Chapter 6 – Architectural Design

Dr. Michael F. Siok, PE, ESEP
UT Arlington
Computer Science and Engineering

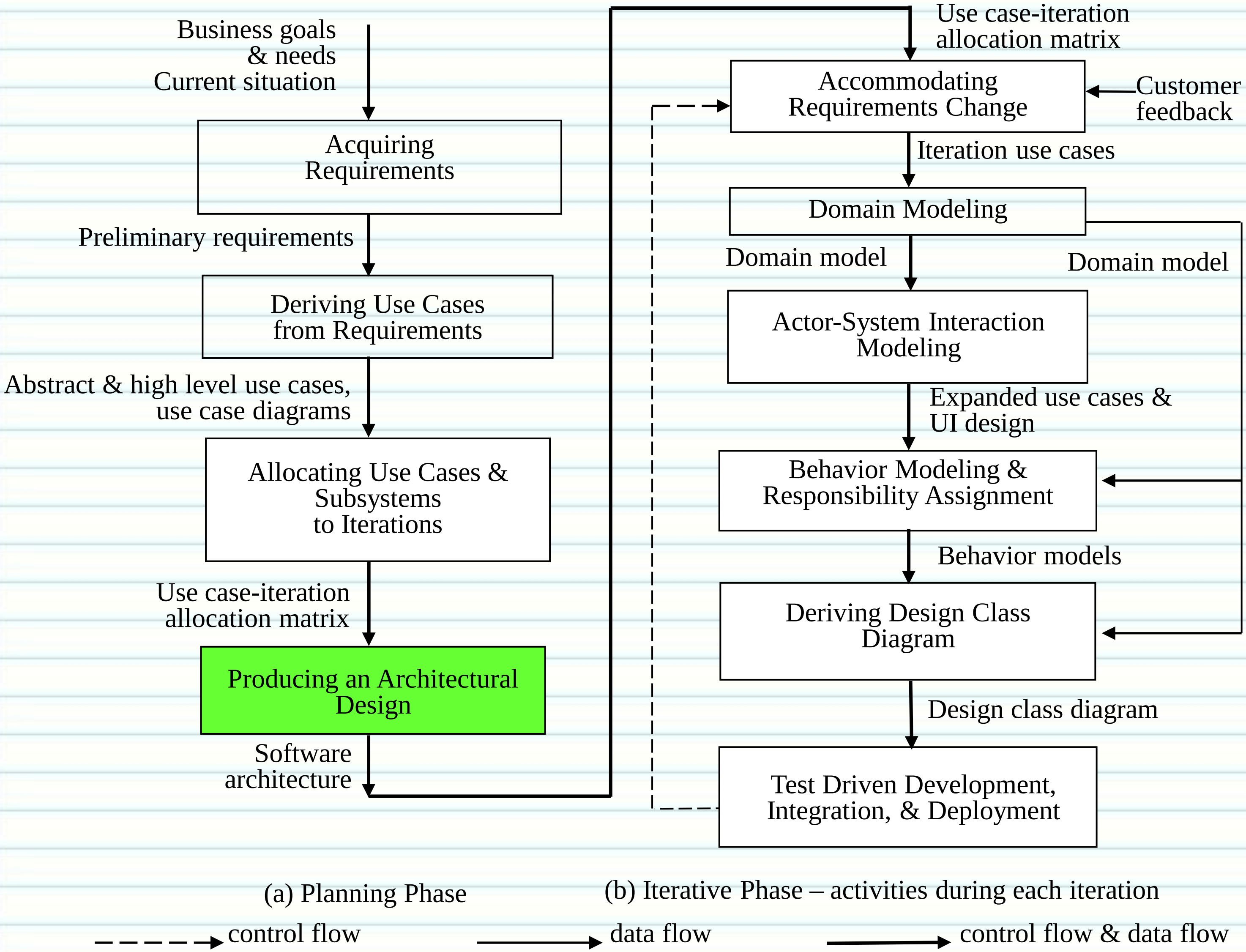
Key Takeaway Points

- The software architecture of a system or subsystem refers to the style of structural design including the interfacing and interaction among its subsystems and components
  - *Components, Connectors, and Constraints (the 3 C’s of architecture)*

- Different types of systems require different design methods and architectural styles
  - Affects system properties of utility, efficiency, security, safety, and others

- Architecture plays a central role for the entire life of the software system

Architectural Design in Methodology Context



What Is Software Architectural Design

- The *software architecture* of a system or subsystem refers to the style of design of the system structure including the interfacing and interaction among its major components
- The Software Architectural Design activity is a decision-making process to determine the software architecture and its representation for the system under development
- According to IEEE 1471-2000, Recommended Practice for Architectural Description of Software Intensive Systems . . .
  - Software architecture is the fundamental organization of a system, embodied in:
    - The ***components*** of the system
    - Relationships (***connections***) among the components
    - Relationships (***connections***) between the components and the environment
    - Principles (***constraints***) governing the design and evolution of the system

The 3 C's of Architecture – Components, Connections, and Constraints

Importance of Architectural Design

- The architecture of a software system has significant impact on performance, efficiency, security, and maintainability

- The Architectural Representation is the primary artifact for conceptualization, constructing, managing, and evolving the system under development
  - And later, the system under maintenance

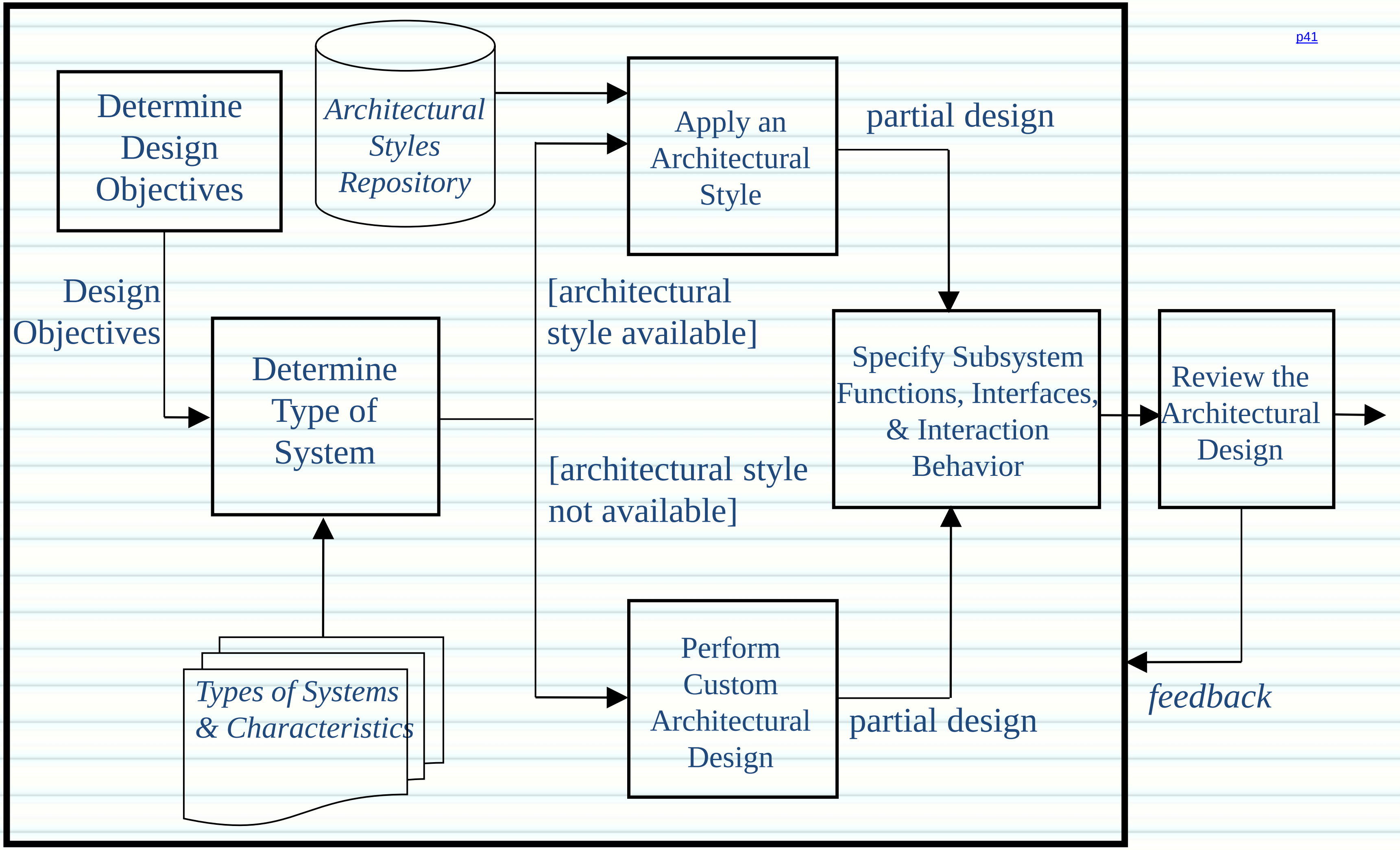
- Architectural design flaws (or even specific 'choices made') could result in project failure
  - Major cost and schedule over-runs
  - Lost functionality
  - Project being abandoned

Guidelines for Architectural Design

1. Adapt an architectural style when possible
2. Apply software design principles
3. Apply design patterns
4. Check against design objectives and design principles
5. Iterate the steps, if needed

Architectural Design Process

p41



Determine Architectural Design Considerations

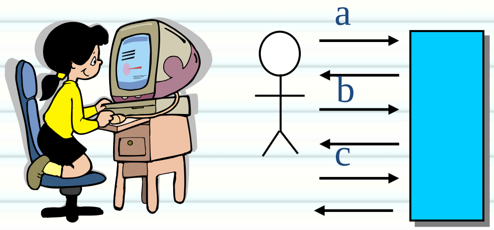
- Ease of change and maintenance
- Use of commercial off-the-shelf (COTS) parts
- System performance
  - e.g., does the system require processing of real-time data or a huge volume of transactions?
- Reliability
- Security
- Software fault tolerance
- Recovery
- . . .

