ARCHITECTURAL PATTERNS

PATTERN

- A pattern for software architecture describes a particular recurring design ***problem*** that arises in specific design ***contexts***, and presents well-proven generic scheme for its ***solution***.
- The solution scheme is specified by describing its **constituent components**, their **responsibilities** and **relationships**, and the ways in which they collaborate.

PATTERNS: USE

- A pattern addresses a **recurring design problem** that arises in specific design situations, and presents a solution to it.
- Patterns **document** existing, well-proven **design** experience.
- Patterns identify and specify **abstractions** that are above the level of single classes and instances, or of components.
- Patterns provide a **common vocabulary** and understanding for design principles.
- Patterns are a means of **documenting** software architectures.
- Patterns support construction of software with defined properties. (Patterns explicitly address non-functional requirements for software systems)
- Patterns help build **complex** and **heterogeneous** software architectures.
- Patterns act as **building-blocks** for constructing more complex designs.
- Patterns help manage software complexity.

PATTERN: SCHEMA

- Three-part schema that underlies every pattern:
  - **Context:** a situation giving rise to a problem
  - **Problem:** the recurring problem arising in that context
  - **Solution:** a proven resolution of the problem

- The **schema** as a whole denotes a type of rule that establishes a relationship between a given **context**, a certain **problem** arising in that context, and an appropriate **solution** to the problem.

PATTERN: SCHEMA

- **Context:** extends the plain problem-solution dichotomy by describing situations in which the problem occurs.
- The context of a pattern may be fairly general (e.g. developing software with human-computer interface) or specific (e.g. implementing MVC mechanism)
- Specifying the correct context for a pattern may be difficult as determining all situations is impossible.

PATTERN: SCHEMA

- **Problem:** describes the problem that arises repeatedly in the given context. (concrete design issue). For example:
  - The problem of varying user interfaces

- **Force:** any aspect of the problem that should be considered when solving it; such as:
  - **Requirements** the solution must fulfill (e.g. efficiency)
  - **Constraints** that should be considered (e.g. following a particular protocol)
  - **Desirable** properties the solution should have (e.g. changing software should be easy)

- Forces may complement or contradict each other. (e.g. extensibility of a system vs. minimization of its code size)

PATTERN: SCHEMA

- **Solution:** shows how to solve the recurring problem, or how to balance the forces associated with it.
- Aspects of Solution:
  - Every pattern specifies a certain **structure**, a spatial configuration of elements (Static aspect).
  - Every pattern specifies a **run-time behavior** (Dynamic aspect).
- Solution does not necessarily resolve all forces associated with the problem.

PATTERN: SUMMARY OF THE SCHEMA

Pattern

- **Context**
  - Design situation giving rise to a design problem
- **Problem**
  - Set of forces repeatedly arising in the context
- **Solution**
  - Configuration to balance the forces
    - Structure with components and relationships
    - Run-time behavior

PATTERNS

- Pattern categories:
  - Architectural Patterns
  - Design Patterns
  - Idioms

ARCHITECTURAL PATTERNS

- An ***architectural pattern*** expresses a fundamental structural organization schema for software system. It provides a set of **predefined subsystem**, specifies their **responsibilities**, and include **rules** and guidelines for organizing the relationship between them.

- Architectural patterns are **templates** for concrete software architectures. They specify the **system-wide structural properties** of an application, and have an impact on the architecture of its sub-systems.

- The **Model-View-Controller pattern** is one of the best-known examples of an architectural pattern. It provides a structure for interactive software systems.

DESIGN PATTERNS

- **Design Pattern:** the subs-systems of a software architecture, as well as the relationships between them, usually consist of several smaller architectural units defined as design patterns.

- A ***design pattern*** provides a scheme for refining the subsystems or components of a software system, or the relationships between them. It describes a commonly-recurring structure of communicating components that solves a general design problem within a particular context.

- Design patterns are medium-scale patterns.

IDIOMS

- An ***idiom*** is a low-level pattern specific to a programming language. An idiom describes how to implement particular aspects of components or the relationships between the using the features of a given language.
- Idioms represent the lowest-level patterns. They address aspects of both **design** and **management.**

PATTERN DESCRIPTION

- Patterns must be presented in an appropriate form to be understood.
  - What is the problem the pattern addresses?
  - What is the proposed solution?
  - The details necessary to implement a pattern and its consequences?

- **Context-Problem-Solution** structure as a description format

PATTERN DESCRIPTION

- **Name:** name and short summary
- **Also Known As:** Other names
- **Example:** a real-world example
- **Context:** Situations to apply pattern
- **Problem:** addressed problem and forces
- **Solution:** Fundamental solution principle
- **Structure:** Structural aspects
- **Dynamics:** run-time behavior
- **Implementation:** Guidelines for implementing
- **Example Resolved:** important aspects
- **Variants:** variants of a pattern
- **Known Uses:** examples of pattern from existing systems
- **Consequences:** benefits and potential liabilities
- **See Also:** References to similar patterns

HETEROGENEOUS ARCHITECTURE

- Patterns must support the **development**, **maintenance**, and **evolution** of complex, large-scale systems. They must also support effective **industrial** software production.
- A single pattern cannot enable the detailed construction of a complete software architecture.
- To meet the needs of software architecture ‘in the large’ we need a rich set of patterns that must cover many different design problems.

