Software architecture:
Architectural Styles

Alexander Serebrenik
True or false?

• Domain-Specific Software Architecture is a part of a Reference Architecture.
True or false?

• Domain-Specific Software Architecture is a part of a Reference Architecture: **FALSE**

• Domain-Specific Software Architecture is broader applicable than a product line.
True or false?

• Domain-Specific Software Architecture is a part of a Reference Architecture: **FALSE**

• Domain-Specific Software Architecture is broader applicable than a product line: **TRUE**

• Model-View-Controller is an examples of a Domain-Specific Software Architecture
Before we start…

True or false?

- Domain-Specific Software Architecture is a part of a Reference Architecture: **FALSE**

- Domain-Specific Software Architecture is broader applicable than a product line: **TRUE**

- Model-View-Controller is an example of a Domain-Specific Software Architecture: **FALSE**
This week sources

Software Architecture
Foundations, Theory, and Practice

Slides by

Rudolf Mak
Johan Lukkien
Recall: Architectural patterns vs. Architectural styles vs. Design patterns

- **Architectural patterns** define the implementation strategies of those components and connectors (‘how?’)
  - More domain specific
- **Architectural styles** define the components and connectors (‘what?’)
  - Less domain specific
- Good architecture makes use of **design patterns** (on a more fine-granular level)
  - We’ll see examples later on
  - Usually domain independent
Architectural Styles

- An **architectural style** is a named collection of architectural design decisions that
  - are applicable in a given development context
  - constrain architectural design decisions that are specific to a particular system within that context
  - elicit beneficial qualities in each resulting system

- Reflect **less domain specificity** than architectural patterns

- Useful in determining everything from subroutine structure to top-level application structure
Benefits of Using Styles

- **Reuse**
  - Design: Well-understood solutions applied to new problems
  - Code: Shared implementations of invariant aspects of a style

- **Understandability** of system organization
  - A phrase such as “client-server” conveys a lot of information

- **Interoperability**
  - Supported by style standardization

- **Style-specificity**
  - Analyses: enabled by the constrained design space
  - Visualizations: depictions matching engineers’ mental models
Basic Properties of Styles

• A vocabulary of **design elements**
  • Component and connector types; data elements
    – e.g., pipes, filters, objects, servers
Recap: Connectors

• “Architectural styles define the components and connectors”

• A software connector is an architectural building block tasked with effecting and regulating interactions among components (Taylor, Medvidovic, Dashofy)
  • Procedure call connectors
  • Shared memory connectors
  • Message passing connectors
  • Streaming connectors
  • Distribution connectors
  • Wrapper/adaptor connectors
  • …
Basic Properties of Styles

- A vocabulary of **design elements**
  - Component and connector types; data elements
    - e.g., pipes, filters, objects, servers

- A set of **configuration rules**
  - Topological constraints that determine allowed compositions of elements
    - e.g., a component may be connected to at most two other components

- A **semantic interpretation**
  - Compositions of design elements have well-defined meanings

- Possible **analyses** of systems built in a style
Some Common Styles

- **Traditional, language-influenced styles**
  - Main program and subroutines
  - Object-oriented
- **Layered**
  - Virtual machines
  - Client-server
- **Data-flow styles**
  - Batch sequential
  - Pipe and filter
- **Shared memory**
  - Blackboard
  - Rule based
- **Interpreter**
  - Interpreter
  - Mobile code
- **Implicit invocation**
  - Event-based
  - Publish-subscribe
- **Peer-to-peer**
- **“Derived” styles**
  - C2
  - CORBA
Architecture Style Analysis

• Summary
• Design elements (components, connectors, data)
• Topology
• Examples of use
• Advantages/disadvantages
• Relation to programming languages/environments
Main program and subroutines

- You should be familiar with this style from a basic programming course

Main program:
- displays greetings and instructions
- enters a loop in which it calls the three subroutines in turn.