Revisit non-functional requirements and constraints

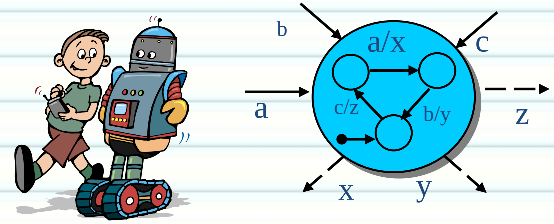
- Depends largely on product application and type of system

Determine System Type

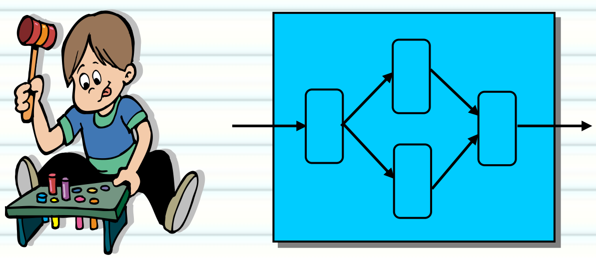
-- Four Common Types of Systems and Behaviors --



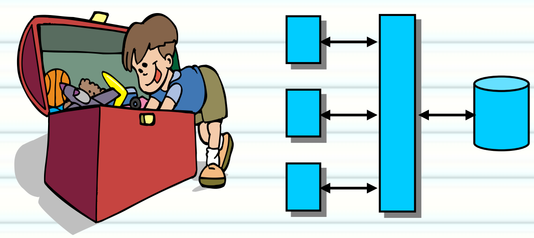
(a) Interactive subsystem



(b) Event-driven subsystem



(c) Transformational subsystem



(d) Database subsystem

Characteristics of Interactive Systems

- The interaction between system and actor consists of a relatively fixed sequence of actor requests and system responses
  - The system has to process and respond to each request
- Often, the system interacts with only one actor during the process of a use case
  - The actor is often a human being although it can also be a device or another subsystem
- The interaction begins and ends with the actor
- The actor and the system exhibit a “client-server” relationship.
- System state reflects the progress of the business process represented by the use case

Characteristics of Event-Driven Systems

- An Event-Driven System receives events from and controls external entities
  - It does not have a fixed sequence of incoming requests
  - Requests arrive at the system randomly
  - It does not need to respond to every incoming event
- Its response is state dependent
  - The same event may result in different responses depending on system state
- It interacts with more than one external entity at the same time
- External entities are often hardware devices or software components rather than human beings
- Its state may not reflect the progress of a computation
- It may need to meet timing constraints, temporal constraints, and timed temporal constraints

Characteristics of Transformational Systems

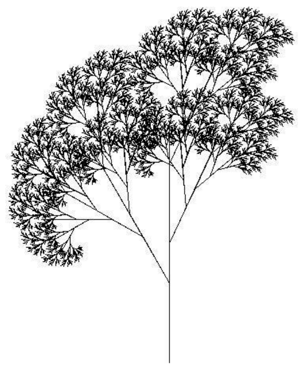
- Transformational systems consist of a network of information-processing activities
  - Transforming activity input to activity output
- Activities may involve control flows that exhibit sequencing, conditional branching, parallel threads, and synchronous and asynchronous behavior
- During the transformation of the input into the output, there is little or no interaction between system and actor
  - It is a batch process
- Transformational systems are usually stateless
- Transformational systems may perform number crunching or computation-intensive algorithm
- The actors can be human beings, devices, or other systems

Characteristics of Object-Persistence Systems

- An Object Persistent System provides capabilities for storing and retrieving objects from a database or file system while hiding the storage media
  - It provides object storage and retrieval capabilities to other subsystems
- It hides the implementation from the rest of the system
- It is responsible only for storing and retrieving objects and does little or no business processing except for performance considerations
- It is capable of efficient storage, retrieval, and updating of a huge amount of structured and complex data

Recursive View of Systems and Subsystems

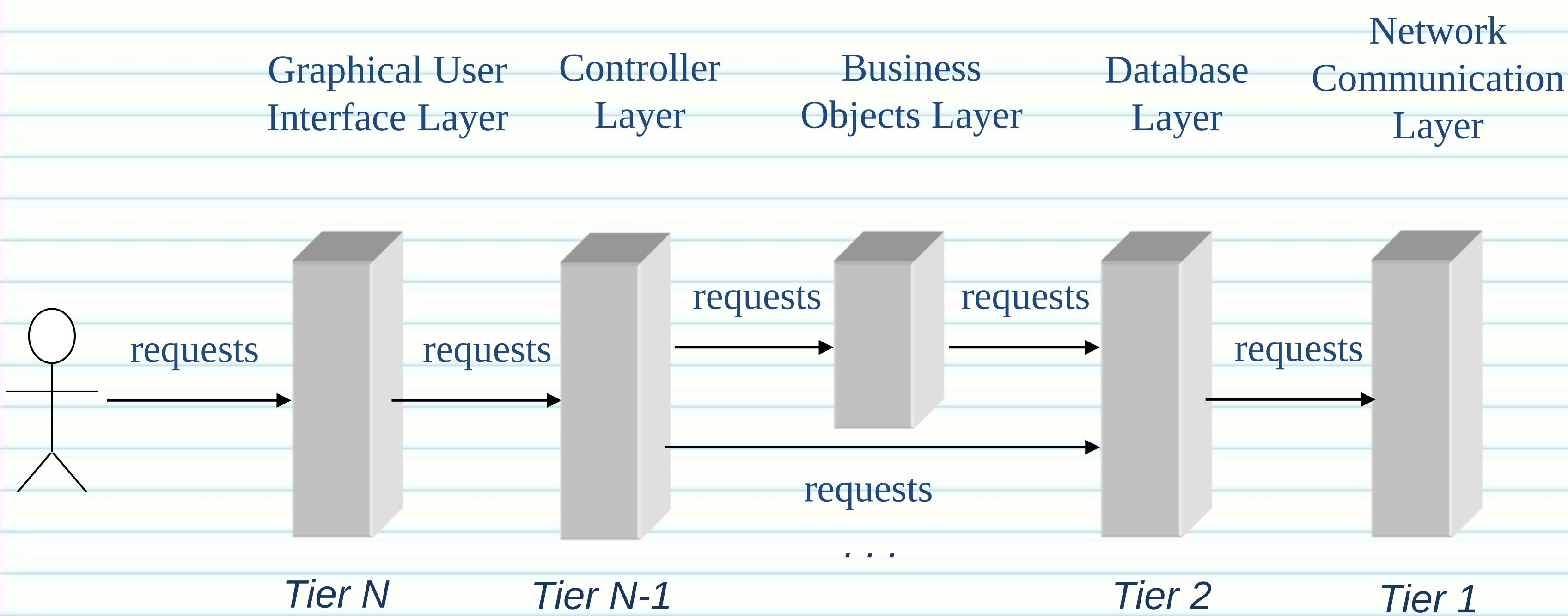
- It is worth mentioning again here . . .
- Systems and subsystems are relative to each other
  - A system is a subsystem of a larger system
  - A subsystem is a system and may consist of other subsystems
- A system may consist of different types of subsystems
- The design method applied needs to match the type of subsystem under development



System Types and Architectural Styles

Type of System	Architectural Style
Interactive System	N-Tier Style
Event-Driven System	Event-Driven Style
Transformational System	Main Program and Subroutines Style
Object-Persistence Subsystem	Persistence Framework Style
Client-server System	Client-server Style
Distributed, decentralized System	Peer-to-peer Style
Heuristic problem-solving System	Blackboard Style