ARCHITECTURAL PATTERNS

- Architectural Patterns:
  - Layers
  - Pipes and Filters
  - Blackboard
  - Broker
  - Model-View-Controller
  - Presentation-Abstraction-Control
  - Microkernel
  - Reflection

ARCHITECTURAL PATTERNS

- Architectural Patterns represent the highest-level patterns in a system. They help to specify the fundamental structure of an application and rule other development activities; such as: detailed design of subsystems, communication and collaboration between different parts and later extensions.
- Categories of patterns (based on supporting properties):
  - **From Mud to Structure** (*Layers*, *Pipes and Filters*, and *Blackboard*)
  - **Distributed Systems** (*Broker*. Also refers to *Microkernel* and *Pipes and Filters*)
  - **Interactive Systems** (*Model-View-Controller* and *Presentation-Abstraction-Control*)
  - **Adaptable Systems** (*Reflection* and *Microkernel*)

ARCHITECTURAL PATTERNS: FROM MUD TO STRUCTURE

- **From Mud to Structure:** helps to avoid huge number of components and objects. Supports a controlled decomposition of an overall system task into cooperating subtasks.
  - **Layer:** helps to structure applications that can be **decomposed into groups of subtasks** in which each group of subtasks is at a particular level of abstraction.
  - **Pipes and Filters:** provides a structure for systems that process a **stream of data**. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows to build families of related systems.
  - **Blackboard:** is useful for problems for which **no deterministic solution strategies** are known. Several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.

ARCHITECTURAL PATTERNS: LAYERS

- The ***Layers*** architectural pattern helps to structure applications that can be decomposed into groups of subtasks in which each group of subtasks is at a particular level of abstraction. Examples:
- **Networking protocols (e.g. TCP/IP):**
  - consist of a set of rules and conventions that describe how computer programs communicate across machine boundaries. The format, contents and meaning of all messages are defined.
  - Specify agreements at a variety of abstraction levels, ranging from the details of bit transmission to high-level application logic. Designers use several sub-protocols and arrange them in layers. Each layer deals with a specific aspect of communication and uses the services of the next lower layer.

ARCHITECTURAL PATTERNS: LAYERS

Application	Layer 7	Provides miscellaneous protocols for common activities
Presentation	Layer 6	Structures information and attaches semantics
Session	Layer 5	Provides dialog control and synchronization facilities
Transport	Layer 4	Breaks messages into packets and guarantees delivery
Network	Layer 3	Selects a route from sender to receiver
Data Link	Layer 2	Detects and corrects errors in bit sequences
Physical	Layer 1	Transmits bits: velocity, bit-code, connection, etc.

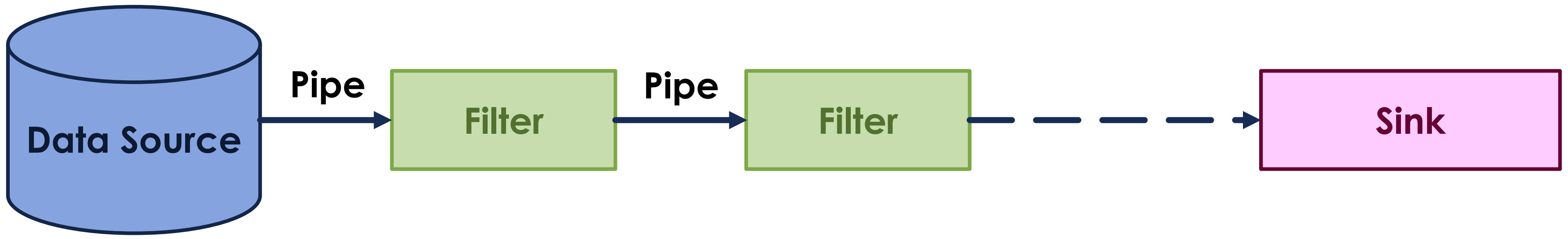
ARCHITECTURAL PATTERNS: LAYERS

- A layered approach is considered better practice than implementing the protocol as a monolithic block. Since implementing conceptually different issues separately reaps several benefits (e.g. aiding development by teams and supporting incremental coding and testing).
- Using semi-independent parts also enables the easier exchange of individual parts at a later date.
- Better implementation technologies such as new languages or algorithms can be incorporated by simply rewriting a delimited section of code.
- **Context:** a large system that requires decomposition.
- **Scenario:** a system whose dominant characteristic is a mix of **low-level** (e.g. sensor input, reading bits from a file) and **high-level** (e.g. interface of multi-user application) issues, where high-level operations rely on the lower-level ones.

ARCHITECTURAL PATTERNS: PIPES AND FILTERS

- The ***Pipes and Filters*** architectural pattern provides a structure for systems that process a stream of data. Each processing step is encapsulated in filter component. Data is passes through pipes between adjacent filters. Recombining filters allows to build families of related systems. Examples: Linked Programs, Compilers, …
- **Context:** processing data streams.

ARCHITECTURAL PATTERNS: PIPES AND FILTERS



ARCHITECTURAL PATTERNS: PIPES AND FILTERS

- **Scenario:** a system must process or transform a stream of input data. Implementing such a system as a single component may not be feasible for several reasons:
  - the system has to be built by several developers
  - The global system task decomposes naturally into several processing stages
  - Requirements are likely to change
- Solution:
  - the pipes and filters architectural pattern divides the task of a system into **several sequential processing steps**. These steps are connected by the data flow through the system.
  - Each processing step is implemented by a ***filter*** component. A filter consumes and delivers data incrementally (in contrast to consuming all its input before producing any output) to achieve low latency and parallel processing.
  - The input to the system is provided by a ***data source*** (e.g. a text file). The output flows into a ***data sink*** (e.g. a file). The data source, the filters and the data sink are connected sequentially by ***pipes*** creating a processing pipeline.

