UNIVERSITY OF TECHNOLOGY SYDNEY

ARCHITECTURAL PATTERNS

Tom McBride

Software Architecture

UTS:
ENGINEERING AND INFORMATION TECHNOLOGY

UTS CRICOS PROVIDER CODE: 00099F

Contents

- Introduction and discussion of patterns
- Sources of patterns and information about patterns
- Review the main architectural patterns:
  - Pipes and filters,
  - Layer,
  - Micro-kernel,
  - Model-View-Controller (MVVC, MVP)
  - Action – Domain - Responder
  - Message bus
- Review the main design patterns;
  - Façade
  - Proxy
  - Broker

Software Architecture Patterns

Origins

Their role in software architecture

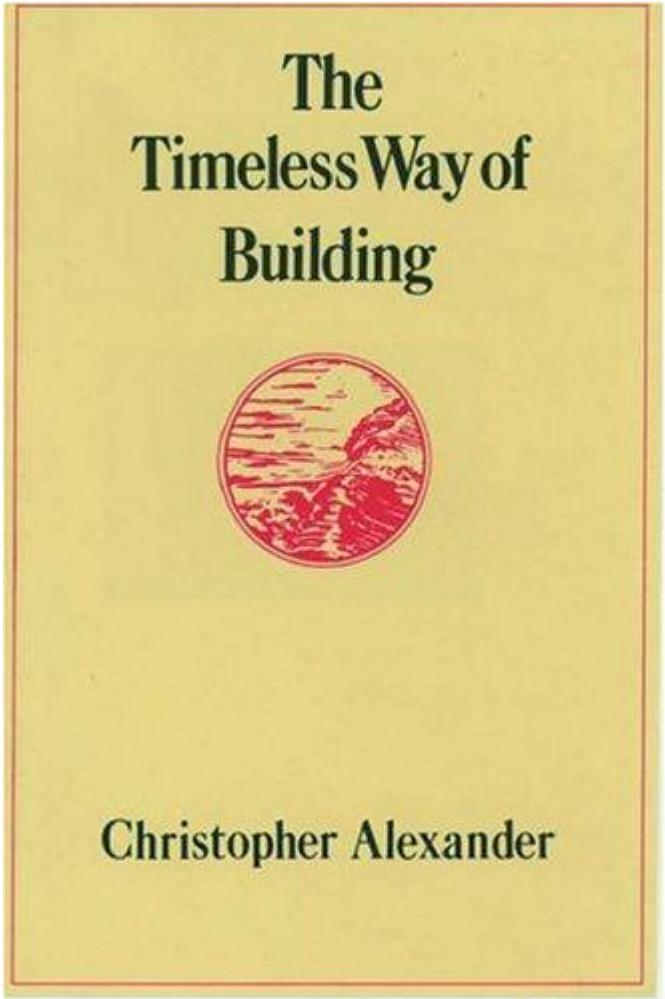
THE CONCEPT OF PATTERNS

The cognition of problem solving

- Experts solve problems differently from novices
- Experts recognise abstractions of the situation and apply adaptations of known solutions
- Recognition primed decision-making used in dynamic situations like fire-fighting, business decision-making, software development
  - More efficient to reuse what works than to develop something from first principles each time.
  - We build up a library of “schemas”
- Software development
  - Schema proposed in early AI work (1980s)
  - Patterns formalised a variety of schema

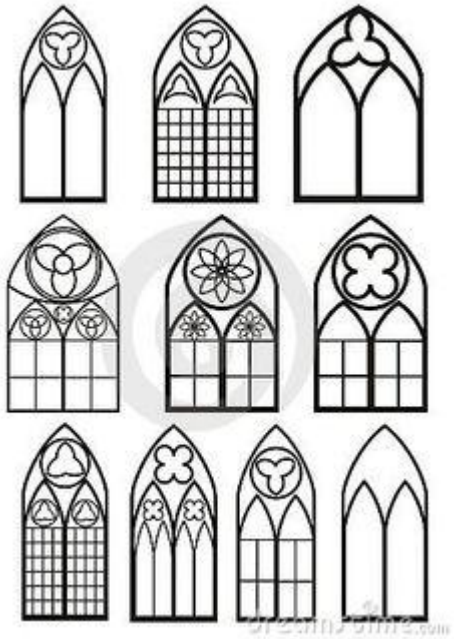
It all started with Christopher Alexander

- Christopher Alexander wrote “A timeless way of building”
- Arose from studies in urban design
- The same problem tends to be solved in the same way
- If you can recognise the abstracted problem then you can apply the abstracted solution – the pattern



Patterns are generalised solutions to recurring problems

- A pattern describes a recurring problem that occurs in a given context and, based on a set of guiding forces, recommends a solution.
- The solution is usually a simple mechanism, a collaboration between two or more classes, objects, services, processes, threads, components, or nodes that work together to resolve the problem identified in the pattern.



Other contributors and sources

- Books
  - Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995), *Design Patterns: Elements of Reusable Object-Oriented Software,* Addison-Wesley Publishing Company, Reading, MA
  - Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P. and Stal, M. (1996), *Pattern-Oriented Software Architecture: A system of patterns,* John Wiley & Sons, New York
  - Schmidt, D., Stal, M., Rohnert, H. and Buschmann, F. (2000), *Pattern-Oriented Software Architecture, Volume 2, Patterns for Concurrent and Networked Objects,* John Wiley & Sons
  - Alur, D., Crupi, J. and Malks, D. (2003), *Core J2EE Patterns: Best practices and design strategies,* Prentice Hall PTR, Upper Saddle River, 2nd ed
  - Larman, C. (2002), *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and the Unified Process.,* Prentice-Hall Inc, Upper Saddle River
- Web sites
  - Microsoft patterns and practices at http://msdn.microsoft.com/en-us/practices/default.aspx
  - Core J2EE patterns at http://www.corej2eepatterns.com/Patterns2ndEd/index.htm
  - The Wikipedia page for Software Architecture has links to many architectural patterns at http://en.wikipedia.org/wiki/Software_architecture
  - The Wikipedia page for software design patterns has references and links to a lot of software design patterns at http://en.wikipedia.org/wiki/Software_design_patterns
  - http://www.vico.org/pages/PatronsDisseny.html
  - Portland Patterns Repository contains more than architecture and design patterns http://c2.com/ppr/
- Other sources
  - IBM's Rational Rose has some patterns available to be invoked from within the tool.

