340CT
Software Quality and Process Management

Dr. Yih-Ling Hedley
Email: aa0817@coventry.ac.uk

Software Architecture and Applications

- Software architecture and design
  - Current software architecture application patterns
  - Real-world examples of applications
  - Evaluation of the patterns

Software Architecture 1

- Software architecture relates to a level of design concerned with issues of the **overall system structure**
  - Structural issues include:
    - **protocols** for communication, synchronization, and data access
    - composition of **design elements**
    - assignment of **functionality** to design elements and selection among design alternatives
    - **scaling and performance**
    - **physical distribution**

 Source: http://epf.eclipse.org/wikis/openup/

Software Application Architecture 1

- Software architecture patterns that
  - Define a **structured solution** that meets all of the technical and operational requirements
  - Optimise common quality attributes such as **performance, security, and manageability**
  - Focus on how the components within an application are used by, or interact with, others within the application
  - Select data structures and algorithms or the implementation details of individual components

 Source: Microsoft Application Architecture Guide, 2nd Edition

Software Application Architecture 2

- Current software application architecture patterns - examples:
  - **Layered architecture**
  - **Event-Driven architecture**
  - **Microkernel architecture**

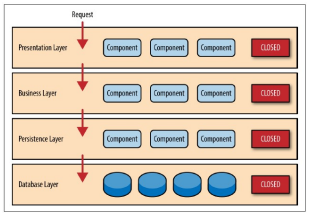
 Source: Software Architecture Patterns by Mark Richards 2015

Software Architecture Patterns 1

- **Layered Architecture**: Components within the layered architecture pattern are organized into **horizontal layers**, each with a specific role within the application
- Apply the concept: **layers of isolation**
  - the changes made in one layer of the architecture **without affecting components in other layers**
  - each layer is independent of the other layers, thereby having **little or no knowledge of the inner workings of other layers** in the architecture

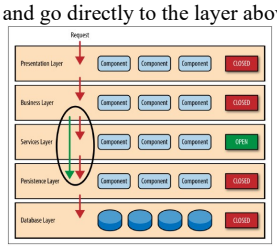
Software Architecture Patterns 1.1

- A basic layered architecture:
  - **Closed layered architecture** requires a request to go from a layer to another layer through the layer right above or below it



Software Architecture Patterns 1.2

- A basic layered architecture
  - **Open layered architecture** allows requests to bypass an open layer and go directly to the layer above or below it



Software Architecture Patterns 1.4

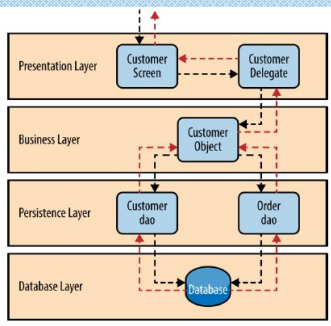
- **Layered Architecture:** Real World Scenario
  - **The customer screen receives a request** to get information for a customer and forwards that request onto the customer delegate module, responsible for knowing which modules in the **business layer** can process that request and what data it needs
  - The customer object in the business layer responsible for aggregating all of the information needed by calling the customer DAO (Data Access Object, used to separate low level data accessing API or operations from high level business logics) module in the **persistence layer** and the order DAO module for customer data and order information.

Software Architecture Patterns 1.4.1

- **Layered Architecture:** Real World Scenario
  - The customer and order DAOs execute SQL statements to retrieve the data and return the data to the customer object in the business layer.
  - Once the customer object receives the data, it aggregates the data and returns the information to the customer delegate, which returns that data to the customer screen to be presented to the user.

Software Architecture Patterns 1.4.2

- **Layered Architecture:** Example: 4 layers



Software Architecture Patterns 1.5.1

- **Evaluation of Layered Architecture:**
  - **Ease of development: High** – Ease of development is relatively high, as the pattern is well known whilst not overly complex to implement.
    - particularly for most business applications in which companies tend to develop applications by separating skill sets by layers (presentation, business, database layers)

Software Architecture Patterns 1.5.2

- **Evaluation of Layered Architecture:**
  - **Testability: High** – As components reside in their specific layers in the architecture, other layers can be tested relatively easy.
    - For example, a developer can use a stub for presentation component or screen to isolate testing within a business component, as well as producing a stub for the business layer to test certain screen functionality.

Activity 1

- Evaluate Layered Architecture in terms of following (a rating of high or low):
  - Performance
- **Performance: Low** – Generally, the pattern does not lend itself to high-performance applications due to the inefficiencies of having to go through multiple layers of the architecture to fulfil a business request.

