

# A Brief Look at Architecture and some patterns for it

Architecture and Architectural Patterns - 1

## Scalability?

Architecture and Architectural Patterns - 2

## Packages for Scalability in UML

Notice: the gray entities aim at decoupling the system

Architecture and Architectural Patterns - 3

## Architecture

- **ROOM (ObjecTime) book:**
  - By "architecture" we mean that part of a system that provides the framework on which *all* other aspects of the system depend. A system's architecture is the principal factor that determines its capacity to evolve and adapt to new requirements.
- **Scalability implies grouping:**
  - Grouping objects into subsystems and layers is referred to as *clustering*.
  - An architecture typically takes the form of a set of *clustering* decisions.
  - An architecture is like a load-bearing wall: You can modify it but it's a lot of work!
  - Focusing exclusively on code and forgetting the decisions behind an architecture during software evolution is a recipe for '**architectural decay**': the gradual deterioration (due to changes over time) of the precise and identifiable boundaries and relationships between system components.

Architecture and Architectural Patterns - 4

### About Clustering

- Design clusters as black-box objects! Try to decouple clusters as much as possible:
  - Avoid point-to-point messages between elements of clusters:
    - » instead possibly use 'service access points'
  - Ultimately, the role of a cluster is to protect its components through a well-defined interface (façade pattern).
- Inter-cluster communication may typically be more expensive! Minimize it!
- **The bin-packing" problem:** it is an NP-complete problem to decide how to group objects into clusters to achieve highest performance.

Architecture and Architectural Patterns - 5

### Architectural Heuristics

- **Separate UI (i.e., input and output), hardware and database responsibilities from the functionality of the core of the system.**
  - Possibly use *connectors* to realize locality of change: for example, if the DB changes, only its connector should be affected.
  - Typically work out the details of this separate clusters during OOD, emphasizing abstraction first (through connectors).
  - Libraries and **legacy code** may influence the exact separation.
    - » **Legacy code:** a procedure, computer system and/or application that continues to be used and must be maintained
  - Avoid overlapping of services between these different clusters.
- **Several architectural patterns exist!**

Architecture and Architectural Patterns - 6

### Architectural Patterns according to [Buschman96]

- Structure of the components of a s/w system together with their interrelationships, principles and guidelines governing their design and evolution over time.
- Express fundamental **structural** organization schemas for s/w systems
  - they provide a way to organize generic modules, specifying their type of responsibilities, together with rules and guidelines for organizing the relationships between them.
- Categories:
  - decomposition of a system's task into cooperating subtasks
    - » aka from mud to structure (**pipes and filters, layers, blackboard**)
  - distributed applications (**broker**)
  - human-machine interactive systems:
    - » **MVC** aka **Presentation-Abstraction-Control**
  - adaptive systems (**microkernel, reflection**)

Architecture and Architectural Patterns - 7

### Example 1: Layer Pattern [Buschman96]

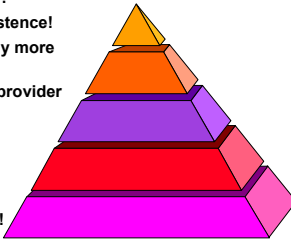
- **Intent:** provide a structure for applications that can be decomposed into groups of subtasks, each group providing a set of services for the layer above it.
- **Context:** a large system that requires decomposition.
- **Problem:** Consider designing a system with a mix of high and low level issues, where the higher ones rely on the lower level ones. Forces to be resolved:
  - later code changes should not ripple through the system,
  - parts of the system should be exchangeable,
  - parts of the system should be configurable,
  - similar responsibilities should be grouped together into coherent components,
  - no "standard" component granularity,
  - complex components may need decomposition,
  - crossing boundaries may be expensive, and
  - built by teams that need clear demarcation lines for what they are responsible for.
- **Solution:** structure the system into an appropriate collection of layers where each layer is a collaboration of components at the same level of abstraction. **Example: 7 OSI layers**

Architecture and Architectural Patterns - 8

### Layering

**From OSI layering principles:**

- One layer does not *contain* another:  
they typically have separate existence!
- Objects at higher levels are typically more application-specific
- Think of a lower layer as a *service provider*
- Minimize the number of layers
- Prefer a *closed* architecture...  
simpler reuse and evolution
- Beware of cloned services:  
locality of change is violated!



Architecture and Architectural Patterns - 9

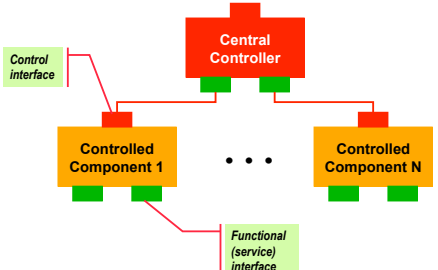
### Example 2: Recursive Control [Selic98]

- **Problem**
  - Structuring control in large real-time systems
    - » reach and maintain operational state
      - synchronize with external world, and
      - maintain operation despite events occurring asynchronously with internal operation.
    - » often inadequately addressed because we tend to first focus on primary functionality
      - exception handlers contain control policies
        - language specific, and
        - fragment system control policy across collection of handlers.
      - function and control are so closely coupled that any addition of new functionality risks compromising the control of the system.
- **Solution**
  - Separate Control from Function.
  - Separate Control Policies from Control Mechanisms
    - » example: detecting vs handling failure.

Architecture and Architectural Patterns - 10

### The Basic Structural Pattern

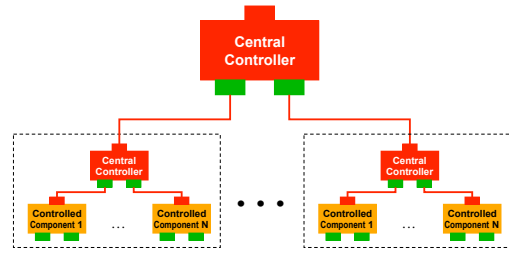
- Set of components that need to be controlled as a unit



Architecture and Architectural Patterns - 11

### Recursive Application

- **Hierarchical control**
  - scales up to arbitrary number of levels



Architecture and Architectural Patterns - 12

**Bottom Line**

- **Applicability:**
  - Event-driven real-time systems requiring non-trivial, dynamic control;
  - Evolution of both control and functionality.
- **Consequences:**
  - system control issues are brought up front
    - » it is more likely that they will be properly addressed.
  - simplifies implementation of complex systems
    - » recursive application of the same structural pattern.
  - minimizes coupling between changes in functionality and control.
  - additional overhead.
- **Related Patterns:**
  - Strategy design pattern
  - Composite design pattern.
  - Chain of Responsibility design pattern.

Architecture and Architectural Patterns - 13

**Example 3: Dynamic Structure [Aubin/Corriveau]**

- In dynamic systems, it is not known in advance which particular components will be involved in a **dynamic relationship!**

Architecture and Architectural Patterns - 14

**The Solution: Plug-In Roles**

- **Static placeholders that are filled in at run-time**

Architecture and Architectural Patterns - 15

**Type Genericity**

- **Plug-in roles can be filled in by any component that has the appropriate ports**
  - provided that the corresponding ports are not already connected in some other composite
  - a capsule can fit in even if it has additional ports that are not required for the role

Architecture and Architectural Patterns - 16

### Ports and Roles

- The roles that a particular capsule can play are determined by the set of its public service ports
- ⇒ A single capsule may be involved in multiple collaborations at the same time
  - e.g., control and functional interactions
  - in true dynamic systems, this is the case for most objects
- ⇒ **Multiple containment**: a capsule may be in more than one container at the same time