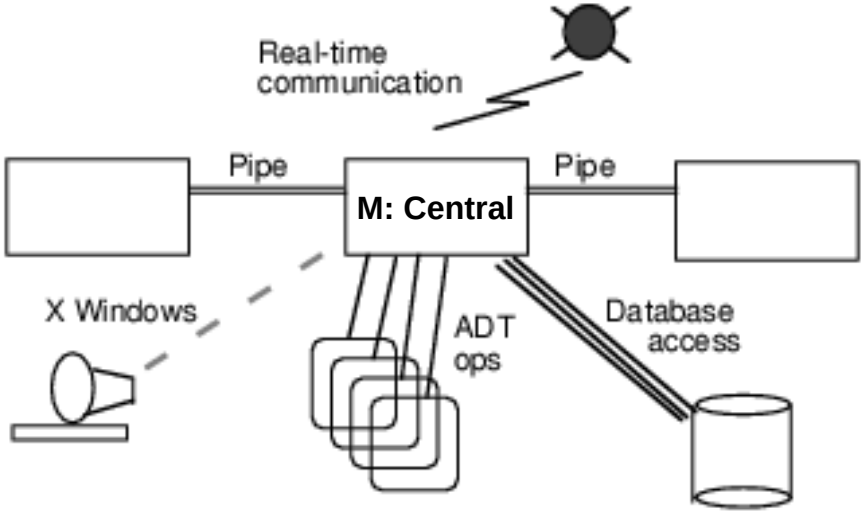


Figure 2: Revised architecture diagram with discrimination among connections

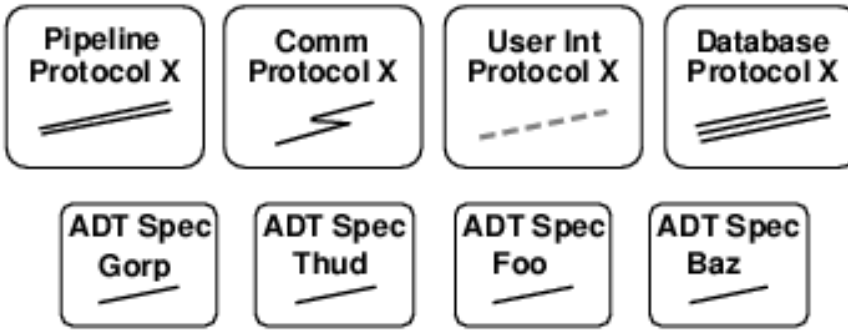


Figure 4: Constellation of protocol specifications required by exampl

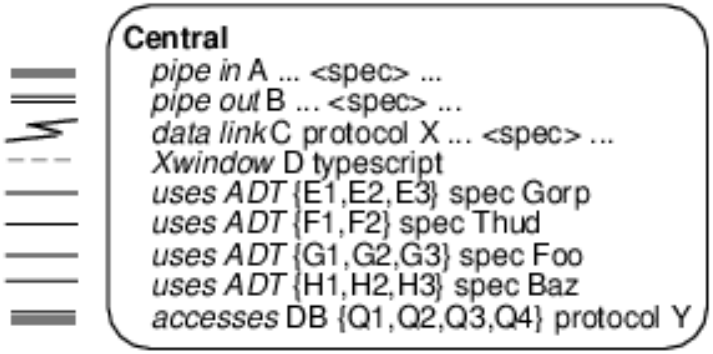


Figure 5: Interface specification of central component, referring to protocols

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.

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.

Architectural Styles (patterns)

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

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

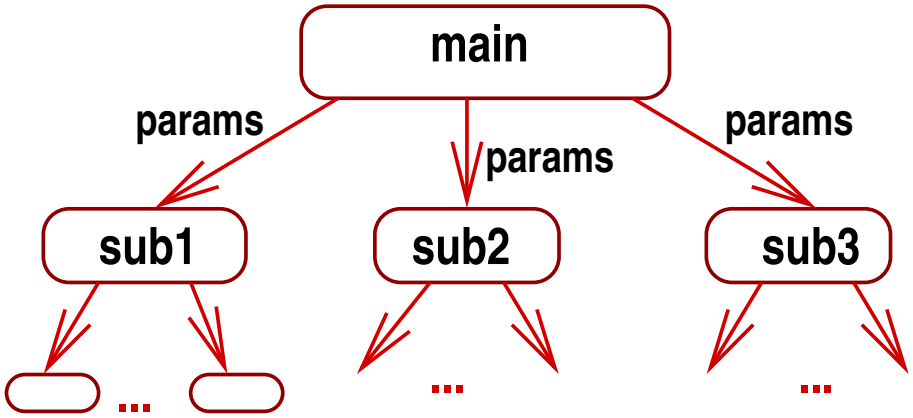
Data-flow systems: Batch-sequential and Pipe-and-filter