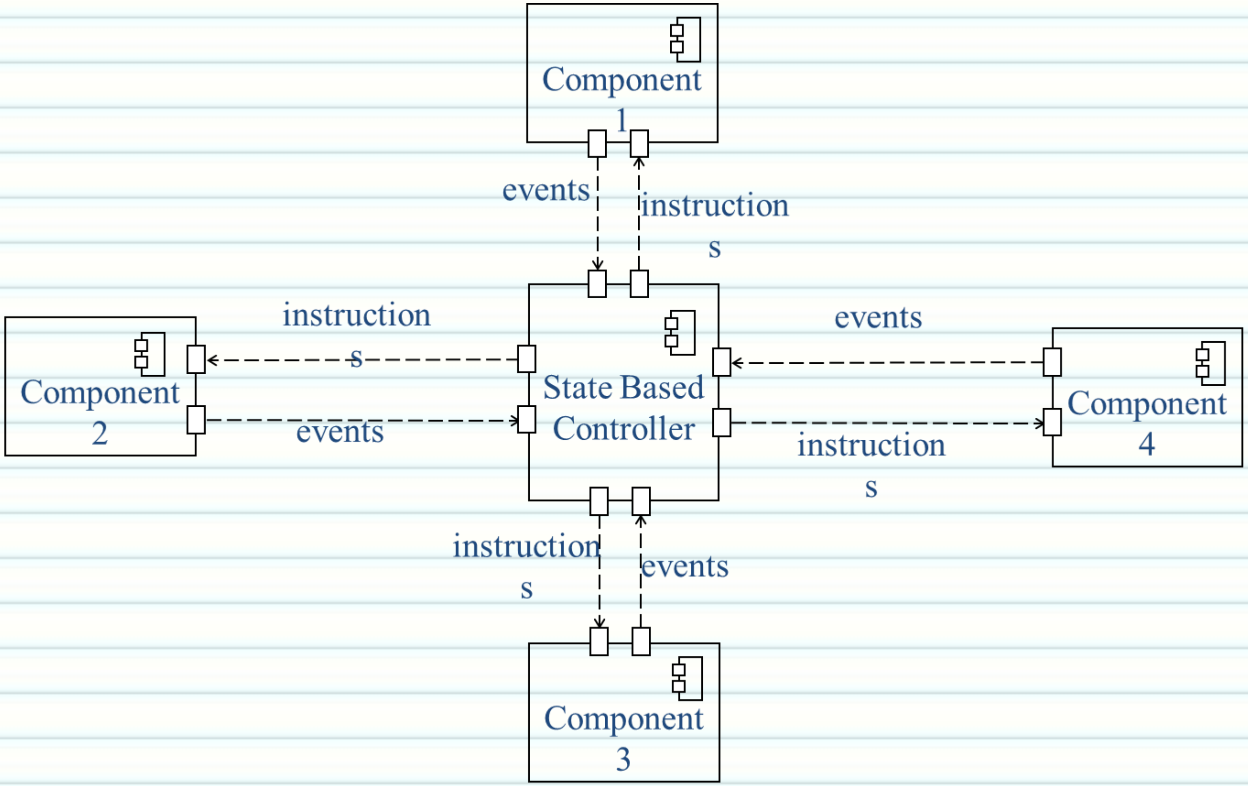
N-Tier Architecture Style



- System is structured into a number of leveled layers or tiers
- Interacts with only 1 actor at a time
- Events arrive in predetermined sequence
- System must respond to each event

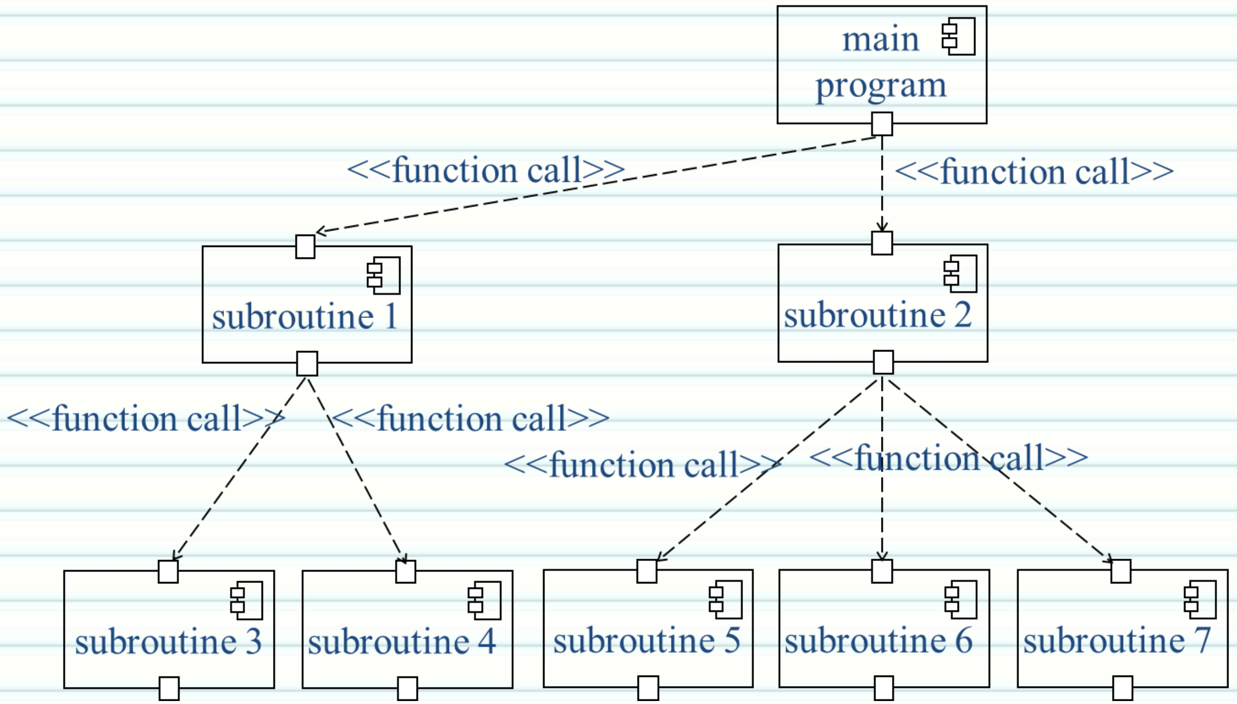
Design Principles
-- Separation of concerns
-- High cohesion
-- Stupid objects
-- Information hiding
-- Design for change

Event-Driven Architecture Style



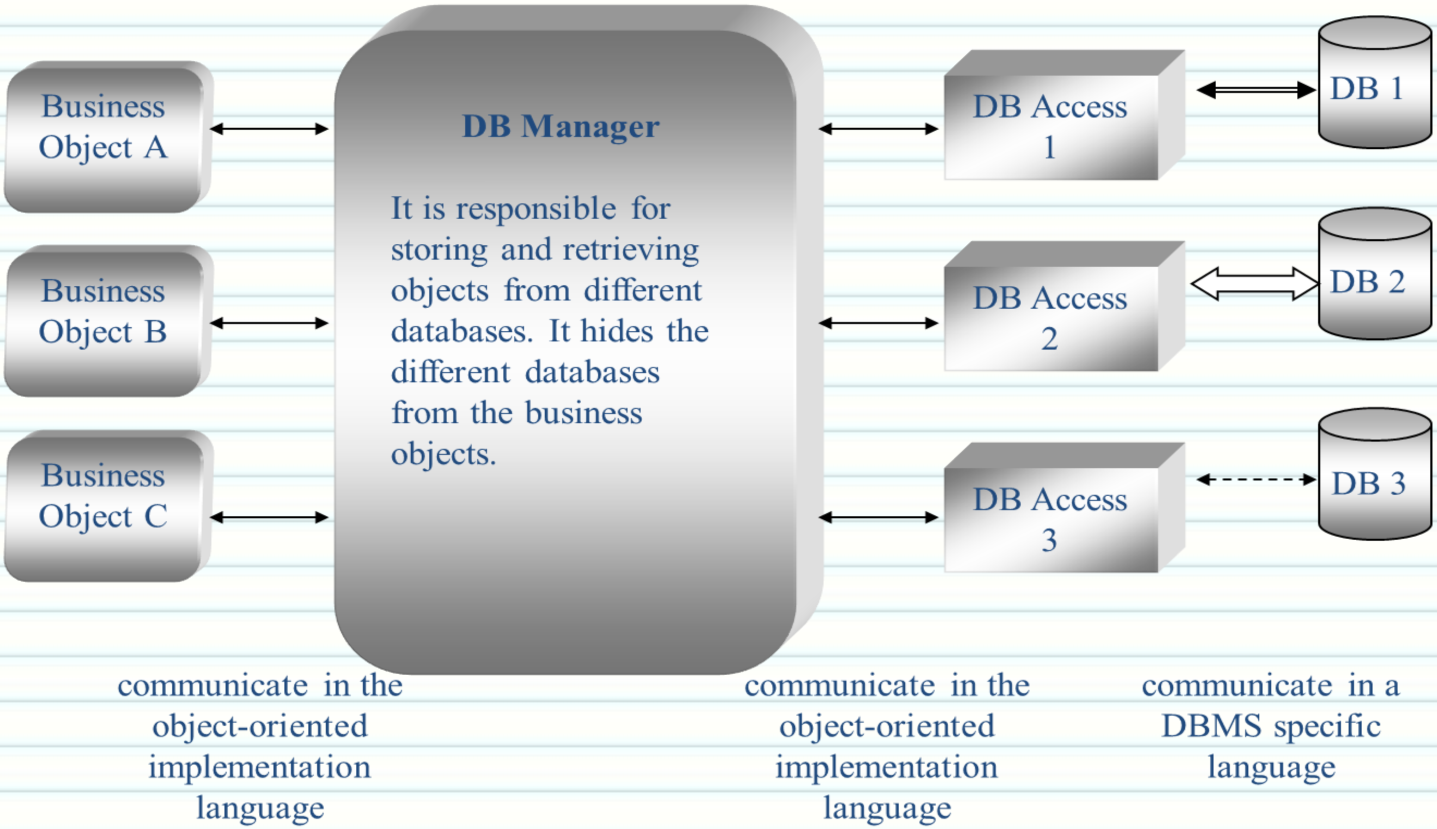
- Consists of a state-based controller and components under its control
- Exhibits state-dependent behavior implemented by the controller
- Components send events to controller
- Controller processes events and sends instructions to components

