6. Software architecture

**Initiate**
- 1 Architecture and architects
- 2 Architecture precursors
- 3 Architecture frameworks

**Govern**
- 11 Architecture in Operations
- 11 Architecture Governance
- 11 Architecture Change Management
- 11 Architecture Implementation

**Plan**
- 10 Migration Planning
- Migration path
- Business case
- Delivery Plans

**Architect**
- 4 Business & 5 Data architecture
- 6 Software & 7 Apps architecture
- 8 Design for NFRs
- 9 Infrastructure architecture

**Intermediate level**
6 Software domain view

- Applications architecture (section 7) depends on the lower level design of fine-grained software components discussed in this section.

- Enterprise and solution architects should be aware of tools and techniques for modularising one application into components and integrating those components.
Encapsulation of modules

- [A technique] that defines a thing by an interface it offers.
- It hides inner workings or processes from external entities.
- It hides internal resources (notably data structures) from external entities.
API (service catalogue)

- [An interface definition] a collection of automated behaviors accessible by software clients.
- It identifies discrete services, may provide access to them, and hides what performs.

<table>
<thead>
<tr>
<th>Software</th>
<th>Required behaviours</th>
<th>Logical structures</th>
<th>Physical structures</th>
</tr>
</thead>
<tbody>
<tr>
<td>Operation</td>
<td>API</td>
<td>Application component</td>
<td></td>
</tr>
</tbody>
</table>

**Address Component: Interface definition**

<table>
<thead>
<tr>
<th>Operation: Get Address (Post Code, House Num): Address</th>
</tr>
</thead>
</table>
Operation (discrete service)

[An automated behavior] that can be requested of an application component.

<table>
<thead>
<tr>
<th>Software</th>
<th>Required behaviours</th>
<th>Logical structures</th>
<th>Physical structures</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Operation</td>
<td>API</td>
<td>Application component</td>
</tr>
</tbody>
</table>

### Address Component: Interface definition

**Operation: Get Address (Post Code, House Num): Address**
- Precondition: Post Code is properly formatted
- Postcondition: Correct Address is returned

**Operation: Get Post Code (Address): Post Code**
- Precondition: Address is properly formatted
- Postcondition: Correct Post Code is returned

**Non-functional characteristics – shared by services above**
- Response time = < 3 seconds
- Throughput = 10 per second
**Application component**

- [A component] capable of performing automated behaviors.
- It can be a whole application or a component within one.
- It may be encapsulated behind an API, and may maintain some business data.

<table>
<thead>
<tr>
<th>Software</th>
<th>Required behaviours</th>
<th>Logical structures</th>
<th>Physical structures</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Operation</td>
<td>API</td>
<td>Application component</td>
</tr>
</tbody>
</table>

![Diagram of Application component]

Copyright Avancier Limited 2013
Component-based design (software level)

- [A technique] a modular design approach that divides a system into components.
- Components may be coded using different technologies and deployed in different locations.

- **Delegation**
- [A process] whereby one component calls another to do the work.
Delegation usually implies invoking a component by passing it a message.
Statefulness

► **Stateful component**
► [An application component] that retains data in its local memory between invocations.
► The state is lost if the component is removed.

► **Stateless component**
► [An application component] that does not retain data between invocations.
► However, its transient state can be copied into a persistent data store.
Component-based design artifacts

► **Component-dependency diagram**
► [An artifact] that shows the design-time structure of a software application.
► It shows which components depend on which other components

► **Sequence diagram**
► [An artifact] that shows how components cooperate at run-time to enable a process.
► How a design-time structure behaves at run time is critical to meeting requirements.
6.2 Component integration

- This section reflects a history of increasing distribution and integration of software systems, and the development of ways to connect loosely-coupled application components.
- However, there will always be use cases in which components are better closely coupled.
**LPC versus RPC**

**Local Procedure Call (LPC)**
- [A process] by which one component calls another component running on the same computer.
- It is simpler, quicker and more secure than a remote procedure call.

