Robotics Programming Laboratory

Bertrand Meyer
Jiwon Shin

Lecture 9: Software Architecture in Robotics
Some design patterns

**Creational**
- Abstract Factory
- Builder
- Factory Method
- Prototype
- Singleton

**Structural**
- Adapter
- Bridge
- Composite
- Decorator
- Façade
- Flyweight
- Proxy

**Behavioral**
- Chain of Responsibility
- Command (undo/redo)
- Interpreter
- Iterator
- Mediator
- Memento
- Observer
- State
- Strategy
- Template Method
- Visitor

Erich Gamma, Ralph Johnson, Richard Helms, John Vlissides: *Design Patterns*, Addison-Wesley, 1994
Benefits of design patterns

- Capture the knowledge of experienced developers
- Publicly available repository
- Common pattern language
- Newcomers can learn & apply patterns
- Yield better software structure
- Facilitate discussions: programmers, managers
Architectural styles
(based in part on material by Peter Müller)
Software architecture styles

An architectural style is defined by

- Type of basic architectural components (e.g. classes, filters, databases, layers)
- Type of connectors (e.g. calls, pipes, inheritance, event broadcast)
Model-View Controller

(Trygve Reenskaug, 1979)
Architecture styles

Overall system organization:
- Hierarchical
- Client-server
- Cloud-based
- Peer-to-peer

Individual program structuring:
- Control-based
  - Call-and-return (Subroutine-based)
  - Coroutine-based
- Dataflow:
  - Pipes and filters
  - Blackboard
  - Event-driven
- REST
- Object-oriented
Hierarchical

Each layer provides **services to the layer above it** and acts as a client of the layer below.

Each layer collects services at a particular level of **abstraction**.

A layer depends only on lower layers:
- Has no knowledge of higher layers.

**Example**
- Communication protocols
- Operating systems
Hierarchical

Components

- Group of subtasks which implement an abstraction at some layer in the hierarchy

Connectors

- Protocols that define how the layers interact
Hierarchical: examples

THE operating system (Dijkstra)
The OSI Networking Model

- Each level supports communication at a level of abstraction
- Protocol specifies behavior at each level of abstraction
- Each layer deals with specific level of communication and uses services of the next lower level

Layers can be exchanged

- Example: Token Ring for Ethernet on Data Link Layer
The system you are designing

Data transformation services, such as byte swapping and encryption

Initializes a connection, including authentication

Reliably transmits messages

Transmits & routes data within network

Sends & receives frames without error

Sends and receives bits over a channel
Hierarchical style example

Use service of lower layer

Virtual connection
Hierarchical: discussion

Strengths:

- Separation into levels of abstraction; helps partition complex problems
- Low coupling: each layer is (in principle) permitted to interact only with layer immediately above and under
- Extendibility: changes can be limited to one layer
- Reusability: implementation of a layer can be reused

Weaknesses:

- Performance overhead from going through layers
- Strict discipline often bypassed in practice
Call-and-return

Components: Objects
Connectors: Messages (routine invocations)

Key aspects
- Object preserves integrity of representation (encapsulation)
- Representation is hidden from client objects

Variations
- Objects as concurrent tasks
Call-and-return

Strengths:
- Change implementation without affecting clients
- Can break problems into interacting agents
- Can distribute across multiple machines or networks

Weaknesses:
- Objects must know their interaction partners; when partner changes, clients must change
- Side effects: if \( A \) uses \( B \) and \( C \) uses \( B \), then \( C \)'s effects on \( B \) can be unexpected to \( A \)
Client-server

Components

- Subsystems, designed as independent processes
- Each server provides specific services, e.g. printing, database access
- Clients use these services

Connectors

- Data streams, typically over a communication network
**Client -server example: databases**

**Clients: user applications**
- Customized user interface
- Front-end processing of data
- Initiation of server remote procedure calls
- Access to database server across the network

**Server: DBMS, provides:**
- Centralized data management
- Data integrity and database consistency
- Data security
- Concurrent access
- Centralized processing
Client-server variants

Thick / fat client
- Does as much processing as possible
- Passes only data required for communications and archival storage to the server
- Advantage: less network bandwidth, fewer server requirements

Thin client
- Has little or no application logic
- Depends primarily on server for processing
- Advantage: lower IT admin costs, easier to secure, lower hardware costs.
Client-server: discussion

Strengths:
- Makes effective use of networked systems
- May allow for cheaper hardware
- Easy to add new servers or upgrade existing servers
- Availability (redundancy) may be straightforward

Weaknesses:
- Data interchange can be hampered by different data layouts
- Communication may be expensive
- Data integrity functionality must be implemented for each server
- Single point of failure
Client-server variant: cloud computing

The server is no longer on a company’s network, but hosted on the Internet, typically by a providing company.