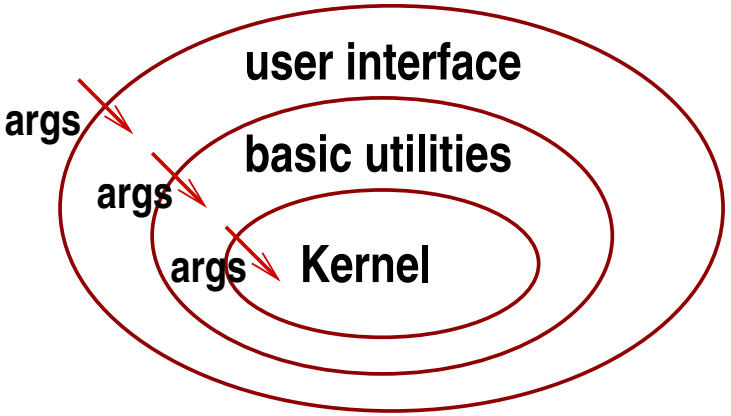
	Batch sequential	*Pipe and filter*
Components:	whole program	filter (function)
Connectors:	conventional input-output	pipe (data flow)
Constraints:	components execute to completion, consuming entire input, producing entire output	data arrives in increments to filters

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

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

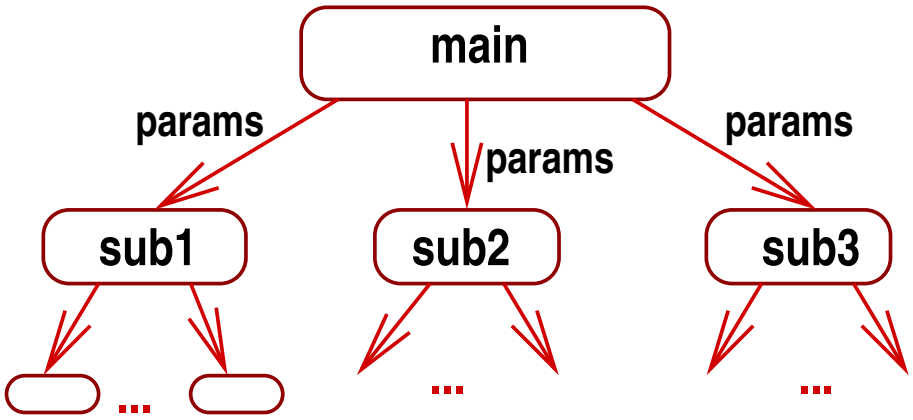
Call-and-return systems: subroutine and layered

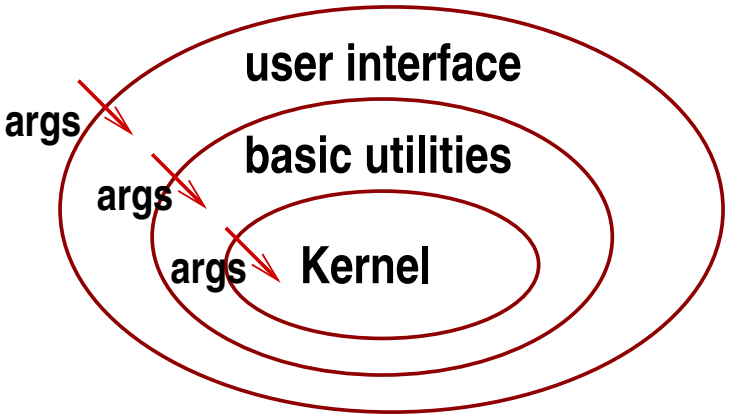




	Subroutine	*Layered*
Components:	subroutines (“servers”)	functions (“servers”)
Connectors:	parameter passing	protocols
Constraints:	hierarchical execution and encapsulation	functions within a layer invoke (API of) others at next lower layer