**Remote Procedure Call (RPC)**
- [A process] by which a process on one computer calls a process on another computer.
- It is more complex, slower and less secure than a local procedure call.
- The term usually implies a synchronous request-reply style of interoperation.

**Local Procedure Call**
- Simple
- Fast
- Available
- Secure

**Remote Procedure Call**
- More complex
- Slower
- Less available
- Less secure
[A technique] for modular design and integration that *initially* assumed the run-time system works like this

- **Instantiation**: an object is an instance of a class (a component type)
- **Identification**: an object is identified by an object identifier
- **Co-location**: client and server objects work in the same name space
- **Statefulness**: an object is stateful
- **Inheritance**: a subtype object can perform the behaviors of a super type class
- **Design pattern**: intelligent domain objects communicate to complete a process
- **Synchronicity**: client objects make request-reply invocations to server objects
- **Blocking**: a server object accepts only one invocation at once.
Instantiation and identification

- An object is an instance of a class (a component type).
- An object is identified by an object identifier.

Bertrand Meyer said classes should be abstract data types, which
- Encapsulate a data structure.
- Define operations performable on that data structure.

Customer

Object Id

998
55 Oak Road, Town 0
Excellent Wife called Mary

changeAddress (CustomerNumber Address): Done
closeOrder (CustomerNumber OrderValue): Done
getDetails (CustomerNumber):
Co-location

- Client and server objects work in the same name space
- You call an object to do work using an "object/method pair"
- 99999/addPupil
an object is stateful

The first OO programs handled a few small stateful objects.

Case studies were
- real-time process control systems – objects live forever
- graphical user interfaces - objects deleted when the UI is closed
Inheritance

- A subtype object can perform the behaviors of a super type class.

- Objects of a subclass **inherit** or **extend** the operations of a super class.

- E.g., you can ask a bowler object to:
  - Update batting average
  - Reply with the player’s name
A controller orchestrates the domain objects (Betrand Meyer)

Intelligent domain objects communicate to complete a process
client objects make request-reply invocations to server objects
- Invokes server object
  - using an “object-method pair”
  - an object id and a method /operation name
- Requests
  - freezes
  - holds a connection until reply
- Receives reply
  - wakes up and carry on

a server object accepts only one invocation at once (so blocks others)
DO: Distributed Objects

[A technique] for modular design and integration that employs an Object Request Broker.

An object request broker (ORB) is RPC-like middleware that enables the objects of an OO program to be distributed.

Software is coded as though all objects are on one computer.

The ORB handles the distribution of objects between computers.

So (in theory) the distributed system behaves like one OO program.

It may provide transaction management, security and other features.
IDL: Interface Definition Language

- A language for defining an API (not the procedures of operations/services in it).
- It enables components coded in different languages and running on different operating systems to interoperate.

1990s
- Sun's ONC RPC
- The Open Group's Distributed Computing Environment
- IBM's System Object Model
- Object Management Group's CORBA,
- WSDL for Web services

2000 - 2010
- Increasing use of WSDL

2010 - 2020
- Increasing use of REST and OData
WSDL: Web Service Description Language:

- [A standard] from the W3C for an interface definition.
- A WSDL includes a signature, protocol and web address for each operation/service.
- Initially it depended on XML and SOAP.
- It is now usable with JSON and HTTP.
SOAP

- A standard application layer protocol devised by Microsoft, then adopted by the W3C.
- A protocol used to invoke operations on remote components without using an ORB.
- It allows distributed components to communicate by sending XML messages over HTTP.

<table>
<thead>
<tr>
<th>DO</th>
<th>SOAP</th>
<th>REST</th>
<th>EDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tightly-coupled</td>
<td>Loosely-coupled</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Web Service

- [A component] invoked over “the web” using an internet protocol and a published interface.
- Initially, implied the use of WSDL, XML and SOAP standards.
- No longer related to any particular IDL, data format or internet protocol.