Example: cloud services by Google, Amazon, Microsoft

Advantages:

- Scalability
- Many issues such as security, availability, reliability are handled centrally

Disadvantages:

- Loss of control
- Dependency on Internet
Peer-to-peer

Similar to client-server style, but each component is both client and server

Pure peer-to-peer style

- No central server, no central router

Hybrid peer-to-peer style

- Central server keeps information on peers and responds to requests for that information

Examples

- File sharing applications, e.g., Napster
- Communication and collaboration, e.g., Skype
Peer-to-peer: discussion

**Strengths:**
- Efficiency: all clients provide resources
- Scalability: system capacity grows with number of clients
- Robustness
  - Data is replicated over peers
  - No single point of failure (in pure peer-to-peer style)

**Weaknesses:**
- Architectural complexity
- Resources are distributed and not always available
- More demanding of peers (compared to client-server)
- New technology not fully understood
Subroutines

Similar to hierarchical structuring at the program level

Functional decomposition

Topmost functional abstraction
Subroutines

Advantages:

- Clear, well-understood decomposition
- Based on analysis of system’s function
- Supports top-down development

Disadvantages:

- Tends to focus on just one function
- Downplays the role of data
- Strict master-slave relationship; subroutine loses context each time it terminates
- Adapted to the design of individual functional pieces, not entire system
Coroutines

A more symmetric relationship than subroutines

Particularly applicable to simulation applications

A simulated form of concurrency
Dataflow systems

Availability of data controls the computation.
The structure is determined by the orderly motion of data from component to component.

Variations:
- Control: push versus pull
- Degree of concurrency
- Topology
Dataflow: batch-sequential

Frequent architecture in scientific computing and business data processing

Components are independent programs
Connectors are media, typically files
Each step runs to completion before next step begins
Batch-sequential

History: mainframes and magnetic tape

Business data processing

- Discrete transactions of predetermined type and occurring at periodic intervals
- Creation of periodic reports based on periodic data updates

Examples

- Payroll computations
- Tax reports
Dataflow: pipe-and-filter

Component: filter
- Reads input stream (or streams)
- Locally transforms data
- Produces output stream(s)

Connector: pipe
- stream, e.g., FIFO buffer
Pipe-and-filter

Data processed **incrementally** as it arrives
Output can begin before input fully consumed

Filters must be **independent**: no shared state
Filters don’t know upstream or downstream filters

Examples

- lex/yacc-based compiler (scan, parse, generate…)
- Unix pipes
- Image / signal processing
Push pipeline with active source

Source of each pipe pushes data downstream

Example with Unix pipes:

```
grep p1 * | grep p2 | wc | tee my_file
```
Pipe-and-filter: discussion

**Strengths:**
- Reuse: any two filters can be connected if they agree on data format
- Ease of maintenance: filters can be added or replaced
- Potential for parallelism: filters implemented as separate tasks, consuming and producing data incrementally

**Weaknesses:**
- Sharing global data expensive or limiting
- Scheme is highly dependent on order of filters
- Can be difficult to design incremental filters
- Not appropriate for interactive applications
- Error handling difficult: what if an intermediate filter crashes?
- Data type must be greatest common denominator, e.g. ASCII
Dataflow: event-based (publish-subscribe)

A component may:

- Announce events
- Register a callback for events of other components

Connectors are the bindings between event announcements and routine calls (callbacks)
Event-based style: properties

Publishers of events do not know which components (subscribers) will be affected by those events

Components cannot make assumptions about ordering of processing, or what processing will occur as a result of their events

Examples

- Programming environment tool integration
- User interfaces (Model-View-Controller)
- Syntax-directed editors to support incremental semantic checking
Event-based style: example

Integrating tools in a shared environment

Editor announces it has finished editing a module
- Compiler registers for such announcements and automatically re-compiles module
- Editor shows syntax errors reported by compiler

Debugger announces it has reached a breakpoint
- Editor registers for such announcements and automatically scrolls to relevant source line
Event-based: discussion

Strengths:
- Strong support for reuse: plug in new components by registering it for events
- Maintenance: add and replace components with minimum effect on other components in the system

Weaknesses:
- Loss of control:
  - What components will respond to an event?
  - In which order will components be invoked?
  - Are invoked components finished?
- Correctness hard to ensure: depends on context and order of invocation
Components

- Central data store component represents state
- Independent components operate on data store
Data-Centered: discussion

**Strengths:**
- Efficient way to share large amounts of data
- Data integrity localized to repository module

**Weaknesses:**
- Subsystems must agree (i.e., compromise) on a repository data model
- Schema evolution is difficult and expensive
- Distribution can be a problem
Blackboard architecture

Interactions among knowledge sources solely through repository

Knowledge sources make changes to the shared data that lead incrementally to solution

Control is driven entirely by the state of the blackboard

Example

- Repository: modern compilers act on shared data: symbol table, abstract syntax tree
- Blackboard: signal and speech processing
Blackboard architecture: example

The EVE architecture
The EVE architecture (ETH chair of SE)

AutoProof

Inter. prover

Alias analysis

Sep. logic prover

Invariant inference

Suggestions

AutoProof

Inter. prover

Sep. logic prover

Test results

AutoTest

Test case generation

EVE

Suggestions

AutoTest

Test case generation

Inter. prover

Sep. logic prover

Invariant inference

Suggestions

AutoFix

Invariants inference
Interpreters

Architecture is based on a **virtual machine** produced in software.

Special kind of a **layered architecture** where a layer is implemented as a true language interpreter.

Components

- “Program” being executed and its data
- Interpretation engine and its state

Example: Java Virtual Machine

- Java code translated to platform independent bytecode
- JVM is platform specific and interprets the bytecode
Object-oriented

Based on analyzing the types of objects in the system and deriving the architecture from them

Compendium of techniques meant to enhance extendibility and reusability: contracts, genericity, inheritance, polymorphism, dynamic binding...

Thanks to broad notion of what an “object” is (e.g. a command, an event producer, an interpreter...), allows many of the previously discussed styles