2IW80 Software specification and architecture

# Software architecture: Architectural Styles

Alexander Serebrenik

Technische Universiteit
Eindhoven
University of Technology

Where innovation starts

# Before we start…

## True or false?

- Domain-Specific Software Architecture is a part of a Reference 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.

# Before we start…

## True or false?

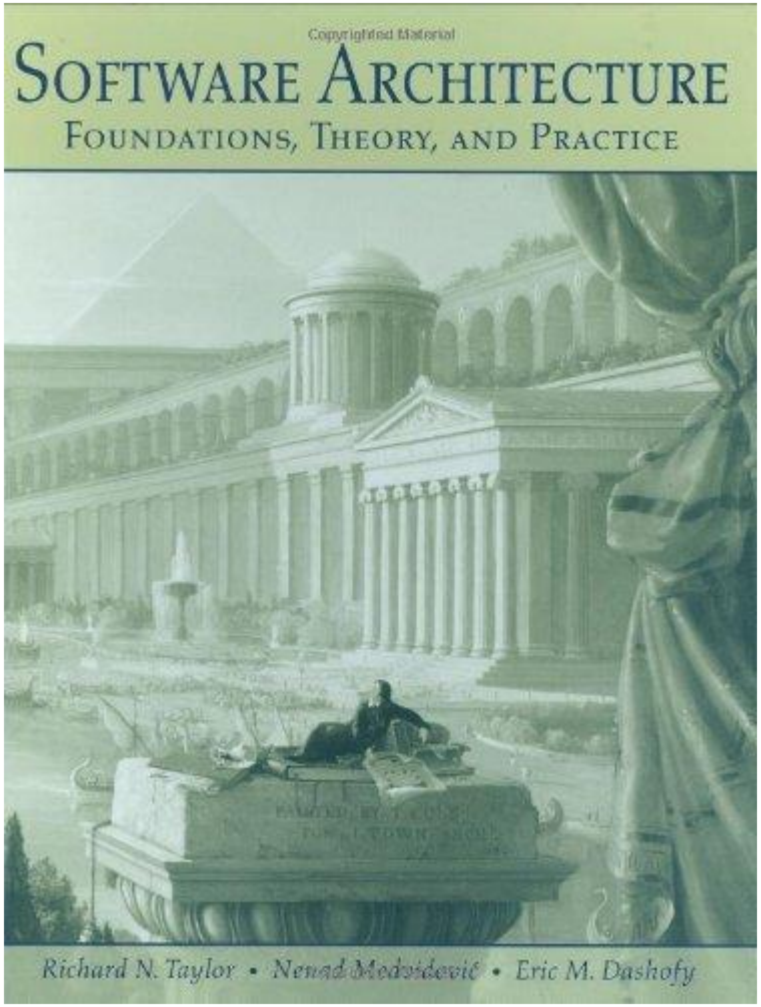
- Domain-Specific Software Architecture is a part of a Reference Architecture: **FALSE**
- Domain-Specific Software Architecture is broader applicable than a product line: **TRUE**
- Model-View-Controller is an examples of a Domain-Specific Software Architecture

# 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 examples of a Domain-Specific Software Architecture **FALSE**

# This week sources


Copyrighted Material
SOFTWARE ARCHITECTURE
FOUNDATIONS, THEORY, AND PRACTICE
Richard N. Taylor • Nenad Medvidović • Eric M. Dashofy

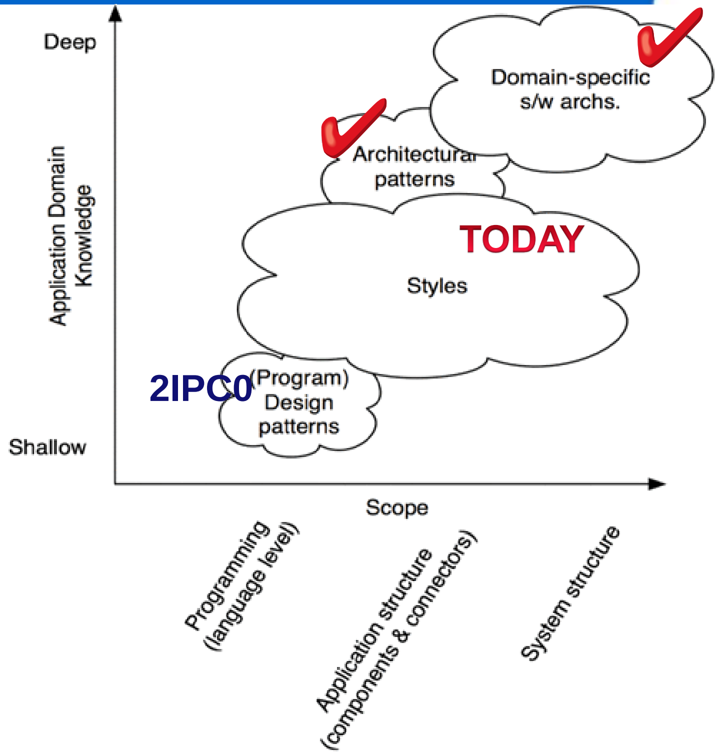
## Slides by

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



Deep
Shallow
Application Domain Knowledge
Domain-specific s/w archs.
Architectural patterns
TODAY
Styles
2IPC0
(Program) Design patterns
Scope
Programming (language level)
Application structure (components & connectors)
System structure

# 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

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# 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

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# 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

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# 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

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# Architecture Style Analysis

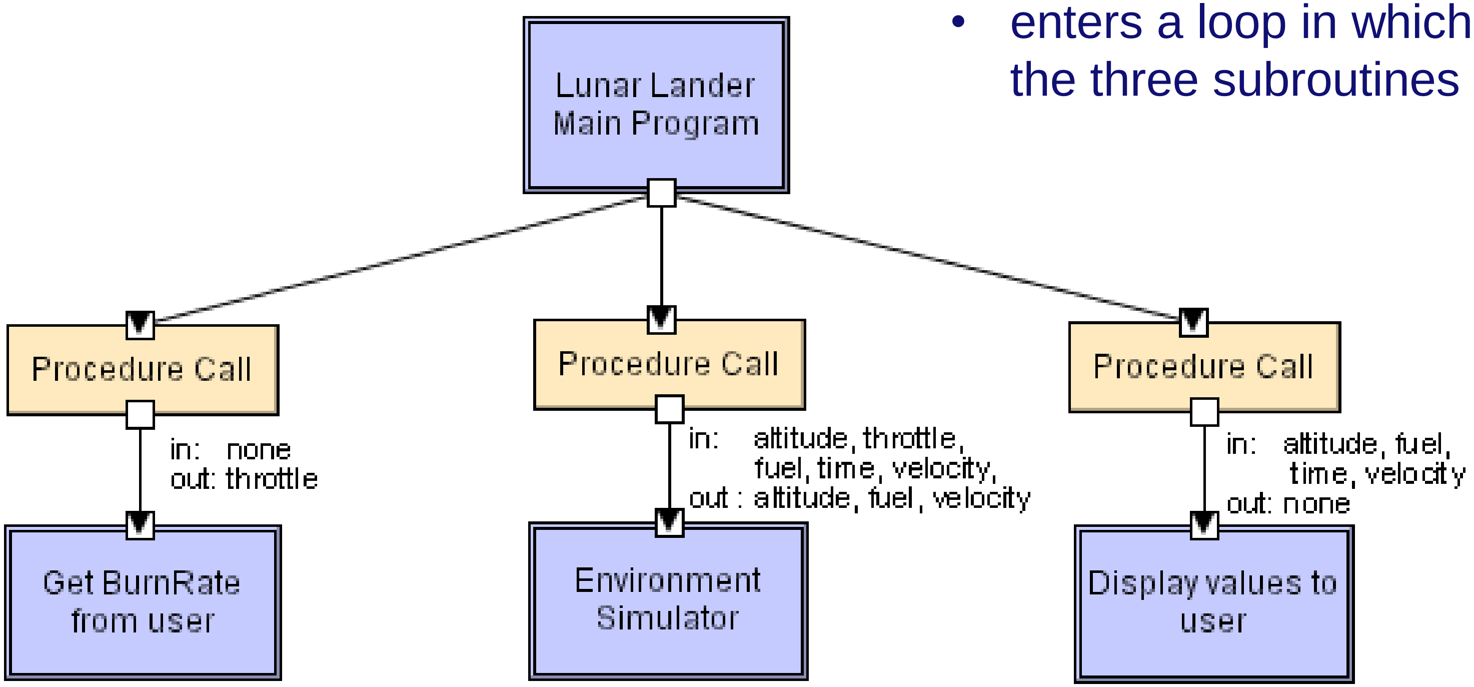
- Summary
- Design elements (components, connectors, data)
- Topology
- Examples of use
- Advantages/disadvantages
- Relation to programming languages/environments