Main Program and Subroutine Architecture Style



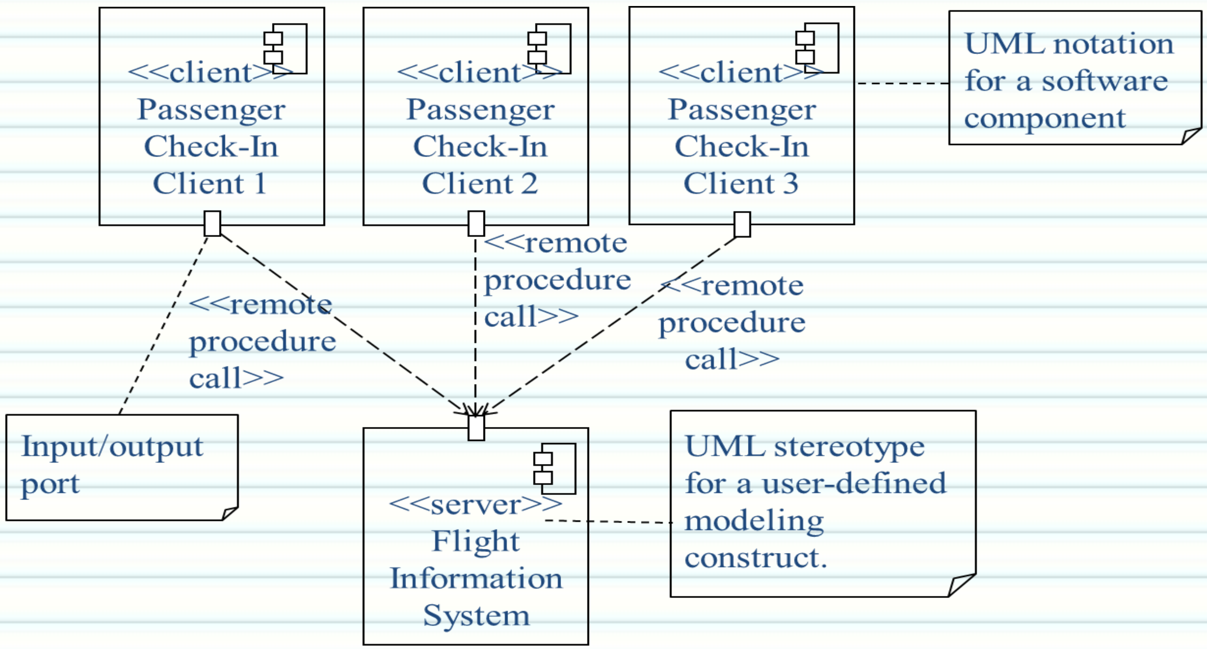
- Consists of a main program and a number of subprograms/subroutines
- Subroutines may call additional lower level subroutines
- Often used as architecture for transformational systems
- Information processing consisting of network of processes
  - Transforms input to a different output

Object-Persistence Framework Style



- Hides databases/file systems by decoupling them from objects that use them
- Objects are '*unaware*' of the specific storage devices used
  - Changes to databases or file systems then have no impact on the objects that use them

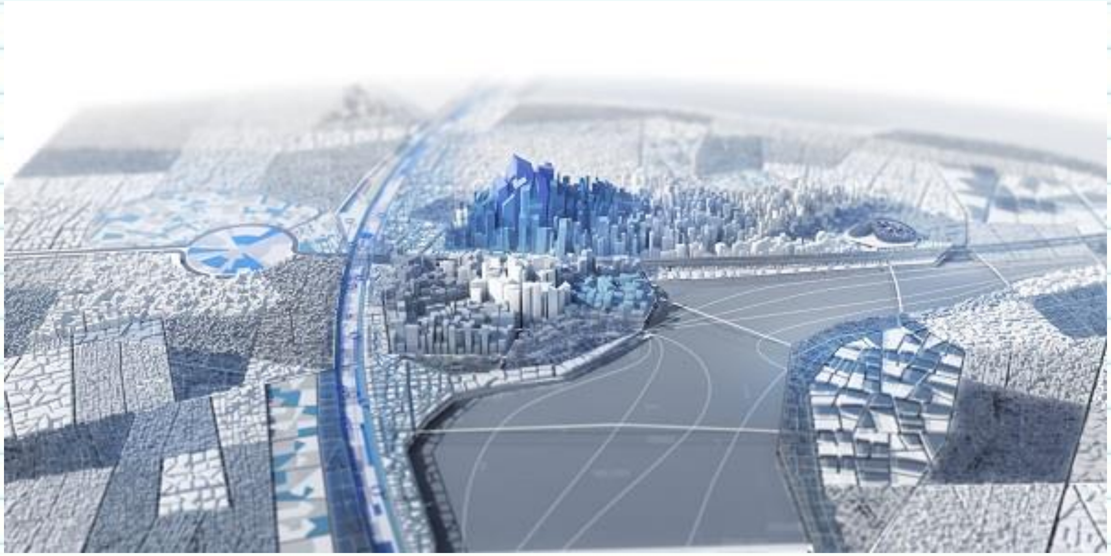
Client-Server Architecture Style



- Consists of a server and some number of clients
- Clients send requests to the server
- Server processes requests and sends results back to the clients
- Clients can be added or deleted from the network without impacting work of the server
  - Clients can be lightweight computers if most services are provided by server

Some More Styles

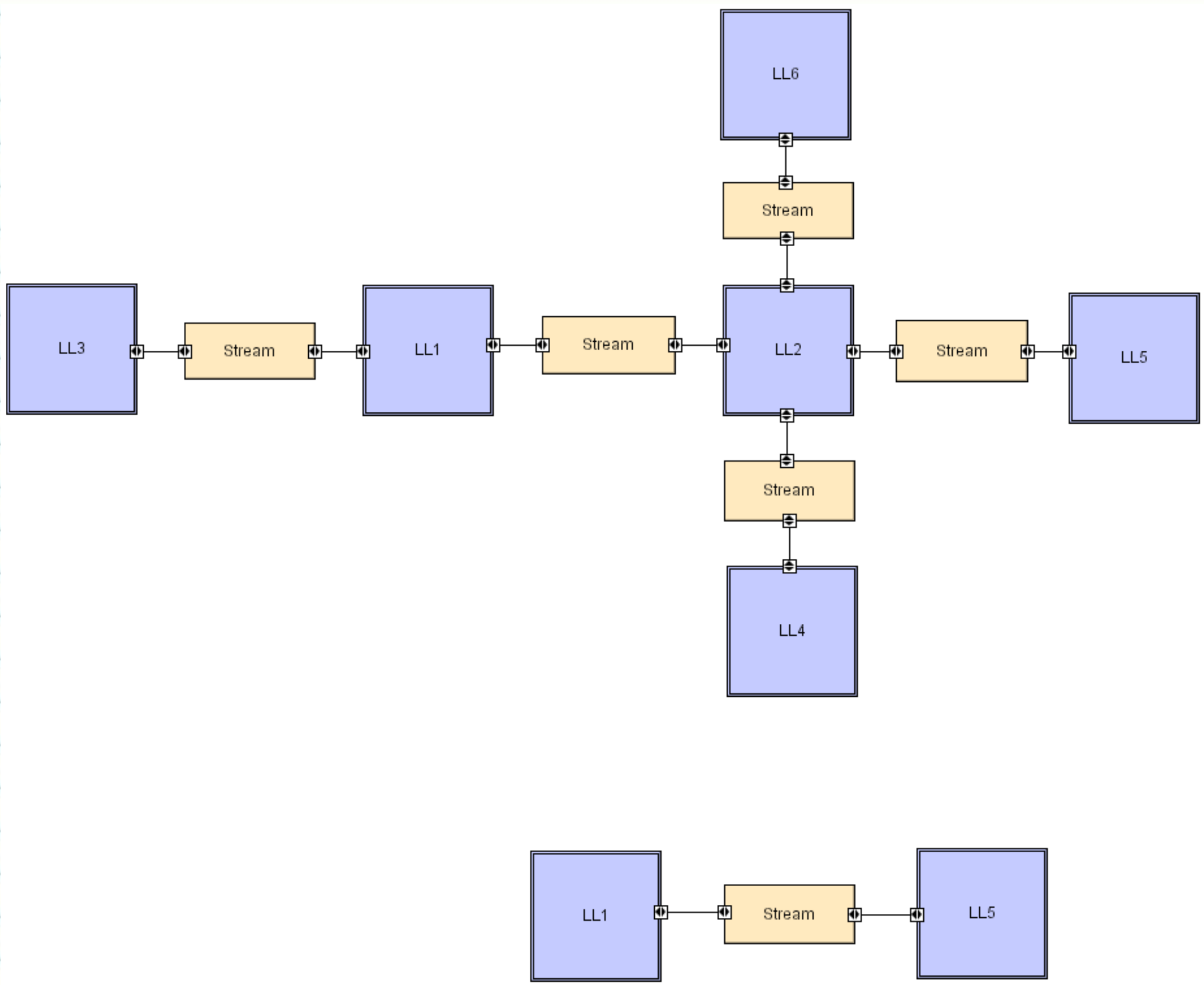
- There are lots of architectural styles
- Note-worthy architectural styles to mention:
  - Peer–to–peer Style
  - Blackboard Style
  - Pipe–and—filter Style
  - Data Flow Style (Batch-Sequential)
  - Rule-Based Style
  - Interpreter Style
  - Mobile Style
  - Publish-Subscribe