<table>
<thead>
<tr>
<th></th>
<th>WSDL1</th>
<th>WSDL2 allows also</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data format</td>
<td>XML messages</td>
<td>JSON</td>
</tr>
<tr>
<td>Protocol</td>
<td>SOAP over HTTP (or perhaps SMTP)</td>
<td>HTTP directly</td>
</tr>
</tbody>
</table>

DO

- SOAP: Tightly-coupled
- REST: Loosely-coupled

EDA

SWAGGER for JSON?
WADL (XML) for REST?
SOA: Service–Oriented Architecture

- [A technique] for modular design and integration that is a more loosely-coupled than Distributed Objects and facilitates the reuse of remotely accessible services.

- It is often associated with the use of Web Services, but does not have to be.
REST: Representational State Transfer

[A technique] for modular design and integration devised by Roy Fielding as a means to connect remote components using standard internet protocols.

It decouples distributed components so that client/sender components need minimal information about server/receiver components.

**A RESTful client**
- invokes a remotely accessible service using a domain name and an operation type available in an internet protocol, usually HTTP.

**A REST-compliant server**
- is identified by a domain name and offers only one service in response to each operation type in an internet protocol, usually HTTP.

---

**DO**  SOAP  **REST**  EDA
Tightly-coupled  Loosely-coupled
OData *(Microsoft 2007, OASIS 2014)*

- [An IDL] an evolution of REST.
- It supports clients wanting to invoke operations on entities in a remote web data store.
- Client applications that speak OData can easily connect to data server applications that provide CRUD operations on a logical data model.

- Four parts
  - Entity Data Model (EDM) in an XML schema
  - Request and reply protocol
  - Client-side libraries
  - Server-side data servers

- DO SOAP REST EDA
  - Tightly-coupled
  - Loosely-coupled
EDA: Event-Driven Architecture

- [A technique] for a modular design and integration that decouples the senders of news or update events from the receivers.
- Any component can receive or read any event/message published by any other component.
- It often implies using the publish and subscribe features of a middleware technology.
- But can be implemented using a shared data space.

<table>
<thead>
<tr>
<th>DO</th>
<th>SOAP</th>
<th>REST</th>
<th>EDA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Tightly-coupled</td>
<td></td>
<td>Loosely-coupled</td>
<td></td>
</tr>
</tbody>
</table>
6.3 Component coupling
Synchronous

1: A request-reply style
- A client must wait for a server to reply before continuing.
- (The usual invocation from one COBOL module or Java object to another.)

2: A blocking style
- A server serves one client at a time.
- The caller and responder hold a channel open, blocking others from using it.
- (The usual invocation style used by CORBA-compliant technologies.)
Asynchronous

1: A so-called **fire-and-forget** style in which a client does not wait for a server to reply.
(The usual style in email conversations.)

2: A **non-blocking** style in which a server can accept requests from several clients before responding to the first.
(The usual style of Web Services.)
Typically, the server has a queue of incoming messages and releases the channel after a message is received.
## Loose coupling c1990

<table>
<thead>
<tr>
<th>Factor</th>
<th>Tight coupling</th>
<th>Decoupling techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Naming</strong></td>
<td>Clients use object identifiers</td>
<td>Clients use domain names</td>
</tr>
<tr>
<td></td>
<td>One name space</td>
<td>Multiple name spaces behind interfaces</td>
</tr>
<tr>
<td><strong>Paradigm</strong></td>
<td>Stateful objects/modules</td>
<td>Stateless objects/modules</td>
</tr>
<tr>
<td></td>
<td>Reuse by OO inheritance</td>
<td>Reuse by delegation</td>
</tr>
<tr>
<td></td>
<td>Intelligent domain objects</td>
<td>Intelligent process controllers</td>
</tr>
<tr>
<td><strong>Time</strong></td>
<td>Synchronous request-reply</td>
<td>Asynchronous messaging</td>
</tr>
<tr>
<td></td>
<td>Blocking servers</td>
<td>Non-blocking servers</td>
</tr>
</tbody>
</table>

