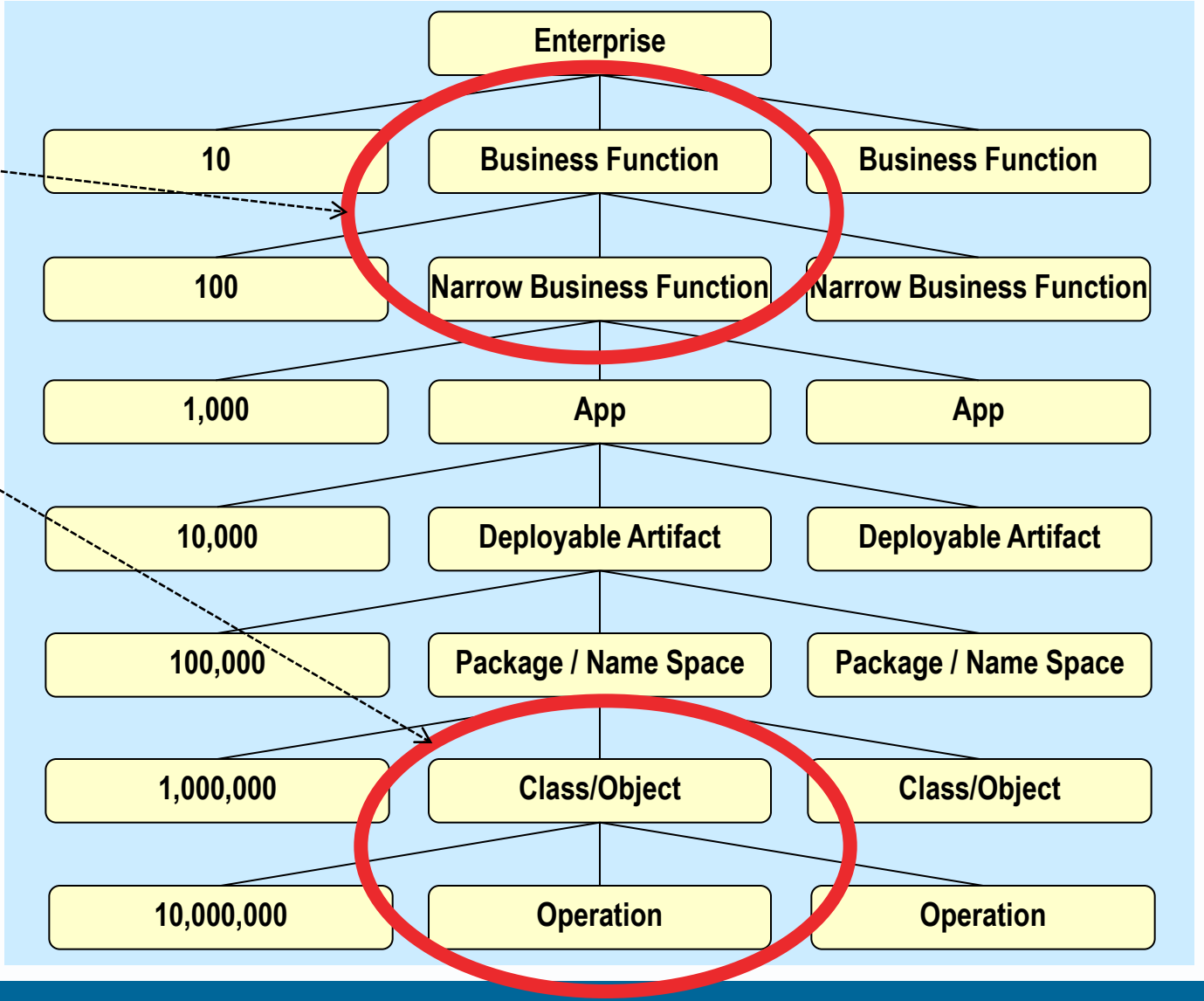
Avancier Methods (AM)

Software Architecture

Modularity and OO presumptions

Radically changing gear!