Diagram:
- Main Program
  - Procedure Call
    - Parameters: altitude, throttle, fuel, time, velocity
    - Output: altitude, fuel, velocity
  - Procedure Call
    - Parameters: altitude, fuel, time, velocity
    - Output: none
  - Procedure Call
    - Parameters: none
    - Output: throttle

Subroutines:
- Get Burn Rate from user
- Environment Simulator
- Display values to user
Main program and subroutines: Style Analysis

• **Summary:**
  • Decomposition based upon separation of functional processing steps

• **Design elements**
  • Components: main program and subroutines
  • Connectors: function/procedure calls
  • Data: Values passed in/out subroutines

• **Topology**
  • Static organization is hierarchical
  • Full structure: a directed graph
Main program and subroutines: Style Analysis

• What are common **examples** of its use?
  • Small programs, pedagogical uses

• What are the **advantages** of using the style?
  • Modularity: subroutines can be replaced as long as interface semantics are unaffected

• What are the **disadvantages** of using the style?
  • Usually fails to scale
  • Inadequate attention to data structures
  • Effort to accommodate new requirements: unpredictable
  • Usually not a good idea unless…

• Relation to **programming languages/environments**
  • Traditional programming languages: BASIC, Pascal, C…
Object-Oriented Lunar Lander

You should be familiar with this style from an OO-programming course.

Identify similarities and differences between the styles.
Object-Oriented Lunar Lander

Similarities:
connectors (procedure calls) + data (arguments)
Object-Oriented Lunar Lander

**Similarities**: connectors (procedure calls) + data (arguments)

**Differences**: encapsulation (UI, SpaceCraft, Environment)
- Procedural: input & output are separated
- OO: input & output are together
How would this look like as a class diagram?

```
SpaceCraft
altitude: double
fuel: double
time: int
velocity: double
SpaceCraft(a:double, f:double, t:int, v: double)
setAltitude(a: double)
setFuel(f: double)
...
getAltitude()
getFuel()
...
```

```
GUI
burnRate: double
getBurnRate(): double
displayStatus(s:SpaceCraft)
```

```
EnvironmentSimulation
moonGravity: double
calculateStatus(burnRate: double, s: SpaceCraft): SpaceCraft
```

1:*  creates  1

1  uses  1
Object-Oriented Style: Style Analysis

- **Summary:**
  - State strongly encapsulated. Internal representation is hidden from other objects
  - Objects are responsible for their internal representation integrity

- **Design elements**
  - Components: objects (data and associated operations)
  - Connectors: method invocations
  - Data: arguments passed to methods

- **Topology**
  - Can vary arbitrarily: data and interfaces can be shared through inheritance
Object-Oriented Style: Style Analysis

- What are common examples of its use?
  - pedagogy
  - complex, dynamic data structures
  - close correlation between physical world entities and entities in the program

- What are the advantages of using the style?
  - Integrity: data is manipulated only by appropriate methods
  - Abstraction: internals are hidden
Object-Oriented Style: Style Analysis

• What are the disadvantages of using the style?
  • Not efficient enough for high performance computing (e.g., scientific computing, data science)
  • Distributed applications require extensive middleware to provide access to remote objects
  • In absence of additional structural principles unrestricted OO can lead to highly complex applications
    – Hence, unrestricted OO is usually a bad idea

• Relation to programming languages/environments
  • OO-languages: Java, C++…
Layered Style Lunar Lander

- **Basic idea:**
  - Each layer exposes an interface (API) to be used by the layer above it
  - Each layer acts as a
    - **Server:** service provider to layer “above”
    - **Client:** service consumer of the layer “below”

- Taylor et al call this style “**virtual machines**”
  - I do not like this name since these virtual machines are not related to simulation or program execution as in “Java Virtual Machine”, Python, etc.
Layering

