# Architectural Patterns and Tactics

# Overview

- Architectural patterns:
  - Package common combinations of design decisions
  - Has known ways of reusing
  - Describes a class of architectures
- Tactics are simpler and more atomic – they capture a step to take for a particular Quality Attribute to change behaviour with respect to that QA.
- Tactics can be seen as the building blocks of patterns

# Architectural Patterns

- An architectural pattern comprises:
  - A **context** that provides the frame for a *problem.*
  - A **problem** that is a generalised description of a class of problems often with QA requirements that should be met.
  - A **solution** that is suitably generalised in the same way as the problem. A solution:
    - Describes the architectural structures that solve the problem that balances the forces at work in the solution.
    - The solution might be static, runtime or deployment oriented.

# Architectural Solution

- An architectural solution comprises:
  - A set of element types (e.g. data repositories, processes, objects)
  - A set of interaction mechanisms (e.g. static relationships, events and handlers, …)
  - The relationship between the elements.
  - Constraints on the relationship between elements, element behaviour and communication mechanisms
- This wikipedia page covers the basic architectural patterns: https://en.wikipedia.org/wiki/List_of_software_architecture_styles_and_patterns

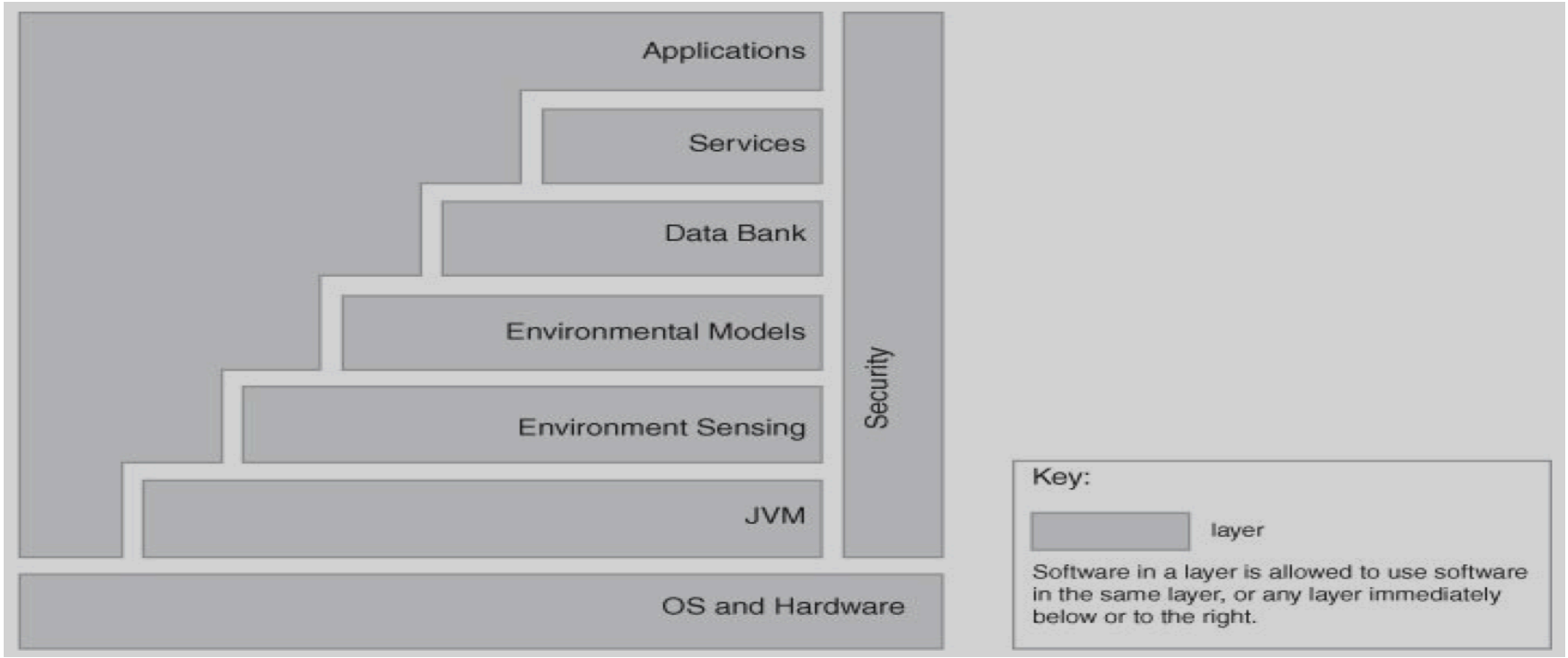
# Architectural Patterns

- We consider:
  - Static Patterns designed to solve issues around development and maintenance of the code base.
  - Connector and component models that consider patterns of interaction, monitoring at runtime and responses to stimuli.
  - Deployment patterns that consider the allocation of elements to hardware resources and the availability of resources in the system