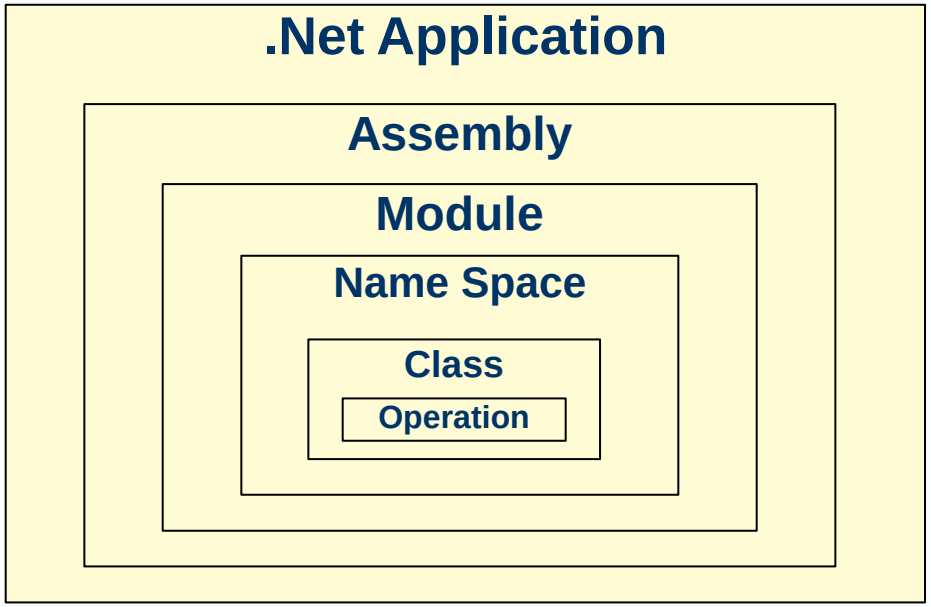
- Earlier discussion of business architecture
- This discussion of software architecture
- Some modular design principles are relevant to both

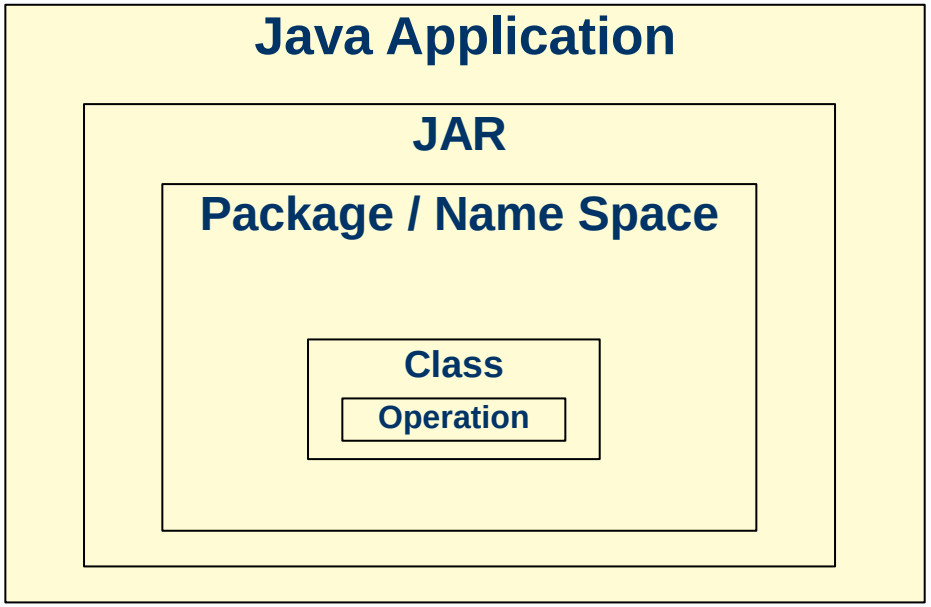


Software architecture as the finest-grained enterprise architecture

- "EA regards the enterprise as a system, or a system of systems".