Strict Layering

Layer 1
  Program A

Layer 2
  Program B  Program C

Layer 3
  Program D

Nonstrict Layering

Layer 1
  Program A

Layer 2
  Program B  Program C

Layer 3
  Program D
Layered Style: Style Analysis

• **Summary:**
  • An ordered sequence of layers, each layer offers services (interfaces) that can be used by programs (components) residing with the layer(s) above it

• **Design elements**
  • Components: layers, each layer usually several programs
  • Connectors: typically procedure calls
  • Data: parameters passed between layers

• **Topology**
  • Linear (strict layering), acyclic (non-strict layering)
Layered Style: Style Analysis

- What are common **examples** of its use?
  - operating systems
    - 2INC0 “Operating systems” SfS:Y3Q1
  - network and protocol stacks
    - 2IC60 “Computer networks and security” SfS, WbS:Y2Q4

Layered Style: Style Analysis

- What are the **advantages** of using the style?
  - Clear dependence structure benefits evolution
    - Lower layers are independent from the upper layers
    - Upper layers can evolve independently from the lower layers as long as the **interface semantics is unchanged**
    - **Strict layering**: limits propagation of change
  - Reuse
    - e.g., standardized layer interfaces for libraries/frameworks

- What are the **disadvantages** of using the style?
  - Not universally applicable
  - Performance (mostly for strict layering and many layers)
Client-Server Style

• Similar to the layered style

• Differences
  • Only two layers
    – Client(s)
    – Server
  • Network-based connection

• Clients
  • Thin – no processing beyond UI
  • Thick – otherwise
Client-Server Style: Style Analysis

• **Summary:**
  • Client initiates communication by sending server a request.
  • Server performs the requested action and replies.

• **Design elements**
  • Components: client(s) and server
  • Connectors: remote procedure call, network protocols
  • Data: parameters and return values

• **Topology**
  • Two-level, multiple clients making requests to server
  • No client-client communication
Client-Server Style: Style Analysis

- What are common examples of its use?
  - centralization of data is required
  - server: high-capacity machine (processing power)
  - clients: simple UI tasks
  - many business applications
    - 2IIC0 “Business Information Systems” SfS, WbS:Y3Q1
Client-Server Style: Style Analysis

- What are common **examples** of its use?
  - centralization of data is required
  - server: high-capacity machine (processing power)
  - clients: simple UI tasks
  - many business applications
    - 2IIC0 “Business Information Systems” SfS, WbS:Y3Q1

- What are the **advantages** of using the style?
  - Data centralization, powerful server serving many clients

- What are the **disadvantages** of using the style?
  - Single point of failure
  - Network bandwidth / amount of requests
Some Common Styles

- **Traditional, language-influenced styles**
  - Main program and subroutines
  - Object-oriented

- **Layered**
  - (Virtual machines)
  - Client-server

- **Data-flow styles**
  - Batch sequential
  - Pipe and filter

- **Shared memory**
  - Blackboard
  - Rule based

- **Interpreter**
  - Interpreter
  - Mobile code

- **Implicit invocation**
  - Event-based
  - Publish-subscribe

- **Peer-to-peer**

- **“Derived” styles**
  - C2
  - CORBA
Batch Sequential

• **Dataflow styles** focus on how data moves between processing elements

• **Batch-sequential**
  • “The Granddaddy of Styles”
  • Separate programs are executed in order
  • Aggregated data (on magnetic tape) transferred by the user from one program to another
What about the Lunar Lander?
Batch Sequential

For tomorrow’s use

Not a recipe for a successful lunar mission!
Batch Sequential: Style Analysis

• **Summary:**
  • Separate programs executed one at a time, till completion

• **Design elements**
  • Components: independent programs
  • Connectors: “the human hand” carrying tapes between the programs, a.k.a. “sneaker-net”
  • Data: aggregated on tapes

• **Topology**
  • Linear

• What are common **examples** of its use?
  • Transaction processing in financial systems
Batch Sequential: Style Analysis

- What are the **advantages** of using the style?
  - Simplicity
  - Severable executions

- What are the **disadvantages** of using the style?
  - No concurrency
  - No interaction between components
Pipe and Filter

• In Batch Sequential the next program waits till the preceding one has finished processing data completely.