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

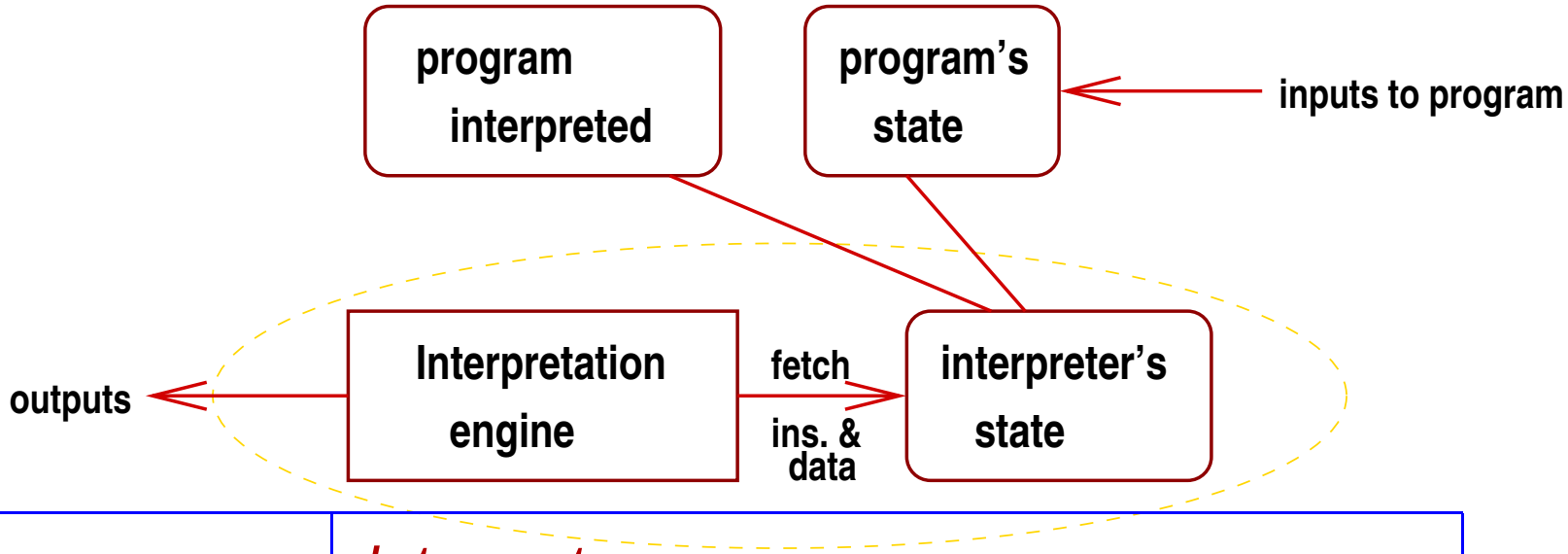




Advantages: hierarchical decomposition of solution; limits range of interactions between components, simplifying correctness reasoning; each layer defines a *virtual machine*; supports portability (by replacing lowest-level components).

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

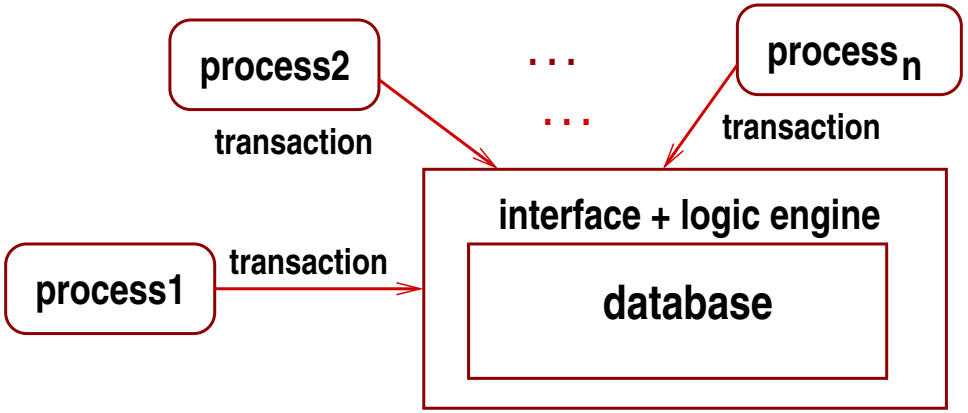


	Interpreter
Components:	"memories" and state-machine engine
Connectors:	fetch and store operations
Constraints:	engine's "execution cycle" controls the simulation of program's execution

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

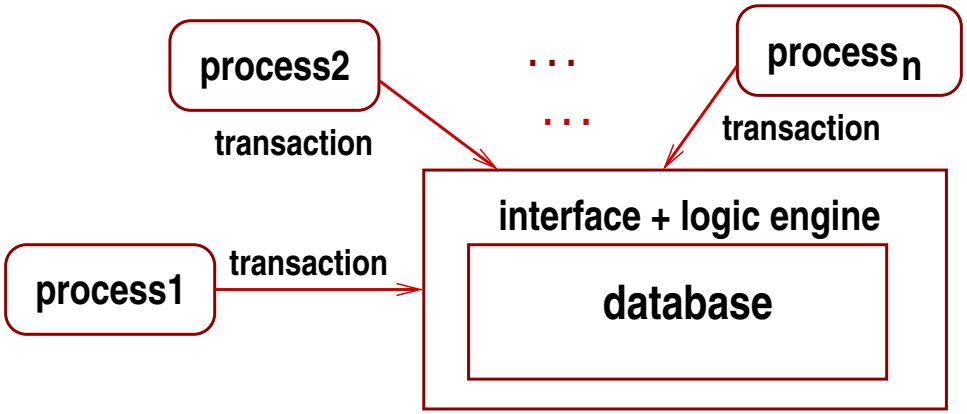
Advantages: rapid prototyping *Disadvantages:* inefficient.