Peer-to-Peer (P2P) Style

- Consists of independent component connected through network protocols
  - Used in distributed or decentralized computing systems
  - Highly scalable, robust
- State and behavior are distributed among peers which can act as either clients or servers
  - **Peers:** independent components, having their own state and control thread
  - **Connectors:** Network protocols, often custom
  - **Data Elements:** Network messages
  - **Topology:** Network (may have redundant connections between peers); can vary arbitrarily and dynamically
- Supports decentralized computing with flow of control and resources distributed among peers
- Highly robust in the face of failure of any given node
- Scalable in terms of access to resources and computing power. But caution on the protocol!

Peer-to-Peer Style



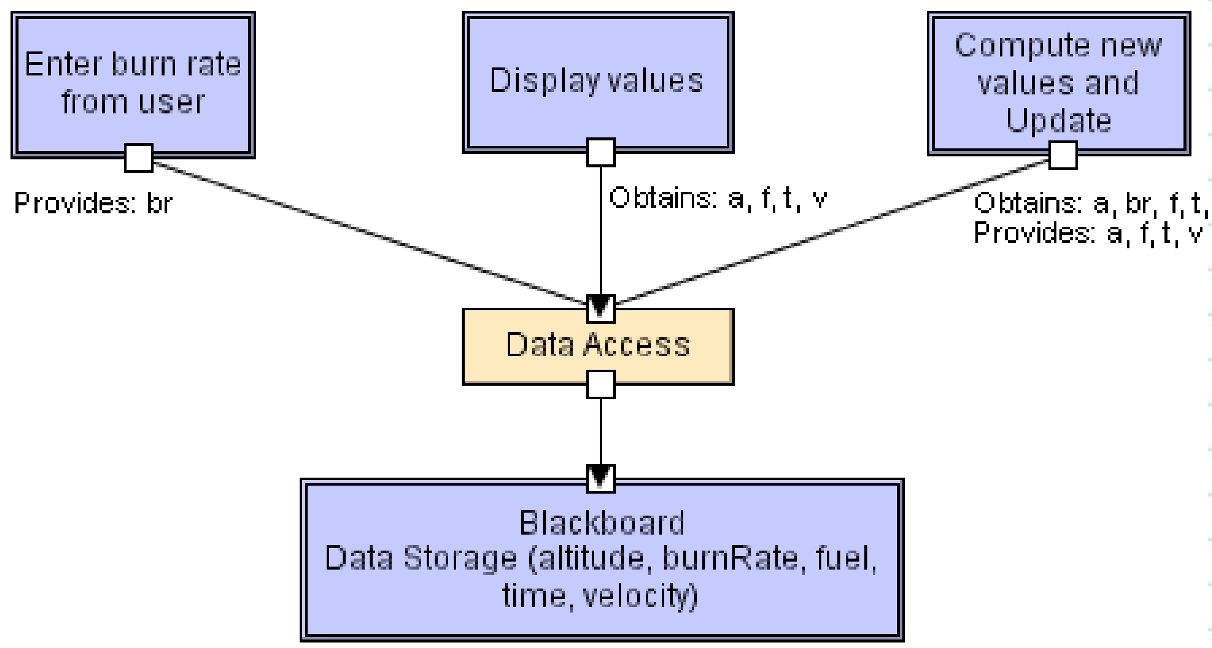
Blackboard Style

- Two kinds of components:
  - Central data structure (data repository) — called "*blackboard*"
  - Independent components (programs) operate on the blackboard

- System control is entirely driven by the blackboard state

- Examples
  - Typically used for AI systems
  - Integrated software environments (e.g., Interlisp)
  - Compiler architecture

Blackboard LL



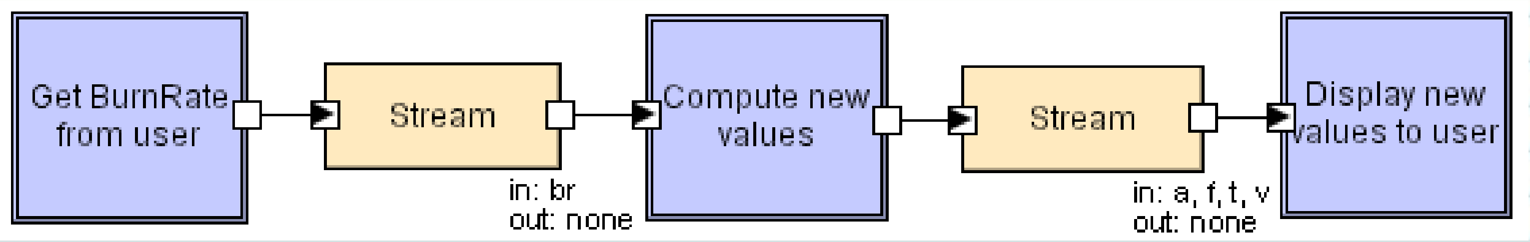
Pipe and Filter Style

- Components are filters (programs)
  - Transform input data streams into output data streams using connectors (pipes)
  - Also known as pipeline architecture style
- Filters are independent (no shared state)
  - Filters have no knowledge of up- or down-stream filters
- Variations
  - Pipelines — linear sequences of filters
  - Bounded pipes — limited amount of data on a pipe
  - Typed pipes — data strongly typed
- Advantages
  - System behavior is a succession of component behaviors
  - Filter addition, replacement, and reuse
    - Possible to hook any two filters together
  - Concurrent execution
- Disadvantages
  - Batch organization of processing
  - Interactive applications

- Examples
  - UNIX shell (signal processing)
  - Distributed systems (parallel programming)

- Example: `ls invoices | grep -e August | sort`

Pipe and Filter LL



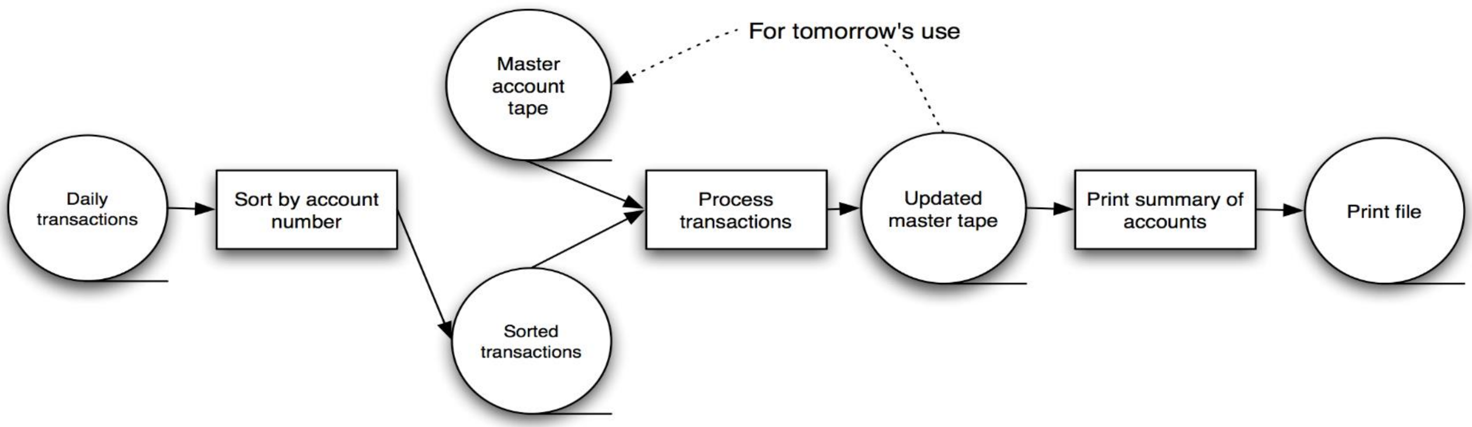
Data-Flow Styles

- Batch Sequential Style
  - Separate programs are executed in order
    - Data is passed as an aggregate from one program to the next
  - Connectors: “The human hand” carrying tapes between the programs, a.k.a. *“sneaker-net”*
  - Data Elements: Explicit, aggregate elements passed from one component to the next upon completion of the producing program’s execution