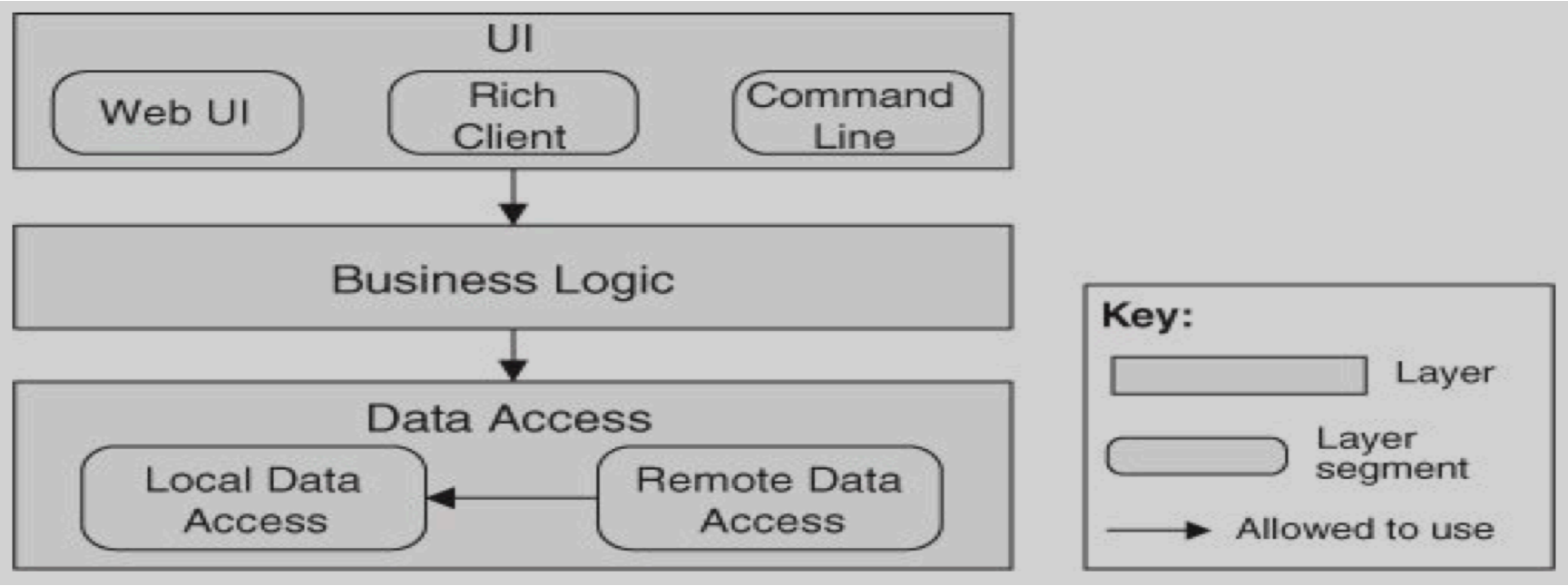
# Static Pattern: Layered Architecture

| | |
|---|---|
| Overview | The layered pattern defines layers (groupings of modules that offer a cohesive set of services) and a unidirectional *allowed-to-use* relation among the layers. The pattern is usually shown graphically by stacking boxes representing layers on top of each other. |
| Elements | *Layer*, a kind of module. The description of a layer should define what modules the layer contains and a characterization of the cohesive set of services that the layer provides. |
| Relations | *Allowed to use*, which is a specialization of a more generic *depends-on* relation. The design should define what the layer usage rules are (e.g., "a layer is allowed to use any lower layer" or "a layer is allowed to use only the layer immediately below it") and any allowable exceptions. |