ARCHITECTURAL PATTERNS: PIPES AND FILTERS

Benefits

- No intermediate files are necessary (but possible)
- Flexibility by filter exchange.
- Flexibility by recombination.
- Reuse of filter components.
- Rapid prototyping of pipelines.
- Efficiency by parallel processing (promotes concurrency).

Liabilities

- Sharing state information is expensive or inflexible.
- Efficiency gain by parallel processing is often an illusion.
- Data transformation overhead.
- Error handling

ARCHITECTURAL PATTERNS: PIPES AND FILTERS

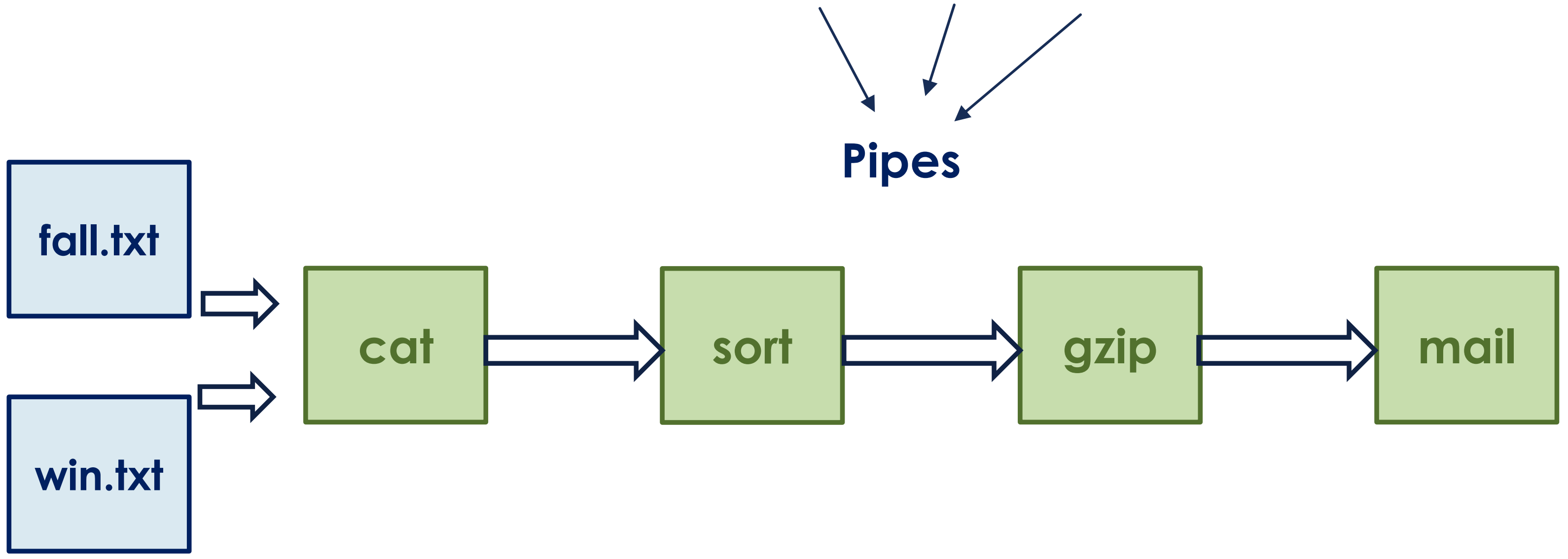
Known Uses

- For large processes that can be broken down into multiple steps
- Examples:
  - Unix Commands
  - Image Processing
  - Compilers

Example

- **Unix Command Pipelines**

```
cat fall.txt win.txt  |  sort  |  gzip  |  mail abc@u.ca
```



ARCHITECTURAL PATTERNS: BLACKBOARD

- The **Blackboard** architectural pattern is useful for problems for which no deterministic solution strategies are known. In Blackboard several specialized subsystems assemble their knowledge to build a possibly partial or approximate solution.

- **Example:** software system for speech recognition.
  - **Input:** Speech as waveform
  - **Output:** machine representation of the corresponding English phrase
  - **Transformations:** acoustic-phonetic, linguistic, statistical

- **Context:** an immature domain in which no closed approach to a solution is known or feasible.