Repositories: databases and blackboards



	Database	*Blackboard*
Components:	processes and database	knowledge sources and blackboard
Connectors:	queries and updates	notifications and updates
Constraints:	transactions (queries and updates) drive computation	knowledge sources respond when enabled by the state of the blackboard. Problem is solved by cooperative computation on blackboard.

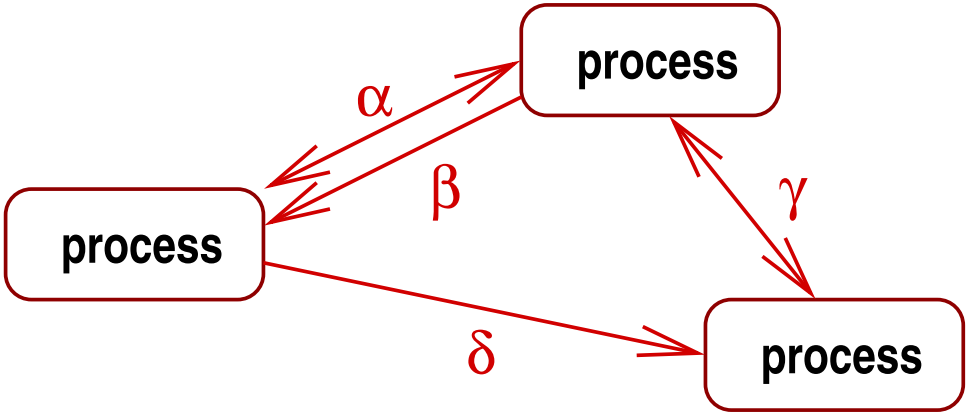
Examples: speech and pattern recognition (blackboard); syntax editors and compilers (parse tree and symbol table are repositories)



Advantages: easy to add new processes.

Disadvantages: alterations to repository affect all components.

Independent components: communicating processes

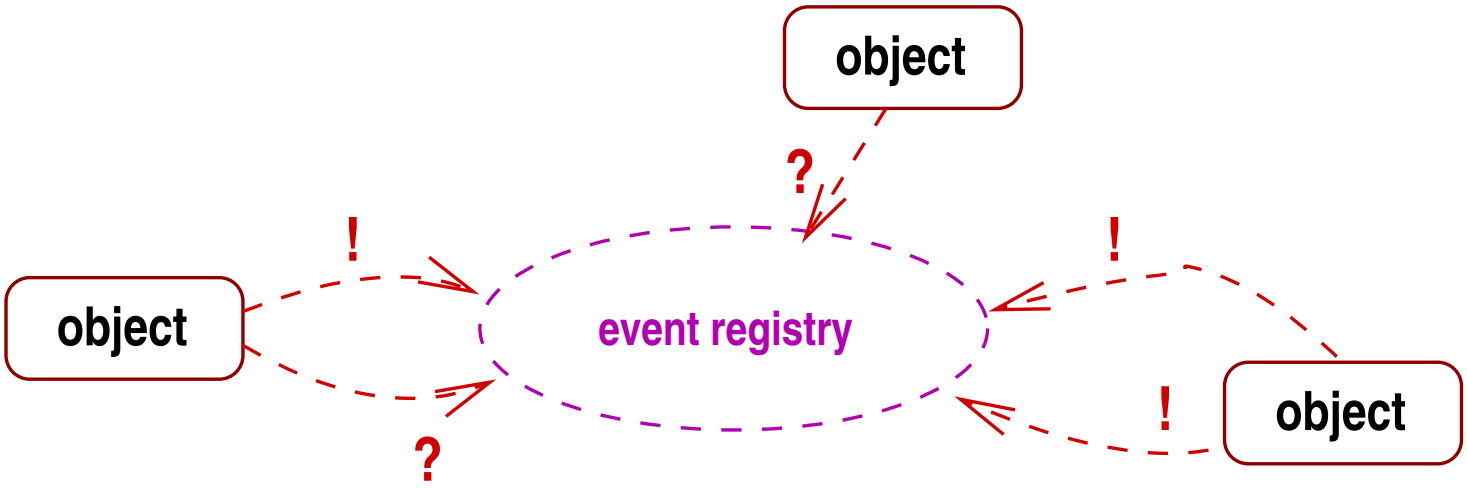


	Communicating processes
Components:	processes (“tasks”)
Connectors:	ports or buffers or RPC
Constraints:	processes execute in parallel and send messages (synchronously or asynchronously)

