DCC / ICEx / UFMG

Architectural Patterns:

Distributed, Interactive, and Adaptable Systems

Eduardo Figueiredo

http://www.dcc.ufmg.br/~figueiredo

Architectural Patterns

- Distributed Systems
  - Client-Server
  - **Broker**
- Interactive Systems
  - **Model-View-Controller (MVC)**
  - Presentation-Abstraction-Control
- Adaptable Systems
  - **Microkernel**
  - Reflection

Broker

Broker

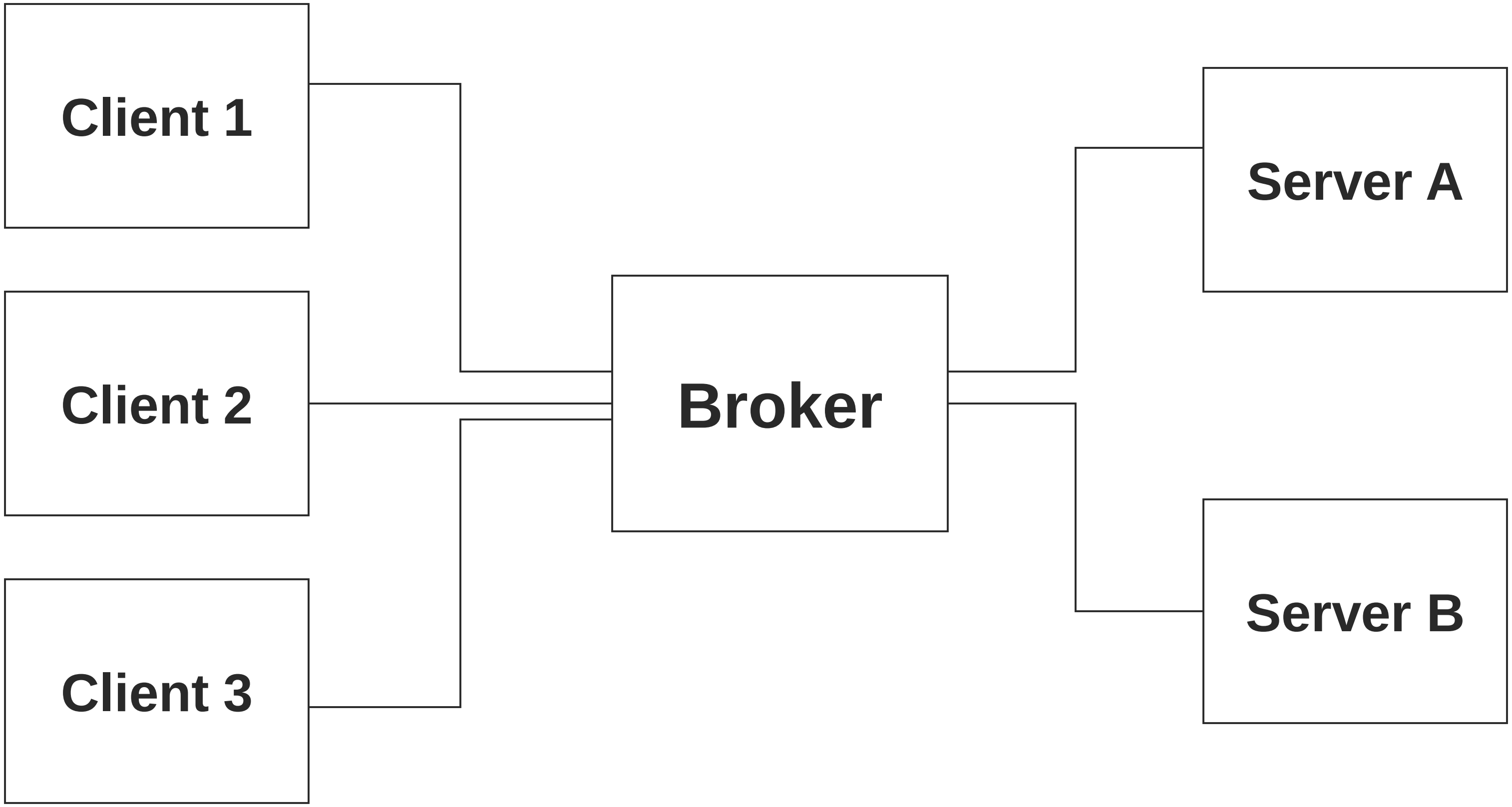
- Used to structure distributed systems with decoupled components
  - Components interact by remote service invocations

- The broker component is responsible for coordinating communication
  - Forwarding requests and transmitting results

Broker Roles

- **Servers** register themselves to the broker
  - They make their services available
- **Clients** do not know servers
  - They access the servers functionalities by sending requests to the broker
- **Broker** finds the appropriate server, forwarding the request to the server
  - And transmitting the results to the client

Broker Structure



Benefits

- Location Transparency
  - Clients do not need to know where servers are located
- Changeability and Extensibility
  - If a server changes but keeps its interface, it can be replaced by an equivalent server
- Interoperability between different Broker systems