• What if the next program could process data elements as soon as they become available?
  • programs can operate concurrently ⇒ speed up
  • data is considered as streams
In Batch Sequential the next program waits till the preceding one has finished processing data completely.

What if the next program could process data elements as soon as they become available?
- programs can operate concurrently ⇒ speed up
- data is considered as streams

Lunar Lander
Pipe and Filter: Style Analysis

• **Summary:**
  • Separate programs executed, potentially concurrently

• **Design elements**
  • Components: independent programs, a.k.a. **filters**
  • Connectors: routers of data streams (**pipes**), provided by an operating system
    – Variations
      – *Pipelines* — linear sequences of filters
      – *Bounded pipes* — limited amount of data on a pipe
      – *Typed pipes* — data strongly typed
  • Data: linear data streams, traditionally – text
Pipe and Filter: Style Analysis

• Topology
  • Usually linear pipelines, sometimes T-joins are possible

• What are common examples of its use? Have you seen this style before?
Pipe and Filter: Style Analysis

- **Topology**
  - Usually linear pipelines, sometimes T-joins are possible

- What are common examples of its use?

Have you seen this style before?

- *Unix*: `ls invoices | grep –e “August” | sort`
- *MS-DOS*: `dir | findstr “Onder*”`
Pipe and Filter: Style Analysis

• **Topology**
  • Usually linear pipelines, sometimes T-joins are possible

• What are common **examples** of its use?

**Have you seen this style before?**

• *Unix*: `ls invoices | grep –e “August” | sort`
• *MS-DOS*: `dir | findstr “Onder*”`

• Operating systems applications, shells
• Massive data processing applications
  - Results of the processing are more important than the process itself
Pipes and Filters: Style Analysis

- What are the **advantages** of using the style?
  - Simplicity
  - Filters are independent
  - New combinations can be easily constructed

- What are the **disadvantages** of using the style?
  - Data structures to be exchanged should be relatively simple
    - Usually text tables
  - No interaction between components

- Relation to **programming languages**
  - Unix shells
Blackboard Style

- **Two kinds** of components
  - Central data structure — blackboard
  - Components operating on the blackboard
  - System control is entirely driven by the blackboard state

- Shared blackboard: problem description
- Multiple experts
  - identify a (sub)problem they can solve,
  - work on it
  - post the solution on the blackboard
  - enable other experts to solve their problem
Experts perform independent tasks.

Blackboard maintains the game state.
Blackboard: Style Analysis

• **Summary:**
  • Separate programs communicate through the shared repository, known as the blackboard

• **Design elements**
  • Components:
    – shared blackboard
    – independent programs, a.k.a. *knowledge sources*
  • Connectors: depending on the context
    – procedure calls, database queries, direct references…
  • Data: stored on the blackboard

• **Topology**: star, the blackboard as the central node
Blackboard: Style Analysis

- What are common **examples** of its use?
  - Heuristic problem solving in artificial intelligence
  - Compiler!

Diagram:
- Lexical analyzer
- Syntactic analyzer
- Semantic analyzer
- Internal representations of the program (stored in **blackboard**)
- Bytecode generator
- Optimizer
Blackboard: Style Analysis

• What are the **advantages** of using the style?
  • Solution strategies should not be preplanned
  • Data/problem determine the solutions!