- **Business architecture** is highest level business system design
- **Software architecture** is lowest level business system design, including the decomposition of an application into components

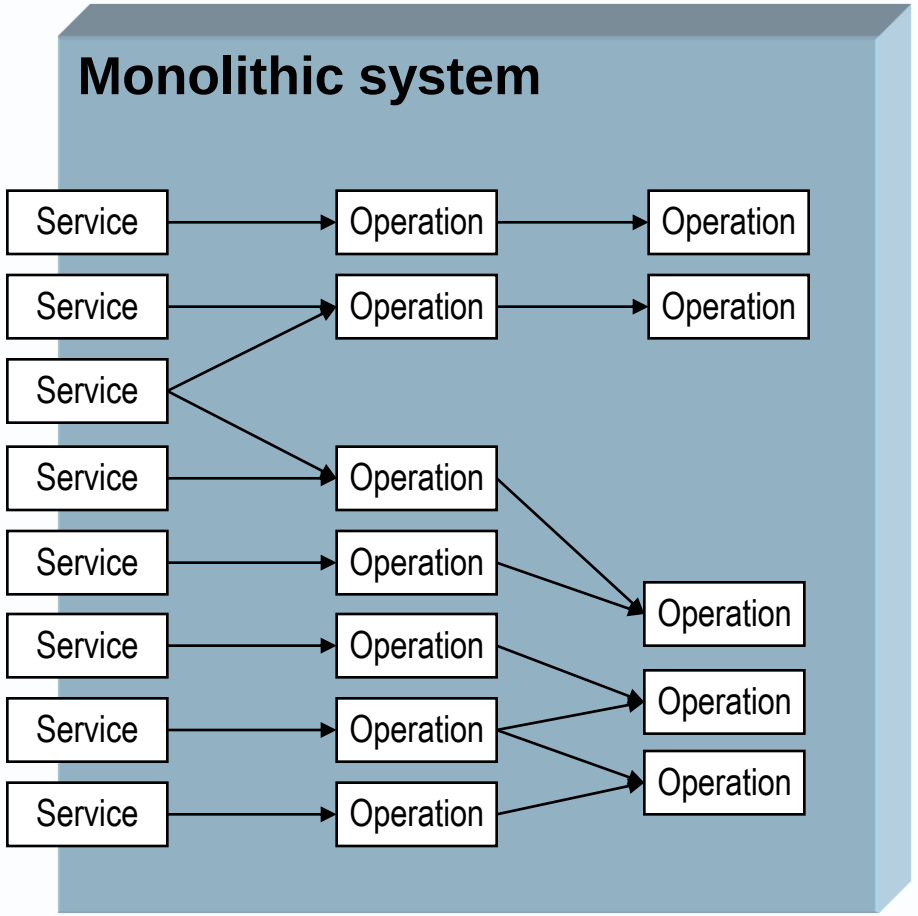


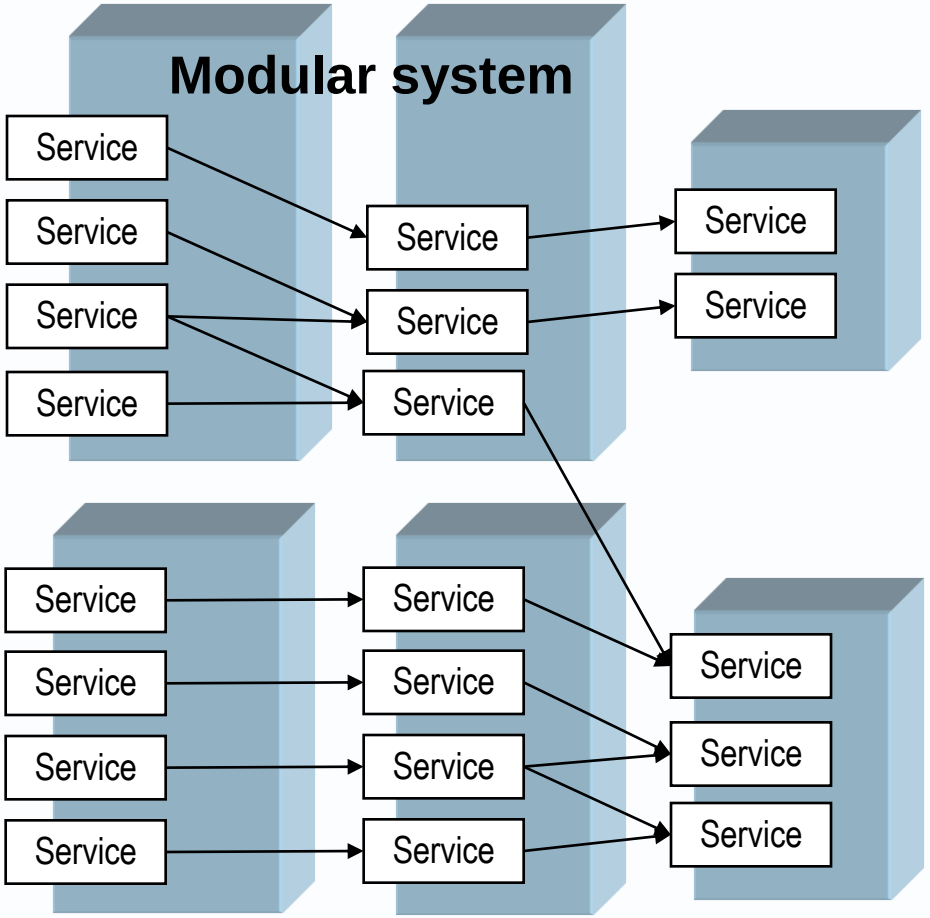


Modular system design

Avancier

Divides the system into components - to perform the required processes





Modular design questions

- For 40 years, software gurus have continually revisited modular design and integration questions.

- How best to
  - scope each module?
  - avoid duplication between modules?
  - separate modules?
  - integrate modules?

- What principles might help?

Architecture principles promoted in a global organisation

1. Separate concerns (for flexibility and maintainability)
2. Build for competitive advantage / Buy for competitive parity
3. Encapsulate components (for CBD and SOA)
4. Use open APIs for inter-component communication
5. Loosely couple components (for flexibility and availability)
6. Use Event-Driven Architecture for broadcast updates
7. Maintain a single source of truth
8. Design for response time / latency
9. Design for graceful failure – informing users
10. Web first: design for browser and client device independence

