Architectural Patterns and Styles

Software Architecture
Lecture 4
How Do You Design?

Where do architectures come from?

**Creativity**

1) Fun!
2) Fraught with peril
3) May be unnecessary
4) May yield the best

**Method**

1) Efficient in familiar terrain
2) Not always successful
3) Predictable outcome (+ & - )
4) Quality of methods varies
Identifying a Viable Strategy

- Use fundamental design tools: abstraction and modularity.
  - But how?
- Inspiration, where inspiration is needed. Predictable techniques elsewhere.
  - But where is creativity required?
- Applying own experience or experience of others.
The Tools of “Software Engineering 101”

- Abstraction
  - Abstraction(1): look at details, and abstract “up” to concepts
  - Abstraction(2): choose concepts, then add detailed substructure, and move “down”
    - Example: design of a stack class

- Separation of concerns
Learning Objectives

- Delineate the role of DSSAs and Patterns in Software architecture, and apply common patterns to problems
- Understand the role and benefits of architectural styles
- Understand and apply common styles in your designs
- Construct complex styles from simpler styles
- Understand the challenges around greenfield design
Patterns, Styles, and DSSAs
Architectural Patterns

- An architectural pattern is a set of architectural design decisions that are applicable to a recurring design problem, and parameterized to account for different software development contexts in which that problem appears.
- Architectural patterns are similar to DSSAs but applied “at a lower level” and within a much narrower scope.
State-Logic-Display: Three-Tiered Pattern

- Application Examples
  - Business applications
  - Multi-player games
  - Web-based applications
Model-View-Controller (MVC)

- Objective: Separation between information, presentation and user interaction.
- When a model object value changes, a notification is sent to the view and to the controller. So that the view can update itself and the controller can modify the view if its logic so requires.
- When handling input from the user the windowing system sends the user event to the controller; If a change is required, the controller updates the model object.
Model-View-Controller

Graphical Display

View (Encapsulation of display choices)

User-interface Events

Controller (Encapsulation of interaction semantics)

Model (Encapsulation of information)
Objective: Structuring embedded control applications
The Lunar Lander: A Long-Running Example

- A simple computer game that first appeared in the 1960’s
- Simple concept:
  - You (the pilot) control the descent rate of the Apollo-era Lunar Lander
    - Throttle setting controls descent engine
    - Limited fuel
    - Initial altitude and speed preset
    - If you land with a descent rate of < 5 fps: you win (whether there’s fuel left or not)
  - “Advanced” version: joystick controls attitude & horizontal motion
Sense-Compute-Control LL

- Altimeter
- Gyro
- Fuel level
- Engine Control Switch
- Attitude joystick

Flight Control Computer

Logic:
- loop
- read all sensor values
- compute control outputs
- send controls to all actuators
- end loop

- Attitude Control Thruster 1
- Main Descent Engine Controller
- Cockpit Displays
Learning Objectives

- Delineate the role of DSSAs and Patterns in Software architecture, and apply common patterns to problems

- Understand the role and benefits of architectural styles

- Understand and apply common styles in your designs

- Construct complex styles from simpler styles

- Understand the challenges around greenfield design
Architectural Styles: Definition

- 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
- A primary way of characterizing lessons from experience in software system design
- Reflect less domain specificity than architectural patterns
- Useful in determining everything from subroutine structure to top-level application structure
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
Benefits of Using Styles

- Design reuse
  - Well-understood solutions applied to new problems
- Code reuse
  - 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-specific analyses
  - Enabled by the constrained design space
- Visualizations
  - Style-specific depictions matching engineers’ mental models
Style Analysis Dimensions

- What is the design vocabulary?
  - Component and connector types
- What are the allowable structural patterns?
- What is the underlying computational model?
- What are the essential invariants of the style?
- What are common examples of its use?
- What are the (dis)advantages of using the style?
- What are the style’s specializations?
Learning Objectives

- Delineate the role of DSSAs and Patterns in Software architecture, and apply common patterns to problems
- Understand the role and benefits of architectural styles
- Understand and apply common styles in your designs
- Construct complex styles from simpler styles
- Understand the challenges around greenfield design
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
  - Mobile code
- Implicit invocation
  - Event-based
  - Publish-subscribe
- Peer-to-peer
Main Program and Subroutines LL

Diagram:

- **Lunar Lander Main Program**
  - Procedure Call
    - Get BurnRate from user
      - in: none
      - out: throttle
    - Environment Simulator
      - in: altitude, throttle, fuel, time, velocity
      - out: altitude, fuel, velocity
  - Procedure Call
    - Display values to user
      - in: altitude, fuel, time, velocity
      - out: none
Object-Oriented Style