A pattern has four essential elements

- **The pattern name**
  - The pattern name is a handle used to describe the problem, its solution and consequences in a few words. Rather like “Dances with wolves” as a name. Finding good names for patterns is generally quite difficult.
- **The problem**
  - The problem describes when to apply the pattern. It explains the problem and its context. Sometimes the problem will include a list of conditions that must be met before it makes sense to apply this pattern.
- **The solution**
  - The solution describes the elements that make up the design, their relationships and responsibilities, and collaborations. The solution doesn’t describe a particular concrete design or implementation, because a pattern is like a template that can be applied in many different situations. Instead, a pattern provides and abstract description of a design problem and how a general arrangement of elements solves it.
- **The consequences**
  - The consequences are the results and trade-offs of applying the pattern. Though the consequences are unvoiced when we describe design decisions, they are critical for evaluating design alternatives and for understanding the costs and benefits of applying the pattern.

Patterns come on a variety of scales and abstractions

- Some patterns help in structuring a software system into sub-systems. Other patterns support the refinement of sub-systems and components, or the relationship between them. Further patterns help in implementing particular design aspects in a specific programming language. Patterns also range from domain dependent ones, such as those for decoupling interacting components, to patterns addressing domain specific aspects such as transaction policies in business applications, or call routing in telecommunications.
- Patterns can be grouped into;
  - Architectural patterns
  - Design patterns
  - Idioms
- However, different vendors have different classification schemes.
  - J2EE has Presentation tier, Business tier and Integration tier patterns.
  - Microsoft provides access to their patterns in different ways
    - Application Type
    - Guidance type
    - Platform and developer tool version
    - Quality attribute

Summary

- A pattern describes a recurring problem that occurs in a given context and, based on a set of guiding forces, recommends a solution.
- A pattern has four essential elements
  - Name
  - Problem
  - Solution
  - Consequences
- Patterns have been used in software architecture and programming very successfully.

ARCHITECTURAL PATTERNS

Architectural patterns

- An architectural patterns expresses a fundamental structural organisation schema for software systems. It provides a set of predefined sub-systems, specifies their responsibilities, and includes rules and guidelines for organising the relationships between them.

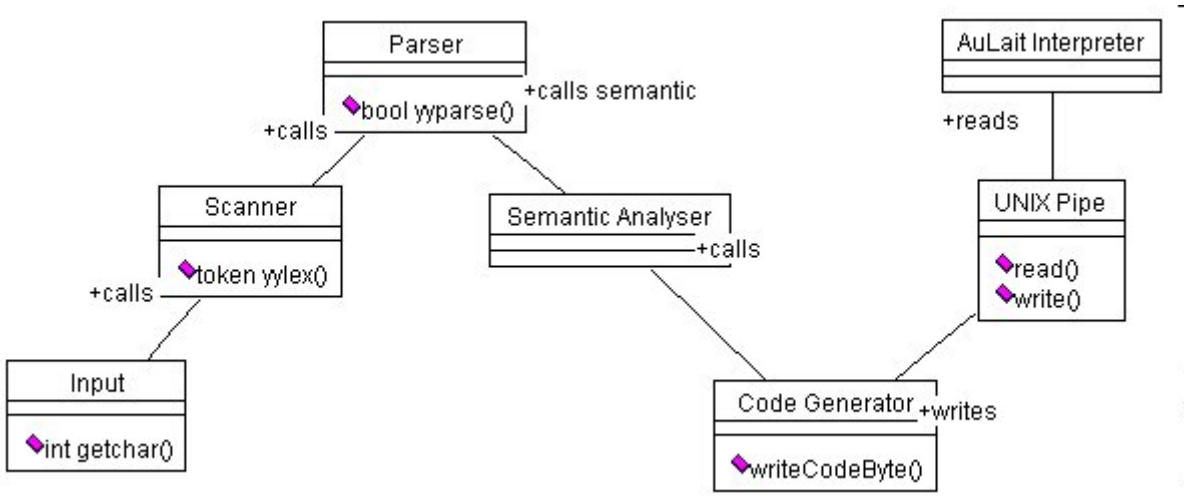
- Common architectural patterns
  - Pipes and filters
  - Layer
  - Microkernel
  - Model-View-Controller
  - Action – Domain - Responder
- Other architectural patterns
  - Blackboard

PIPES AND FILTERS

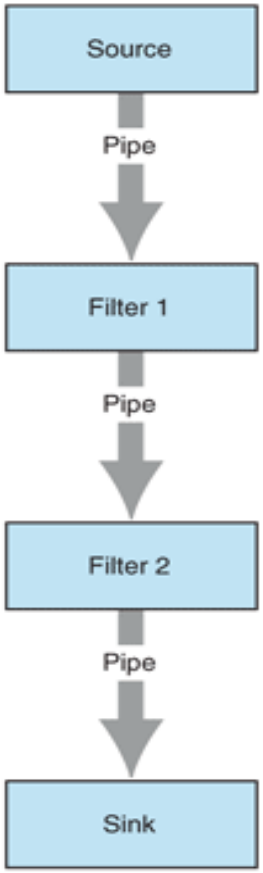
Pattern: Pipes and filters

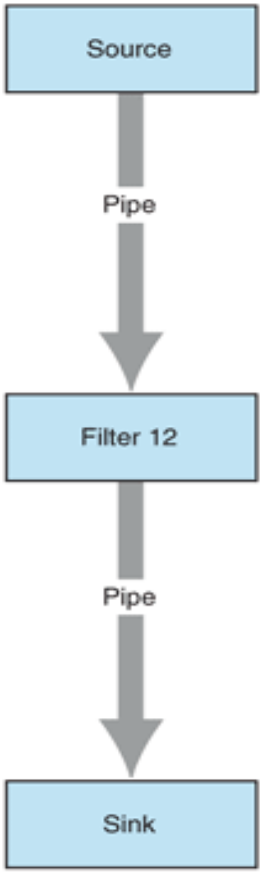
- Context
  - Data can arrive in different formats
- Problem
  - Need to apply a series of ordered but independent computations
- Forces
  - Maintain the precise order of operations.
  - Preserve the order of shared data among all operations.
  - Consider the introduction of parallelism, in which different step-operations can process different pieces of data at the same time.
  - Distribute process into similar amounts among all step-operation.
  - Improvement in performance is achieved when execution time decreases.
- Solution
  - Implement the transformations by using a sequence of filter components, where each filter component receives an input message, applies a simple transformation, and sends the transformed message to the next component. Conduct the messages through *pipes* that connect filter outputs and inputs and that buffer the communication between the filters.