| Constraints | ▪ Every piece of software is allocated to exactly one layer.<br>▪ There are at least two layers (but usually there are three or more).<br>▪ The *allowed-to-use* relations should not be circular (i.e., a lower layer cannot use a layer above). |
| Weaknesses | ▪ The addition of layers adds up-front cost and complexity to a system.<br>▪ Layers contribute a performance penalty. |

# Layers: Clear Access Rules



# Clear Access Rules



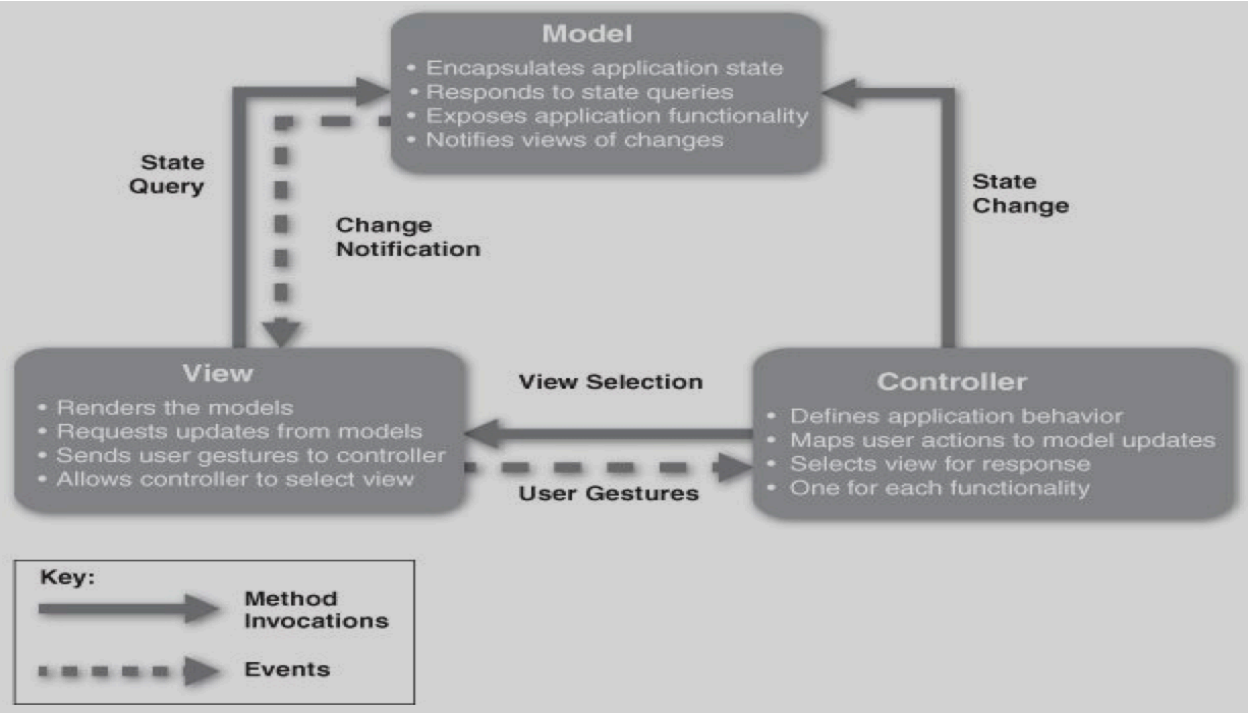
# Component Connector: Model-View-Controller

- Context:
  - Here we have a situation where the user interface is subject to continuing change either to meet the needs of the application or diversity in the user group.
- Problem:
  - Isolating the UI functionality from the Application functionality.
  - Maintaining multiple views in the presence of change in the underlying data.