- Faster simpler
- More flexible

- COBOL modules and Java objects
- Web Services
- CORBA
## Loose coupling today

<table>
<thead>
<tr>
<th>Factor</th>
<th>Tight coupling</th>
<th>Decoupling techniques</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Naming</strong></td>
<td>Clients use object identifiers</td>
<td>Clients use domain names</td>
</tr>
<tr>
<td></td>
<td>One name space</td>
<td>Multiple name spaces behind interfaces</td>
</tr>
<tr>
<td><strong>Paradigm</strong></td>
<td>Stateful objects/modules</td>
<td>Stateless objects/modules</td>
</tr>
<tr>
<td></td>
<td>Reuse by OO inheritance</td>
<td>Reuse by delegation</td>
</tr>
<tr>
<td></td>
<td>Intelligent domain objects</td>
<td>Intelligent process controllers</td>
</tr>
<tr>
<td><strong>Time</strong></td>
<td>Synchronous request-reply</td>
<td>Asynchronous messaging</td>
</tr>
<tr>
<td></td>
<td>Blocking servers</td>
<td>Non-blocking servers</td>
</tr>
<tr>
<td><strong>Location</strong></td>
<td>Remember remote addresses</td>
<td>Use brokers/directories/facades</td>
</tr>
<tr>
<td><strong>Data types</strong></td>
<td>Complex data types</td>
<td>Simple data types</td>
</tr>
<tr>
<td><strong>Version</strong></td>
<td>Version dependency</td>
<td>Design to avoid version dependence</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Apply the open-closed principle</td>
</tr>
<tr>
<td><strong>Protocol</strong></td>
<td>Protocol dependency</td>
<td>Design for multiple protocols</td>
</tr>
<tr>
<td><strong>Integrity</strong></td>
<td>ACID transactions</td>
<td>BASE: compensating transactions and eventual consistency</td>
</tr>
</tbody>
</table>

Often faster and/or simpler

Often more flexible, but more complex
6.4 Design patterns [not to be examined]

- Design pattern
- A shape or structure of elements that commonly appears in solution design.
- A tried and tested design that is tailored to address particular problems or requirements.
Basic design pattern pairs

Hierarchical/centralisation

**Hierarchy**
- Top
- Middle
- Middle
- Middle
- Atom
- Atom
- Atom
- Atom

**Hub and Spoke**
- Hub
- CRM
- ERP
- Billing
- DW
- A
- B

**Fork/Orchestration**
- Controller
- CRM
- ERP
- Billing

**Hierarchical layering**
- Client layer
- Client
- Server layer
- Server

Anarchical/distribution

**Network**
- Atom
- Atom
- Atom

**Point to Point**
- CRM
- ERP
- Billing
- DW
- A
- B

**Chain/Choreography**
- CRM
- ERP
- Billing

Peer to Peer
- System
- Component
- System
- Component
Design pattern pair

► A pair of contrasting patterns that suit different situations.
► Architects choose between alternative patterns by trading off their pros and cons.

<table>
<thead>
<tr>
<th>Hierarchical or centralisation pattern</th>
<th>Anarchical or distribution pattern</th>
</tr>
</thead>
<tbody>
<tr>
<td>centralises control in one place or component.</td>
<td>distributes control to many places or components.</td>
</tr>
<tr>
<td>Hierarchy</td>
<td>Anarchy or Network</td>
</tr>
<tr>
<td>Hub and Spoke</td>
<td>Point-to-Point or Mesh</td>
</tr>
<tr>
<td>Client-Server</td>
<td>Peer-to-Peer</td>
</tr>
<tr>
<td>Fork or Orchestration</td>
<td>Chain or Choreography</td>
</tr>
</tbody>
</table>
Structural patterns

**Hierarchical structure**
- [A design pattern] that divides a node into subordinate nodes.
- A rule of thumb for division: divide one into about seven.
- A rule of thumb for decomposition: stop at three or four levels.

**Network structure**
- [A design pattern] in which a node can be connected to any other node.
Communication path patterns

► Hub and spoke
  ■ [A design pattern] in which components communicate via a mediator.
  ■ It can be good where the endpoints are volatile; but can be more complex and slower.