Pipes and Filters - UML



Pipes and filters - diagram





LAYER

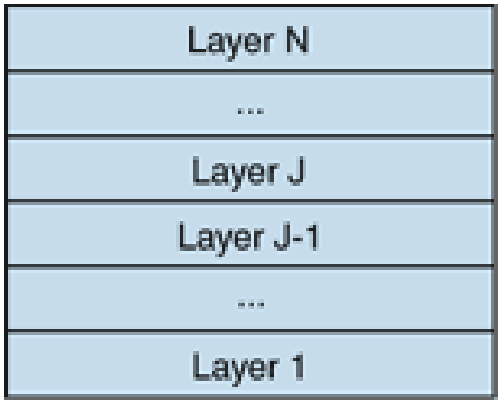
Pattern: Layer

- Context
  - You are designing a complex enterprise application that is composed of a large number of components across multiple levels of abstraction.
- Problem
  - The problem is usually that the system contains a mix of high level and low level issues where the high level operations rely on the low level ones.
- Forces
  - Localize changes.
  - Separate concerns among components.
  - Components should be reusable by multiple applications.
  - Individual components should be cohesive.
  - Unrelated components should be loosely coupled.
- Solution
  - Separate the components of your solution into layers. The components in each layer should be cohesive and at roughly the same level of abstraction. Each layer should be loosely coupled to the layers underneath. The upper level achieves its goals by calling a number of “services” from the lower level.

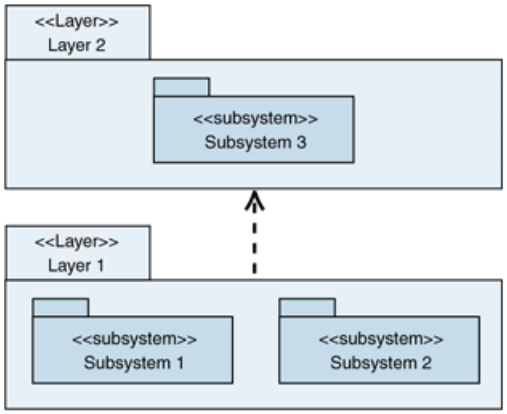
Layers are not components

- Common mistake is to treat a layer as a component.
- Something in one layer achieves its objective by calling on services from the layer below it.
- The higher layer coordinates the sequence of services.
- Several different parts of the same layer call on the lower layer services in varied sequences to achieve their specific objectives.
- Example is the services provided by a data access layer – create, read, update, delete, commit, etc.