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

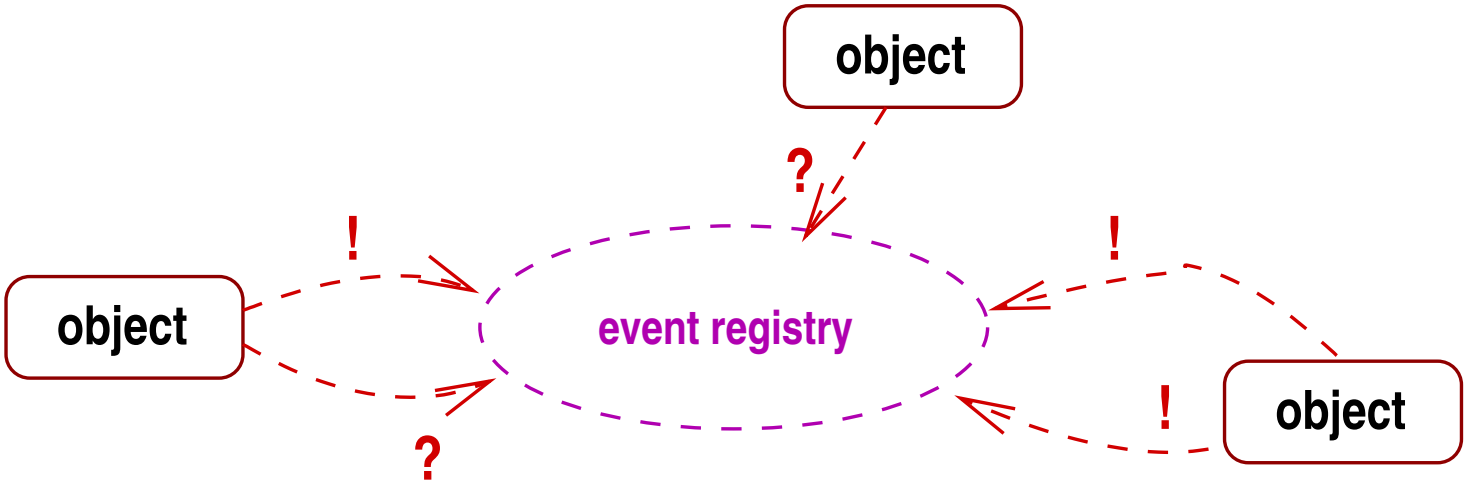
Advantages: easy to add and remove processes. *Disadvantages:* difficult to reason about control flow.

Independent components: event systems



	Event systems
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.

Three architectures for a compiler (Garlan and Shaw)

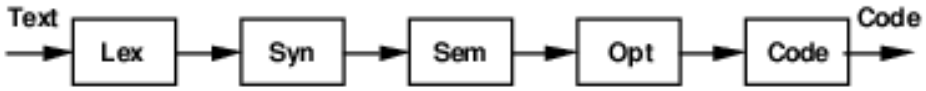


Figure 15: Traditional Compiler Model

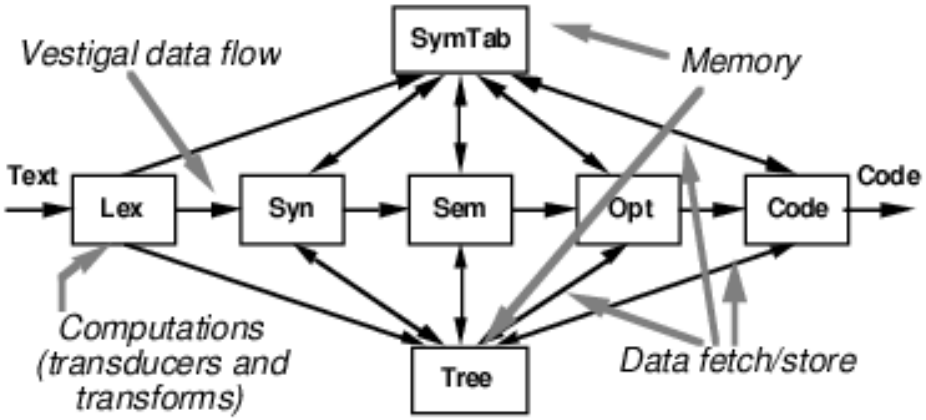


Figure 17: Modern Canonical Compiler

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

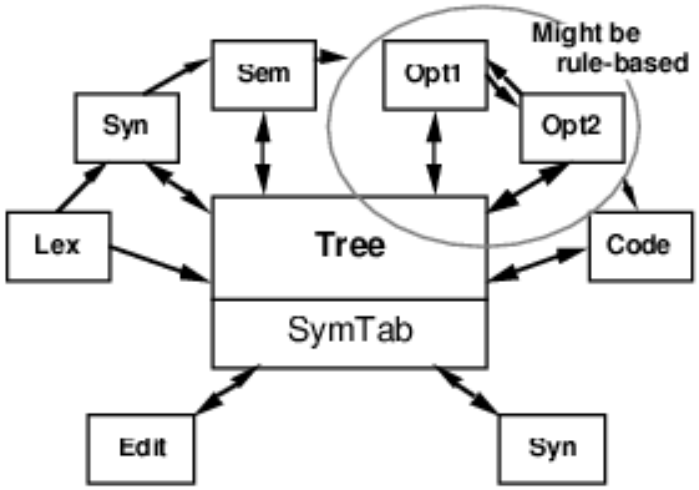


Figure 18: Canonical Compiler, Revisited

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

4. Architecture Description Languages

Wright: CSP-based description language

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.