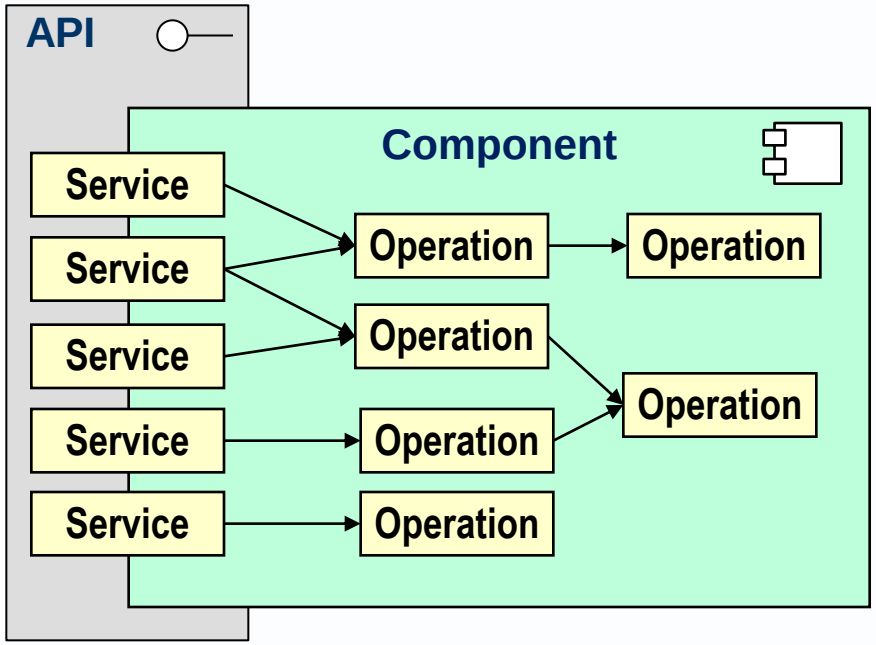
Encapsulation

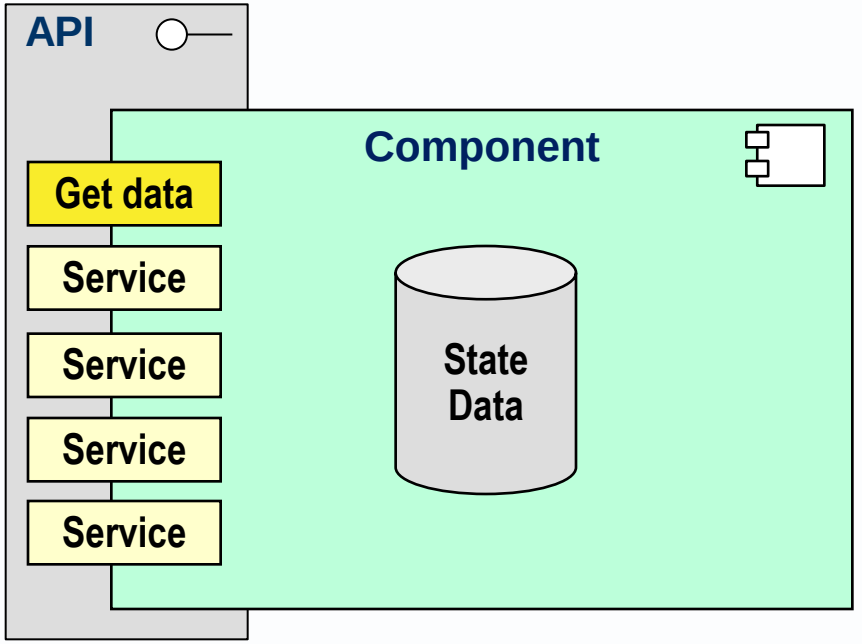
- **Encapsulation**
- Modularisation in the 1970s
- Remote Procedure Call
- The OOP revolution
- The MVC pattern

Deliberate repetition of core concepts from day 1

“Encapsulate components” (“a building block has a defined boundary”)