Layer pattern - diagrams

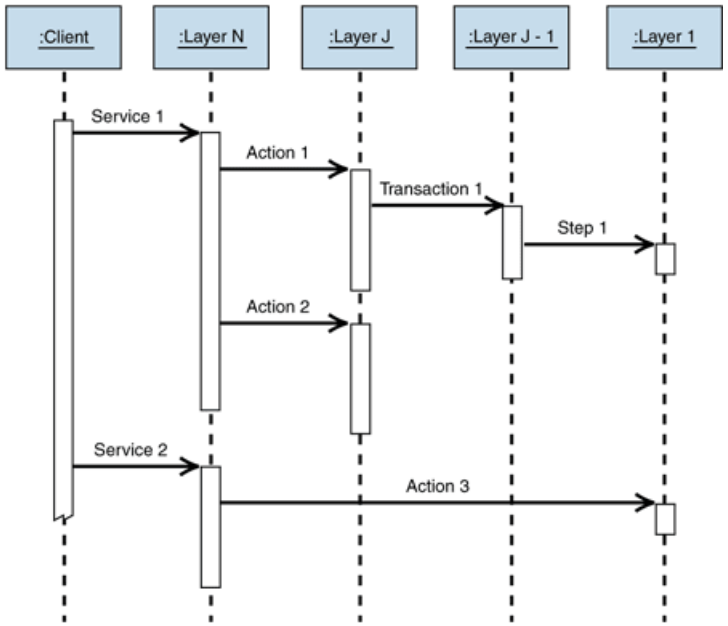


General layer pattern



Layers can include subsystems and components

Layer pattern – sequence diagram



MICROKERNEL

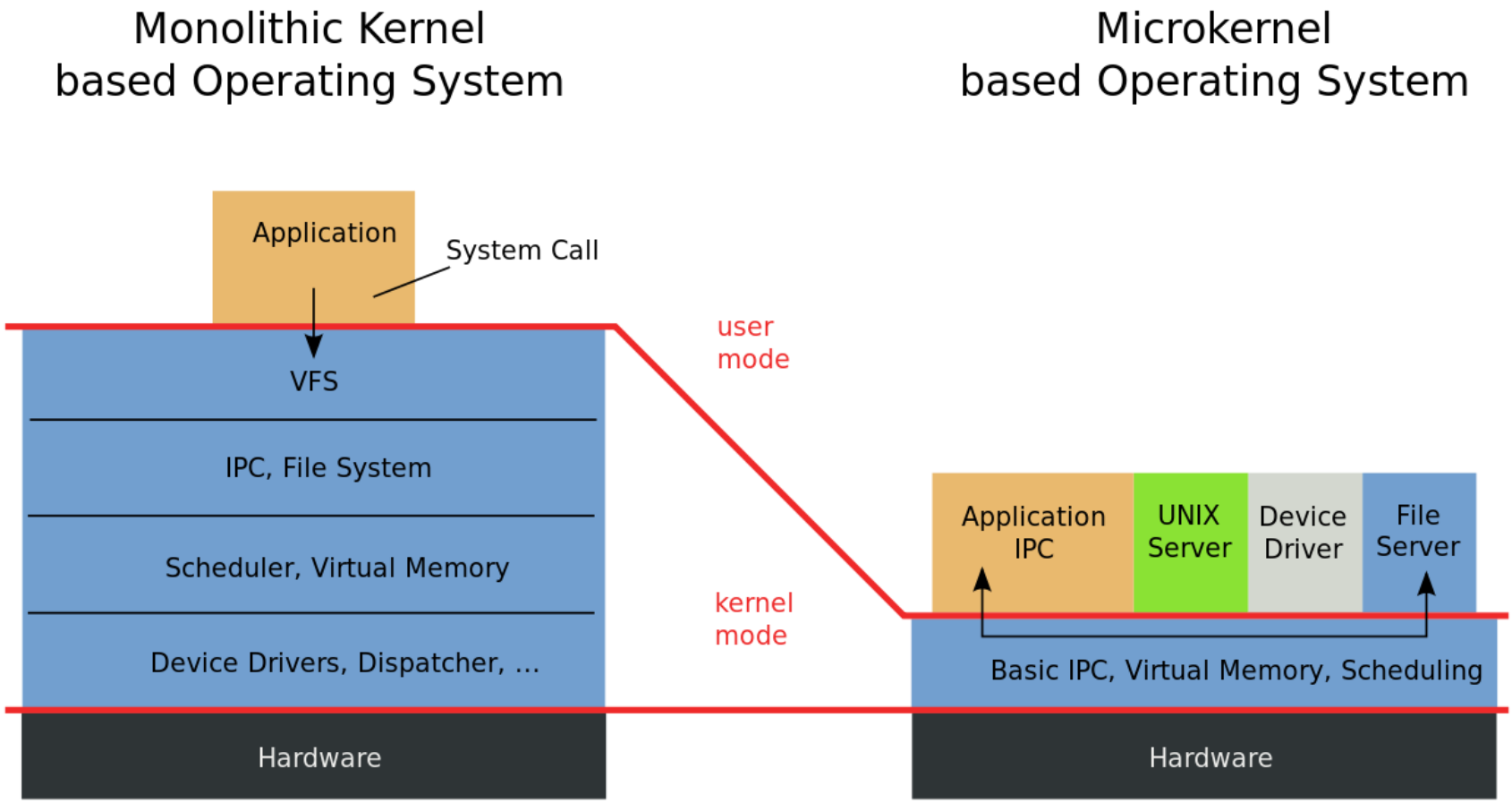
Pattern: Microkernel

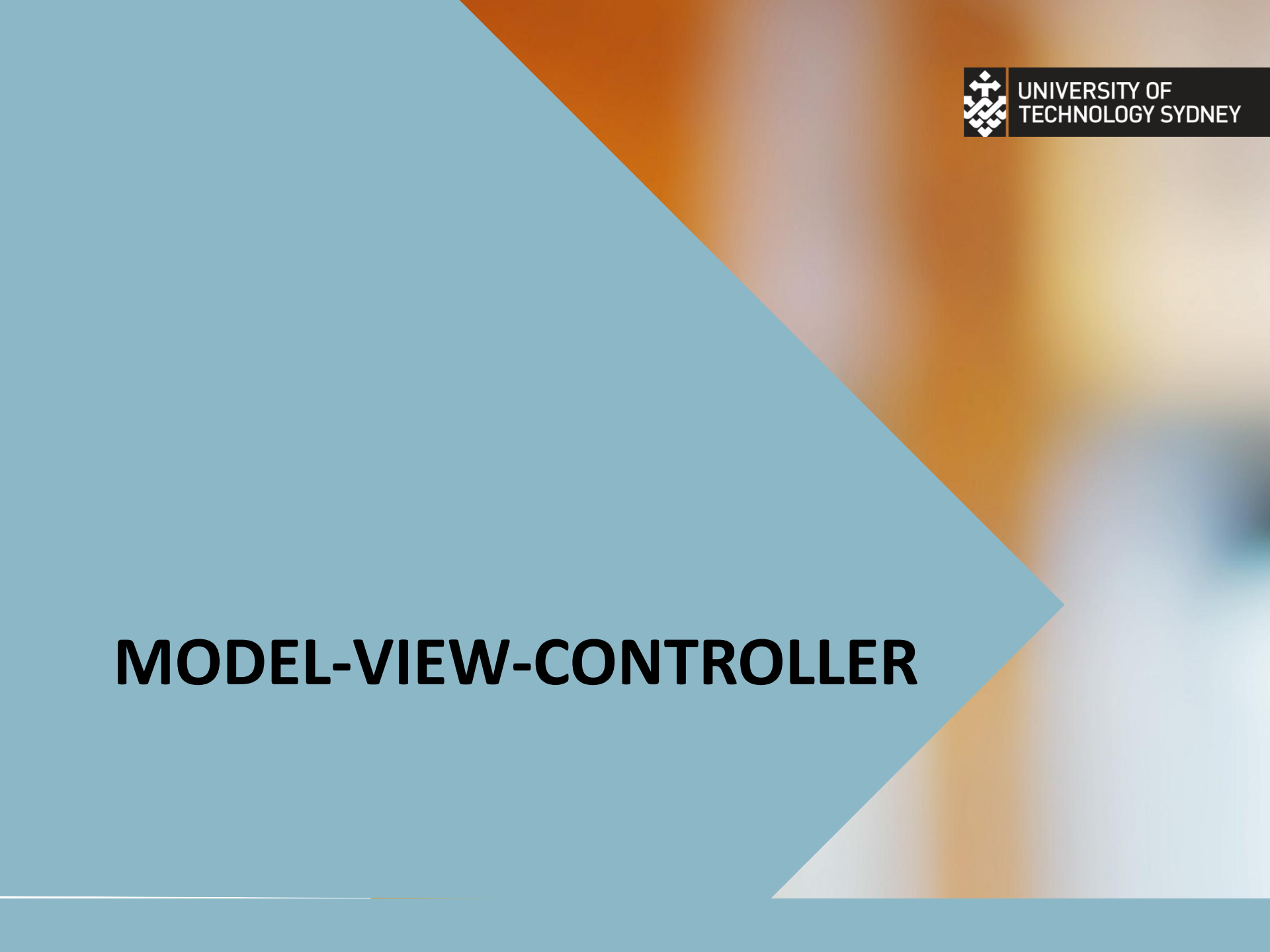
- Context
  - Several applications use similar programming interfaces that build on the same core functionality that may be deployed on several different platforms
- Problem
  - Core functionality is fairly stable but the connections to the outside world or peripheral devices is subject to change
- Forces
  - The applications in your domain need to support similar but different application platforms
  - The applications use the same functional core in different ways
  - The core must be converted to run on each different platform
- Solution
  - Encapsulate the fundamental services of your application platform in a microkernel component. The microkernel includes functionality that enables other components running in separate processes to communicate with each other. It is also responsible for maintaining system-wide resources such as files or processes. In addition, it provides interfaces that enable other components to access its functionality.

Misuses of Microkernel

- Microkernel is useful when something must be able to be ported to different hardware but retain its essential functions.
- Most of the time it involves different control chips
- If portability to different hardware is not required, use a layer pattern