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

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

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

Forms of CSP processes:

- prefixing: $e \rightarrow P$

  $plusOne?x \rightarrow return!x + 1 \rightarrow \cdots \parallel plusOne!2 \rightarrow return?y \rightarrow \cdots$

  $\Rightarrow\ return!2 + 1 \rightarrow \cdots \parallel return?y \rightarrow \cdots$

- external choice: $P \Box Q$

  $plusOne?x \rightarrow \cdots \Box plusTwo?x \rightarrow \cdots x + 2 \cdots \parallel plusTwo!5 \rightarrow \cdots$

  $\Rightarrow\ \cdots 5 + 2 \cdots \parallel \cdots$

- internal choice: $P \sqcap Q$

  $plusOne?x \rightarrow \cdots \parallel plusOne!5 \rightarrow \cdots \sqcap plusTwo!5 \rightarrow \cdots$

  $\Rightarrow\ plusOne?x \rightarrow \cdots \parallel plusTwo!5 \rightarrow \cdots$

- parallel composition: $P \| Q$

- halt: $\S$

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

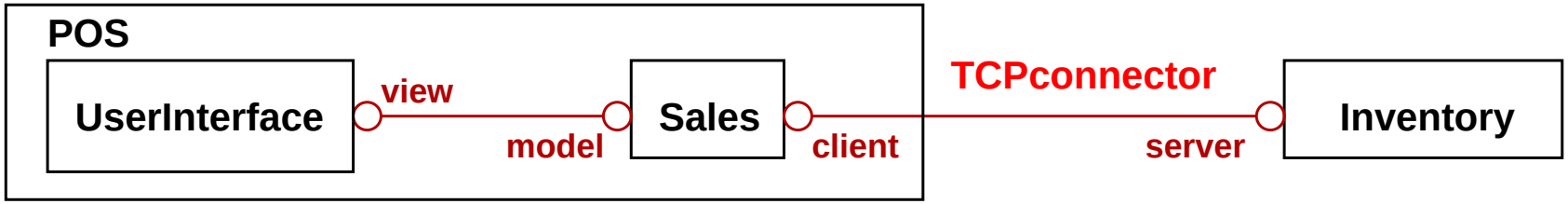
A pipe protocol in Wright

```
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.*

ArchJava: coding connectors in Java

- Each component (class) has its own interfaces (*ports*) that list which methods it `requires` and `provides`
- Connectors are coded as classes, too, and extend the basic classes, `Connector`, `Port`, `Method`, etc.
- The ArchJava run-time platform includes a run-time type checker that enforces correctness of run-time connections (e.g., RPC, TCP)
- There is an open-source implementation and Eclipse plug-in
- `www.archjava.org`



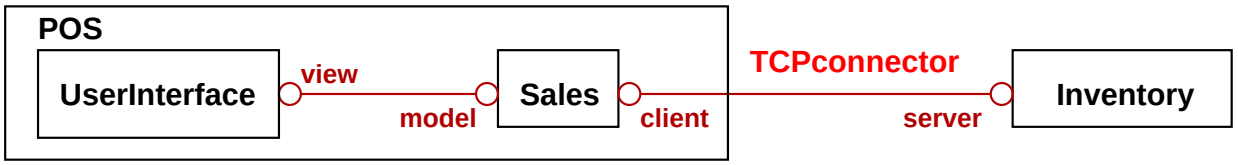

```
package pos;
…
public component class POS {
    …
    private final Sales sales = new Sales();
    private final UserInterface userInterface = new UserInterface();

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

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

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

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



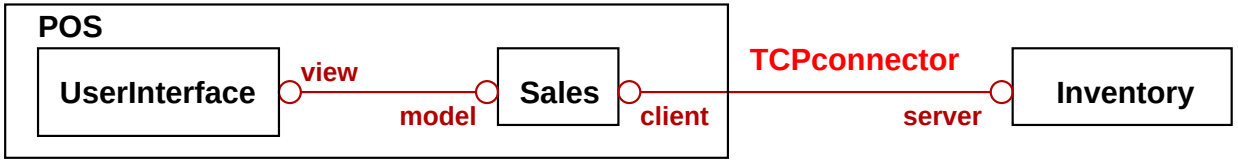

```
package pos;
…
public component class Sales {
    private String data;

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

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

    public String getData() {
        return data;
    }

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



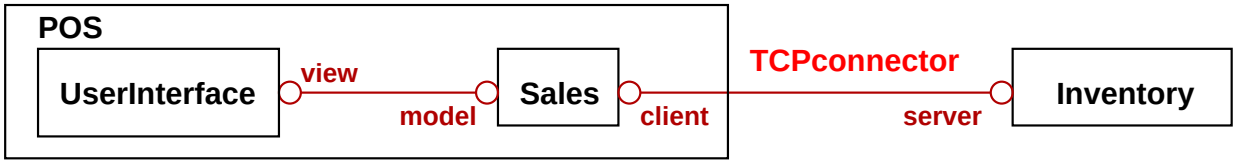