ARCHITECTURAL PATTERNS: BLACKBOARD

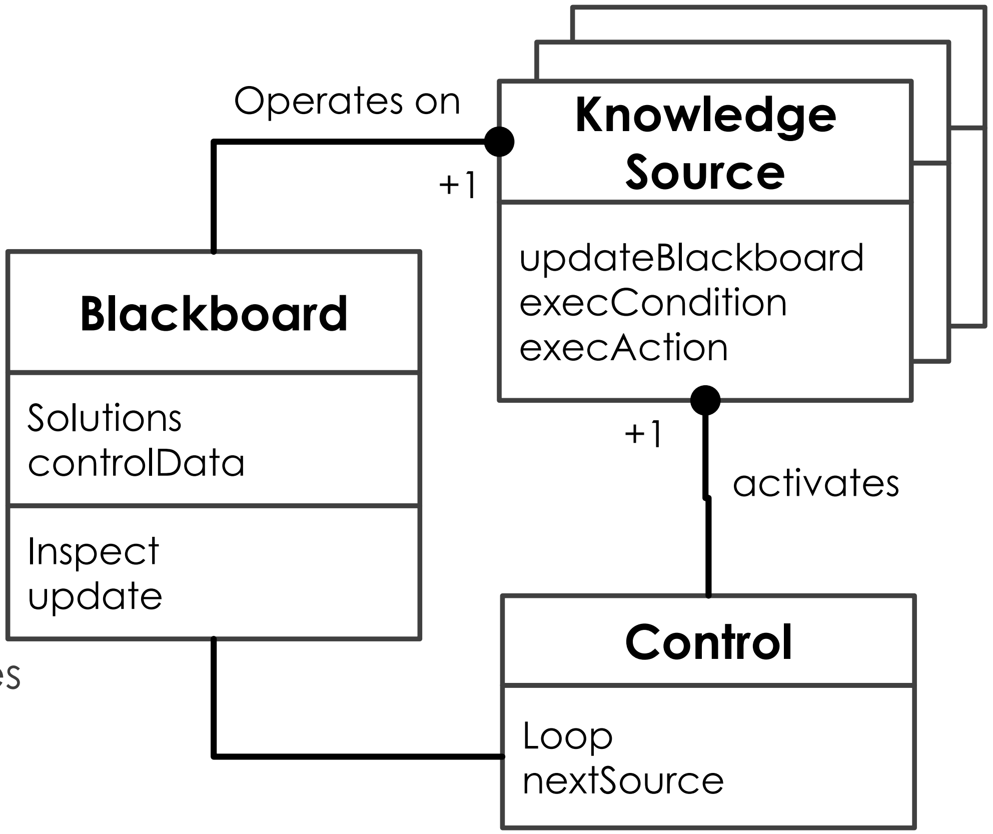
- The Blackboard pattern tackles problem that do not have a feasible deterministic solution for the transformation of raw data into high level data structures (e.g. diagrams, tables or English phrases).
- **Vision**, **image recognition**, **speech recognition** and **surveillance** are examples of domains in which such problems occur. They are characterized by a problem that, when decomposed into sub-problems span several fields of expertise.

ARCHITECTURAL PATTERNS: BLACKBOARD

- **Solution:** The idea behind the Blackboard architecture is a **collection of independent programs** that work cooperatively on a common data structure. Each program is specialized for solving a particular part of the overall task, and all programs work together on the solution. These specialized programs are independent of each other. They do not call each other, nor is there a predetermined sequence for their activation.

ARCHITECTURAL PATTERNS: BLACKBOARD

- **Structure:** Divide the system into a component called **blackboard**, a collection of **knowledge sources**, and a **control** component:
  - The blackboard is the central data store.
  - Knowledge sources are separate, independent subsystems that solve specific aspect of the overall problem. Knowledge sources read from and write to the blackboard.
  - The control component runs a loop that monitors the changes on the blackboard and decides what action to take next.



ARCHITECTURAL PATTERNS: DISTRIBUTED SYSTEMS

- **Distributed Systems:** there are 3 patterns related to distributed systems
  - **Pipes and Filters:** provides a structure for systems that process a stream of data. Each processing step is encapsulated in a filter component. Data is passed through pipes between adjacent filters. Recombining filters allows to build families of related systems. (used more for structuring the functional core of an application than for distribution)
  - **Microkernel:** applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts. Microkernel systems employ Client-Server architecture in which clients and servers run on top of the microkernel component.
  - **Broker:** can be used to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communications.

ARCHITECTURAL PATTERNS: BROKER

- The **Broker** architectural pattern can be used to structure distributed software systems with decoupled components that interact by remote service invocations. A broker component is responsible for coordinating communication, such as forwarding requests, as well as for transmitting results and exceptions.

- **Example:** a city information system on a wide-area network. Some computers on the network host one or more services about events, restaurants, hotels, historical monuments, or public transit. Users use WWW to get access to services. System changes and grows continuously, so the individual services should be decoupled from each other.

ARCHITECTURAL PATTERNS: BROKER

- **Context:** the environment is a distributed one and possible heterogeneous systems with independent cooperating components exist.

- **Problem:** building a complex software system as a set of decoupled and inter-operating components, rather than as a monolithic application, results in greater flexibility, maintainability and changeability. By partitioning functionality into independent components the system becomes potentially **distributable** and **scalable**.

- **Solution:** introduce a broker component to achieve better **decoupling** of clients and servers.