Microkernel pattern



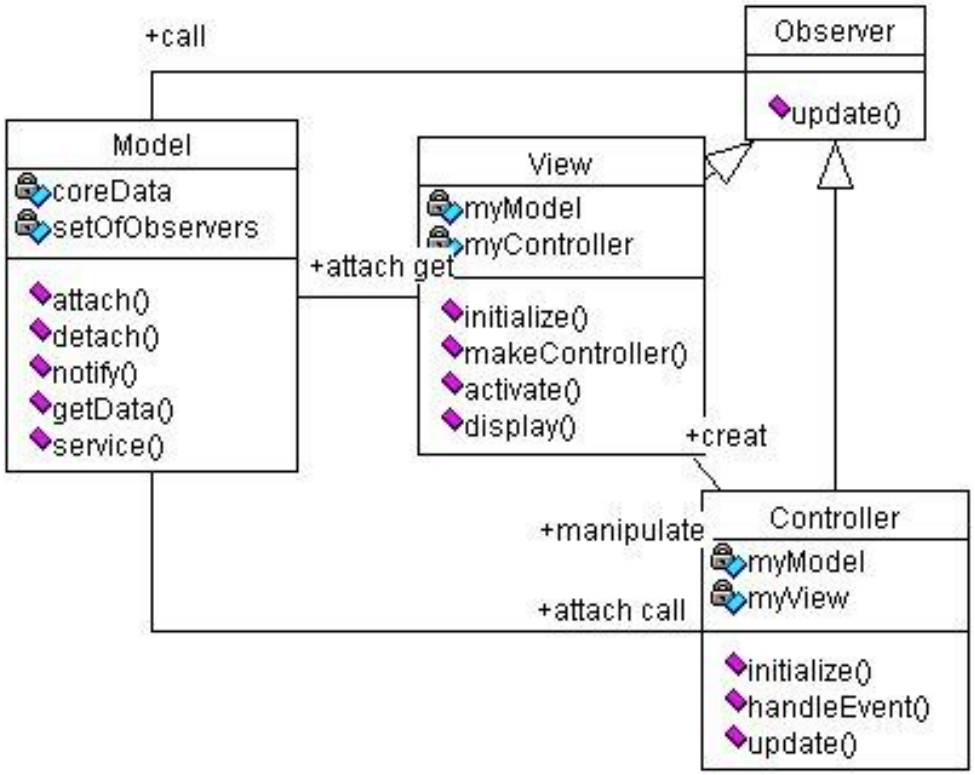


MODEL-VIEW-CONTROLLER

Pattern: Model-View-Controller

- Context
  - Many computer systems retrieve and display data in different forms.
  - User interface changes more often than the data model
  - Business logic needs to be incorporated
- Problem
  - How do you modularize the user interface functionality of a Web application so that you can easily modify the individual parts?
- Forces
  - User interface changes more often than business logic
  - Application can display the same data in different ways
  - User activity consists of two parts – presentation and update
- Solution
  - The *Model-View-Controller (MVC)* pattern separates the modelling of the domain, the presentation, and the actions based on user input into three separate classes (Burbeck, 1992):
  - **Model**. The model manages the behaviour and data of the application domain, responds to requests for information about its state (usually from the view), and responds to instructions to change state (usually from the controller).
  - **View**. The view manages the display of information.
  - **Controller**. The controller interprets the mouse and keyboard inputs from the user, informing the model and/or the view to change as appropriate.