# MVC: Solution

| | |
|---|---|
| Overview | The MVC pattern breaks system functionality into three components: a model, a view, and a controller that mediates between the model and the view. |
| Elements | The *model* is a representation of the application data or state, and it contains (or provides an interface to) application logic.<br>The *view* is a user interface component that either produces a representation of the model for the user or allows for some form of user input, or both.<br>The *controller* manages the interaction between the model and the view, translating user actions into changes to the model or changes to the view. |
| Relations | The *notifies* relation connects instances of model, view, and controller, notifying elements of relevant state changes. |
| Constraints | There must be at least one instance each of model, view, and controller.<br>The model component should not interact directly with the controller. |
| Weaknesses | The complexity may not be worth it for simple user interfaces.<br>The model, view, and controller abstractions may not be good fits for some user interface toolkits. |

# MVC Pattern


Model
• Encapsulates application state
• Responds to state queries
• Exposes application functionality
• Notifies views of changes
State Query
Change Notification
State Change
View
• Renders the models
• Requests updates from models
• 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
Key:
Method Invocations
Events

# Other Component-Connector Patterns

- Pipe and Filter Pattern
- Broker Pattern
- Client-Server Pattern
- Peer-to-Peer Pattern
- Service-Oriented Architecture Pattern
- Publish-Subscribe Pattern
- Shared Data Pattern

# Deployment/Allocation Patterns

- Here we are looking at contexts:
  - Where we are concerned with resource use
  - We might consider flexible deployment of resource
  - The QAs we care about are sensitive to the pattern of deployment and the use of resources.

# Allocation: Map-Reduce Pattern

- Context:
  - We have large quantities of data we wish to treat as “population” data.
  - This encourages an approach that involves significant amounts of independent processing.
- Problem:
  - Where for ultra-large data sets doing some individual processing to a portion of the data set and then sorting and analyzing grouped data, map reduce provides a simple way of doing this processing.