# 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.



# 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

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

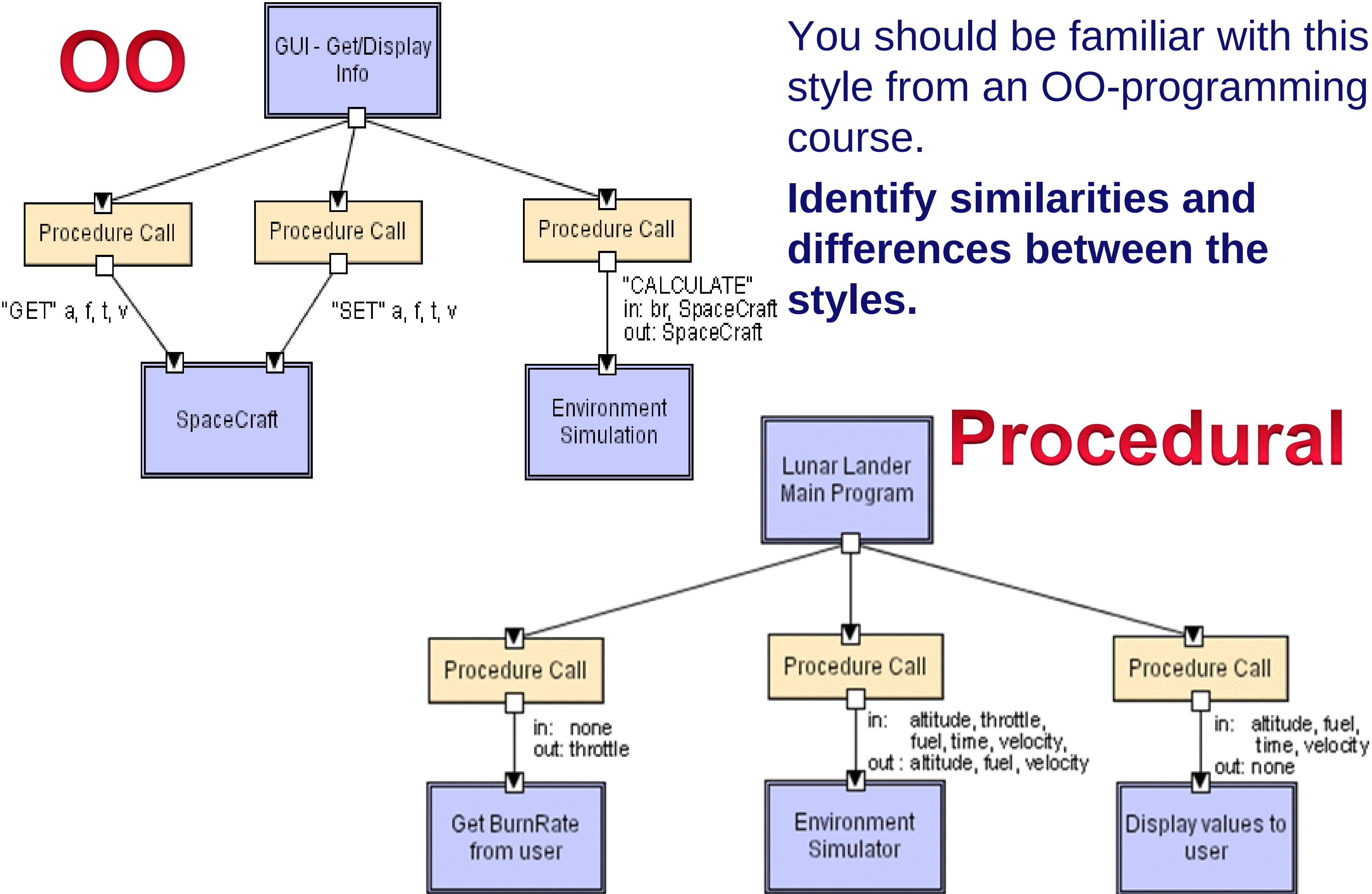
# Main program and subroutines: Style Analysis

- 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
- Relation to **programming languages/environments**
    - Traditional programming languages: BASIC, Pascal, C…