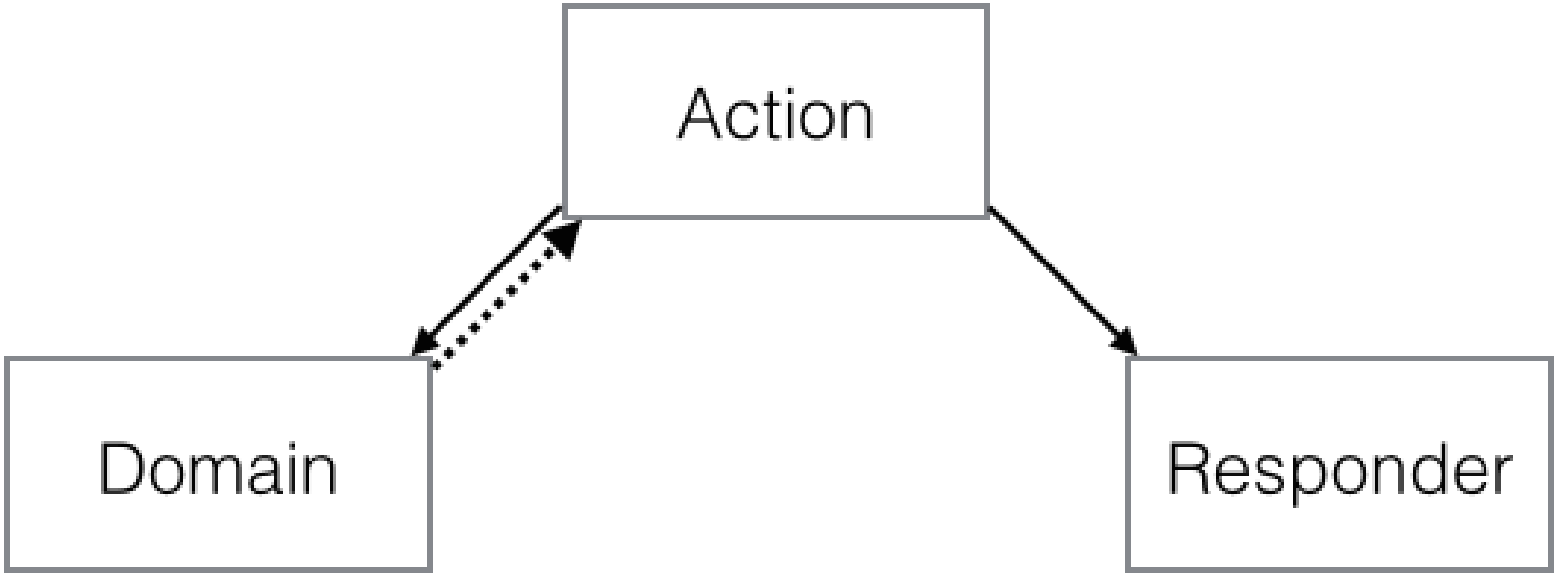
Model-View-Controller diagram

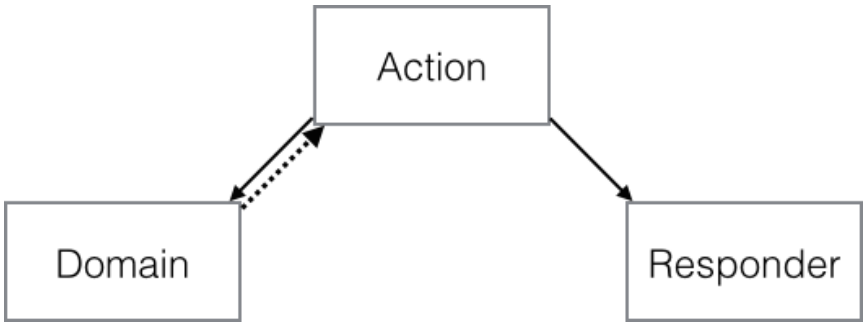


UNIVERSITY OF TECHNOLOGY SYDNEY

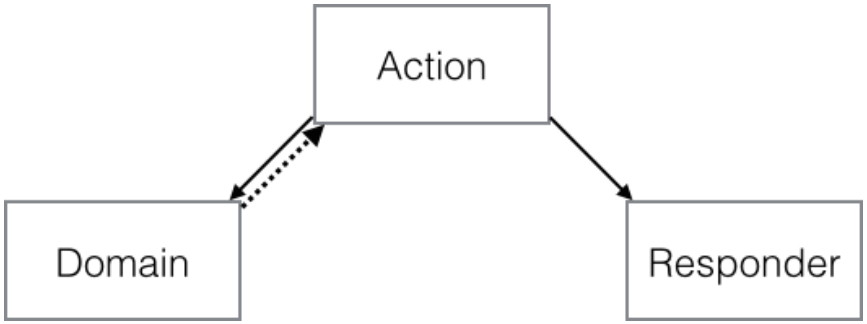
ACTION – DOMAIN - RESPONDER

Pattern: Action – Domain - Responder

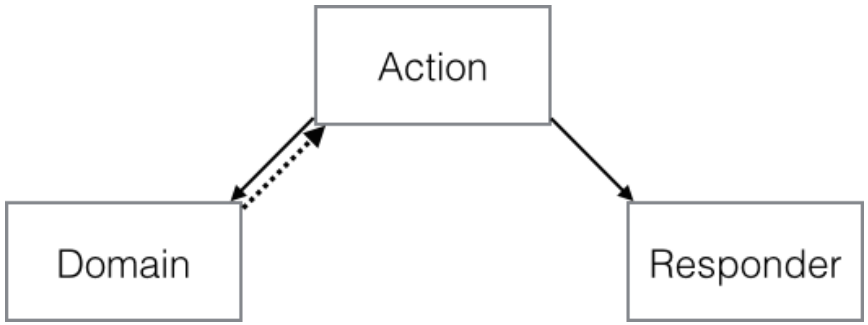




- *Action*
  - is the logic that connects the *Domain* and *Responder*. It uses the request input to interact with the *Domain*, and passes the *Domain* output to the *Responder*.
- *Domain*
  - is the logic to manipulate the domain, session, application, and environment data, modifying state and persistence as needed.
- *Responder*
  - is the logic to build an HTTP response or response description. It deals with body content, templates and views, headers and cookies, status codes, and so on.



- The web handler receives a client request and dispatches it to an ***Action***.
- The ***Action*** interacts with the ***Domain***.
- The ***Action*** feeds data to the *Responder*. (N.b.: This may include results from the ***Domain*** interaction, data from the client request, and so on.)
- The ***Responder*** builds a response using the data fed to it by the ***Action***.
- The web handler sends the response back to the client.



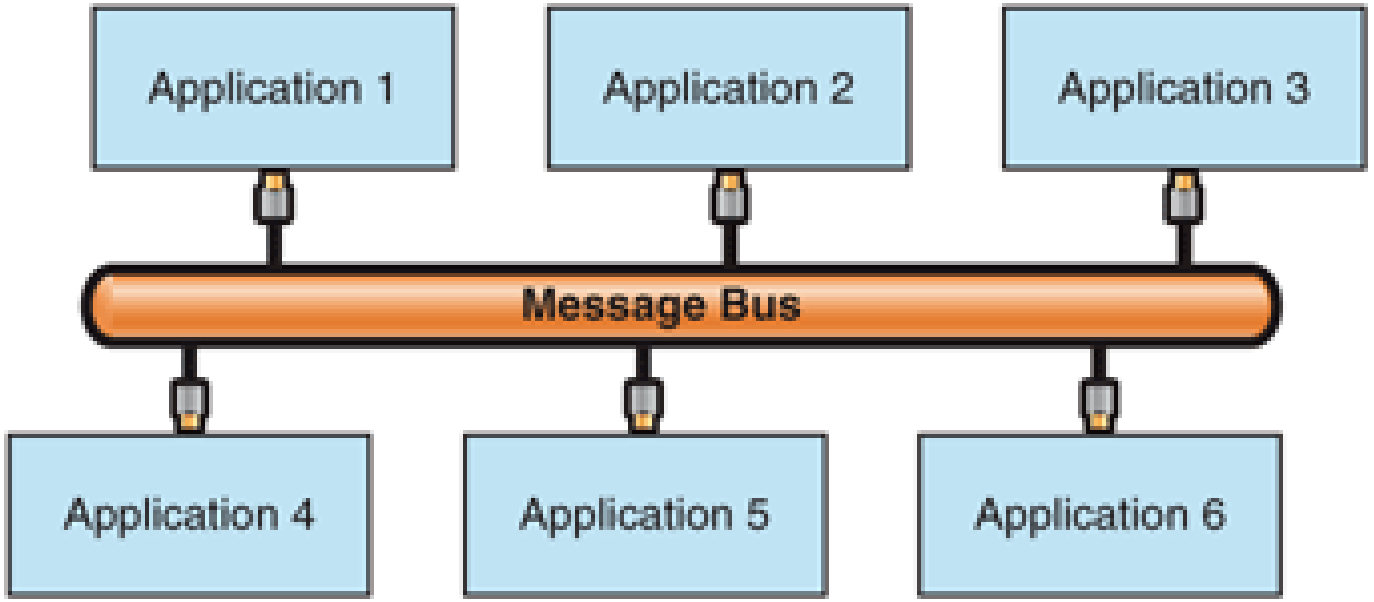
- Paul Jones;
  - I think ADR more closely fits what we actually do in web development on a daily basis. For example, this pattern is partly revealed by how we generally do web routing and dispatch. We generally route and dispatch *not* to a controller class per se, but to a particular action method within a controller class.
  - It is also partly revealed by the fact that we commonly think of the template as the *View*, when in a web context it may be more accurate to say that the HTTP response is the *View*. As such, I think ADR may represent a better separation of concerns than MVC does in a web context
- https://github.com/pmjones/adr

MESSAGE BUS

Pattern: Message Bus

- Context
  - Several co-operating applications, possibly running on several servers, need to exchange data but don't want to build separate interfaces for each application
  - Potential to add more applications
- Problem
  - As an integration solution grows, how can you lower the cost of adding or removing applications?
- Forces
  - Communication between applications usually creates dependencies between the applications.
  - In a configuration where point-to-point connectivity exists, the coupling has a quadratic (or *O[n2]*) growth with the number of applications.
  - Usually, the applications of an integration solution have different interfaces.
  - Some integration solutions consist of a fixed set of applications.
- Solution
  - Connect all applications through a logical component known as a message bus. A message bus specializes in transporting messages between applications. A message bus contains three key elements:
    - A set of agreed-upon message schemas
    - A set of common command messages
    - A shared infrastructure for sending bus messages to recipients

Message bus - diagram



UNIVERSITY OF TECHNOLOGY SYDNEY

DESIGN PATTERNS

UTS: ENGINEERING AND INFORMATION TECHNOLOGY

UTS CRICOS PROVIDER CODE: 00099F

Design patterns

- A design pattern provides a scheme for refining the sub-systems or components of a software system, or the relationships between them. It describes commonly recurring structure of communicating components that solves a general design problem within a particular context.
- Common design patterns include
  - Façade
  - Proxy
  - Broker
- There are many more design patterns but most have little effect on the architectural structure of the system