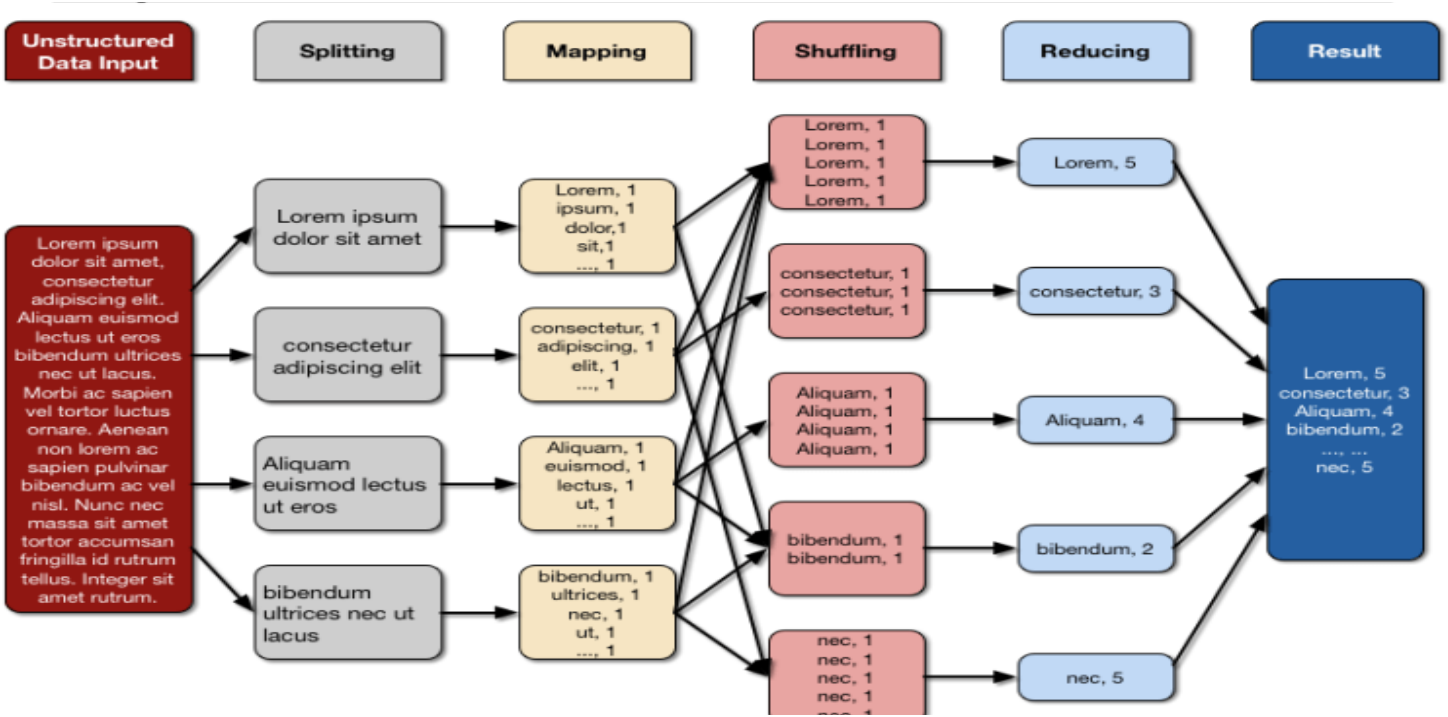
# Map Reduce - Solution

| | |
|---|---|
| Overview | The map-reduce pattern provides a framework for analyzing a large distributed set of data that will execute in parallel, on a set of processors. This parallelization allows for low latency and high availability. The map performs the *extract* and *transform* portions of the analysis and the reduce performs the *loading* of the results. (*Extract-transform-load* is sometimes used to describe the functions of the map and reduce.) |
| Elements | *Map* is a function with multiple instances deployed across multiple processors that performs the extract and transformation portions of the analysis.<br>*Reduce* is a function that may be deployed as a single instance or as multiple instances across processors to perform the load portion of extract-transform-load.<br>The *infrastructure* is the framework responsible for deploying map and reduce instances, shepherding the data between them, and detecting and recovering from failure. |

