Architectural Styles

Reid Holmes
Lunar lander example

Flight Control Computer

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

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

- Attitude Control Thruster 1
- Main Descent Engine Controller
- Cockpit Displays
Language-based

- Influenced by the languages that implement them
- Lower-level, very flexible
- Often combined with other styles for scalability

Examples:
- Main & subroutine
- Object-oriented

We won’t cover these in any great detail
Style: Main program & subroutine

- Decomposition of functional elements.

- Components:
  - Main program and subroutines.

- Connections:
  - Function / procedure calls.

- Data elements:
  - Values passed in / out of subroutines.

- Topology:
  - Directed graph between subroutines and main program.
Style: Main program & subroutine

- Additional constraints:
  - None.

- Qualities:
  - Modularity, as long as interfaces are maintained.

- Typical uses:
  - Small programs.

- Cautions:
  - Poor scalability. Data structures are ill-defined.

- Relations to languages and environments:
  - BASIC, Pascal, or C.
Style: Object-oriented

- Encapsulation of state and actions.

- Components:
  - Objects or ADTs.

- Connections:
  - Method calls.

- Data elements:
  - Method arguments.

- Topology:
  - Varies. Data shared through calls and inheritance.
Style: Object-oriented

- Additional constraints:
  - Commonly used with shared memory (pointers). Object preserves identity of representation.

- Qualities:

- Typical uses:
  - With complex, dynamic data. Correlation to real-world entities.

- Cautions:
  - Distributed applications hard. Often inefficient for sci. computing. Potential for high coupling via constructors. Understanding can be difficult.

- Relations to languages and environments:
  - C++, Java.
**Dataflow**

- A data flow system is one in which:
  - The availability of data controls computation
  - The structure of the design is determined by the orderly motion of data between components
  - The pattern of data flow is explicit

- Variations:
  - Push vs. pull
  - Degree of concurrency
  - Topology

**Examples:**
- Batch-sequential
- Pipe-and-filter

[Reference: CZARNECKI]
Style: Batch-sequential

- Separate programs executed in order passed, each step proceeding after the previous finishes.

- Components:
  - Independent programs.

- Connections:
  - Sneaker-net.

- Data elements:
  - Explicit output of complete program from preceding step.

- Topology:
  - Linear.
Style: Batch-sequential

- Additional constraints:
  - One program runs at a time (to completion).

- Qualities:
  - Interruptible execution.

- Typical uses:
  - Transaction processing in financial systems.

- Cautions:
  - Programs cannot easily feed back into one another.
Style: Pipe-and-filter
Style: Pipe-and-filter

- Streams of data are passed concurrently from one program to another.

- Components:
  - Independent programs (called filters).

- Connections:
  - Explicitly routed by OS.

- Data elements:
  - Linear data streams, often text.

- Topology:
  - Typically pipeline.
Style: Pipe-and-filter

- Qualities:
  - Filters are independent and can be composed in novel sequences.

- Typical uses:
  - Very common in OS utilities.

- Cautions:
  - Not optimal for interactive programs or for complex data structures.
Layered

- Layered systems are hierarchically organized providing services to upper layers and acting as clients for lower layers.

- Lower levels provide more general functionality to more specific upper layers.

- In strict layered systems, layers can only communicate with adjacent layers.

Examples:
- Virtual machine
- Client-server
Style: Client-server

CLIENT 1
Get/Display Info
Graphics Processing

CLIENT 2
Get/Display Info
Graphics Processing

CLIENT n
Get/Display Info
Graphics Processing

Procedure Call

Procedure Call

Procedure Call

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

SERVER:
Game State
Game Logic
Environment
Simulation
Style: Client-server

- Clients communicate with server which performs actions and returns data. Client initiates communication.

- Components:
  - Clients and server.

- Connections:
  - Protocols, RPC.

- Data elements:
  - Parameters and return values sent / received by connectors.

- Topology:
  - Two level. Typically many clients.
Style: Client-server

› Additional constraints:
  › Clients cannot communicate with each other.

› Qualities:
  › Centralization of computation. Server can handle many clients.

› Typical uses:
  › Applications where: client is simple; data integrity important; computation expensive.

› Cautions:
  › Bandwidth and lag concerns.
Interpreter

- Commands interpreted dynamically
- Programs parse commands and act accordingly, often on some central data store

Examples:
- Interpreter
- Mobile code
Style: Mobile code

- Game Server
- Stream: in: game code, out: none
  - Lunar Lander Game Applet
- Stream
  - Lunar Lander Game Applet
- Stream
  - Lunar Lander Game Applet
Style: Mobile code

- Code and state move to different hosts to be interpreted.

- Components:
  - Execution dock, compilers / interpreter.

- Connections:
  - Network protocols.

- Data elements:
  - Representations of code, program state, data.

- Topology:
  - Network.
Style: Mobile code

- Variants:
  - Code-on-demand, remote evaluation, and mobile agent.