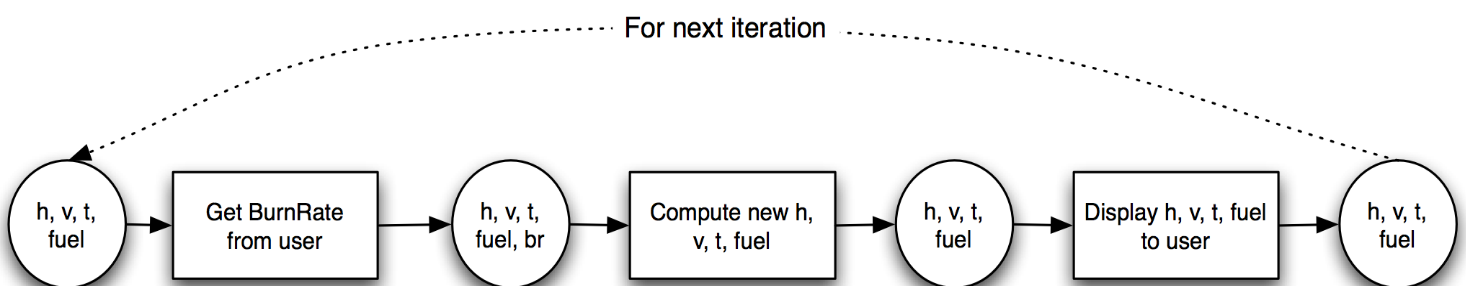
- Typical uses
  - Transaction processing in financial systems
    - *“The Granddaddy of Styles”*

Batch-Sequential Style

A Financial Application



Rocket Fuel Burn Example



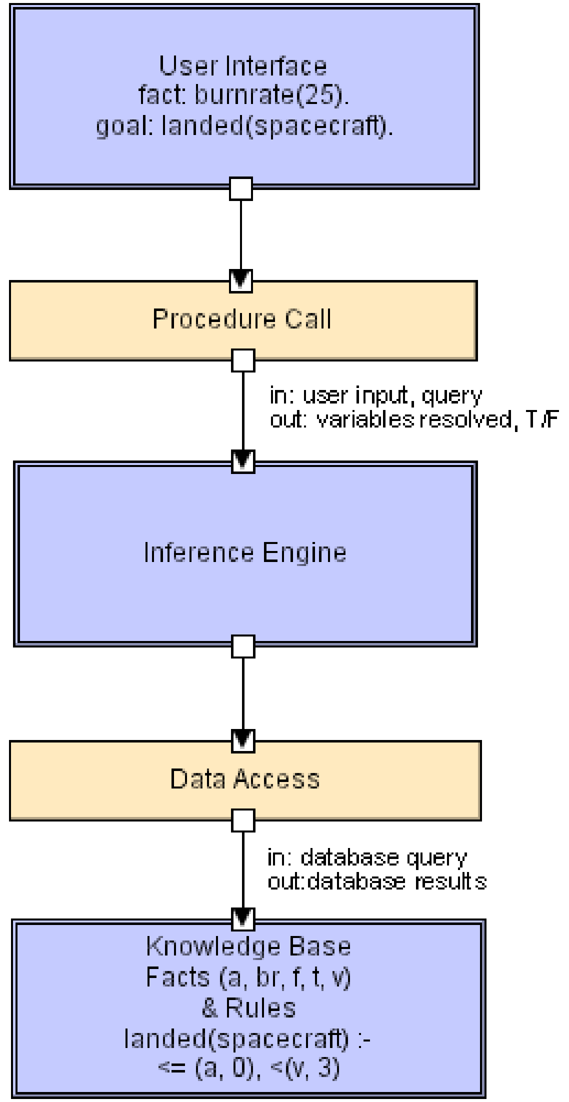
Rule-Based Style

- 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

- **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 Style

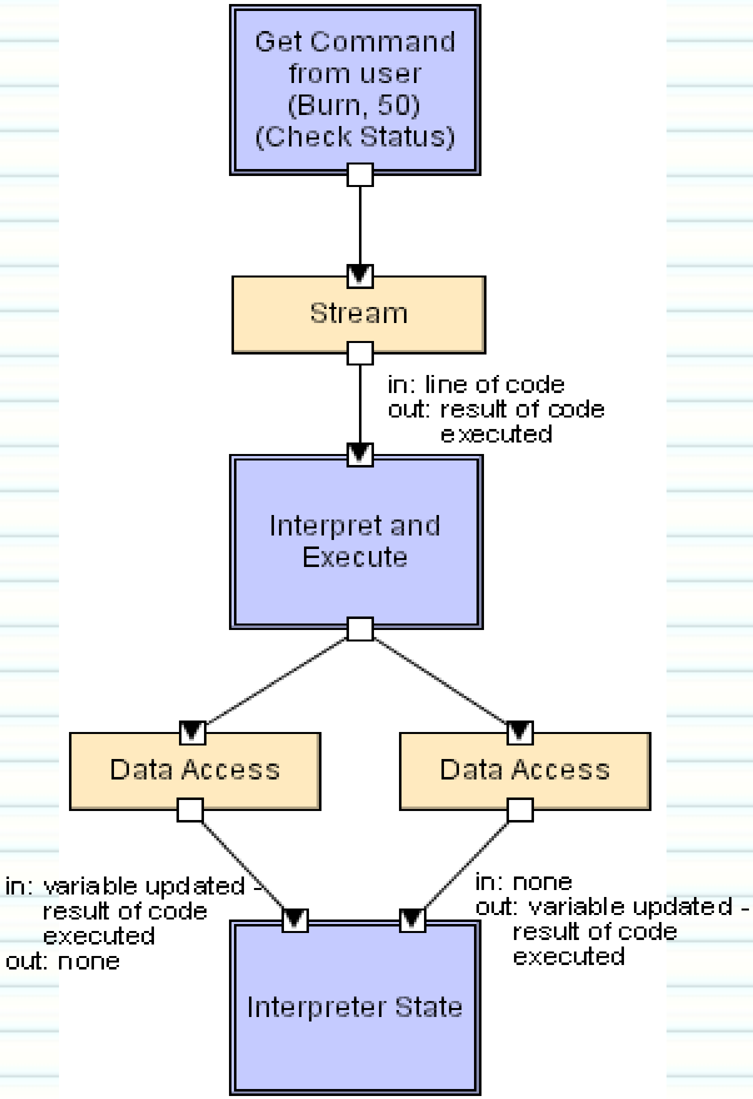


Interpreter Style

- Interpreter parses and executes input commands, updating the state maintained by the interpreter
  - **Components**: Command interpreter, program/interpreter state, user interface
  - **Connectors**: Typically very closely bound with direct procedure calls and shared state

  - 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
  - Superb for end-user programmability
    - Supports dynamically changing set of capabilities
    - e.g., Lisp and Scheme