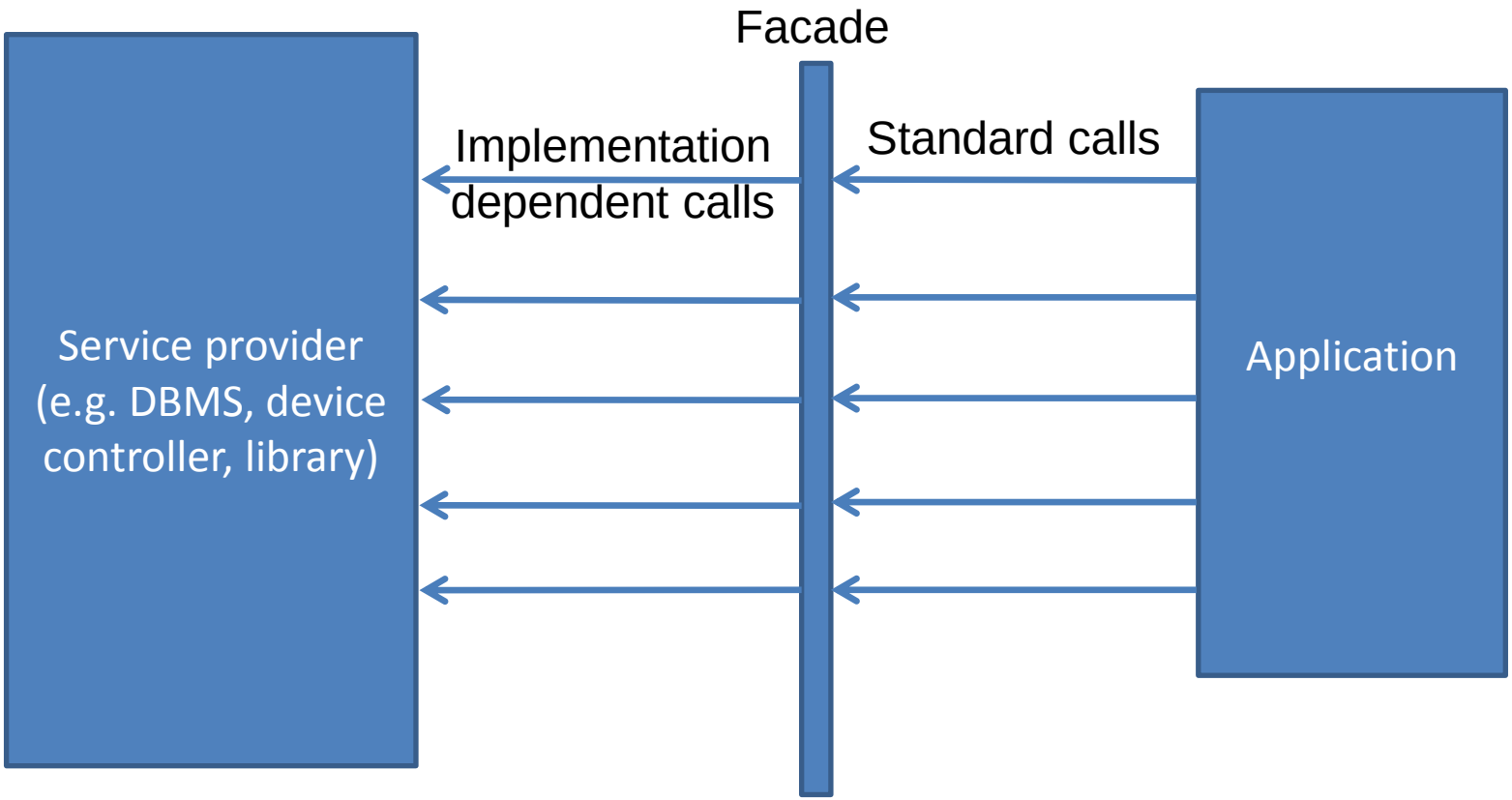
UNIVERSITY OF TECHNOLOGY SYDNEY

FACADE

Pattern: Façade

- Context
  - Several parts of the system need to interface to subsystems in different ways. Maintaining the different interfaces and sequences can become messy
- Intent
  - Provide a unified interface to a set of interfaces in a subsystem. Facade defines a higher-level interface that makes the subsystem easier to use. This can be used to simplify a number of complicated object interactions into a single interface.
- Forces
  - Provide a simple interface to a complex subsystem
  - Decouple the system from its clients
- Solution
  - Put a “Façade” between the clients and the subsystem
  - Analogous to a concierge who takes a request "we'd like to go out to an evening dinner, and a show, and they return to the hotel for an intimate dessert in our room“
  - The concierge handles all the nitty-gritty details (a taxi, restaurant reservations, theatre tickets, housekeeping prep of the room, the kitchen preparing the dessert, room-service delivery, etc...)