• What are the **disadvantages** of using the style?
  • Overhead when
    - a straight-forward solution strategy is available
    - interaction between “independent” programs need a complex regulation
    - data on the blackboard is a subject to frequent change (and requires propagation to all other components)
Interpreter Style

• **Compilers** translate the (source) code to the executable form **at once**

• **Interpreters** translate the (source) code instructions **one by one** and execute them
  • To pass data from one instruction to the other we need to keep the Interpreter state
Interpreter Style

- **Compilers** translate the (source) code to the executable form **at once**

  - **C**

- **Interpreters** translate the (source) code instructions **one by one** and execute them
  - To pass data from one instruction to the other we need to keep the Interpreter state

- **Python**
Interpreter Style

- **Compilers** translate the (source) code to the executable form **at once**

- **Interpreters** translate the (source) code instructions **one by one** and execute them
  - To pass data from one instruction to the other we need to keep the Interpreter state

**What about Java?**

a) **Compiler**
b) **Interpreter**
Interpreter Style

- **Compilers** translate the (source) code to the executable form **at once**

- **Interpreters** translate the (source) code instructions **one by one** and execute them
  - To pass data from one instruction to the other we need to keep the Interpreter state

What about Java?

a) Compiler  
b) Interpreter  
Both!
How is this related to architecture?

Interpreter Lunar Lander

- User commands constitute a **language**
  
  “Burn 50” – set the burnrate to 50
  
  “Check status”

  ...

- Example of a domain-specific language (DSL)
  - Do you recall Domain-Specific Software Architectures?
  - Active research topic in Eindhoven
    - 2IS15 Generic language technology
  - This language is being interpreted by the rest of the implementation
**Interpreter Style: Style Analysis**

- **Summary:**
  - Interpreter parses and executes input commands, updating the state maintained by the interpreter

- **Design elements**
  - Components:
    - command **interpreter**
    - program/interpreter **state**
    - user interface.
  - Connectors: typically very closely bound with direct procedure calls and shared state.
  - Data: commands
• **Topology**
  - Tightly-coupled three-tier, state can be separate

• **What are common examples of its use?**
  - Great when the user should be able to program herself
    - e.g., Excel formulas
    - domain-specific languages become more and more popular
      - Not all of them are interpreted, but many of them are…
• What are the **advantages** of using the style?
  • Highly dynamic behavior possible, where the set of commands is dynamically modified.
  • System architecture may remain constant while new capabilities are created based upon existing primitives.

• What are the **disadvantages** of using the style?
  • Performance
    - it takes longer to execute the interpreted code
    - but many optimizations might be possible
  • Memory management
    - when multiple interpreters are invoked simultaneously
Mobile Code Style

- Sometimes interpretation cannot be performed locally
- **Code-on-demand**
  - Client has resources and processing power
  - Server has code to be executed
  - Client requests the code, obtains it and runs it locally

Client → Server:
- Request webpage
- Return JavaScript code

Run in the browser
Mobile Code Style

- Sometimes interpretation cannot be performed locally
- Code-on-demand
- Remote execution/evaluation
  - client has code but does not have resources to execute it
  - software resources (e.g., interpreter)
  - or hardware resources (e.g., processing power)
    - 2IN28 Grid and cloud computing

Client ➔ Server (grid)

- code ➔ run
- results ➔ run
Mobile Code Style

- Sometimes interpretation cannot be performed locally
  - Code-on-demand
  - Remote execution/evaluation
- **Mobile agent**
  - Initiator has code and some resources but not all
  - Can autonomously decide to migrate to a different node to obtain additional resources

http://maf.sourceforge.net/
Mobile Code Style: Major challenge – Security

• Code being executed might be malicious!
  • privacy invasion
  • denial of service

• Solutions:
  • Sandboxing
    – Mobile code runs only in a restricted environment, “sandbox”, and does not have access to vital parts of the system
  • Signing
    – Only mobile code signed by a trusted party can be executed
  • Responsibility: execution dock handling receipt and execution of code and state

• 2IC60 Computer networks and security – Y2Q4
• Master track IST
Mobile Code Style: Style Analysis

• **Summary:**
  - Code moves to be interpreted on another host
  - Variants: code on demand, remote execution, mobile agent