Activity 1.2

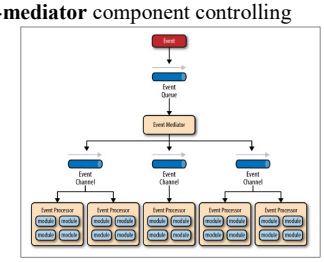
- Evaluate Layered Architecture in terms of following (a rating of high or low):
  - Scalability
- **Scalability:**
  - **Low** if the applications based on this architecture pattern are tightly coupled, then it is generally difficult to scale
  - **High** if a layered architecture distributes its layers over multiple physical tiers, then it can improve scalability

Software Architecture Patterns 2

- **Event-Driven Architecture**
  - A distributed architecture pattern, which consists of two main topologies:
    - **Mediator**: to coordinate events through **a central mediator**
    - **Broker**: to coordinate events through **individual event processors**

Software Architecture Patterns 2.1

- **Mediator:** four components: **event queues, an event mediator, event channels, and event processors**.
  - A **central event-mediator** component controlling the initial event



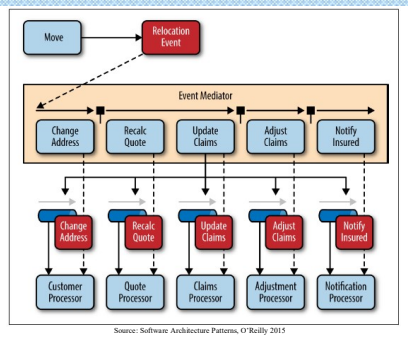
Source: Software Architecture Patterns, O'Reilly 2015

Software Architecture Patterns 2.1.1

- **Mediator:** Real World Scenario
  - The move requests starts with **a relocation event** as the initial event (the event mediator contains the steps involved in processing the relocation event)
  - For each initial event step, the event mediator creates a processing event (e.g., change address), sends the **change address processing event** to its **change address event channel** and waits for the processing event to be processed by the **customer process event processor**
  - The process continues until all of the steps in the initial event have been processed.

Software Architecture Patterns 2.1.2

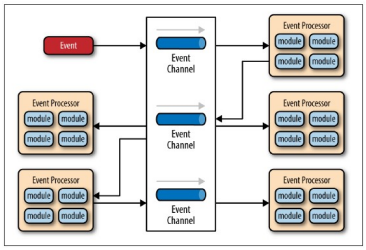
- **Mediator:** Real World Scenario



Source: Software Architecture Patterns, O'Reilly 2015

Software Architecture Patterns 2.2

- **Broker:** each event-processor component is responsible for processing an event



Source: Software Architecture Patterns, O'Reilly 2015

Software Architecture Patterns 2.2.1

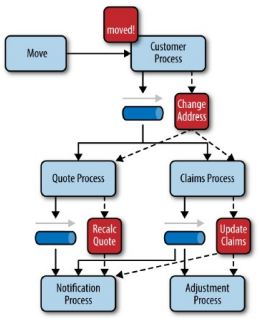
- **Broker:** Real World Scenario
  - The **customer-process component** receives the move event directly, changes the customer address, and publishes an update change address event to the system
  - The **quote processor component** recalculates the new auto insurance rates and publishes an update recalc quote event

Software Architecture Patterns 2.2.2

- **Broker:** Real World Scenario
  - The **claims processing component**, also receives the change address event, and updates an outstanding insurance claim and publishes an event to the system as an update claim event.
  - These new events are then received by other event processor components, and the event chain continues through the system until there are no more events are published for the initiating event.

Software Architecture Patterns 2.3.3

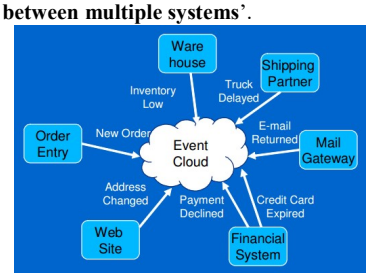
- **Broker:** Example



Source: Software Architecture Patterns, O'Reilly 2015

Software Architecture Patterns 2.4

- **Event Cloud** – referred to '**the interchange of many events between multiple systems**'.



Source: David Luckham, The Power of Events

Software Architecture Patterns 2.5

- **Evaluation of Event-Driven Architecture:**
  - **Ease of development: Low** – Development can be complicated, due to:
    - the **asynchronous** nature of the pattern
    - the need for more a**dvanced error handling** conditions within the code for unresponsive and failed event processors.

Software Architecture Patterns 2.5.1

- **Evaluation of Event-Driven Architecture:**
  - **Testability: Low**
    - no overly difficult for individual unit testing,
    - **specialized testing clients or tools** would be required to generate events.
    - testing is also more difficult and complicated by the asynchronous nature of this pattern.

Activity 2

- Evaluate Event-Driven Architecture in terms of following (a rating of high or low):
  - Performance

- **Performance: High** – generally the pattern achieves high performance through its **asynchronous capabilities**; that is, the ability to perform decoupled, parallel asynchronous operations, however with the cost of queuing and de-queuing messages (in the messaging infrastructure)

Activity 2.1

- Evaluate Event-Driven Architecture in terms of following (a rating of high or low):
  - Scalability

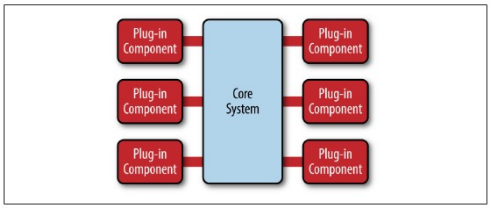
- **Scalability: High** – achieved through:
  - highly independent and decoupled event processors
  - each event processor can be scaled separately, allowing for fine-grained scalability.