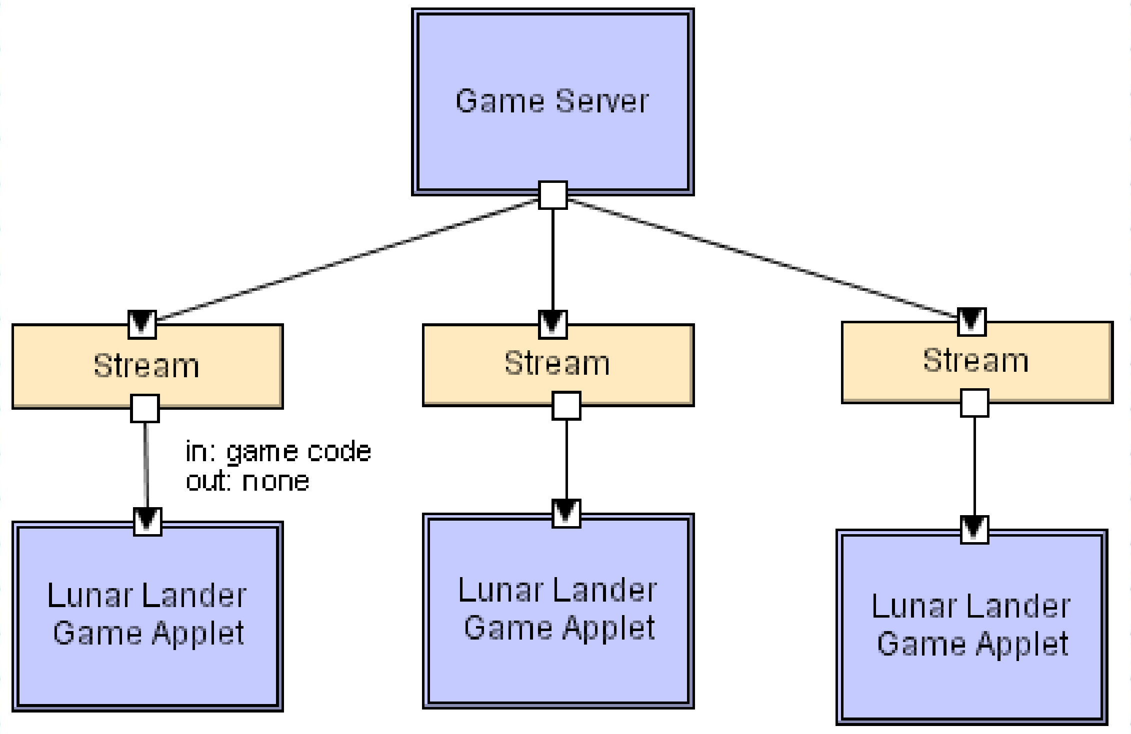
Interpreter Style



Mobile-Code Style

- Mobile-Code Style -- a data element (some representation of a program) is dynamically transformed into a data processing component
  - **Components**: “Execution dock”, which handles receipt of code and state; code compiler/interpreter
  - **Connectors**: Network protocols and elements for packaging code and data for transmission
  - **Data Elements:** Representations of code as data; program state; data
  - **Variants**: Code-on-demand, remote evaluation, and mobile agent.

Mobile-Code Style

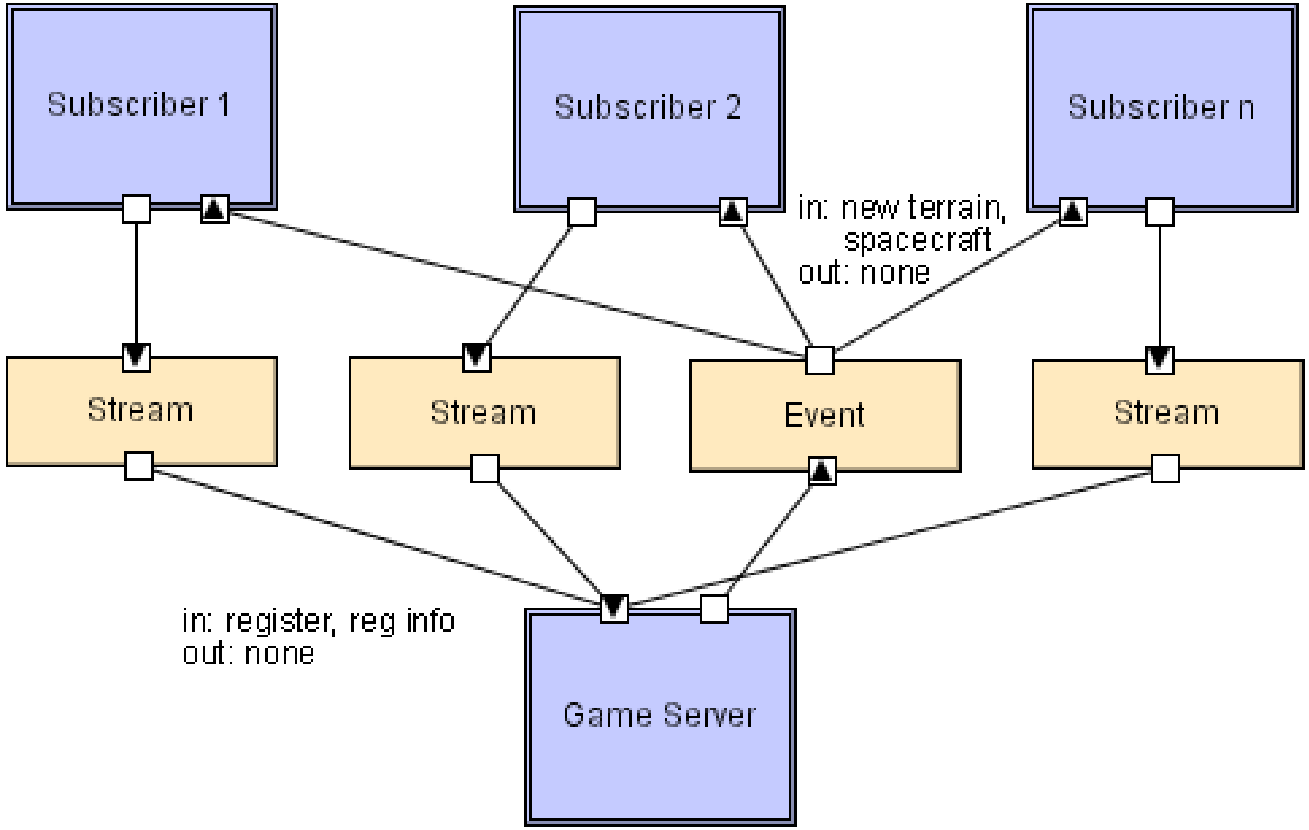


Scripting languages (i.e. JavaScript, VBScript), ActiveX control, embedded Word/Excel macros

Publish-Subscribe

- Subscribers register/deregister to receive specific messages or specific content
- Publishers broadcast messages to subscribers either synchronously or asynchronously
  - **Components:** Publishers, subscribers, proxies for managing distribution
  - **Connectors**: Typically a network protocol is required. Content-based subscription requires sophisticated connectors.
  - **Data Elements:** Subscriptions, notifications, published information
  - **Topology:** Subscribers connect to publishers either directly or may receive notifications via a network protocol from intermediaries
  - **Qualities yielded**: Highly efficient one-way dissemination of information with very low-coupling of components

Publish-Subscribe Style



Perform Custom Architectural Design

- Reusing an architectural style is usually preferred
  - Saves time
  - Ensures a level of quality
- Not all application systems development projects can reuse an existing architectural style
  - Custom architectural design may be required to meet the needs of an application system
- Design patterns are useful for custom architectural design
  - Known solutions to common design problems
  - Covered in a later module

Specify Subsystem Functions and Interfaces

- Once the architecture is developed for the system, requirements are typically allocated to subsystems

- Architectural design continues with:
  - Allocate requirements and architectural design objectives to subsystems and components
  - Specify the functionality of each subsystem and component
  - Specify the interfaces of the subsystems and components
  - Specify the interaction behavior of the subsystems and components

The 3 C's of Architecture – Components, Connections, and Constraints

Review the Architectural Design

- Architectural design is an iterative process
- Review the design to ensure that requirements are allocated correctly and architectural design objectives are met
  - With Architectural Design Team
  - With other stakeholders
- Review to ensure that the architectural design satisfies software design principles
- Review to ensure that the architectural design satisfies security requirements, constraints, and secure software design principles (i.e., the non-functional requirements and constraints)