```
package pos;
…
public component class Inventory {
    public port interface server {
      provides String executeUpdate(String statement);
    }

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

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

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

From K. M. Hansen, www.daimi.dk/~marius/teaching/ATiSA2005




```
public class TCPConnector extends Connector {
        // data members
        protected TCPEndpoint endpoint;
        // public interface
        public TCPConnector(InetAddress host, int prt, String objName) throws IOException {
                endpoint = new TCPEndpoint(this, host, prt, objName);
        }

        public Object invoke(Call call) throws Throwable {
                Method meth = call.getMethod();
                return endpoint.sendMethod(meth.getName(), meth.getParameterTypes(), call.getArguments());
        }

        public static void registerObject(Object o, int prt, String objName) throws IOException {
                TCPDaemon.createDaemon(prt).register(objName, o);
        }
        // interface used by TCPDaemon
        TCPConnector(TCPEndpoint endpoint, Object receiver, String portName) {
                super(new Object[] { receiver }, new String[] { portName });
                this.endpoint = endpoint;
                endpoint.setConnector(this);
        }
        Object invokeLocalMethod(String name, Type parameterTypes[], Object arguments[]) throws Throwable {
                // find method with parameters that match parameterTypes
                Method meth = findMethod(name, parameterTypes);
                return meth.invoke(arguments);
        }
```

So, what is an architectural description language?

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

It should specify

- *structure*: components (interfaces), connectors (protocols), configuration (both static and dynamic structure)
- *behavior*: semantical properties of individual components and connectors, patterns of acceptable communication, global invariants,
- *design patterns*: global constraints that support correctness-reasoning techniques, design- and run-time tool support, and implementation.

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

5. Domain-specific design

Domain-specific design

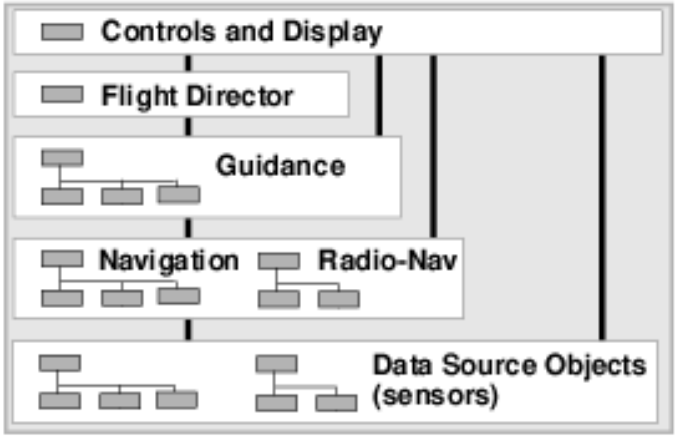
If the problem domain is a standard one (e.g., flight-control or telecommunications or banking), then there are precedents to follow.

A *Domain-Specific Software Architecture* has

- a *domain*: defines the problem area domain concepts and terminology; customer requirements; scenarios; configuration models (entity-relationship, data flow, etc.)
- *reference requirements*: *features* that restrict solutions to fit the domain. ("Features" are studied shortly.) Also: platform, language, user interface, security, performance
- a *reference architecture*
- a *supporting environment/infrastructure*: tools for modelling, design, implementation, evaluation; run-time platform
- a *process* or *methodology* to implement the reference architecture and evaluate it.

Avionics DSSA

- **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.interface = x_j.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 (modelling) language (DSL)

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

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

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

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

When a DSL can generate computer implementations, it is a *domain-specific programming language*.

Domain-specific programming language

In the Unix world, these are "little languages" or "mini-languages," designed to solve a specific class of problems. Examples are `awk`, `make`, `lex`, `yacc`, `ps`, and `Glade` (for GUI-building in X).

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

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

Tradeoffs in using (top-down) DSLs

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

Reference: J. Lawall and T. Mogensen. Course on Scripting Languages and DSLs, Univ. Copenhagen, 2006, `www.diku.dk/undervisning/2006f/213`

6. Software product lines

A software product line

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

The CMU Software Engineering Institute definition:

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

Key issues:

variability: Can we state precisely the products' variations (*features*) ?

guidance: Is there a precise recipe that guides feature selection and product assembly?