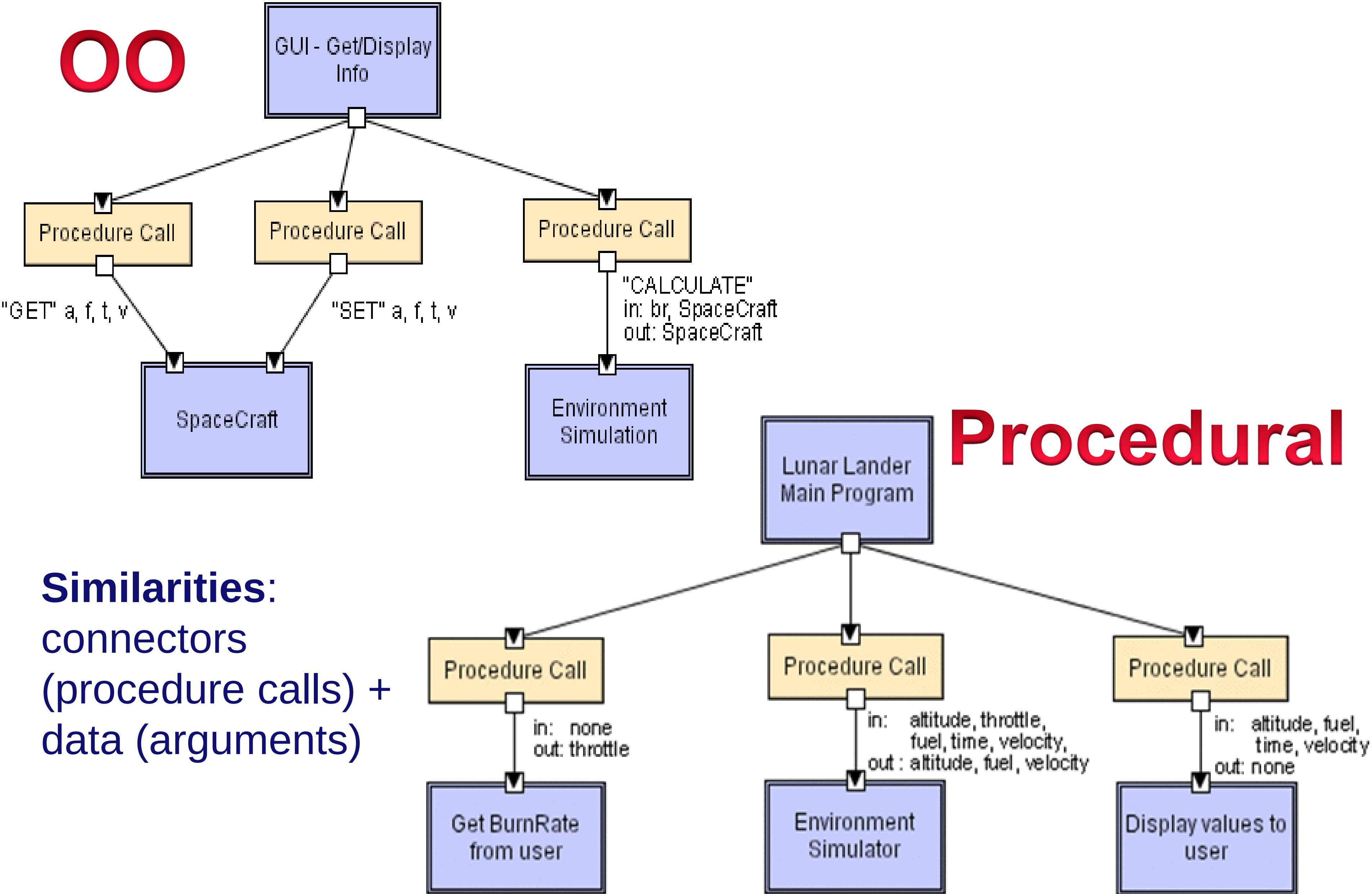
# Object-Oriented Lunar Lander

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

**Identify similarities and differences between the styles.**

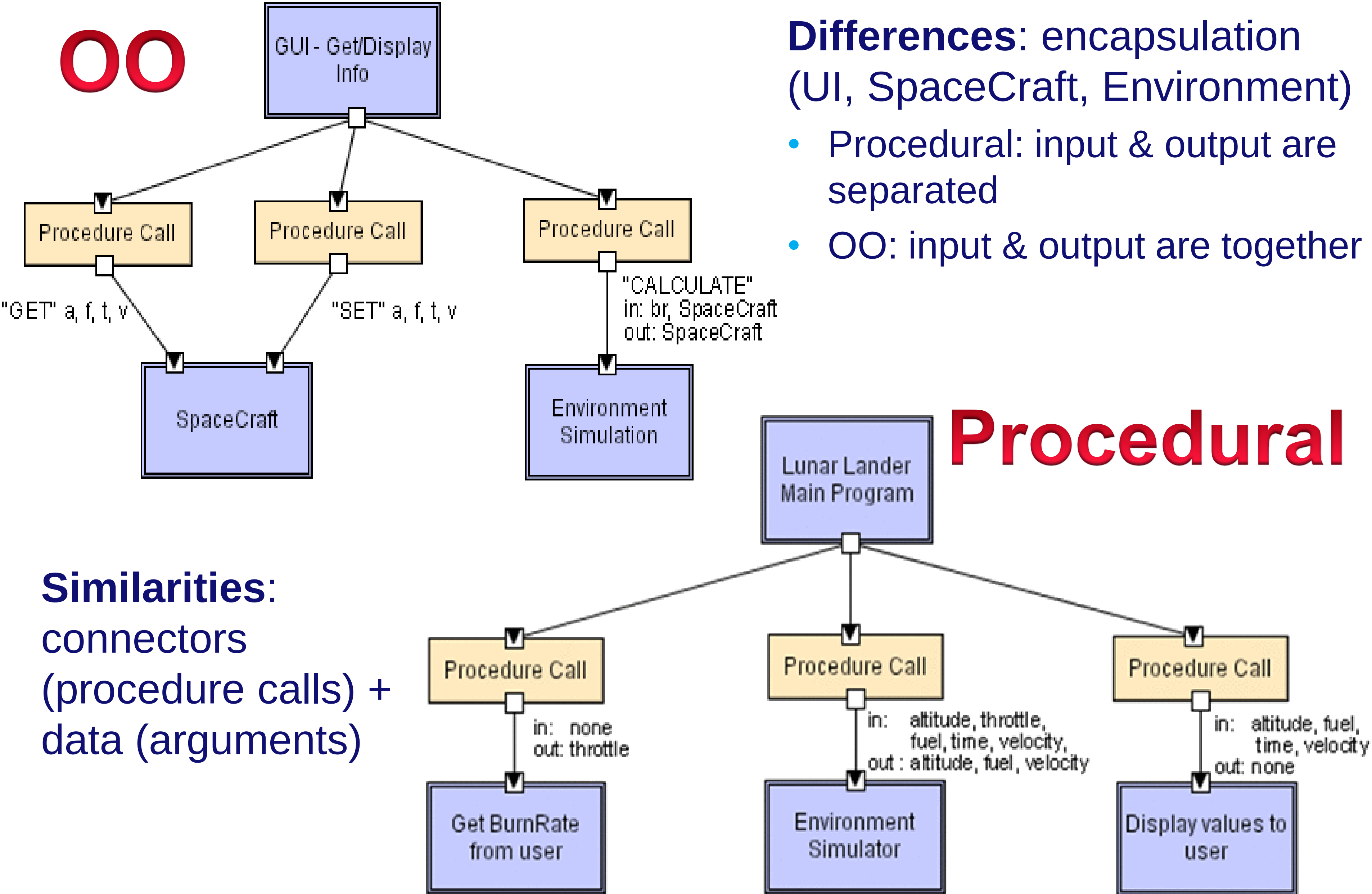


# Object-Oriented Lunar Lander



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

# Object-Oriented Lunar Lander

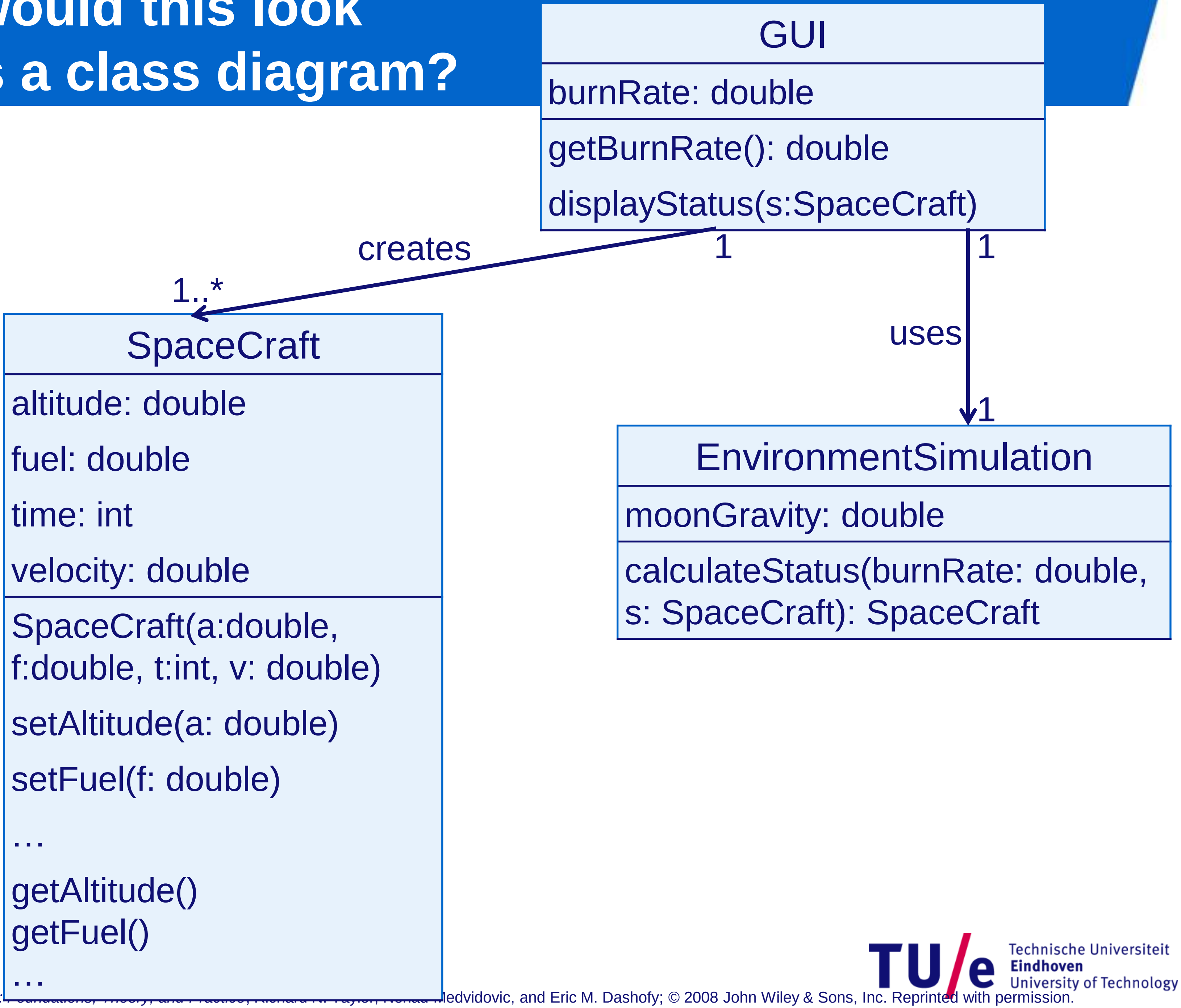


**Differences**: encapsulation (UI, SpaceCraft, Environment)

- Procedural: input & output are separated
- OO: input & output are together

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

# How would this look like as a class diagram?



GUI

burnRate: double

getBurnRate(): double

displayStatus(s:SpaceCraft)

creates

1

1..*

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()

…

1

uses

1

EnvironmentSimulation

moonGravity: double

calculateStatus(burnRate: double, s: SpaceCraft): SpaceCraft

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# 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

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# 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

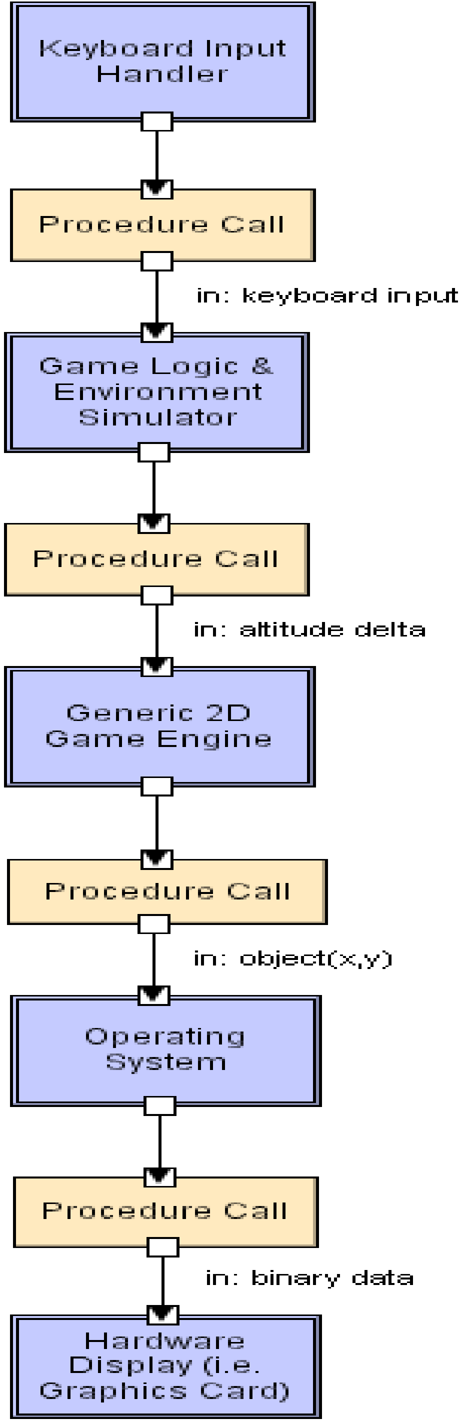
- Relation to **programming languages/environments**
  - OO-languages: Java, C++…

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# Layered Style Lunar Lander

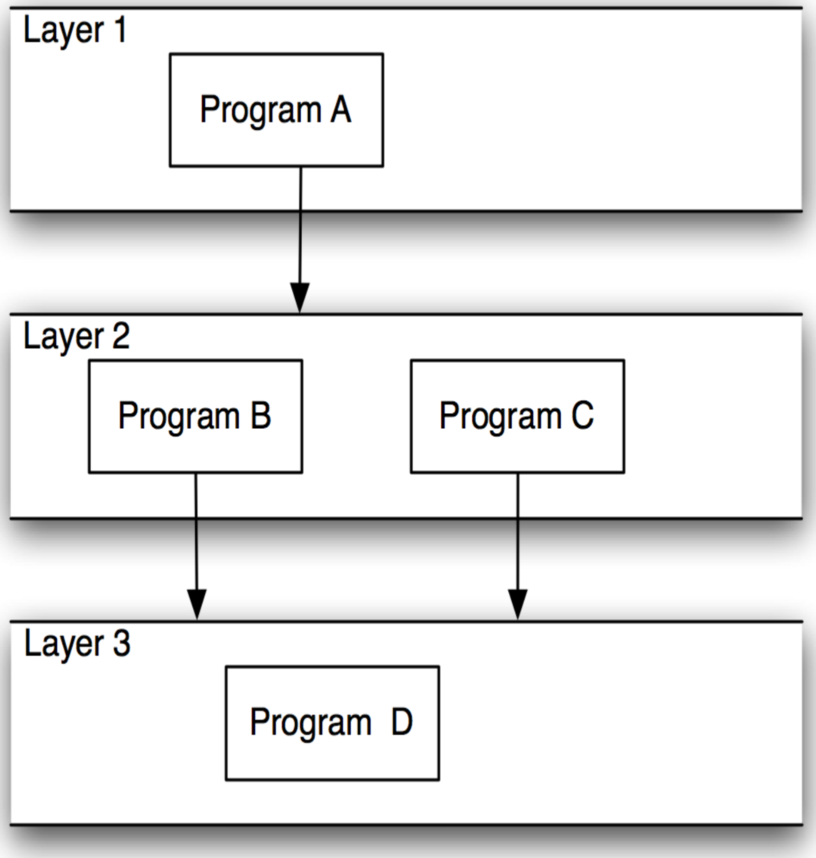
- **Basic idea:**
  - Each layer exposes an interface (API) to be used by the layer above it
  - Each layer acts as a
    - *Server:* service provider to layer “above”
    - *Client:* service consumer of the layer “below”

- 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.