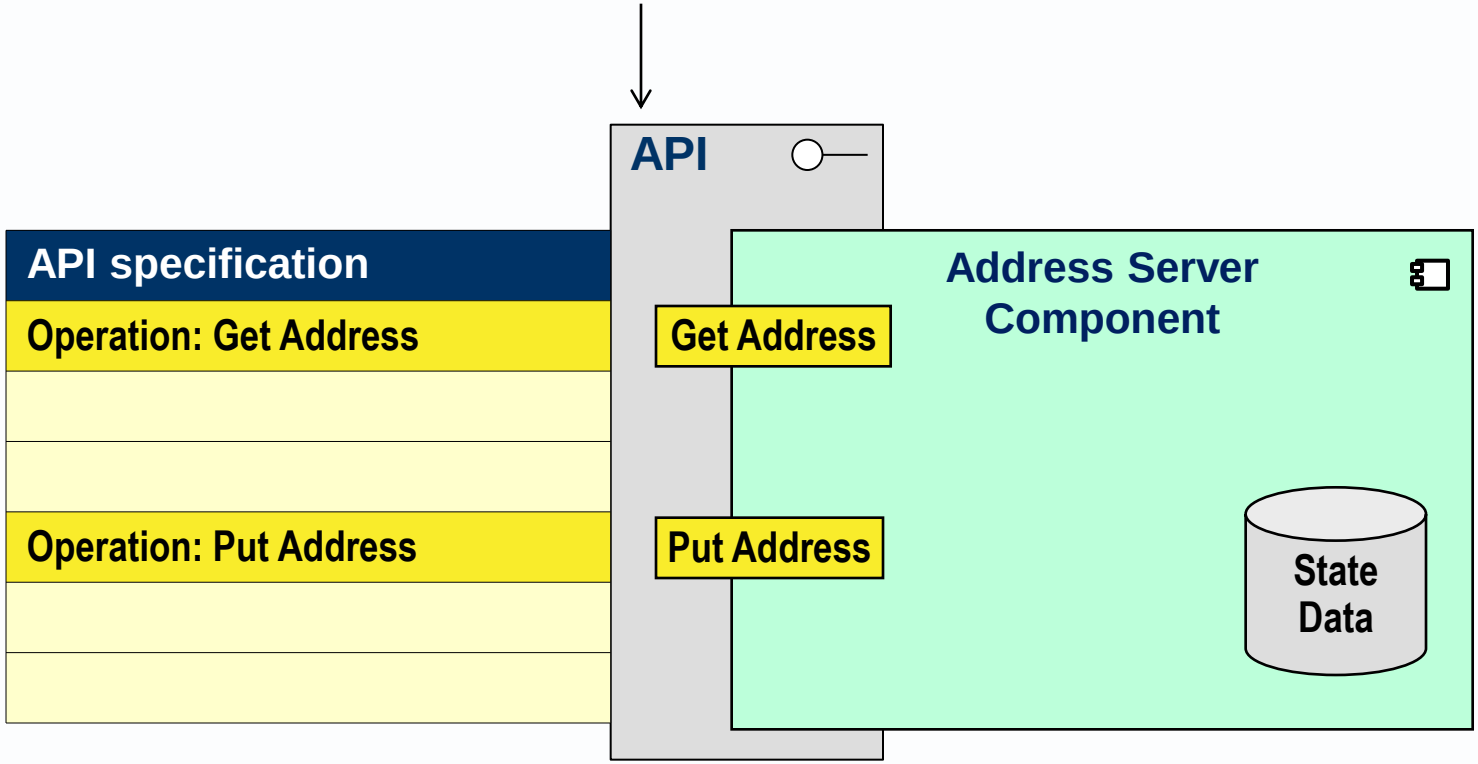
- The enclosure within a component of **processes**, meaning that the inner workings are invisible to outsiders.
- The enclosure within a component of **data**, so the only way to access that data is by using the interface of the component.