► Point to point
  ■ [A design pattern] in which components talk to each other directly.
  ■ It can be faster and simpler where inter-component communication is 1 to 1 and endpoints are stable; but can hinder change.
Delegation patterns

► Hierarchical (or layered):
- [A design pattern] in which higher components delegate work to lower (server) components.
- This pattern is widely used to structure complicated systems (machines, networks, software applications and enterprise architecture).

► Peer-to-peer:
- [A design pattern] in which any two components can delegate to work to each other.
- Such cyclic dependencies are said to create a fragile structure, difficult to understand and maintain, but are sometimes inevitable.
Process control patterns

► **Fork/Orchestration**
  - [A design pattern] in which one component schedules other components.
  - It centralises intelligence about the overall process or workflow.

► **Chain/Choreography**
  - [A design pattern] in which components interact directly.
  - It distributes intelligence about the overall processes
  - Each component calls the next component.
Design pattern pairs - reminder

Hierarchical/centralisation

- Hierarchy
  - Root
  - Middle
    - Atom
    - Atom
    - Atom
    - Atom

- Hub and Spoke
  - Hub
    - CRM
    - ERP
    - Billing
  - A
  - B
  - DW

- Fork/Orchestration
  - Controller
    - CRM
    - ERP
    - Billing

Client-Server layering

- Peer to Peer
  - System
    - Component
    - Component
  - Client
    - Server

Anarchical/distribution

- Network
  - Atom
  - Atom
  - Atom
  - Atom

- Point to Point / Mesh
  - CRM
  - ERP
  - Billing
  - A
  - B
  - DW

- Chain/Choreography
  - CRM → ERP → Billing

Copyright Avancier Ltd 2018
### 6.5 “Design Patterns: Elements of Reusable OO Software”

11 of 23 patterns

<table>
<thead>
<tr>
<th>Name</th>
<th>Description or purpose</th>
</tr>
</thead>
<tbody>
<tr>
<td>Façade</td>
<td>Consistent interface coarse-grained services that encapsulate a subsystem.</td>
</tr>
<tr>
<td>Adapter</td>
<td>A wrapper that converts a provided service into a required service. Facilitates the reuse of existing technologies.</td>
</tr>
<tr>
<td>Proxy</td>
<td>A surrogate for a distributed component. Used in distribution of code between different name spaces.</td>
</tr>
<tr>
<td>Singleton</td>
<td>A component (or class) with only one instance (or object).</td>
</tr>
<tr>
<td>Observer</td>
<td>A component that monitors the state of another component.</td>
</tr>
<tr>
<td>Composite</td>
<td>Enables a class to process operations on every level of a hierarchical structure, including composite and leaf nodes</td>
</tr>
<tr>
<td>Template method</td>
<td>Offers several variations of an algorithm.</td>
</tr>
<tr>
<td>Strategy</td>
<td>Adds a façade to a template method.</td>
</tr>
<tr>
<td>Manager</td>
<td>Often used to manage a set of objects.</td>
</tr>
<tr>
<td>Factory method</td>
<td>Create the right server object for the client - hiding how the server object is initialized.</td>
</tr>
<tr>
<td>Abstract object factory</td>
<td>Create the right factory object for the client - hiding which factory class is used.</td>
</tr>
</tbody>
</table>
[A pattern] in which an interface that shields external entities from internal changes to a system or component.
- It should reduce the coupling between client and server components.
- It may aggregate fine-grained services into a coarse-grained interface.
- It is usually stateless and does little or nothing but delegate work.
[A pattern] that separates
- the processing of an input message (controller),
- the display of a user interface (view) and
- the retrieval and processing of persistent data (model).
How the MVC pattern may be used in enterprise applications

- Separates modules that handle client-side data structures from modules that handle server-side data structures

Client device (screen state)

receives an input request, may modify it, asks the Model to return or change its state.

Web/app server (session state)

after the Model has done its work, organises the state of the Model in the form a client wants to see.