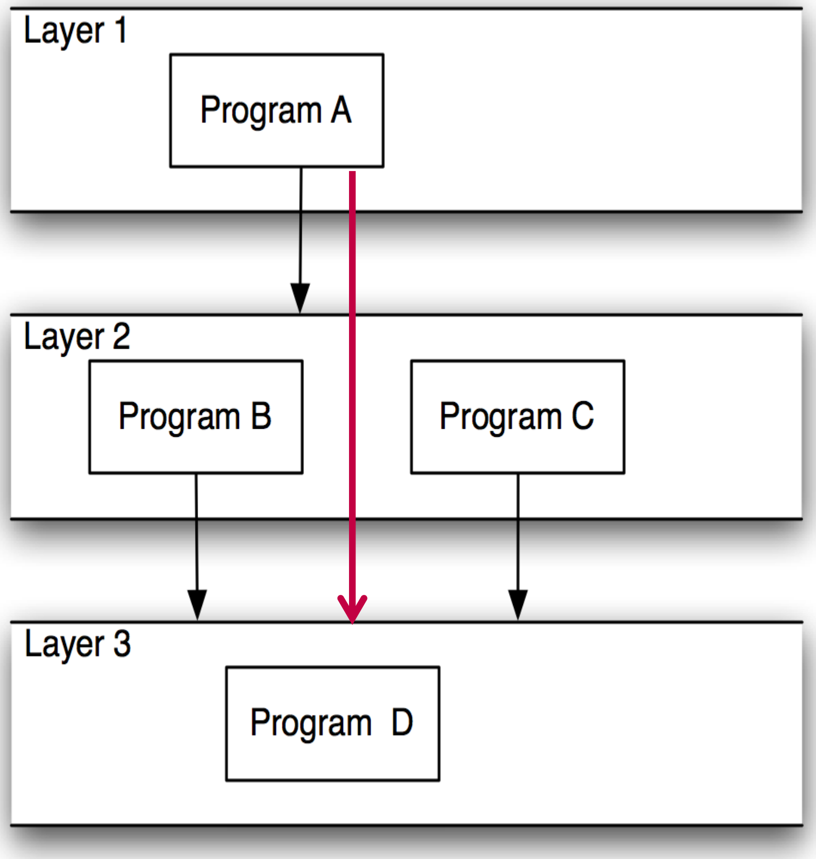


# Layering

## Strict Layering



## Nonstrict Layering



*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

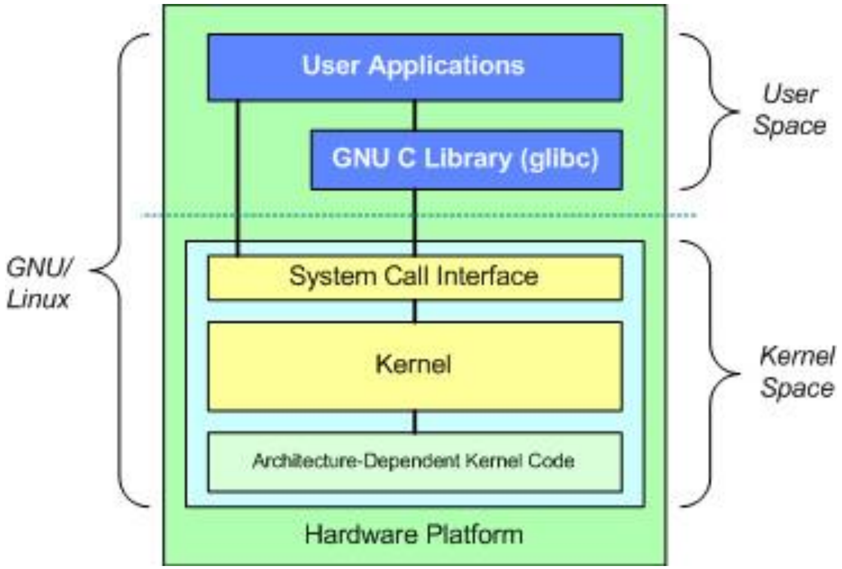
# Layered Style: Style Analysis

- **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)

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# 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



http://www.ibm.com/developerworks/linux/library/l-linux-kernel/

# 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)

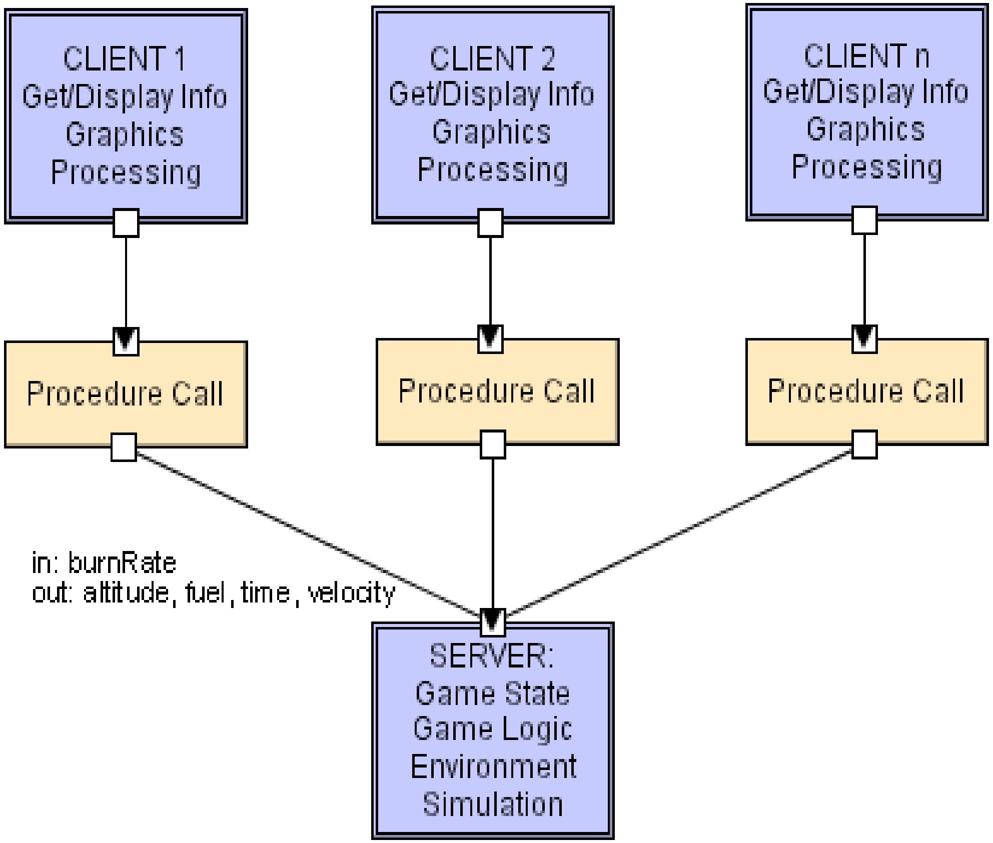
*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# 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



*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# 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

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# 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