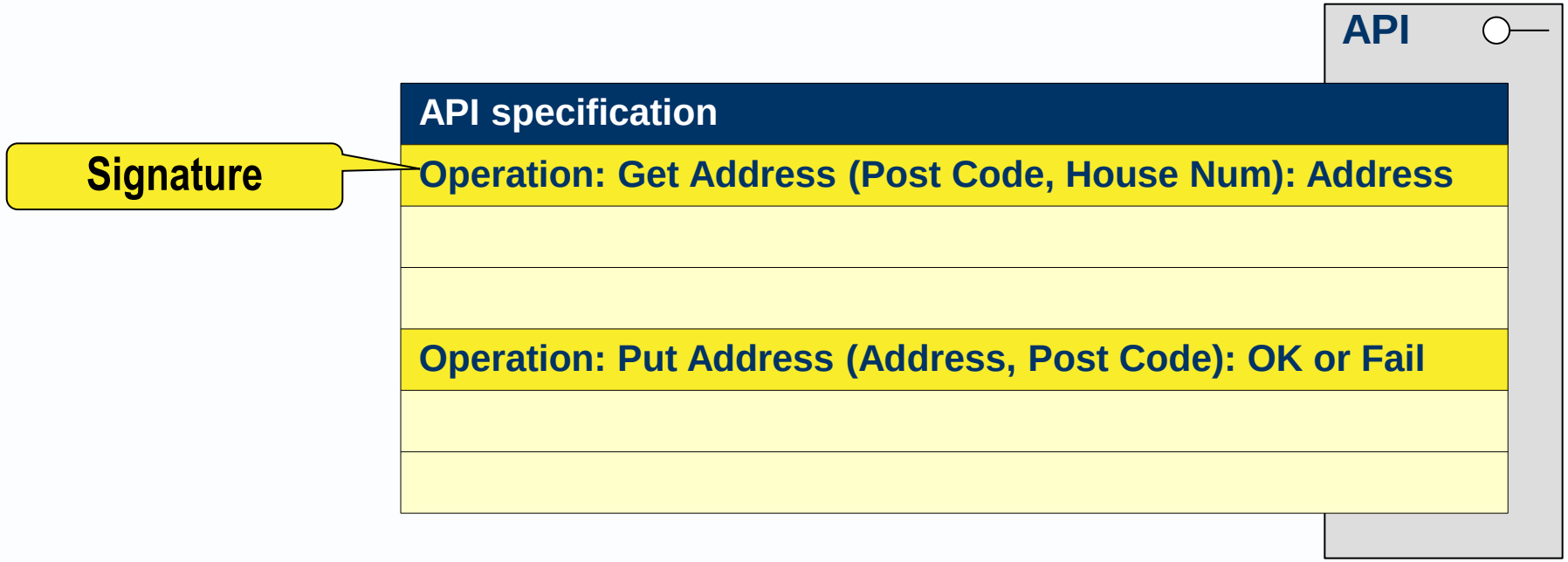
“Use open APIs for inter-component communication”