• **Design elements**
  - Components: code interpreter, execution dock
  - Connectors:
    - network protocols
    - code/data packaging for transmission
  - Data: code, program state, data for the code

• **Topology:** network
Mobile Code Style: Style Analysis

- What are common **examples** of its use?
  - processing large amounts of distributed data
  - dynamic behavior / customization

- What are the **advantages** of using the style?
  - dynamic adaptability
  - performance (resources)

- What are the **disadvantages** of using the style?
  - security challenges
  - network/transmission costs
Some Common Styles

- **Traditional, language-influenced styles**
  - Main program and subroutines
  - Object-oriented

- **Layered**
  - (Virtual machines)
  - Client-server

- **Data-flow styles**
  - Batch sequential
  - Pipe and filter

- **Shared memory**
  - Blackboard
  - Rule based

- **Interpreter**
  - Interpreter
  - Mobile code

- **Implicit invocation**
  - Event-based
  - Publish-subscribe

- **Peer-to-peer**

- **“Derived” styles**
  - C2
  - CORBA
Architectural Styles

- An architectural style is a named collection of architectural design decisions that
  - are applicable in a given development context
  - constrain architectural design decisions that are specific to a particular system within that context
  - elicit beneficial qualities in each resulting system

- Reflect less domain specificity than architectural patterns

- Useful in determining everything from subroutine structure to top-level application structure

- Many styles exist and we will discuss them in detail in the next lecture

Some Common Styles

- Traditional, language-influenced styles
  - Main program and subroutines
  - Object-oriented

- Layered
  - (Virtual machines)
  - Client-server

- Data-flow styles
  - Batch sequential
  - Pipe and filter

- Shared memory
  - Blackboard
  - Rule based

- Interpreter
  - Interpreter
  - Mobile code

- Implicit invocation
  - Event-based
  - Publish-subscribe

- Peer-to-peer
  - “Derived” styles
    - C2
    - CORBA
Scripting languages (i.e. JavaScript, VBScript), ActiveX control, embedded Word/Excel macros.
# Style Summary (1/4)

<table>
<thead>
<tr>
<th>Style Category &amp; Name</th>
<th>Summary</th>
<th>Use It When</th>
<th>Avoid It When</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Language-influenced styles</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Main Program and Subroutines</td>
<td>Main program controls program execution, calling multiple subroutines.</td>
<td>Application is small and simple.</td>
<td>Complex data structures needed. Future modifications likely.</td>
</tr>
<tr>
<td>Object-oriented</td>
<td>Objects encapsulate state and accessing functions</td>
<td>Close mapping between external entities and internal objects is sensible. Many complex and interrelated data structures.</td>
<td>Application is distributed in a heterogeneous network. Strong independence between components necessary. High performance required.</td>
</tr>
<tr>
<td><strong>Layered</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Virtual Machines</td>
<td>Virtual machine, or a layer, offers services to layers above it</td>
<td>Many applications can be based upon a single, common layer of services. Interface service specification resilient when implementation of a layer must change.</td>
<td>Many levels are required (causes inefficiency). Data structures must be accessed from multiple layers.</td>
</tr>
<tr>
<td>Client-server</td>
<td>Clients request service from a server</td>
<td>Centralization of computation and data at a single location (the server) promotes manageability and scalability; end-user processing limited to data entry and presentation.</td>
<td>Centrality presents a single-point-of-failure risk; Network bandwidth limited; Client machine capabilities rival or exceed the server’s.</td>
</tr>
</tbody>
</table>
### Data-flow styles

**Batch sequential**  
Separate programs executed sequentially, with batched input  
Problem easily formulated as a set of sequential, severable steps.  
Interactivity or concurrency between components necessary or desirable. Random-access to data required. Interaction between components required. Exchange of complex data structures between components required.

**Pipe-and-filter**  
Separate programs, a.k.a. filters, executed, potentially concurrently. Pipes route data streams between filters  
[As with batch-sequential] Filters are useful in more than one application. Data structures easily serializable.