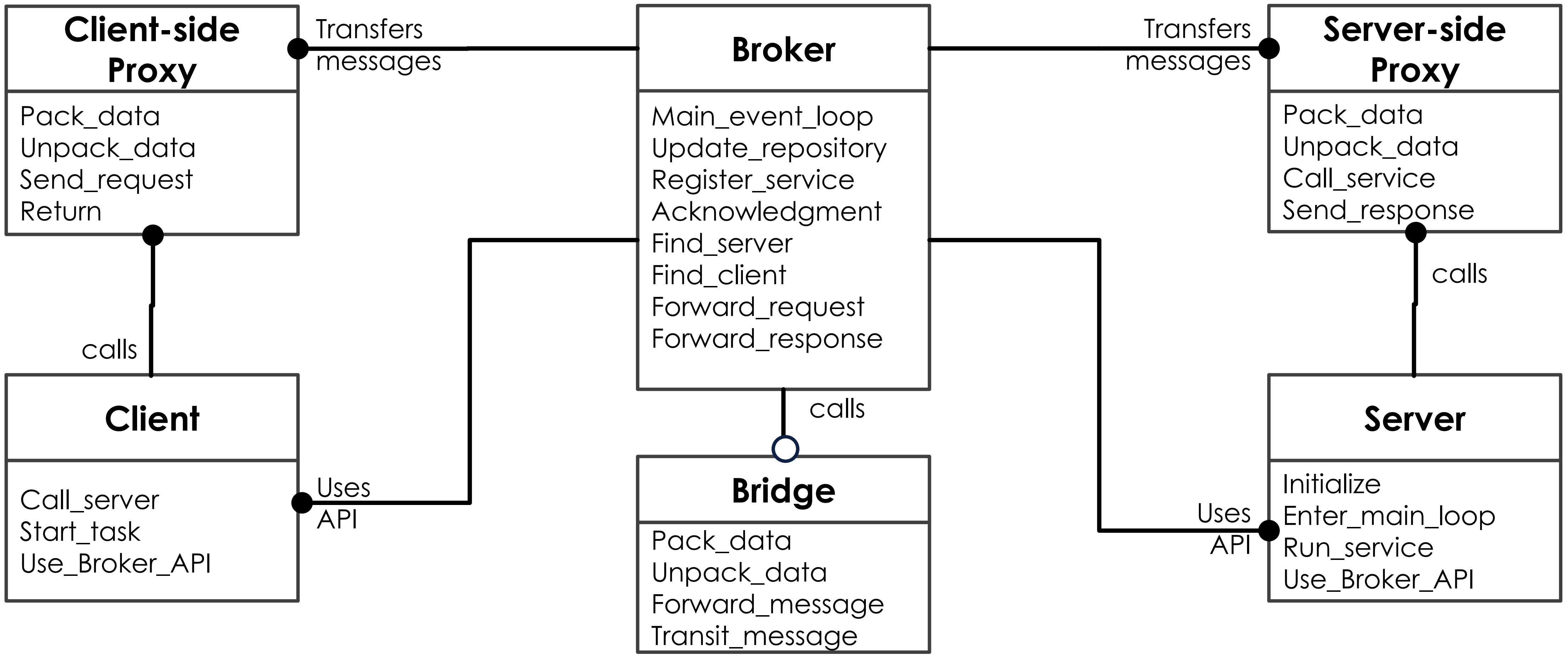
ARCHITECTURAL PATTERNS: BROKER

- **How it works:**
  - **Servers** register themselves with the broker, and make their services available to clients through method interfaces.
  - **Clients** access the functionality of servers by sending requests via the broker.
  - A **broker's** task include locating the appropriate server, forwarding the request to the server and transmitting results and exceptions back to the client.

ARCHITECTURAL PATTERNS: BROKER

- **Structure:** The Broker architectural pattern comprises six types of participating components: clients, servers, brokers, bridges, client-side proxies and server-side proxies.
  - A **Servers** implements objects that expose their functionality through interfaces that consist of operations and attributes (e.g. web servers)
  - **Clients** are applications that access the services of at least one server (e.g. web browsers)
  - A **broker** is a messenger that is responsible for the transmission of request from clients to servers, as well as the transmission of responses and exceptions back to the client. (e.g. the combination of an Internet gateway and the Internet Infrastructure).
  - **Client-side proxies** represent a layer between clients and the broker.
  - **Server-side proxies** are generally analogous to Client-side proxies. The difference is that they are responsible for receiving requests, unpacking incoming messages and calling the appropriate service.
  - **Bridges** are optional components used for hiding implementation details when two brokers interoperate.

ARCHITECTURAL PATTERNS: BROKER



ARCHITECTURAL PATTERNS: INTERACTIVE SYSTEMS

- **Interactive Systems:** systems with high degree of user interaction, mainly with the help of graphical user interfaces. The objective is to enhance the **usability** of an application.
  - **Model-View-Controller:** divides an interactive application into three components. The **model** contains the core functionality and data. **Views** display information to the user. **Controllers** handle user input. <u>Views and Controller together comprise the user interface.</u>
  - **Presentation-Abstraction-Control:** defines a structure for interactive software systems in the form of a hierarchy of cooperating agents. Every agent is responsible for a specific aspect of the application’s functionality and consists of three components: presentation, abstraction, and control.

ARCHITECTURAL PATTERNS: MODEL-VIEW-CONTROLLER

- The **Model-View-Controller** architectural pattern (MVC) divides an interactive application into three component.
  - The model contains the core functionality and data.
  - Views display information to the user.
  - Controllers handle user input.
- Views and controllers together comprise the user interface.