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

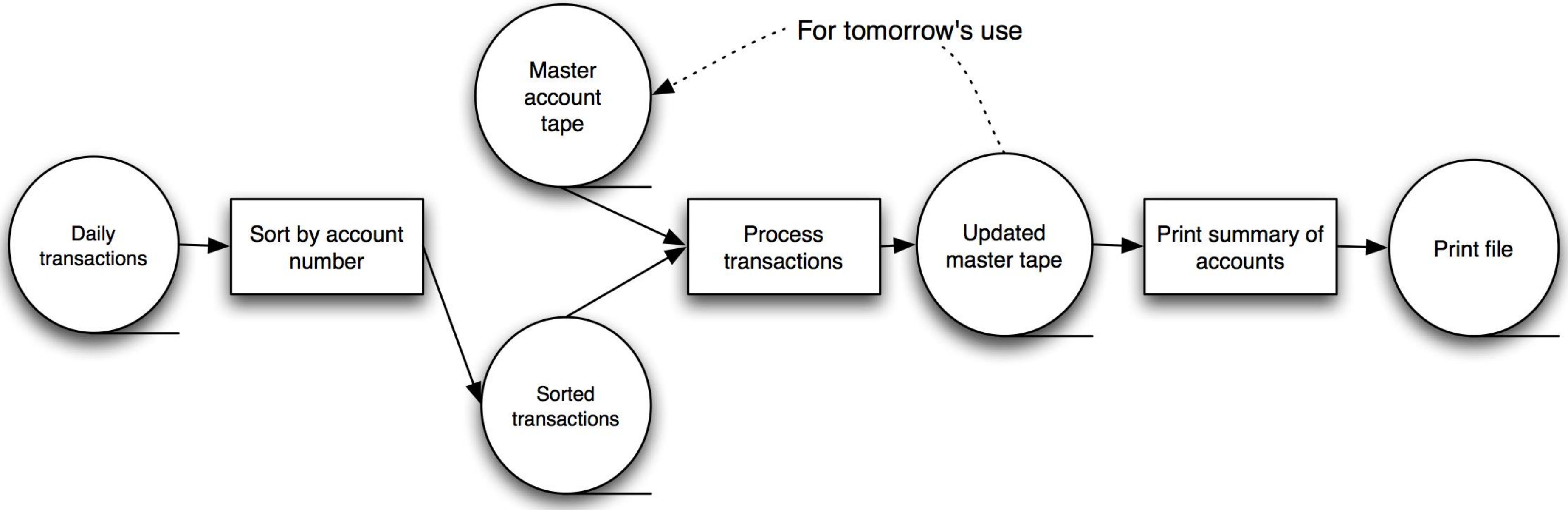
# Batch Sequential

- **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

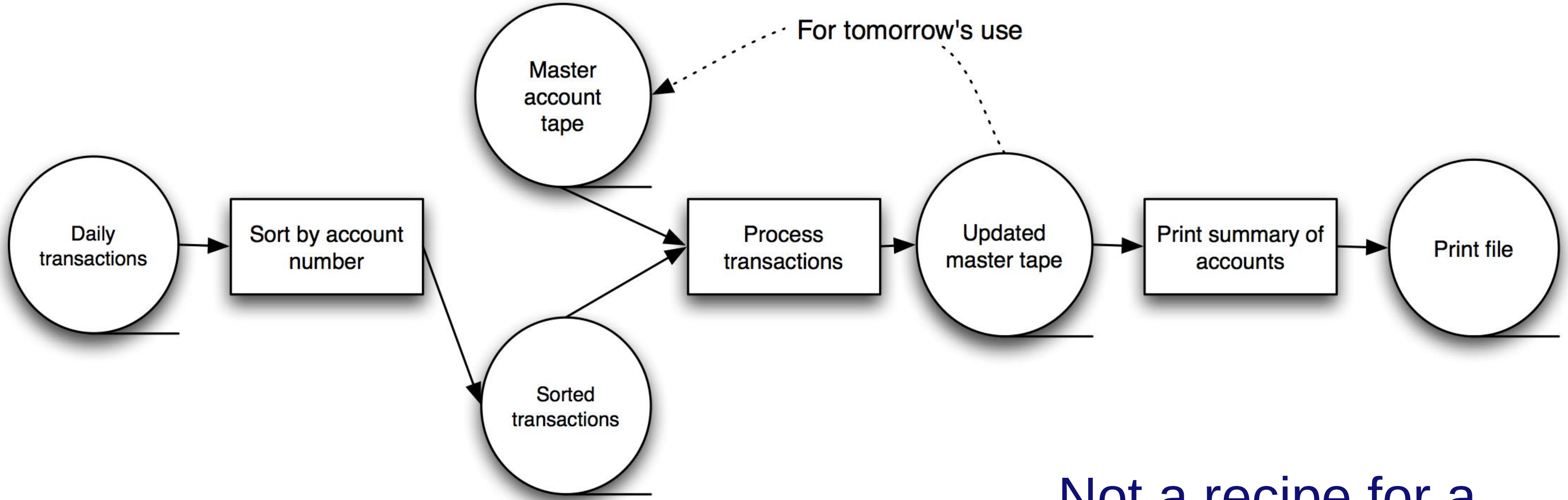
*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

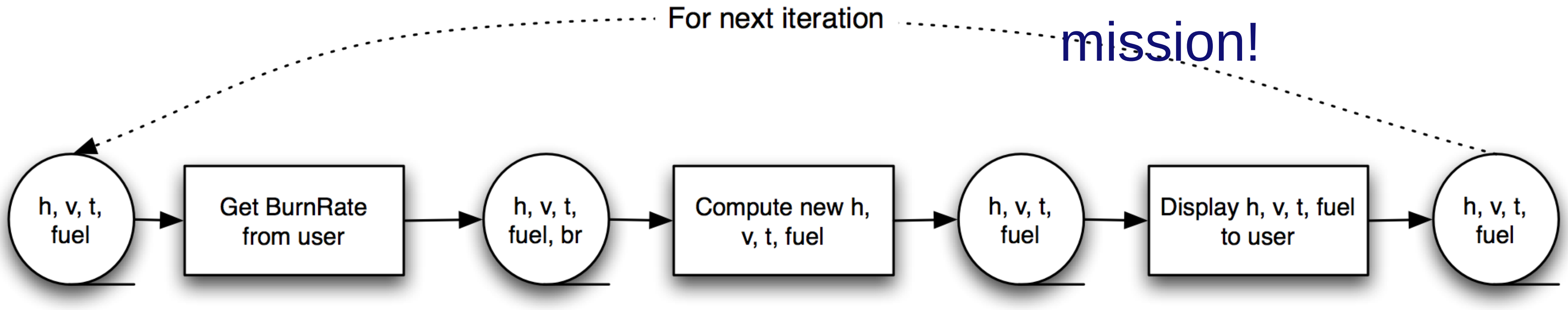
# Batch Sequential



What about the Lunar Lander?

# Batch Sequential





Not a recipe for a successful lunar mission!

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# 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

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# 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

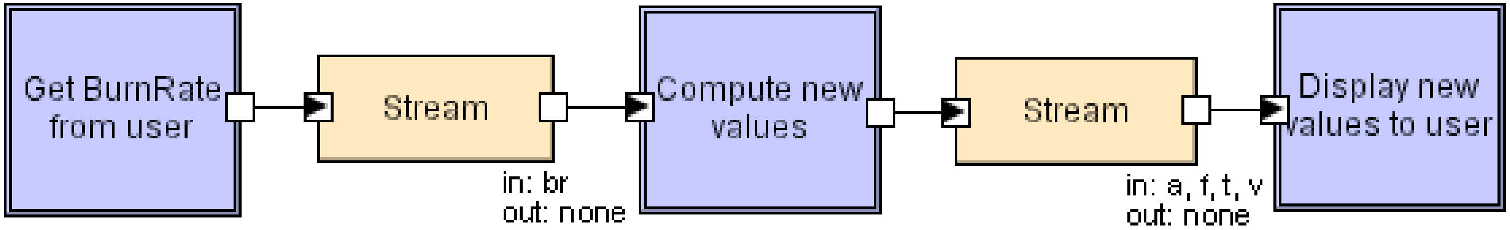
*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# 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

- Lunar Lander



*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# 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?**

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# 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

Technische Universiteit
Eindhoven
University of Technology

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# Blackboard Style

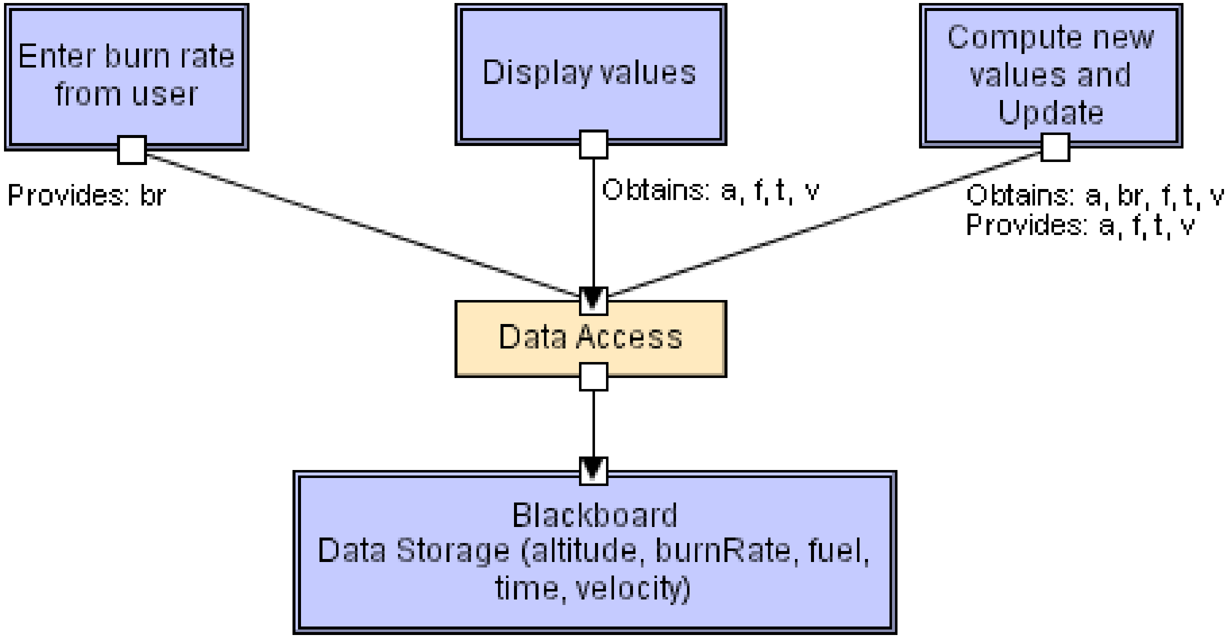
- **Two kinds** of components
  - Central data structure — blackboard
  - Components operating on the blackboard
- System control is entirely driven by the blackboard state

https://www.google.nl/url?sa=i&rct=j&q=&esrc=s&source=images&cd=&cad=rja&docid=_lcXWx6tOHnO2M&tbnid=vKc0m3nirDdc6M:&ved=0CAUQjRw&url=https%3A%2F%2Fwww.facebook.com%2Fmontereyinstitute%3Fref%3Dstream%26viewer_id%3D0&ei=HenjUrWLC4XGswaUh4GoAg&psig=AFQjCNHy_ehnRWgddxY0e9t-Uxvsrd8Hsg&ust=1390754460129161

- 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

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# Blackboard Lunar Lander


Enter burn rate from user
Display values
Compute new values and Update
Provides: br
Obtains: a, f, t, v
Obtains: a, br, f, t, v
Provides: a, f, t, v
Data Access
Blackboard
Data Storage (altitude, burnRate, fuel, time, velocity)