### Shared memory

**Blackboard**  
Independent programs, access and communicate exclusively through a global repository known as blackboard  
All calculation centers on a common, changing data structure; Order of processing dynamically determined and data-driven.  
Programs deal with independent parts of the common data. Interface to common data susceptible to change. When interactions between the independent programs require complex regulation.

**Rule-based**  
Use facts or rules entered into the knowledge base to resolve a query  
Problem data and queries expressible as simple rules over which inference may be performed.  
Number of rules is large. Interaction between rules present. High-performance required.
### Style Summary, continued (3/4)

<table>
<thead>
<tr>
<th><strong>Interpreter</strong></th>
<th>Interpreter parses and executes the input stream, updating the state maintained by the interpreter</th>
<th>Highly dynamic behavior required. High degree of end-user customizability.</th>
<th>High performance required.</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Mobile Code</strong></td>
<td>Code is mobile, that is, it is executed in a remote host</td>
<td>When it is more efficient to move processing to a data set than the data set to processing. When it is desirous to dynamically customize a local processing node through inclusion of external code</td>
<td>Security of mobile code cannot be assured, or sandboxed. When tight control of versions of deployed software is required.</td>
</tr>
</tbody>
</table>
### Implicit Invocation

<table>
<thead>
<tr>
<th>Style</th>
<th>Description</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Publish-subscribe</td>
<td>Publishers broadcast messages to subscribers</td>
<td>Components are very loosely coupled. Subscription data is small and efficiently transported.</td>
<td>When middleware to support high-volume data is unavailable.</td>
</tr>
<tr>
<td>Event-based</td>
<td>Independent components asynchronously emit and receive events communicated over event buses</td>
<td>Components are concurrent and independent. Components heterogeneous and network-distributed.</td>
<td>Guarantees on real-time processing of events is required.</td>
</tr>
<tr>
<td>Peer-to-peer</td>
<td>Peers hold state and behavior and can act as both clients and servers</td>
<td>Peers are distributed in a network, can be heterogeneous, and mutually independent. Robust in face of independent failures. Highly scalable.</td>
<td>Trustworthiness of independent peers cannot be assured or managed. Resource discovery inefficient without designated nodes.</td>
</tr>
</tbody>
</table>

### More complex styles

<table>
<thead>
<tr>
<th>Style</th>
<th>Description</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>C2</td>
<td>Layered network of concurrent components communicating by events</td>
<td>When independence from substrate technologies required. Heterogeneous applications. When support for product-lines desired.</td>
<td>When high-performance across many layers required. When multiple threads are inefficient.</td>
</tr>
<tr>
<td>Distributed</td>
<td>Objects instantiated on different hosts</td>
<td>Objective is to preserve illusion of location-transparency</td>
<td>When high overhead of supporting middleware is excessive. When network properties are unmaskable, in practical terms.</td>
</tr>
</tbody>
</table>
Inference engine parses user input and determines whether it is a fact/rule or a query. If it is a fact/rule, it adds this entry to the knowledge base. Otherwise, it queries the knowledge base for applicable rules and attempts to resolve the query.
Rule-Based Style (cont’d)

- **Components**: User interface, inference engine, knowledge base
- **Connectors**: Components are tightly interconnected, with direct procedure calls and/or shared memory.
- **Data Elements**: Facts and queries
- **Behavior of the application** can be very easily modified through addition or deletion of rules from the knowledge base.
- **Caution**: When a large number of rules are involved understanding the interactions between multiple rules affected by the same facts can become very difficult.
Rule Based LL

User Interface
fact: burnrate(25).
goal: landed(spacecraft).

Procedure Call
in: user input, query
out: variables resolved, T/F

Inference Engine

Data Access
in: database query
out: database results

Knowledge Base
Facts (a, br, f, t, v)
& Rules
landed(spacecraft) :-
<= (a, 0), <=(v, 3)