Reference: `www.softwareproductlines.com`

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

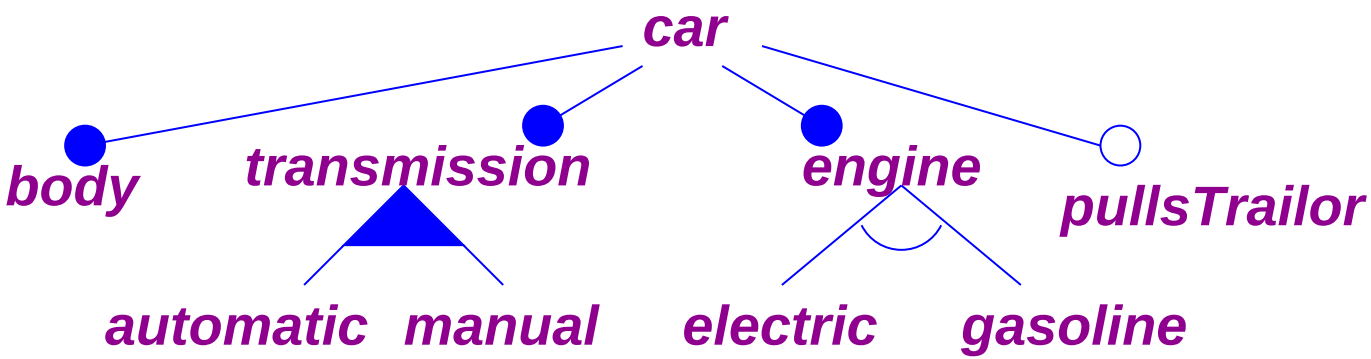
are a development tool for domain-specific architectures and product lines. They help define a domain's reference requirements and guide implementions of instances of the reference architecture.

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.

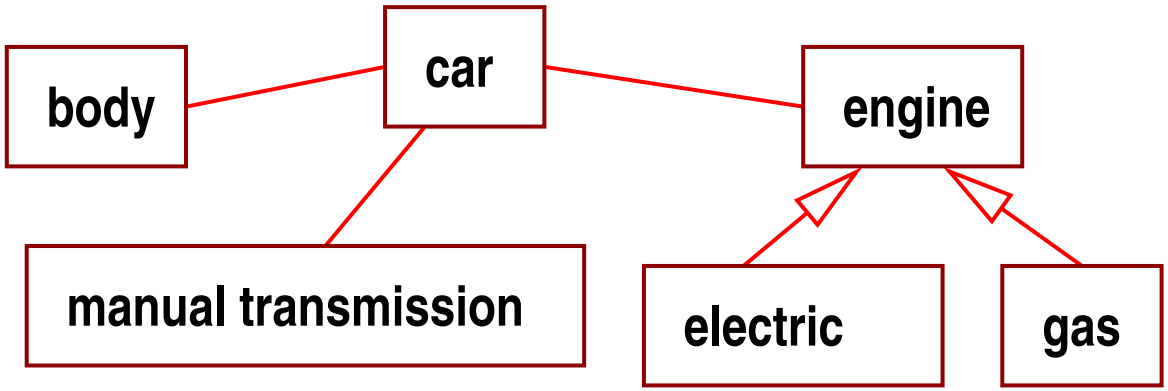
It is a form of decision tree with *and-or-xor* branching, and its hierarchy reflects dependencies of features as well as modification costs.

Feature diagram for assembling automobiles

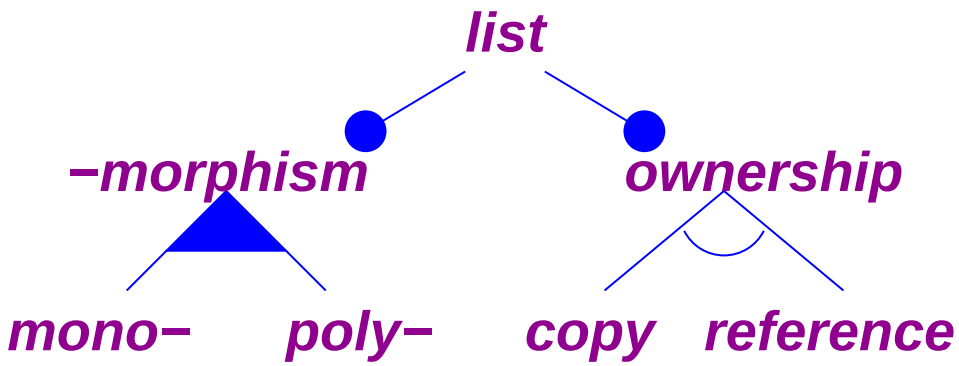


Filled circles label required features; unfilled circles label optional ones. Filled arcs label xor-choices; unfilled arcs label or-choices (where at least one choice is selected).

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.

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.