- Components are objects
  - Data and associated operations
- Connectors are messages and method invocations
- Style invariants
  - Objects are responsible for their internal representation integrity
  - Internal representation is hidden from other objects
- Advantages
  - “Infinite malleability” of object internals
  - System decomposition into sets of interacting agents
- Disadvantages
  - Objects must know identities of servers
  - Side effects in object method invocations
Object-Oriented LL
**OO/LL in UML**

```
GUI
    getBurnRate(): void, float
    displayStatus(a: float, f: float, i: int, v: float)

    creates

Spacocraft
    a: float
    f: float
    i: int
    v: float

    setAttitude(a: float): void
    setFuel(f: float): void
    setTime(i: int): void
    setVelocity(v: float): void
    geta(): float
    getf(): float
    geti(): int
    getv(): float

    uses

Environment Simulation
    calculateStatus(b: int, s: Spacocraft): Spacocraft
```
Layered Style

- Hierarchical system organization
  - “Multi-level client-server”
  - Each layer exposes an interface (API) to be used by above layers
- Each layer acts as a
  - *Server*: service provider to layers “above”
  - *Client*: service consumer of layer(s) “below”
- Connectors are protocols of layer interaction
- Example: operating systems
- *Virtual machine* style results from fully opaque layers
Layered Style (cont’d)

- Advantages
  - Increasing abstraction levels
  - Evolvability
  - Changes in a layer affect at most the adjacent two layers

- Reuse
  - Different implementations of layer are allowed as long as interface is preserved
  - Standardized layer interfaces for libraries and frameworks
Layered Style (cont’d)

- Disadvantages
  - Not universally applicable
  - Performance
- Layers may have to be skipped
  - Determining the correct abstraction level
Layered Systems/Virtual Machines

Layer 1
- Program A

Layer 2
- Program B
- Program C

Layer 3
- Program D
Layered LL
Client-Server Style

- Components are clients and servers
- Servers do not know number or identities of clients
- Clients know server’s identity
- Connectors are RPC-based network interaction protocols
Client-Server LL

CLIENT 1
Get/Display Info
Graphics Processing

CLIENT 2
Get/Display Info
Graphics Processing

CLIENT n
Get/Display Info
Graphics Processing

Procedure Call

SERVER:
Game State
Game Logic
Environment
Simulation

in: burnRate
out: altitude, fuel, time, velocity

Data-Flow Styles

Batch Sequential

- 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.

Batch-Sequential: A Financial Application
Batch-Sequential LL

Not a recipe for a successful lunar mission!
Pipe and Filter Style

- Components are filters
  - Transform input data streams into output data streams
  - Possibly incremental production of output
- Connectors are pipes
  - Conduits for data streams
- Style invariants
  - Filters are independent (no shared state)
  - Filter has no knowledge of up- or down-stream filters
- Examples
  - UNIX shell
  - Distributed systems
  - signal processing
  - parallel programming
- Example: `ls invoices | grep -e August | sort`
Pipe and Filter (cont’d)

- **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
  - Certain analyses
    - Throughput, latency, deadlock
  - Concurrent execution
Pipe and Filter (cont’d)

- Disadvantages
  - Batch organization of processing
  - Interactive applications
  - Lowest common denominator on data transmission
Pipe and Filter LL

Get BurnRate from user -> Stream (in: br, out: none) -> Compute new values -> Stream (in: a, f, t, v, out: none) -> Display new values to user
Blackboard Style

- Two kinds of components
  - Central data structure — blackboard
  - Components operating 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

Enter burn rate from user

Display values

Compute new values and Update

Data Access

Blackboard Data Storage (altitude, burnRate, fuel, time, velocity)
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.
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)
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
- Lisp and Scheme
Interpreter LL
Mobile-Code Style

- **Summary**: 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 LL

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

- Event announcement instead of method invocation
  - “Listeners” register interest in and associate methods with events
  - System invokes all registered methods implicitly
- Component interfaces are methods and events
- Two types of connectors
  - Invocation is either explicit or implicit in response to events
- Style invariants
  - “Announcers” are unaware of their events’ effects
  - No assumption about processing in response to events
Implicit Invocation (cont’d)

- Advantages
  - Component reuse
  - System evolution
    - Both at system construction-time & run-time
- Disadvantages
  - Counter-intuitive system structure
  - Components relinquish computation control to the system
  - No knowledge of what components will respond to event
  - No knowledge of order of responses
Publish-Subscribe

Subscribers register/deregister to receive specific messages or specific content. Publishers broadcast messages to subscribers either synchronously or asynchronously.
Publish-Subscribe (cont’d)