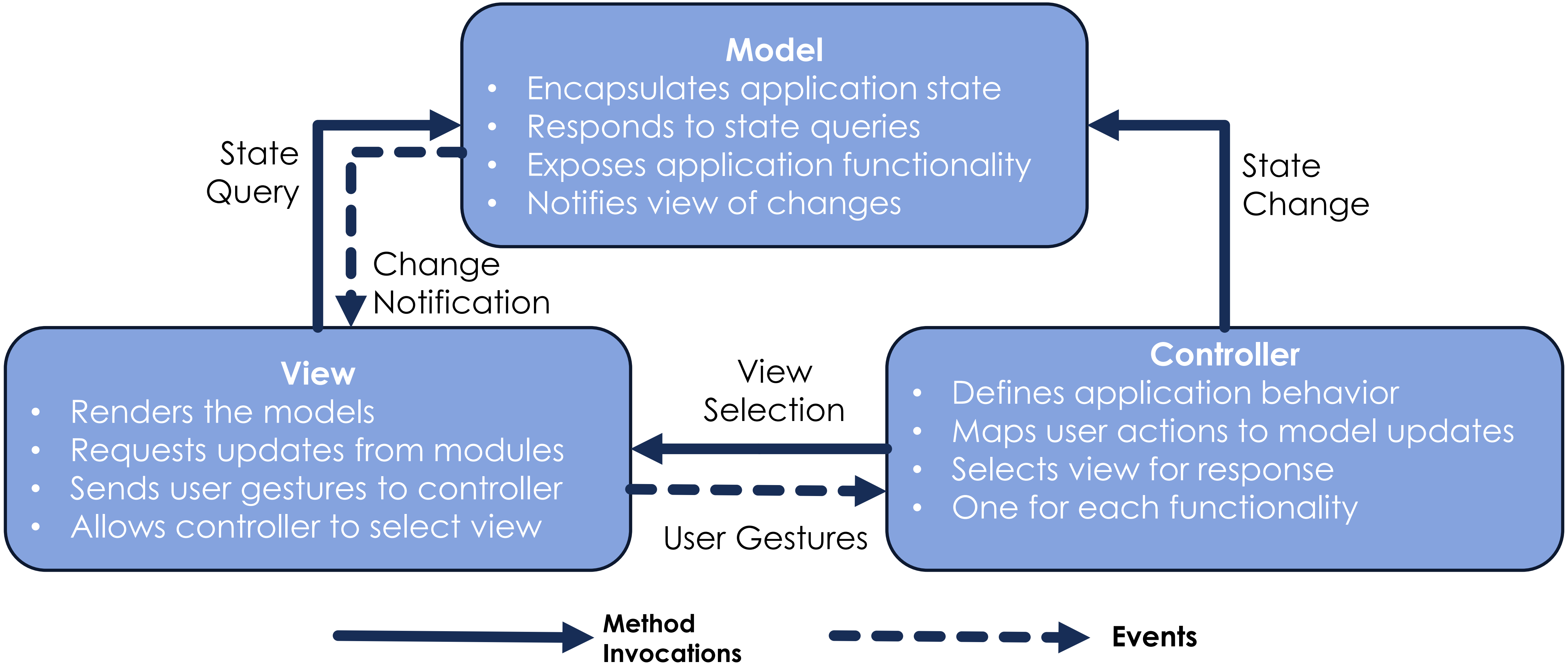
ARCHITECTURAL PATTERNS: MODEL-VIEW-CONTROLLER

- **Context:** User interface software is typically the most frequently modified portion of an interactive application. For this reason it is important to keep modifications to the user interface software separate form the rest of the system.
- **Solution:** The model-view-controller (MVC) pattern separates application functionality into three kinds of components to facilitate its modifiability.

ARCHITECTURAL PATTERNS: MODEL-VIEW-CONTROLLER

- The **model** component contains the functional core of the application. It encapsulates the appropriate data, and exports procedures that perform application specific processing. **Controllers** call these procedures on behalf of the user. The **model** also provides functions to access its data that are used by **view** components to acquire the data to be displayed.
- **View** components present information to the user. Different **views** present the information of the model in different ways.
- The **controller** components accept user input as events.

ARCHITECTURAL PATTERNS: MODEL-VIEW-CONTROLLER


Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies view of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from modules
• Sends user gestures to controller
• Allows controller to select view
View Selection
User Gestures
Controller
• Defines application behavior
• Maps user actions to model updates
• Selects view for response
• One for each functionality
Method Invocations
Events

ARCHITECTURAL PATTERNS: **PRESENTATION-ABSTRACTION-CONTROL**

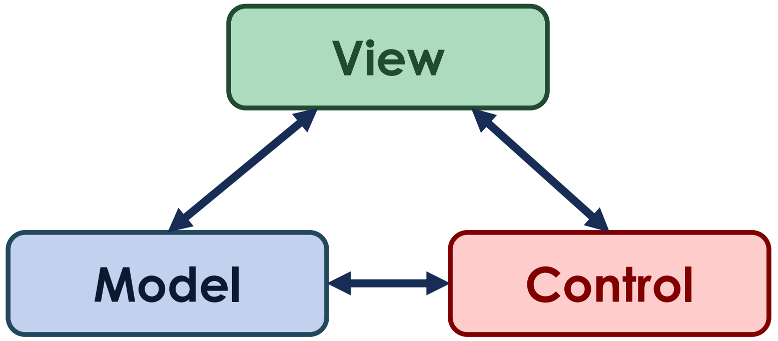
- The **Presentation-Abstraction-Control** architectural pattern (PAC) defines a structure for interactive software systems in the form of a hierarchy of cooperating **agents**.
- Every **agent** is responsible for a specific aspect of the application's functionality and consists of three components: **presentation**, **abstraction**, and **control**.
- This subdivision separates the human-computer interaction aspects of the agent from its functional core and its communication with other agents.