Experts perform independent tasks

Blackboard maintains the game state

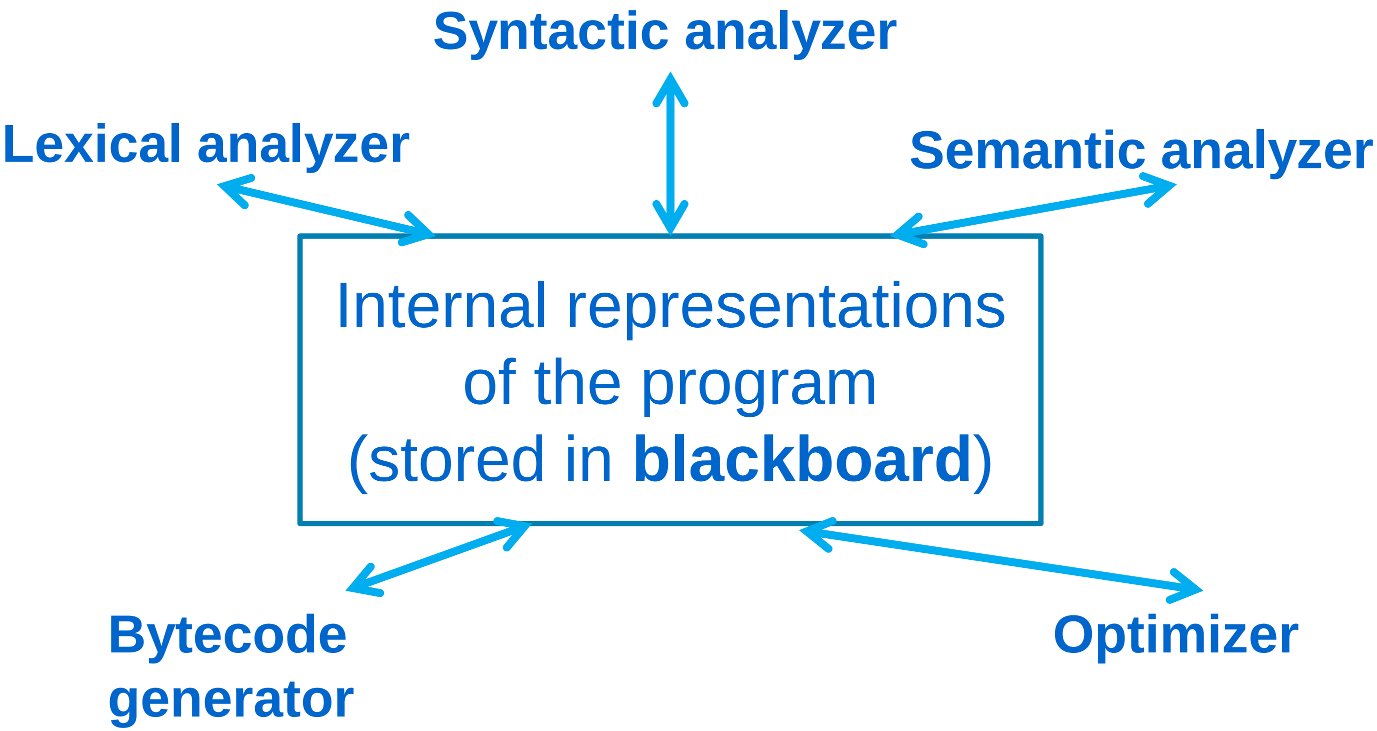
*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# 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!



# 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)

Technische Universiteit
Eindhoven
University of Technology

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# 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

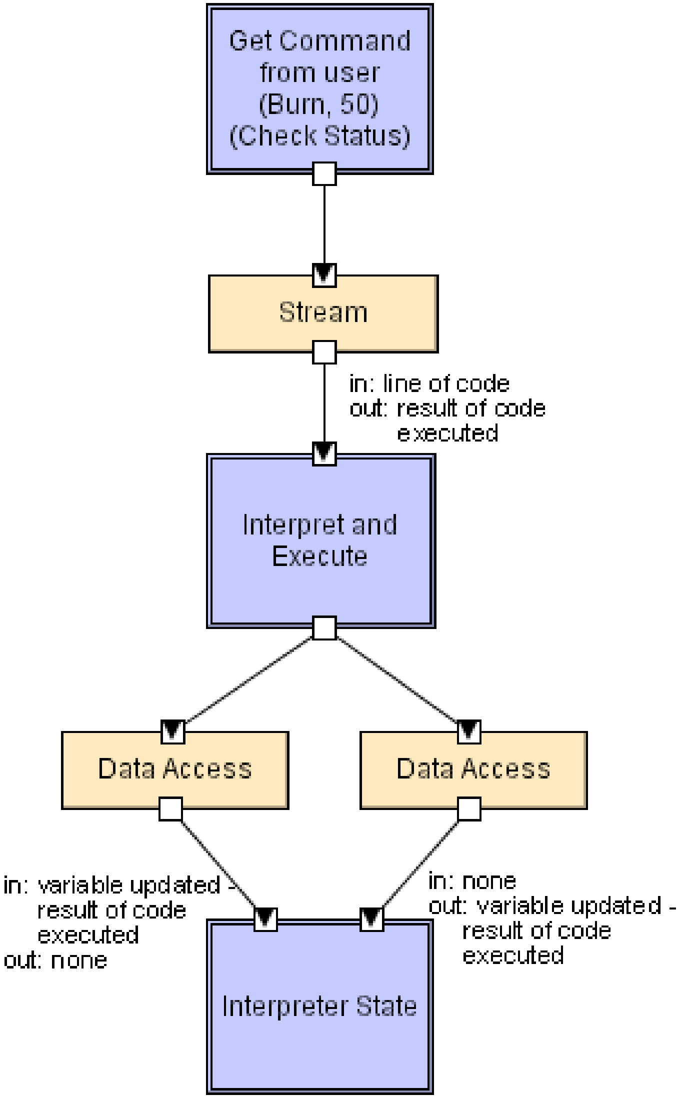
- User commands constitute a ***language***

  “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


Get Command from user (Burn, 50) (Check Status)
Stream
in: line of code
out: result of code executed
Interpret and Execute
Data Access
Data Access
in: variable updated - result of code executed
out: none
in: none
out: variable updated - result of code executed
Interpreter State

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# 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

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# Interpreter Style: Style Analysis

- **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…

# Interpreter Style: Style Analysis

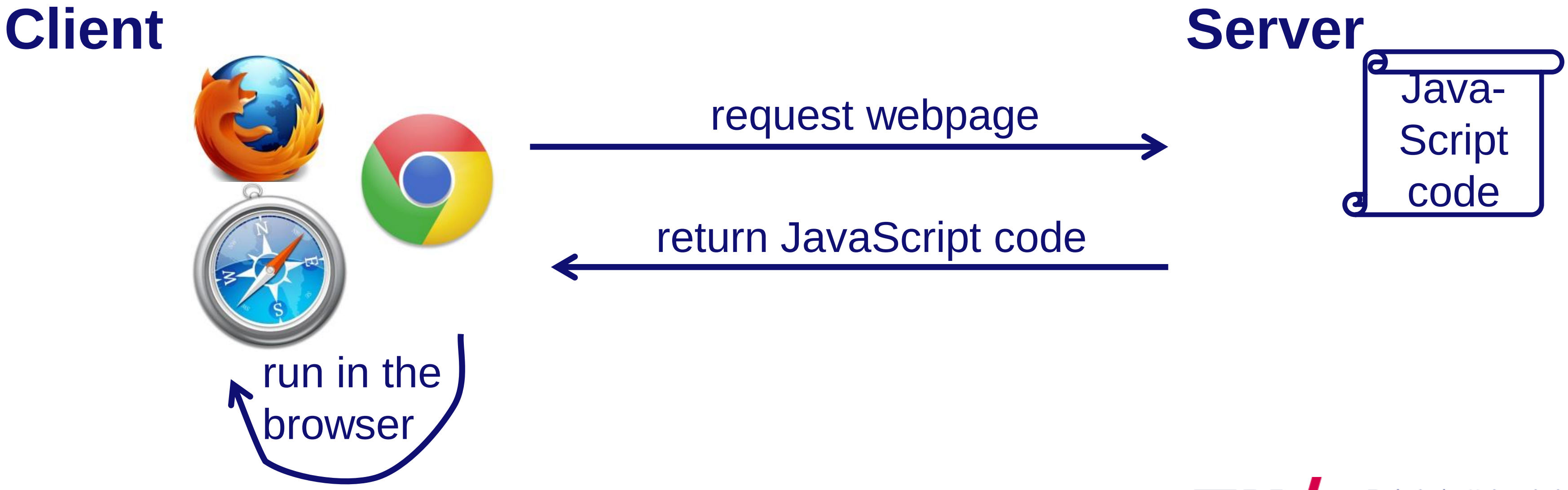
- What are the **advantages** of using the style?
  - Highly dynamic behavior possible, where the set of commands is dynamically modified.
  - System architecture may remain constant while new capabilities are created based upon existing primitives.

- 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

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# 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