- 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
Pub-Sub LL
Event-Based Style

- Independent components asynchronously emit and receive events communicated over event buses
- Components: Independent, concurrent event generators and/or consumers
- Connectors: Event buses (at least one)
- Data Elements: Events – data sent as a first-class entity over the event bus
- Topology: Components communicate with the event buses, not directly to each other.
- Variants: Component communication with the event bus may either be push or pull based.
- Highly scalable, easy to evolve, effective for highly distributed applications.
Event-based LL
Peer-to-Peer Style

- 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
Peer-to-Peer Style (cont’d)

- 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 LL
Learning Objectives

- Delineate the role of DSSAs and Patterns in Software architecture, and apply common patterns to problems

- Understand the role and benefits of architectural styles

- Understand and apply common styles in your designs

- Construct complex styles from simpler styles

- Understand the challenges around greenfield design
Heterogeneous Styles

- More complex styles created through composition of simpler styles
- REST (from the first lecture)
  - Complex history presented later in course
- C2
  - Implicit invocation + Layering + other constraints
- Distributed objects
  - OO + client-server network style
C2 Style

An indirect invocation style in which independent components communicate exclusively through message routing connectors. Strict rules on connections between components and connectors induce layering.
C2 Style (cont’d)

- Components: Independent, potentially concurrent message generators and/or consumers
- Connectors: Message routers that may filter, translate, and broadcast messages of two kinds: notifications and requests.
- Data Elements: Messages – data sent as first-class entities over the connectors. Notification messages announce changes of state. Request messages request performance of an action.
- Topology: Layers of components and connectors, with a defined “top” and “bottom”, wherein notifications flow downwards and requests upwards.
C2 LL

Diagram:

- SpaceCraft
- Clock
- Event
- Game Logic
- GUI

Connections:
- SpaceCraft: Request: BurnFuel, Request: UpdateData, Notification: CalculateStatus, Notification: DisplayStatus
- Clock: Notification: Tick
- Event: Notification: Tick, Notification: DisplayStatus, Request: BurnFuel
- Game Logic: Request: UpdateData
KLAX
KLAX in C2
Distributed Objects: CORBA

- “Objects” (coarse- or fine-grained) run on heterogeneous hosts, written in heterogeneous languages. Objects provide services through well-defined interfaces. Objects invoke methods across host, process, and language boundaries via remote procedure calls (RPCs).
- Components: Objects (software components exposing services through well-defined provided interfaces)
- Connector: (Remote) Method invocation
- Data Elements: Arguments to methods, return values, and exceptions
- Topology: General graph of objects from callers to callees.
- Additional constraints imposed: Data passed in remote procedure calls must be serializable. Callers must deal with exceptions that can arise due to network or process faults.
CORBA Concept and Implementation

Conceptual View

Operation on object requested by client

Generated according to language-specific mapping

Implementation Architecture

### 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>
### Style Summary, continued (2/4)

#### Data-flow styles

<table>
<thead>
<tr>
<th>Style</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Batch</strong></td>
<td>Separate programs executed sequentially, with batched input</td>
</tr>
<tr>
<td><strong>Pipe-and-filter</strong></td>
<td>Separate programs, a.k.a. filters, executed, potentially concurrently. Pipes route data streams between filters</td>
</tr>
<tr>
<td><strong>Shared memory</strong></td>
<td>All calculation centers on a common, changing data structure; Order of processing dynamically determined and data-driven.</td>
</tr>
<tr>
<td><strong>Rule-based</strong></td>
<td>Use facts or rules entered into the knowledge base to resolve a query</td>
</tr>
</tbody>
</table>

#### Shared memory

<table>
<thead>
<tr>
<th>Feature</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Blackboard</td>
<td>Independent programs, access and communicate exclusively through a global repository known as blackboard</td>
</tr>
<tr>
<td>Rule-based</td>
<td>Problem data and queries expressible as simple rules over which inference may be performed.</td>
</tr>
</tbody>
</table>

### Additional Notes

- 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.
### Style Summary, continued (3/4)

<table>
<thead>
<tr>
<th><strong>Interpreter</strong></th>
<th>Description</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Interpreter</td>
<td>Interpreter parses and executes the input stream, updating the state maintained by the interpreter</td>
<td>Highly dynamic behavior required. High degree of end-user customizability.</td>
<td>High performance required.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th><strong>Mobile Code</strong></th>
<th>Description</th>
<th>Advantages</th>
<th>Disadvantages</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mobile Code</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>
## Style Summary, continued (4/4)