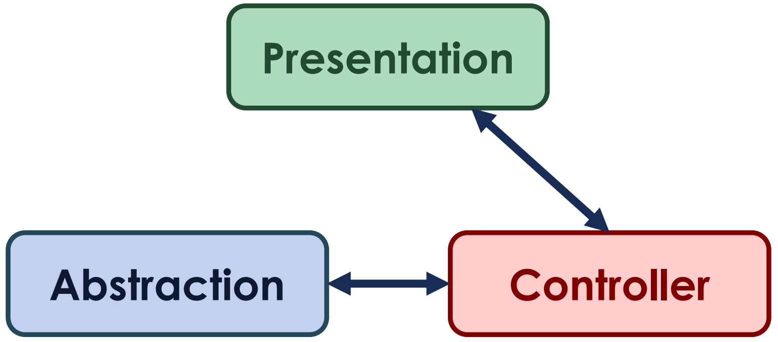
ARCHITECTURAL PATTERNS: **PRESENTATION-ABSTRACTION-CONTROL**

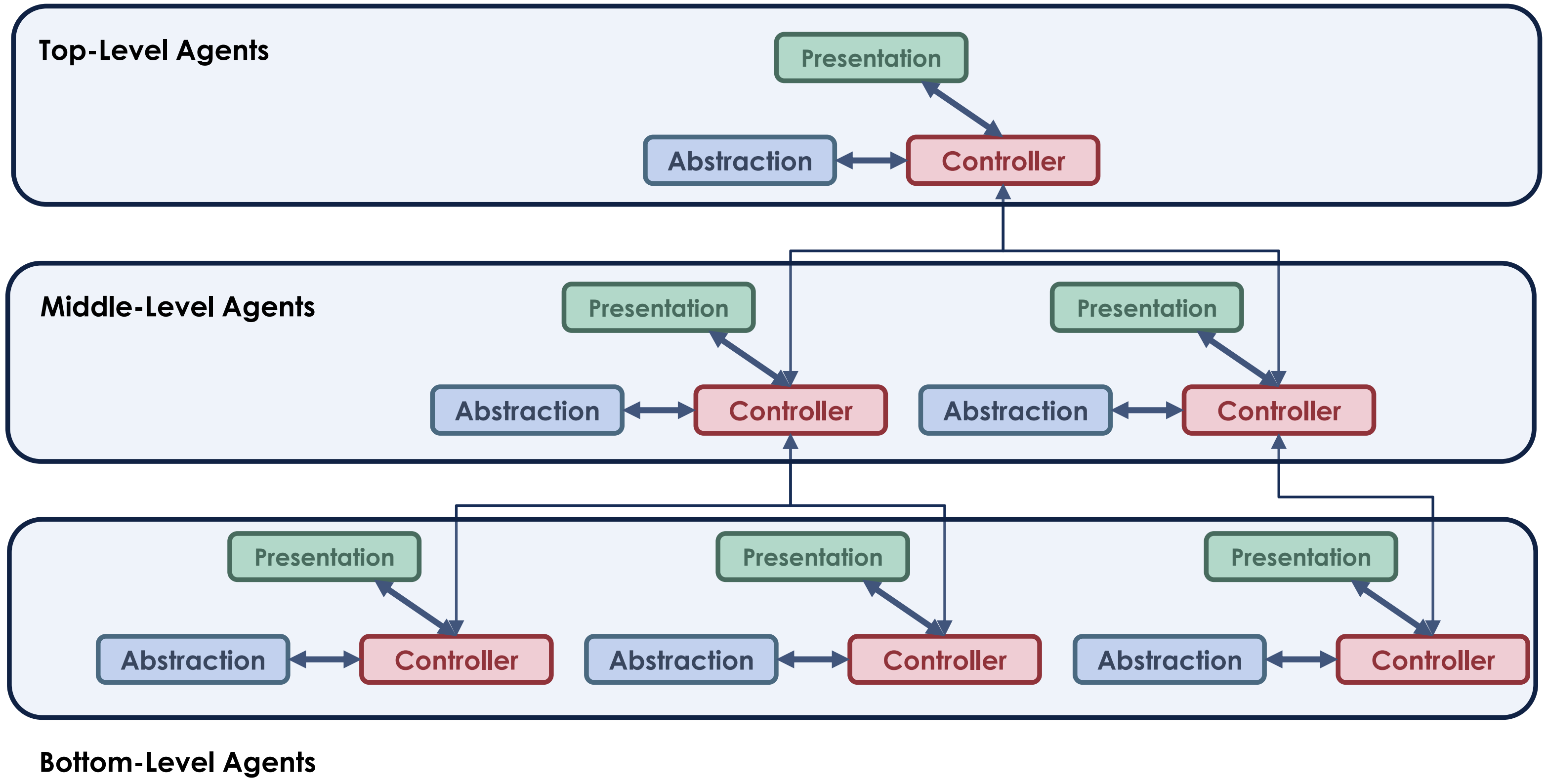
- PAC structures the interactive application as a **tree-like hierarchy of agents**. (1 top-level agent, several intermediate-level agents, and more bottom-level agents)
- The whole architecture reflects transitive dependencies between agents. Each agent depends on all higher-level agents.
- The agent's **presentation** component provides the visible behavior of the PAC agent.
- Its **abstraction** component maintains the data model that underlies the agent
- Its **control** component connects the presentation and abstraction components and provides functionality to communicate with other agents.

ARCHITECTURAL PATTERNS: **PRESENTATION-ABSTRACTION-CONTROL**

Similar to **MVC** pattern







ARCHITECTURAL PATTERNS: **PRESENTATION-ABSTRACTION-CONTROL**

- The **top-level PAC agent** provides the functional core of the system. Most other PAC agents depend or operate on this core.
- **Bottom-level PAC agents** represent self-contained semantic concepts on which users of the system can act (e.g. spreadsheets, charts, and their functions).
- Intermediate-level PAC agent represent either combinations of, or relationships between lower-level agents.