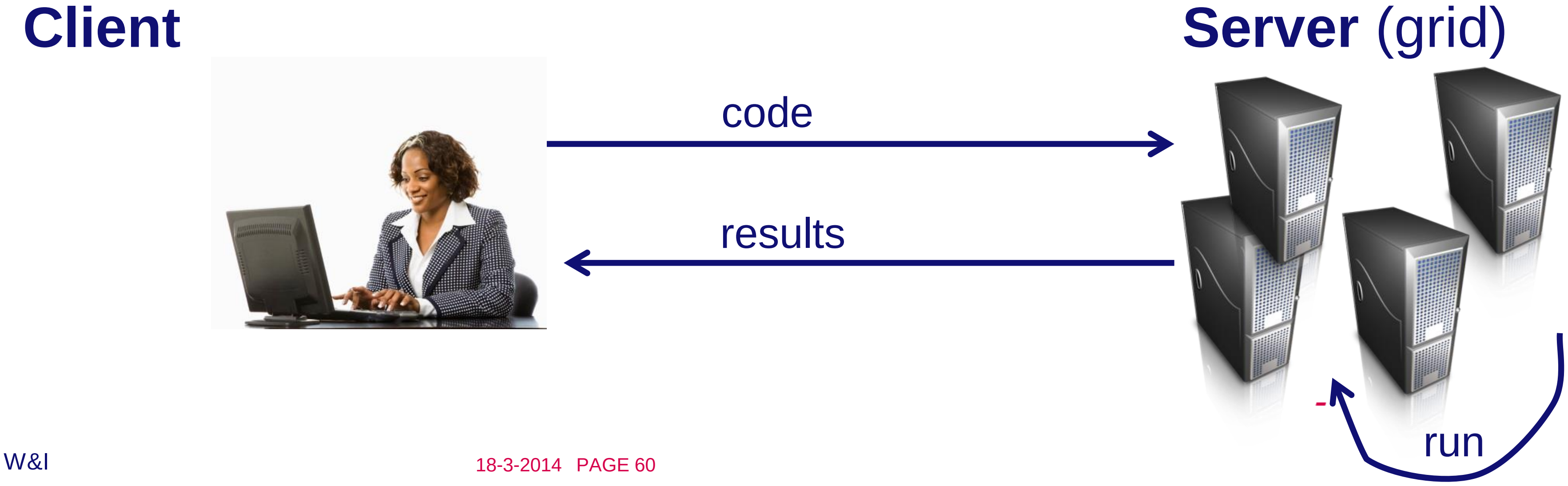
return JavaScript code

Java-Script code

run in the browser

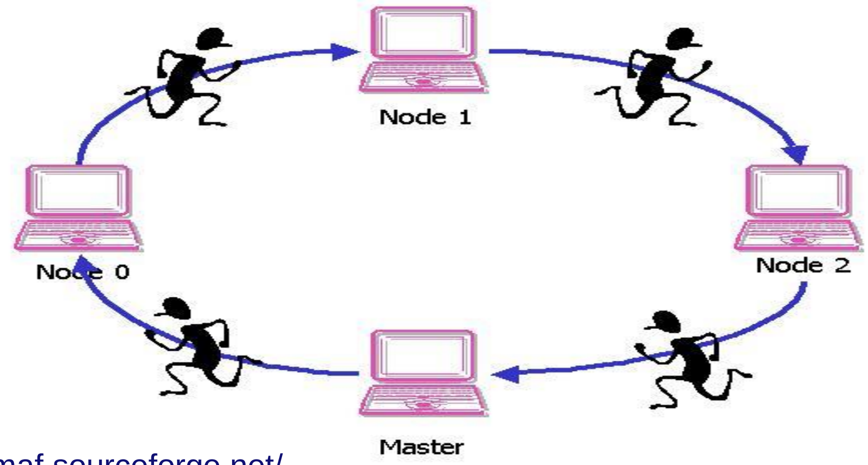
# Mobile Code Style

- 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



# 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

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# 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

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# 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

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# Implicit Invocation Styles

- **Basic idea**
  - Event announcement instead of method invocation
  - “Listeners” register interest in and associate methods with events
  - System invokes all registered methods implicitly

- **Style invariants**
  - “Announcers” are unaware of their events’ effects
  - No assumption about processing in response to events

# Publish-Subscribe

- **Subscribers** register/deregister to receive specific messages or specific content.
- **Publishers** broadcast messages to subscribers.

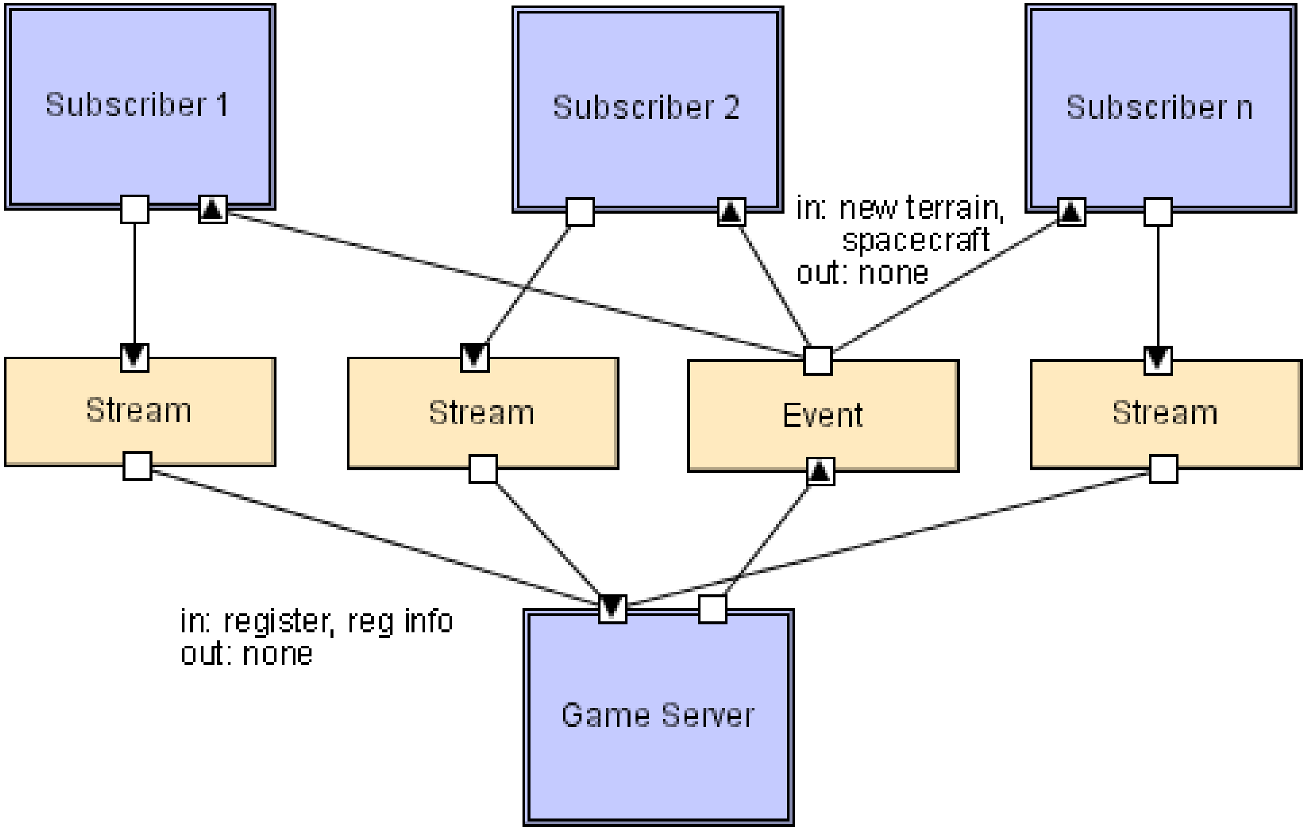
- **Analogy**: newspaper subscription
  - Subscriber chooses the newspaper
  - Publisher delivers only to subscribers
  - *Ergo*, publisher has to maintain a list of subscribers

  - Sometimes we'll need proxies to manage distribution.

http://israel21c.org/israel-in-the-spotlight/going-on-vacation-dont-stop-your-newspaper-subscription-donate-it/

# Publish-Subscriber Lunar Lander

Players



*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# Publish-Subscribe Style: Style Analysis

- **Summary**:
  - Subscribers register/deregister to receive specific messages or specific content.
  - Publishers broadcast messages to subscribers synchronously or asynchronously.
- **Design elements**
  - Components: publishers, subscribers
  - Connectors: procedure calls/network protocols
  - Data: subscriptions, notifications, published information
- **Topology**:
  - Either subscribers directly connected to publishers
  - Or via intermediaries

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# Publish-Subscribe Style: Style Analysis

- What are common **examples** of its use?
  - Social media “friending”
  - GUI
  - Multi-player network-based games

- What are the **advantages** of using the style?
  - Subscribers are independent from each other
  - Very efficient one-way information dissemination

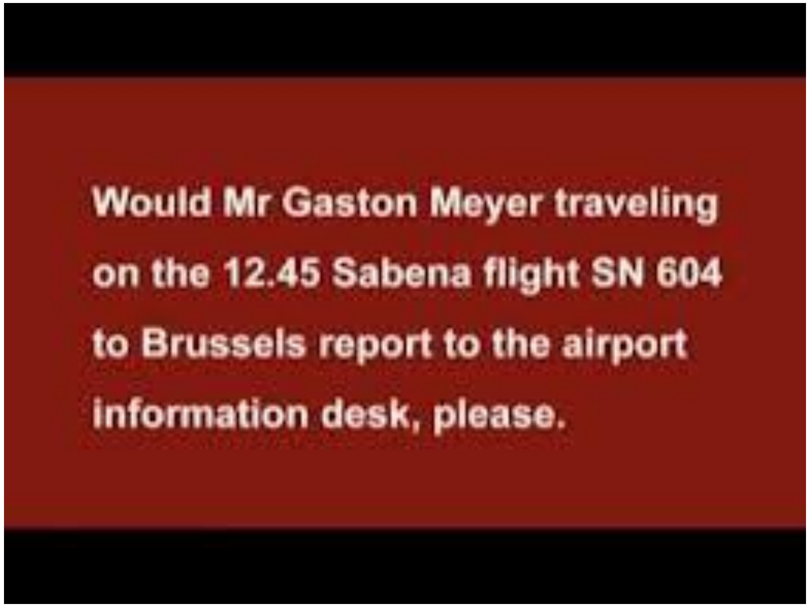
- What are the **disadvantages** of using the style?
  - When a number of subscribers is very high, special protocols are needed

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# Event-Based Style

- In **Publish-Subscribe** the publisher is responsible for maintaining the list of subscribers

- What if the subscribers were responsible for knowing their publishers?