- Qualities:
  - Dynamic adaptability.

- Typical uses:
  - For moving code to computing locations that are closer to the large data sets being operated on.

- Cautions:
Style: Interpreter

Diagram:

1. Get Command from user (Burn, 50) (Check Status)
2. Stream
   - in: line of code
   - out: result of code executed
3. Interpret and Execute
4. Data Access
   - in: variable updated
   - out: result of code executed
   - out: none
5. Data Access
   - in: none
   - out: variable updated
   - out: result of code executed
6. Interpreter State
Style: Interpreter

- Interpret commands on the fly.
- Based on a virtual machine produced in SW.
- Components are the ‘program’, its data, its state, and the interpretation engine.
- e.g., Java Virtual Machine. JVM interprets Java bytecode).
Style: Interpreter

- Update state by parsing and executing commands.

- Components:
  - Command interpreter, program state, UI.

- Connections:
  - Components tightly bound; uses procedure calls and shared state.

- Data elements:
  - Commands.

- Topology:
  - Tightly coupled three-tier.
Style: Interpreter

- Qualities:
  - Highly dynamic behaviour. New capabilities can be added without changing architecture by introducing new commands.

- Typical uses:
  - End-user programming.

- Cautions:
  - May not be performant.
Shared state

- Characterized by:
  - Central store that represents system state
  - Components that communicate through shared data store
  - Central store is explicitly designed and structured

Examples:
- Blackboard
- Rule-based
Style: Blackboard

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

- Independent programs communicate exclusively through shared global data repository.

- Components:
  - Independent programs (knowledge sources), blackboard.

- Connections:
  - Varies: memory reference, procedure call, DB query.

- Data elements:
  - Data stored on blackboard.

- Topology:
  - Star; knowledge sources surround blackboard.
Style: Blackboard

- Variants:
  - Pull: clients check for blackboard updates.
  - Push: blackboard notifies clients of updates.

- Qualities:
  - Efficient sharing of large amounts of data. Strategies to complex problems do not need to be pre-planned.

- Typical uses:
  - Heuristic problem solving.

- Cautions:
  - Not optimal if regulation of data is needed or the data frequently changes and must be updated on all clients.
Implicit invocation

- In contrast to other patterns, the flow of control is “reversed”
- Commonly integrate tools in shared environments
- Components tend to be loosely coupled
- Often used in:
  - UI applications (e.g., MVC)
  - Enterprise systems
    - (e.g., WebSphere)

Examples:
- Publish-subscribe
- Event-based
Style: Publish-subscribe

- Subscriber 1
- Subscriber 2
- Subscriber n
- Game Server
- Stream
- Stream
- Stream
- Event

- in: register, reg info
- out: none
- in: new terrain, spacecraft
- out: none
Style: Publish-subscribe

- Subscribers register for specific messages or content. Publishers maintain registrations and broadcast messages to subscribers as required.

- Components:
  - Publishers, subscribers, proxies.

- Connections:
  - Typically network protocols.

- Data elements:
  - Subscriptions, notifications, content.

- Topology:
  - Subscribers connect to publishers either directly or through intermediaries.
Style: Publish-subscribe

- **Variants:**
  - Complex matching of subscribers and publishers can be supported via intermediaries.

- **Qualities:**
  - Highly-efficient one-way notification with low coupling.

- **Typical uses:**
  - News, GUI programming, network games.

- **Cautions:**
  - Scalability to large numbers of subscriber may require specialized protocols.
Style: Event-based
Style: Event-based

- Independent components asynchronously emit and receive events.

- Components:
  - Event generators / consumers.

- Connections:
  - Event bus.

- Data elements:
  - Events.

- Topology:
  - Components communicate via bus, not directly.
Style: Event-based

- Variants:
  - May be push or pull based (with event bus).

- Qualities:
  - Highly scalable. Easy to evolve. Effective for heterogeneous applications.

- Typical uses:
  - User interfaces. Widely distributed applications (e.g., financial markets, sensor networks).

- Cautions:
  - No guarantee event will be processed. Events can overwhelm clients.
Peer to Peer

- Network of loosely-coupled peers
- Peers act as clients and servers
- State and logic are decentralized amongst peers
- Resource discovery a fundamental problem
Peer-to-peer
Style: Peer-to-peer

- State and behaviour are distributed among peers that can act as clients or servers.

- Components:
  - Peers (aka independent components).

- Connections:
  - Network protocols.

- Data elements:
  - Network messages.

- Topology:
  - Network. Can vary arbitrarily and dynamically.
Style: Peer-to-peer

- **Qualities:**
  - Decentralized computing. Robust to node failures. Scalable.

- **Typical uses:**
  - When informations and operations are distributed.

- **Cautions:**
  - Security. Time criticality.
Activity

- Design Quest using an assigned pattern.
- What are the components, connectors, and topology?