Liabilities

- Restricted efficiency
  - Applications using a broker implementation are usually slower
- Lower fault tolerance
  - If the broker fails during the program execution, clients are unable to access the servers

Model-View-Controller (MVC)

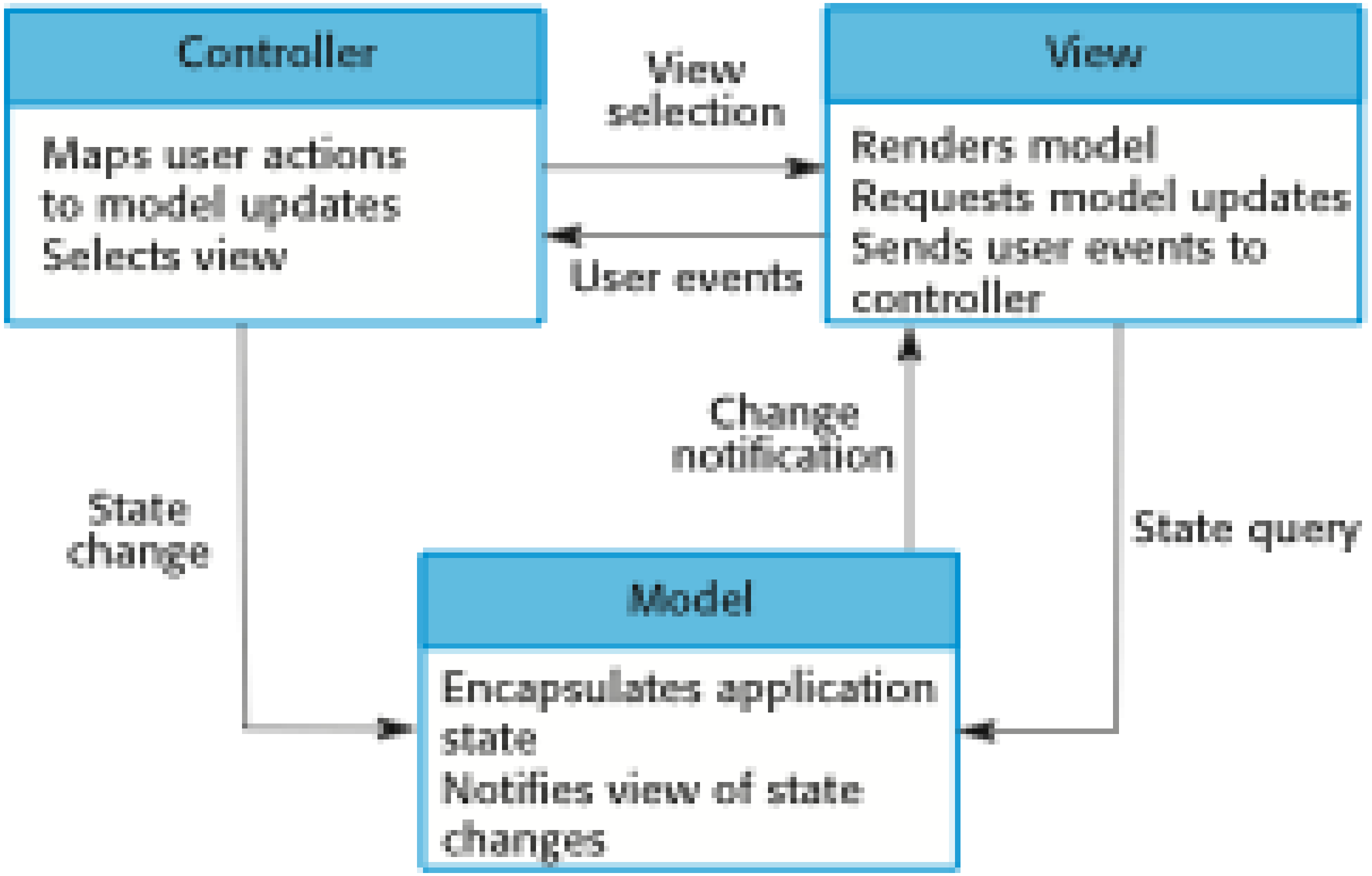
MVC Pattern

- This pattern is commonly used in interactive Web applications
- It organizes the system in three components
  - **Model**: it has the main functionalities and data
  - **View**: it is responsible for presenting data
  - **Controller**: it handles events from the view component

Highlights

- MVC separates presentation and interaction from the system data
  - Each component has its role, but they interact among them
- When should you use this pattern?
  - When there are several way to represent data and views have to be synchronized
  - When interaction requirements are complex or unknown

Representation of Roles



Controller

Maps user actions to model updates
Selects view

View

Renders model
Requests model updates
Sends user events to controller

Model

Encapsulates application state
Notifies view of state changes

Benefits

- It allows data to change independently of their presentation, and vice versa
- It supports the presentation of the same data in multiple ways
- Synchronized views.
  - Changes made in on presentation reflect both on the data and on other presentations