Software Architecture Patterns 3

- **Microkernel Architecture: (also plug-in architecture pattern)**
  - Applied to product-based applications (which is **packaged** and made available for download in versions as a **third-party product**)
  - Allowing to add additional application features as plug-ins to the core application
  - consists of two types of architecture components: a **core system and plug-in modules.**

Software Architecture Patterns 3.1

- **Architecture Components:**
  - Core system
  - Plug-in modules



Source: Software Architecture Patterns, O'Reilly 2015

Software Architecture Patterns 3.2

- **Core Components:**
  - Contains only the **minimal functionality** required by the system.
  - Example: for business-applications, the core system contains the **general business logic** without custom code for special cases, special rules, or complex conditional processing

Software Architecture Patterns 3.3

- **Plug-in Components:**
  - **stand-alone, independent components** that contain specialized processing, additional features, and custom code to enhance or extension of the core system to produce additional business capabilities.

Software Architecture Patterns 3.4

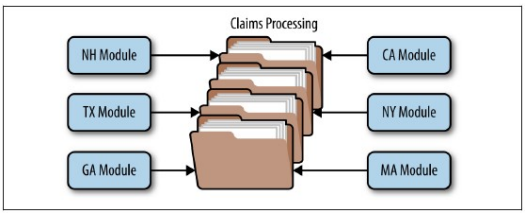
- **Core and Plug-in Components:**
  - Communication through a **plug-in registry**, which contains information about each plug-in module, including:
    - **name**
    - **data contract**
    - **remote access protocol details** (depending on how the plug-in is connected to the core system).

Software Architecture Patterns 3.5

- **Microkernel Architecture:** Real World Scenario
  - The core system for claims processing, contains the basic business logic required to process a claim
  - Each plug-in module contains the specific rules to be implemented using custom source code or separate rules engine instances.
    - It **separates state-specific rules and processing from the core claims system** and can be added, removed, and changed with little or no effect on the rest of the system or other plug-in modules.

Software Architecture Patterns 3.5.1

- **Microkernel Architecture:** Real World Scenario



Source: Software Architecture Patterns, O'Reilly 2015

Software Architecture Patterns 3.6

- **Evaluation of Microkernel Architecture:**
  - **Ease of development: Low**
    - rather complex to implement, as thoughtful design and contract consideration is required, including:
      - ✓**Contract versioning**
      - ✓internal plug-in **registries**
      - ✓plug-in **connectivity**

Software Architecture Patterns 3.6.1

- **Evaluation of Microkernel Architecture:**
  - **Testability: High**
    - Plug-in modules can be tested in isolation
    - Stubs can be easily provided by the core system to demonstrate or prototype a particular feature with little or no change to the core system.

Activity 3

- Evaluate Microkernel Architecture in terms of following (a rating of high or low):
  - Performance
- **Performance: High**
  - generally perform well as applications can be customised and streamlined to only include those features required.
  - Example: the **JBoss** Application Server, removing expensive non-used features such as remote access, messaging, and caching that consume memory, CPU, and threads and slow down the server.

Activity 3.1

- Evaluate Microkernel Architecture in terms of following (a rating of high or low):
  - Scalability
- **Scalability: Low**
  - overall not highly scalable , due to most **implementations that are product based (packaged) and generally smaller in size**, which are implemented as single units

Activity 4

- Identify the architecture pattern for the example below:
  - Anti-lock braking embedded system (an automobile safety system that allows the wheels on a motor vehicle to maintain contact with the road surface according to driver inputs while braking, preventing the wheels from ceasing rotation)
- **Anti-lock braking embedded system - Event driven Architecture**

Activity 4.1

- Identify the architecture pattern for the example below:
  - Eclipse Integrated Developer Environment (IDE)
- **Eclipse IDE - Microkernel Architecture:**
  - The basic Eclipse product (core system) provides an development editor
  - plug-ins can be added to provide a highly customizable and useful product, e.g. **Eclipse SQL Explorer,** a thin SQL client to query and browse any JDBC compliant databases

Activity 4.2

- Identify the architecture pattern for the example below:
  - Internet browsers
- **Internet browsers - Microkernel Architecture**
  - plug-ins add additional capabilities can be added to the basic browser (i.e., core system), e.g. viewing Flash animations or PDF documents

Activity 4.3

- Identify the architecture pattern for the example below:
  - Application Programming Interface (API): encapsulates frequently-used functionality through high level function specifications

- **API - Layered Architecture :**
  - An Application Programming Interface (API) is a layer that encapsulates lower layers of frequently-used functionality. usually a collection of function specifications. Examples:
    - **Java APIs**: Android API, Java Persistence API
    - **OpenGL (Open Graphics Library)**: cross-platform graphics API