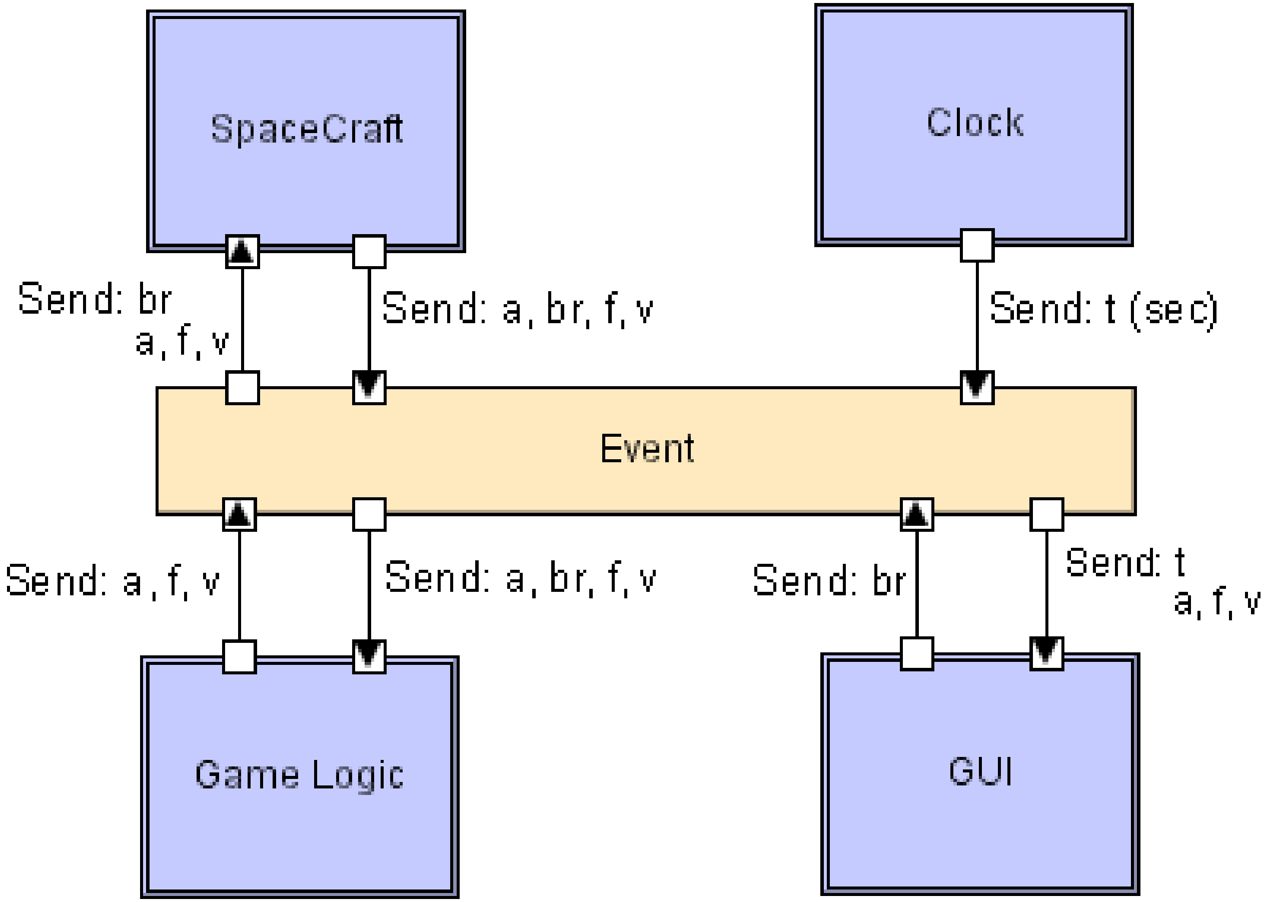
Would Mr Gaston Meyer traveling on the 12.45 Sabena flight SN 604 to Brussels report to the airport information desk, please.

We no longer need to distinguish publishers and subscribers!

# Event-based Lunar Lander

Frequently called “**event bus**”

Commercial middleware



*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# Event-Based Style: Style Analysis

- **Summary**:
  - Independent components asynchronously emit and receive events communicated over event buses
- **Design elements**
  - Components: concurrent event generators/consumers
  - Connectors: event bus (may be more than one)
  - Data: events
- **Topology**:
  - Communication via the event bus only

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# Event-Based Style: Style Analysis

- What are common **examples** of its use?
  - User interface software
  - Enterprise information systems with many independent components (financial, HR, production, …)

- What are the **advantages** of using the style?
  - Scalable
  - Easy to evolve (just add another component!)
  - Heterogeneous (as long as components can communicate with the bus they can be implemented in any possible way)

- What are the **disadvantages** of using the style?
  - No guarantee when the event will be processed

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# Peer-to-Peer Style

- In the Event-Based approach we no longer distinguish between publishers and subscribers
  - “Every component can act as publisher and/or subscriber”

- What if we try to do the same for **“client-server”**?
  - We had it in the **layered (virtual machine)** style
  - But it was restricted to the layered structure!

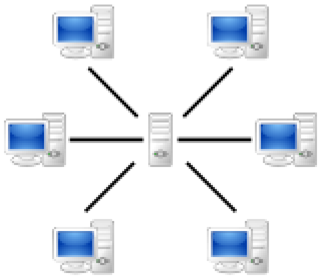
# Peer-to-Peer Style

- In the Event-Based approach we no longer distinguish between publishers and subscribers
  - “Every component can act as publisher and/or subscriber”

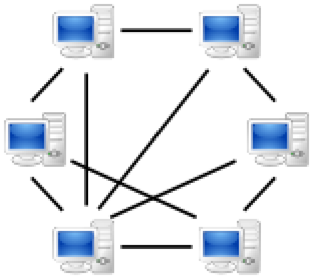
- What if we try to do the same for **“client-server”**?
  - We had it in the **layered (virtual machine)** style
  - But it was restricted to the layered structure!

Peers:

- independent components
- can act as either clients or servers



*Client-Server*



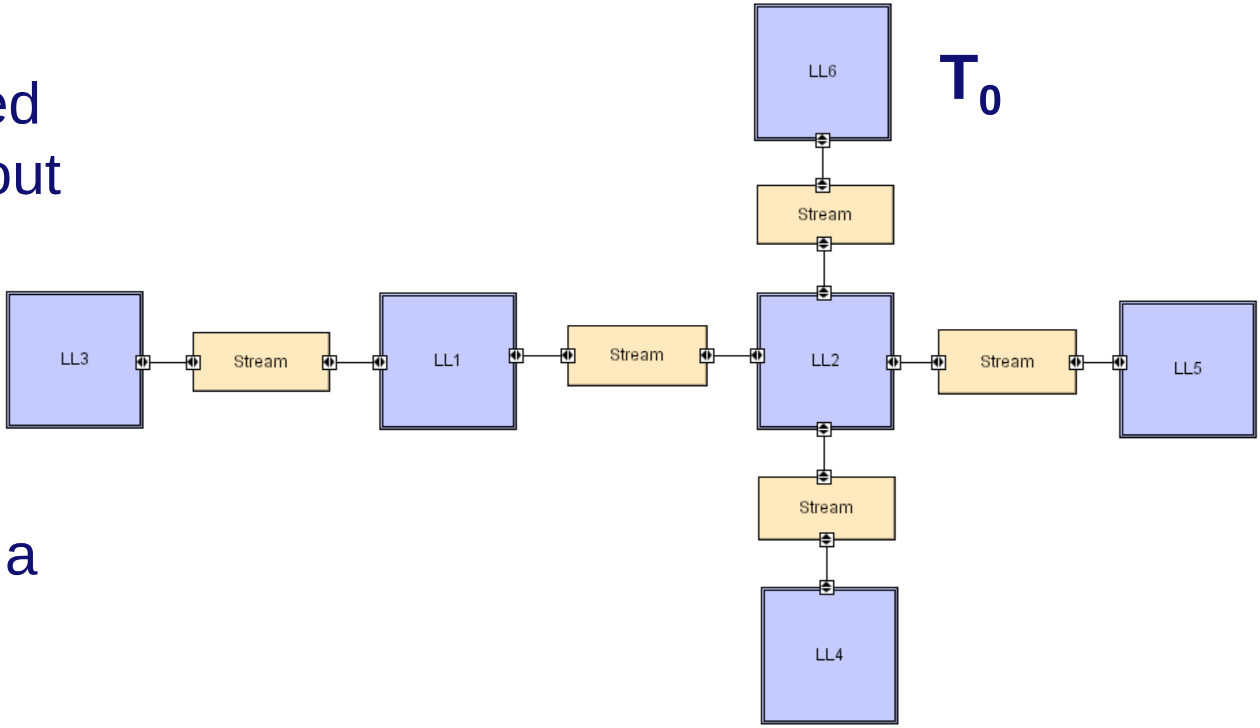
*Peer-to-Peer*

# Peer-to-Peer Lunar Lander

## Adapted version:

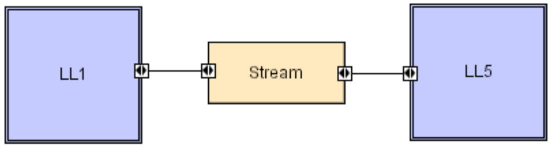
- multiple landers need to communicate about the landing area to avoid collisions

- communication is possible only within a certain range.



$T_0$

$T_n$



*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

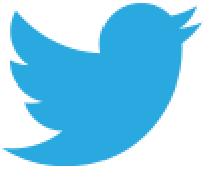
# Peer-to-Peer: Style Analysis

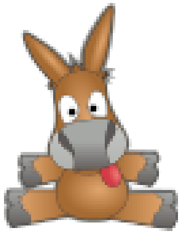
- **Summary**:
  - State and behavior are distributed among peers which can act as either clients or servers.
- **Design elements**
  - Components: peers
  - Connectors: network protocols, often custom
  - Data: network messages
- **Topology**:
  - Network, usually dynamically and arbitrarily varying

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# Publish-Subscribe Style: Style Analysis

- What are common **examples** of its use?
  - sources of information are distributed
  - network is ad-hoc





*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# Peer-to-Peer Style: Style Analysis

- What are the **advantages** of using the style?
  - Robustness (if a node is not available the functionality is taken over)
  - Scalability
  - Decentralization

- What are the **disadvantages** of using the style?
  - Security (peers might be malicious or egoistic)
  - Latency (when information retrieval time is crucial)

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# Heterogeneous Styles

- More complex styles created through composition of simpler styles
  - **REST**
  - **C2**
    - Implicit invocation + Layering + other constraints
  - **Distributed objects**
    - OO + client-server network style
    - CORBA
    - **2II45 Architecture of Distributed Systems**

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# Style Summary (1/4)

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

*Software Architecture: Foundations, Theory, and Practice*; Richard N. Taylor, Nenad Medvidovic, and Eric M. Dashofy; © 2008 John Wiley & Sons, Inc. Reprinted with permission.

# Style Summary, continued (2/4)

| | | | |
|---|---|---|---|
| ***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.<br>Random-access to data 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. | Interaction between components required. Exchange of complex data structures between components required. |
| ***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;<br>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. |

# Style Summary, continued (3/4)

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

# Style Summary, continued (4/4)

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

# Summary

- Different styles result in
  - Different architectures
  - Architectures with greatly differing properties

- A style does not fully determine resulting architecture
  - A single style can result in different architectures
  - Considerable room for
    - Individual judgment
    - Variations among architects

- A style defines domain of discourse
  - About problem (domain)
  - About resulting system