Façade - diagram



Façade – in use

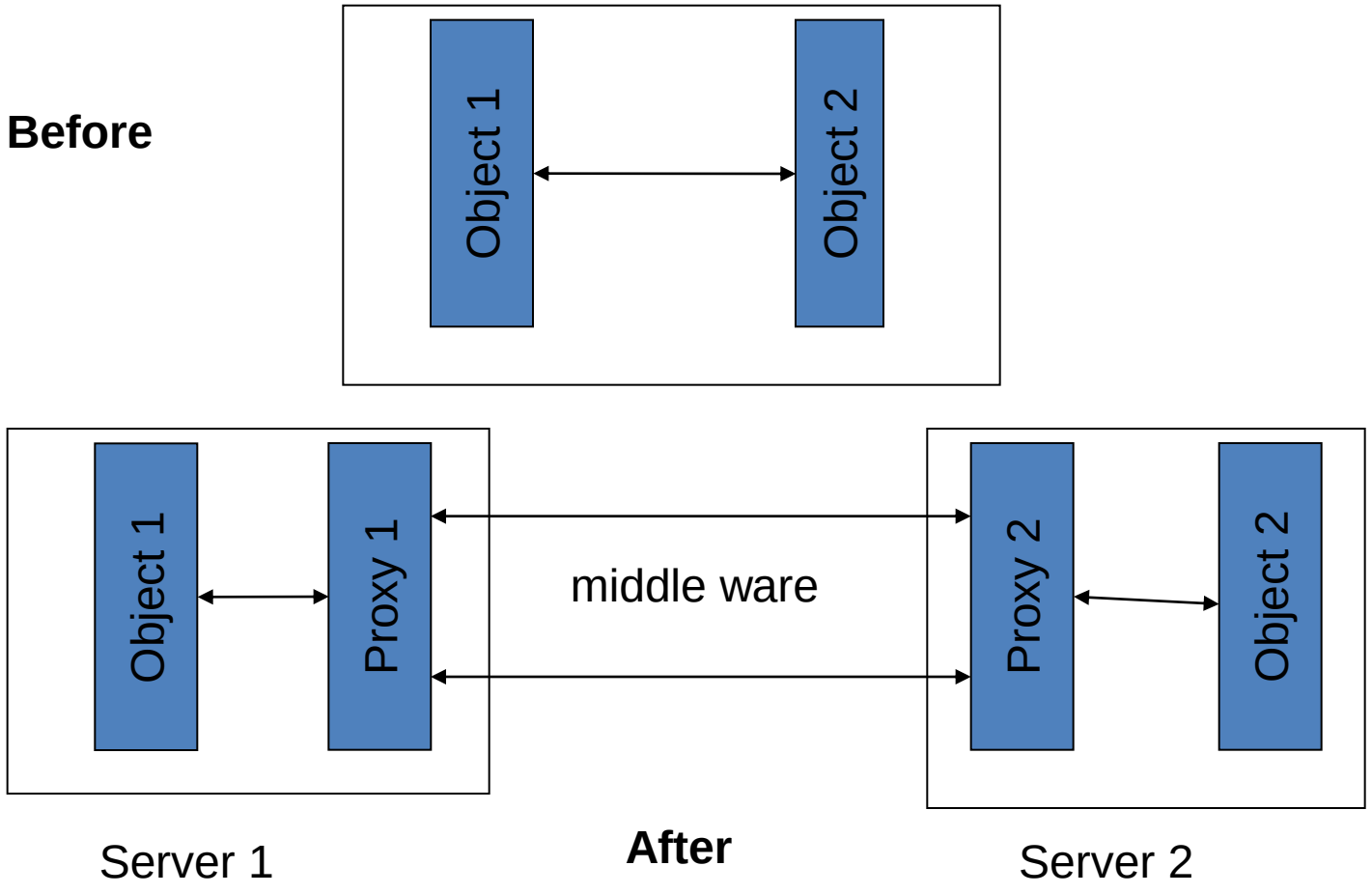
- A façade is useful when you want a simple interface to something more complex
- A façade is useful when you need a stable interface to something that will change
- Usually all parts are on the same server

PROXY

Pattern: Proxy

- Motivation
  - We wish to add an optional behaviour to an existing class. This new behaviour may protect the class, or log access to it, or delay its behaviour or instantiation, or any other single additional behaviour. We also need to be able to use the existing class at times without this additional behaviour.
- Examples
  - **Logging Proxy:** Logs all calls to the method of the original class, either before or after the base behaviour, or both.
    Protection Proxy: Block access to one or more methods in the original class. When those methods are called, the Protection Proxy may return a null, or a default value, or throw an exception, etc…
    Remote Proxy: The Proxy resides in the same process space as the client object, but the original class does not. Hence, the Proxy contains the networking, piping, or other logic required to access the original object across the barrier. This cannot be accomplished with a class proxy.

Proxy – in pictures



Proxy – in use

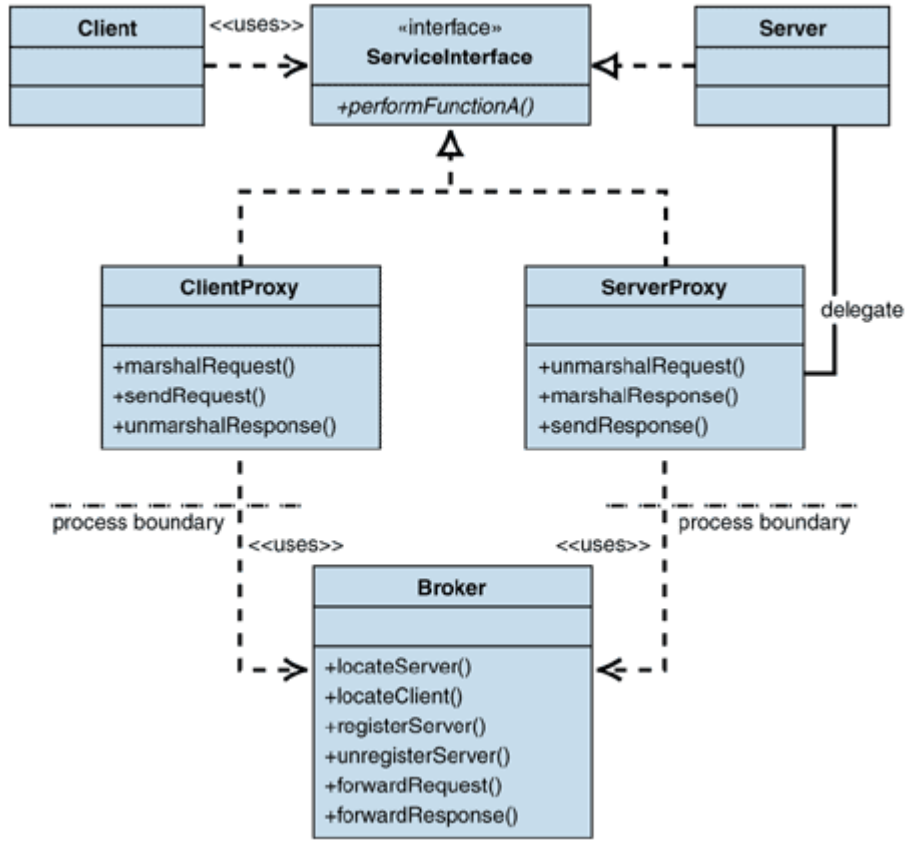
- A proxy is useful when you want a way to separate the interface to something from the details of communicating with that something.
- Imagine calling one object from another object. While they are on the same server or within the same component this is not a problem. But put one of the objects on another server and you now have the difficulty of handling all the coordination and communication. Put matching proxies between the two and let the proxies deal with the communication and coordination.

BROKER

Pattern: Broker

- Context
  - Several servers are cooperating in an application but the location of the different services within the servers is not fixed.
- Problem
  - When building a distributed system, provide a means to find out what systems are available and arrange access to them.
- Forces
  - Components should be able to access service provided by others through remote, location-transparent service invocations
  - You need to exchange, add or remove components at run time.
  - The architecture should hide system and implementation dependent details from the users of those services.
- Solution
  - Introduce a broker
  - Services register themselves with a broker
  - Client request services from the broker
  - The broker locates the service, forwards the request to the server and transmit the results back to the requestor.

Broker - Diagram



Broker – in use

- When applications or servers are spread around, it is undesirable to require that applications always execute from a fixed place.
- So, have the broker who knows where all the applications are at any instant.
- When an application starts, tell the broker where you are.
- When a client starts, ask the broker where the application is.
- Broker tells one where the other one is, then play not further part.