ARCHITECTURAL PATTERNS: **PRESENTATION-ABSTRACTION-CONTROL**

- **Example**: Air Traffic Control System
  - **Top-level agent:** Support Site, Operational Approval, Support Safety Activities
  - **Intermediate-level agents:** Performance Analysis, Post Processing Investigation, Service Volume, Performance Prediction, …
  - **Bottom-level agents:** Data Recording, Data Processing and Analysis, Data Visualization

ARCHITECTURAL PATTERNS: **PRESENTATION-ABSTRACTION-CONTROL**

- **Benefits**:
  - Support of **multi-tasking** and **multi-viewing**
  - Support agent **reusability** and **extensibility**
  - Easy to **plug-in** new agent or **replace** an existing one
  - Support for **Concurrency** where multiple agents are running in **parallel** in different threads or different devices.

ARCHITECTURAL PATTERNS: ADAPTABLE SYSTEMS

- Systems evolve over time – new functionality is added and existing services are changed. They must support new versions of operating systems, user interface platforms or third-party components or libraries. Adaptation to new standards or hardware platforms is also necessary. **Adaptable systems** with two main patterns help when designing for change:
  - The **Microkernel** pattern applies to software systems that must be able to adapt to changing system requirements.
  - The **Reflection** pattern provides a mechanism for changing structure and behavior of software systems dynamically.

ARCHITECTURAL PATTERNS: MICROKERNEL

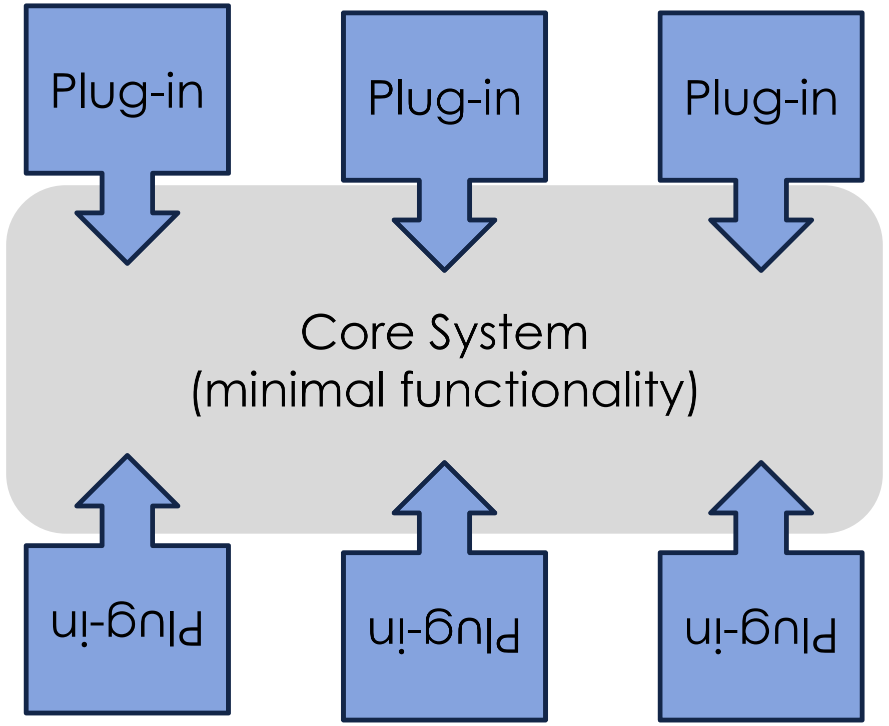
- The **Microkernel** architectural pattern applies to software systems that must be able to adapt to changing system requirements. It separates a minimal functional core from extended functionality and customer-specific parts.
- The microkernel also serves as a socket for plugging in these extensions and coordinating their collaboration.

ARCHITECTURAL PATTERNS: MICROKERNEL

- The **Microkernel** pattern defines five kinds of participating components:
  - Internal servers:
    - Implements additional services
    - Encapsulates some system specifics
  - External servers: Provides programming interfaces for its clients
  - Adapters:
    - Hides system dependencies such as communication facilities from the client
    - Invokes methods of external servers on behalf of clients
  - Clients: Represents an application
  - Microkernel:
    - Provides core mechanisms
    - Offers communication facilities
    - Encapsulates system dependencies
    - Manages and controls resources

ARCHITECTURAL PATTERN: MICROKERNEL

- **Examples:** Operating Systems, Browsers, Text Editors, …



ARCHITECTURAL PATTERN: REFLECTION

- The **Reflection** architectural pattern provides a mechanism for changing structure and behavior of software systems dynamically.
- It supports the modification of fundamental aspects, such as: type structures and function call mechanisms.
- In this pattern, an application is split into two parts:
  - A **meta level** provides information about selected system properties and makes the software self-aware.
  - A **base level** includes the application logic.

ARCHITECTURAL PATTERN: REFLECTION

- **Context:** Building systems that support their own modifications
- **Problem:** Design an architecture that is open to modification and extension. The system should be adapted to changing requirements on demand.
- **Solutions:** Make the software self-aware, and make selected aspects of its structure and behavior accessible for adaptation and change.

ARCHITECTURAL PATTERN: REFLECTION

- This architecture has two major parts: a meta level and a base level.
- The meta level provides a self-representation of the software to give it knowledge of its own structure and behavior. (consists of metaobjects that encapsulate and represent information about the software)
- The base level defines the application logic. Its implementation uses the metaobjects to remain independent of those aspects that are likely to change.

- **Example:** iOS apps, Android apps

REFERENCES

- Pattern-Oriented Software Architecture – A System of Patterns; F. Buschmann, R. Meunier, H. Rohnert, P. Sommerlad, M. Stal; Wiley