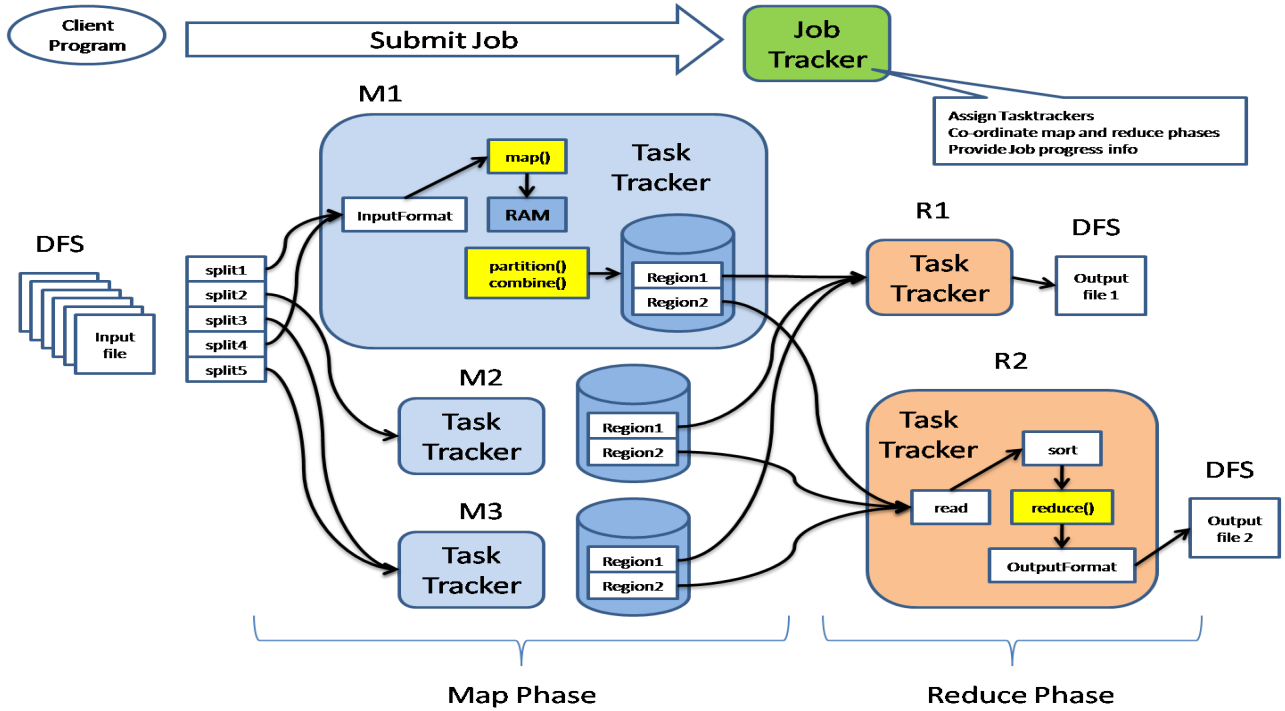
# Map-Reduce: Solution

| | |
|---|---|
| Relations | *Deploy on* is the relation between an instance of a map or reduce function and the processor onto which it is installed.<br>*Instantiate, monitor, and control* is the relation between the infrastructure and the instances of map and reduce. |
| Constraints | The data to be analyzed must exist as a set of files.<br>The map functions are stateless and do not communicate with each other.<br>The only communication between the map instances and the reduce instances is the data emitted from the map instances as <key, value> pairs. |
| Weaknesses | If you do not have large data sets, the overhead of map-reduce is not justified.<br>If you cannot divide your data set into similar sized subsets, the advantages of parallelism are lost.<br>Operations that require multiple reduces are complex to orchestrate. |

# Map Reduce – Specific Example



# Map Reduce Pattern



# Other Allocation Patterns

- Multi-tier architecture pattern
- Cloud architectures

# Relationships Between Tactics and Patterns: Patterns package Tactics

| | Modifiability | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Increase Cohesion | | Reduce Coupling | | | | | Defer Binding Time | | | |
| Pattern | Increase Semantic Coherence | Abstract Common Services | Encapsulate | Use a Wrapper | Restrict Comm. Paths | Use an Intermediary | Raise the Abstraction Level | Use Runtime Registration | Use Startup-Time Binding | Use Runtime Binding |
| Layered | X | X | X | | X | X | X | | | |
| Pipes and Filters | X | | X | | X | X | | | X | |
| Blackboard | X | X | | | X | X | X | X | | X |
| Broker | X | X | X | | X | X | X | X | | |
| Model View Controller | X | | X | | | X | | | | X |
| Presentation Abstraction Control | X | | X | | | X | X | | | |
| Microkernel | X | X | X | | X | X | | | | |
| Reflection | X | | X | | | | | | | |

# Summary

- We have seen the use of architectural patterns.
- Patterns are more problem focussed than Tactics
- Pattersn can help in static, dynamic and deployment contexts.
- Patterns capture experience – they are discovered NOT invented…
- Next time we will consider some other patterns and their relationship to tactics and analysis.