### Implicit Invocation

<table>
<thead>
<tr>
<th>Publish-subscribe</th>
<th>Publishers broadcast messages to subscribers</th>
<th>Components are very loosely coupled. Subscription data is small and efficiently transported.</th>
<th>When middleware to support high-volume data is unavailable.</th>
</tr>
</thead>
<tbody>
<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

| C2 | Layered network of concurrent components communicating by events | When independence from substrate technologies required. Heterogeneous applications. When support for product-lines desired. | When high-performance across many layers required. When multiple threads are inefficient. |
| Distributed Objects | Objects instantiated on different hosts | Objective is to preserve illusion of location-transparency | When high overhead of supporting middleware is excessive. When network properties are unmaskable, in practical terms. |
Learning Objectives

- Delineate the role of DSSAs and Patterns in Software architecture, and apply common patterns to problems
- Understand the role and benefits of architectural styles
- Understand and apply common styles in your designs
- Construct complex styles from simpler styles
- Understand the challenges around greenfield design
Google-like problem: process a huge collection of documents (Web-pages)

- **[Distributed] Grep:** Produce a list of documents that contain a certain word.
- **Count of URL Access Frequency:** Process logs of web page requests and output the number of times each of them has been accessed.
- **Reversed Web-Link Graph:** For a list of web pages produce the set of links that point to these pages.
- **Term-Vector per Host:** A N-term vector summarizes the most N frequent words that occur in a document or a set of documents as a list of <word, frequency> pairs. The goal is to produce the term vector for all domain hosts.
- **Inverted Index:** For each word, produce the list of documents where it appears.

Common styles

- **Traditional, language-influenced**
  - Main program and subroutines
  - Object-oriented
- **Layered**
  - Virtual machines
  - Client-server
- **Data-flow styles**
  - Batch sequential
  - Pipe and filter
- **Shared-state**
  - Blackboard
  - Rule based
- **Interpreter**
  - Interpreter
  - Mobile code
- **Implicit invocation**
  - Event-based
  - Publish-subscribe
- **Peer-to-peer**
The first effort you should make in addressing a novel design challenge is to attempt to determine that it is genuinely a novel problem.

Basic Strategy
- Divergence – shake off inadequate prior approaches and discover/admit a variety of new ideas that offer a potentially workable solution
- Transformation – combine analysis and selection of these potential ideas. New understanding and changes to the problem statement
- Convergence – select and further refine ideas

Repeatedly cycling through the basic steps until a feasible solution emerges.
Analogy Search

- Examine other fields and disciplines unrelated to the target problem for approaches and ideas that are analogous to the problem.
- Formulate a solution strategy based upon that analogy.
- A common “unrelated domain” that has yielded a variety of solutions is nature, especially the biological sciences.
  - E.g., neural networks
Brainstorming

- Technique of rapidly generating a wide set of ideas and thoughts pertaining to a design problem
  - without (initially) devoting effort to assessing the feasibility.
- Brainstorming can be done by an individual or, more commonly, by a group.
- Problem: A brainstorming session can generate a large number of ideas... all of which might be low-quality.
- Chief value: identifying categories of possible designs, not any specific design solution suggested during a session.
“Literature” Search

- Examining published information to identify material that can be used to guide or inspire designers

- Digital library collections make searching much faster and more effective
  - IEEE Xplore
  - ACM Digital Library
  - Google Scholar

- The availability of free and open-source software adds special value to this technique.
Morphological Charts

- The essential idea:
  - identify all the primary functions to be performed by the desired system
  - for each function identify a means of performing that function
  - attempt to choose one means for each function such that the collection of means performs all the required functions in a compatible manner.

- The technique does not demand that the functions be shown to be independent when starting out.

- Sub-solutions to a given problem do not need to be compatible with all the sub-solutions to other functions in the beginning.
Removing Mental Blocks

- If you can’t solve the problem, change the problem to one you can solve.
  - If the new problem is “close enough” to what is needed, then closure is reached.
  - If it is not close enough, the solution to the revised problem may suggest new venues for attacking the original.
Controlling the Design Strategy

- Exploring diverse approaches
  - Potentially chaotic
  - \(\rightarrow\) care in managing the activity
- Identify and review *critical* decisions
- Relate the costs of research and design to the penalty for taking wrong decisions
- Insulate uncertain decisions
- Continually re-evaluate system “requirements” in light of what the design exploration yields
Learning Objectives

- Delineate the role of DSSAs and Patterns in Software architecture, and apply common patterns to problems
- Understand the role and benefits of architectural styles
- Understand and apply common styles in your designs
- Construct complex styles from simpler styles
- Apply styles in Greenfield design