Data server (record state)

Model

Controller

View

DB

retrieves state data relevant to the request, can return or change the state of the Model, may hold session data relevant to a client-side form or process.
Variations of the MVC pattern

Three variations

Other MVC variants include
- hierarchical model–view–controller (HMVC),
- model–view–adapter (MVA)
- model–view–presenter (MVP),
- model–view–viewmodel (MVVM)
MVC in web frameworks (after Wikipedia)

- Web frameworks divide MVC differently between client and server tiers.
- Early web frameworks mostly put MVC components on the server.
  - Ruby on Rails,
  - Django,
  - ASP.NET MVC and
  - Express
- Other frameworks allow MVC components to execute partly on the client
  - AngularJS,
  - EmberJS,
  - JavaScriptMVC and
  - backbone (also see Ajax).
Proxy

[A pattern] in which proxy components act as surrogates for remote components.

It is used in distribution of code between different name spaces, as in “Distributed Objects”
[A pattern] in which observer components monitor the state of a subject component.

A primitive kind of “Event-Driven Architecture”
Moving notification into a separate publisher

**Observer**
- A subject
  - notifies observers of changes to its state
- Observers
  - register with the subject to be notified of changes.
  - unregister when no longer interested

**Event-Driven Architecture (EDA)**
- Inserts a publisher (broker) between Subjects and Observers, and so decouples
[A pattern] in which a component (or class) has only one instance (or object).

We only need one copy of:
- Global variables – e.g. current date and time
- A commonly required table – e.g. exchange rates
- A façade – e.g. a stateless controller

So, a single-object class can hold the data or do the job.
6.6 Communication Patterns

► **Communication pattern** [a pattern] in which a client/sender application (or other actor) connects to a server/receiver application (or other actor).

► Two broad communication styles, each subdivided into two narrower styles, are listed below.

► There are other subcategories, not listed here.

---

Tightly-coupled

![Diagram of communication styles](image)

- **Connected**
  - Point to Point
  - Introduction Agent

- **Disconnected (Mediator)**
  - Message Broker
  - Passive Mediator

Loosely-coupled
Direct connection: point to point

- **Point-to-point connection** [a pattern] in which a message sent by one client/sender is received by one server/receiver.
- The client/sender knows the location of the receiver.
- The client knows what protocols and data formats the server/receiver understands.
- Strengths: simple and fast.
- Weaknesses: potential duplication of data transformation and routing code, reconfiguration costs on receiver address changes.
**Direct connection: direct broker**

- **Direct broker connection** [a pattern] in which parties willing to communicate are registered (with end point locations) in a directory.
- When a client/sender wants to send a message to a server/receiver, the broker makes the introduction, and may establish client-side and server-side proxies.
- From then on, the parties talk directly or through proxies, as though using point-to-point connection.
- Not so simple and fast, but decouples clients/senders from server/receiver locations.
Indirect communication

[a pattern] in which clients/senders never talk directly to servers/receivers, they talk only through a mediator or shared resource. There are two subcategories.

- **Tightly-coupled**
  - Under the covers, all communication is point to point!

- **Loosely-coupled**
  - Communication Style
    - Connected
      - Point to Point
    - Disconnected (Mediator)
      - Introduction Agent
      - Message Broker
      - Passive Mediator
[a pattern] in which an **indirect broker** decouples communicating parties; it adds a layer of indirection between clients/senders and servers/receivers.

- This can enable communicating parties to work at different places and different times (asynchronously).
- It can shield one party from the effects of some changes to the other party.
- Mediator technologies include message brokers, message routers, message buses and publish-subscribe middleware.

**Message bus** = message broker + schema-based distribution
Shared data space communication

- [a pattern] in which parties communicate indirectly by reading and writing messages in a common data store, which might be shared memory, a message queue, a serial file or a database.
- Aka “shared memory”, “space-based architecture” or “blackboard design pattern” or “passive mediator”.
Different communication styles may be used at different levels of a communication stack.

Under the covers is always some point-to-point communication.