Liabilities

- It involve additional code complexity when the interactions are simple
- Communication overhead
  - Potential excessive number of updates
- Close coupling of views and controllers to a model

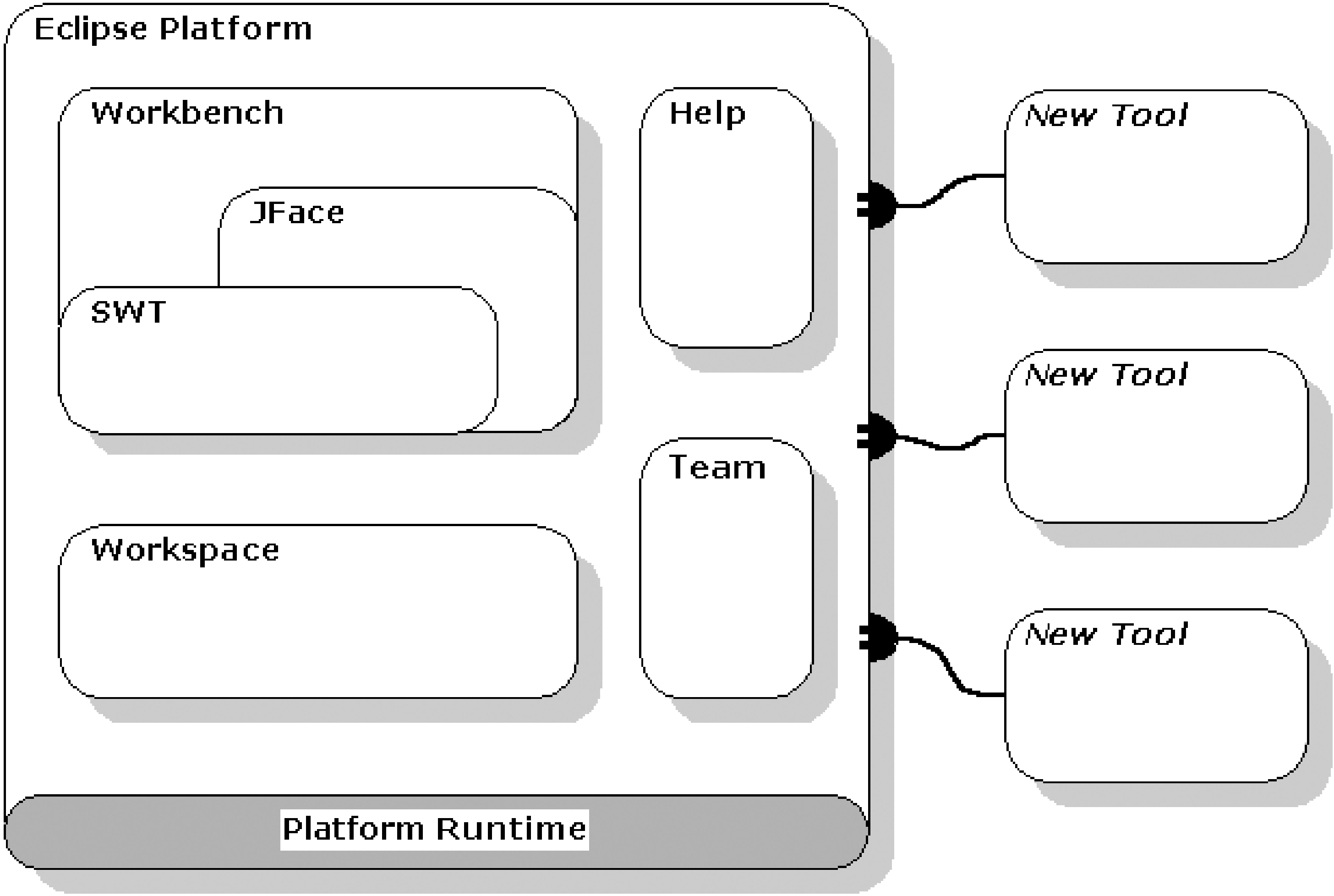
Microkernel

Microkernel

- This pattern applies to systems that must be able to adapt to changing requirements

- It separates a minimal functional core from extended functionality
  - The microkernel serves as a socket for plugging in new extensions

Example of Microkernel



Benefits

- Portability
  - A Microkernel system offers a high degree of portability
- Flexibility and extensibility
  - It can easily includes and removes functionalities (plug-ins)
- Scalability
  - Each new functionality tends to be simple and self-contained

Liabilities

- Performance
  - Overhead of communication in a microkernel system tends to be high
- Complex design and implementation
  - Developing a microkernel system is not trivial
  - Implement plug-ins also requires knowledge about the system structure

Bibliography

- F. Buschmann et al. **Pattern-Oriented Software Architecture: A System of Patterns**. John Wiley & Sons, 1996.
  - Chap. 2 Architectural Patterns