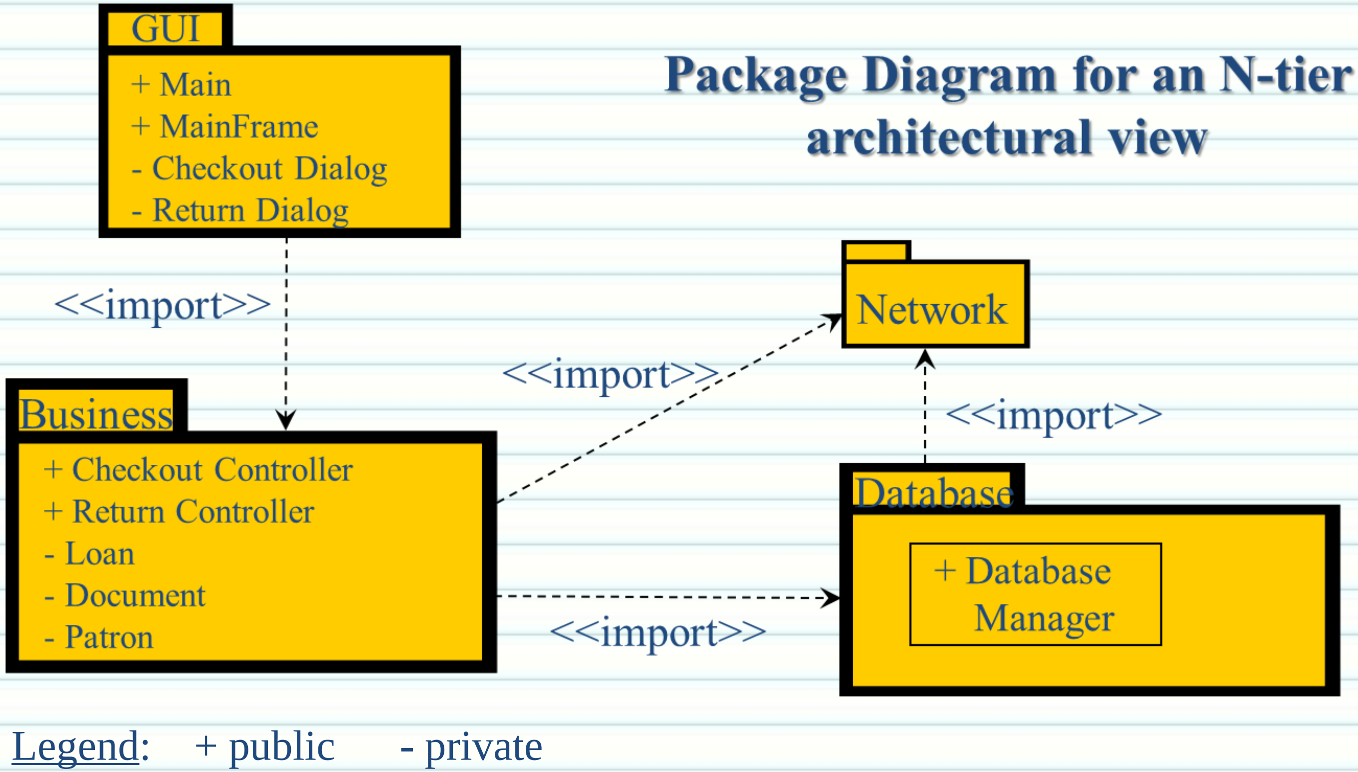
Architectural Style and Package Diagram

- Software Architecture defines structure of the software system, subsystems and relationships
  - Components
  - Connections
  - Constraints

- Software architecture provides for:
  - **Conceptualization** – helps design team think of the software in terms of its overall structure – provides a working system model for the team
  - **Construction** – helps team organize the software development work providing a reference model for team for the development builds
  - **Managing** -- helps team organize and keep organized the software development and software process artifacts (i.e., classes, use cases, data, etc.) during development and construction (see package diagram as example)
  - **Evolving** – establishes a 'system model' from which to evolve and expand the developed system

- The package diagram gives us a way to organize the software artifacts of the development process
  - Provides a way to realize the benefits of 'architecture'

Architectural Style and Package Diagram



Package Diagram for an N-tier architectural view

Package diagram defines a logical organization of artifacts and packages and supports CM of classes and other process artifacts

Applying Software Design Principles

- Design principles are widely accepted rules for software design; correctly applying these rules leads to better quality designs
  - **Design for Change** – design with a “built-in mechanism” to adapt to or facilitate anticipated changes (*see section 6.5.2 in textbook*)
  - **Separation of Concerns** – focus on one aspect of the problem in isolation rather than tackling all aspects at the same time
  - **Information Hiding** – shield implementation detail of a module to reduce its change impact to other parts of the software system
  - **High Cohesion** – from modular design in SA/SD, achieve a higher degree of relevance of the module’s functions to the module’s core functionality
  - **Low Coupling** – from SA/SD, reduce the run-time effect and change impact of a subsystem to other subsystems
  - **Keep It Simple and Stupid** – design simple and “stupid objects”

DP1 – Design for Change

- Change in a software system is ‘a way of life’ – *Change Happens!*
- Numerous events could cause software to change
  - Changes to requirements
  - Changes to fix problems/bugs
  - Changes due to hardware, tech, or OS changes
  - Changes in government policies, regulations, operating procedures
  - Changes to improve performance, reliability, security, etc.
  - Changes to the project schedule(s) or budget(s)
- Design for change means that the architectural design of the software should consider likely anticipated changes and include mechanisms to accommodate those changes
  - Textbook example, LDAP database exclusive use instead of choice of DBMS

DP 2 -- Separation of Concerns

- A problem solving technique
  - Focus on the problem or issue in isolation (by itself, at least temporarily), rather than tackling all aspects of the problem simultaneously
- This DP slows up repeatedly in our software practice
  - *UML* – each drawing type handles on aspect of the application or system
  - *Software design* – usually separated into high-level design and low-level design activities
    - High level – concerned with how to handle the overall design process and artifacts
    - Low level – concerned with designing individual components
  - *Our class process* – activities are separated in sequence to handle aspects of the overall software OO design process
  - *Design of software components* – each component should focus on one aspect of the subject-matter at hand (e.g., GUI handler, DBMS, other business objects)
- For architecture, separation of concerns DP is about making sure that different responsibilities of the system are assigned to different subsystems

DP 3 -- Information Hiding

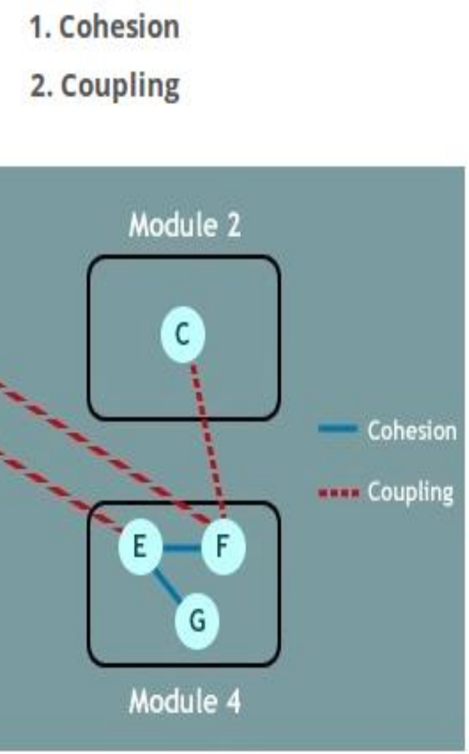
- Shields module implementation details from other modules in the system
  - To reduce change impacts to the rest of the system
- This DP also occurs frequently in software design
  - DBMS design approach hides access implementation details from other parts of the system
    - Don't need to know which DB is used to store and retrieve data
    - Allows developers to be flexible which DBMS choices (RE: Bridge Pattern)
  - Applying an algorithm to elements of data structure
    - Apply this algorithm regardless of the data structure used (RE: Iterator Pattern)
- For architecture, Information Hiding DP is about designing the software system to shield implementation details of parts of the system from the rest of the software system

DP 4 -- High Cohesion

- This DP came from the earlier Structured Analysis/Structured Design (SA/SD) approach
  - Software system is decomposed into hierarchy of modules
    - Higher-level modules call lower-level modules
    - Each module implements a function or set of functions
    - Cohesion =: degree of relevance of function(s) to the module's core functionality
- High Cohesion => better module understanding, easier reuse, and easier software maintenance
- For architecture, High Cohesion DP is about ensuring that components of the architecture have a high degree of relevance to the core functionality of that component

DP 5 -- Low Coupling

- Also came from SA/SD approach
- Coupling
  - Measures change impact effect of one module to all other modules
  - Measures the degree of run-time behavior impact of one module on other modules in the system (due to dependencies and interactions)
- High coupling => higher uncertainty of runtime effects; higher change impact to other modules
- Low Coupling => reduces runtime effects uncertainty, reduces change impacts to other modules, and facilitates program understanding, testing, reuse, and maintenance



- For architecture, Low Coupling DP is about reducing the runtime effects and change impacts of each subsystem to the other subsystems

DP 6 -- Keep it Simple and Stupid (KISS)

- The KISS DP favors simple, straight-forward, and easy-to-understand designs
- Stupid object =: simple-minded, dumb enough
  - Simple-minded – doesn’t ask questions when processing a request
  - Dumb enough – it only knows how to do one thing and nothing else
- For Architecture, the KISS DP means designing the architecture to use simple, stupid objects

Guidelines for Architectural Design

1. Adapt an architectural style, when possible
   – Saves time and effort, typically
   – Based on “*type of system*”
2. Apply software design principles
   – Brings quality and features to the product design
3. Apply design patterns (see module 09)
   – Save time and effort
   – Provides known tried solutions to common problems
4. Check architecture against design objectives and design principles
   – Ensures these were not lost or forgotten during design work
5. Iterate the steps
   – And review . . . .

Applying Agile Principles

1. *Value working software over comprehensive documentation*

2. *Apply the 20/80 rule - that is, "good enough is enough"*

Summary

- Presented the architectural design process
  - Determine Architectural Design
  - Determine System Type
  - Apply Architectural Styles
  - Perform Custom Architectural Design
  - Specify Subsystem functions and Interfaces
  - Review the Architectural Design
- Four Types of Systems
  - Interactive Systems
  - Event-driven Systems
  - Transformational Systems
  - Persistent Storage Systems
- Design Methods and techniques differ for each system type
- Architectural styles and their advantages/disadvantages were discussed
- Software design Principles key to architecture were presented
  - Guide the entire design process, not just architecture