- To encapsulate means hiding the internal data and procedures of a system (component or object) behind its interface.
- The interface is a collection of accessible services.



Service/Operation signature

- Define an operation's signature
  - Name (Input Parameters): Output Reply



- Clients can make it work using only the signature

Service/Operation semantics or rules

- Designers have to understand, and ideally document, the rules governing an operation.

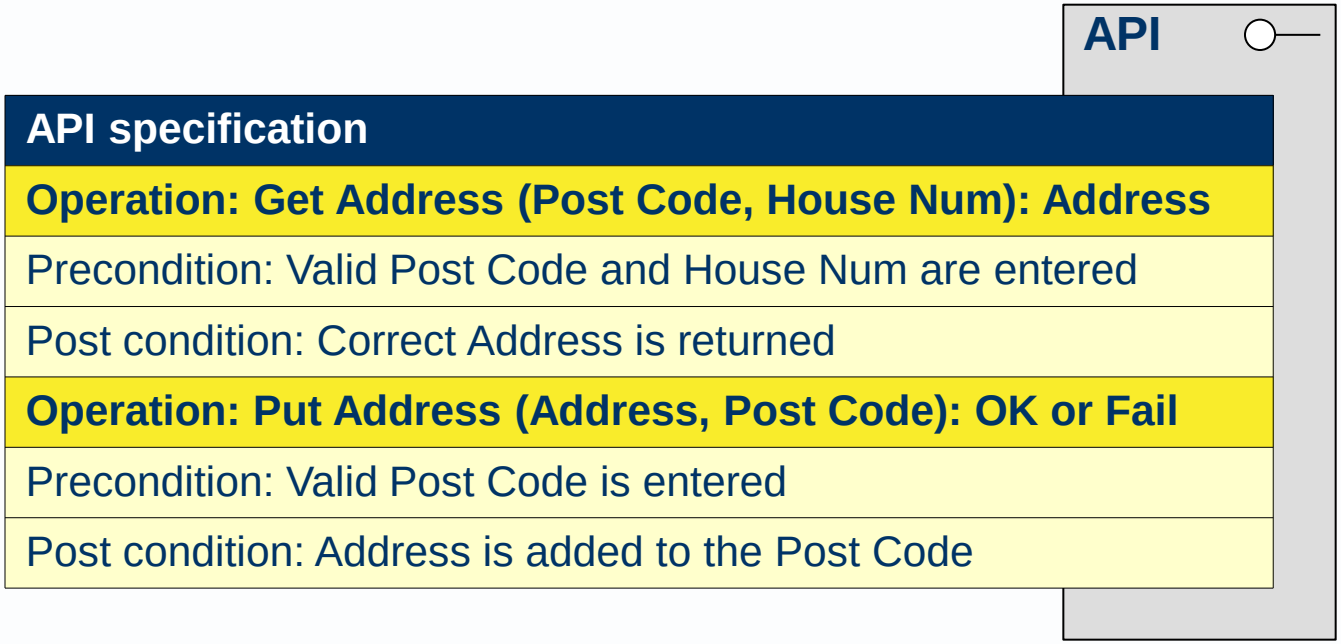
Semantics Or Rules

API specification
Operation: Put Address (Address, Post Code): OK or Fail
Precondition: Valid Post Code is entered
Post condition: Address is added to the Post Code

- **If** the preconditions are true,
- **and** the operation proceeds to completion,
- **then** the post conditions will be true.

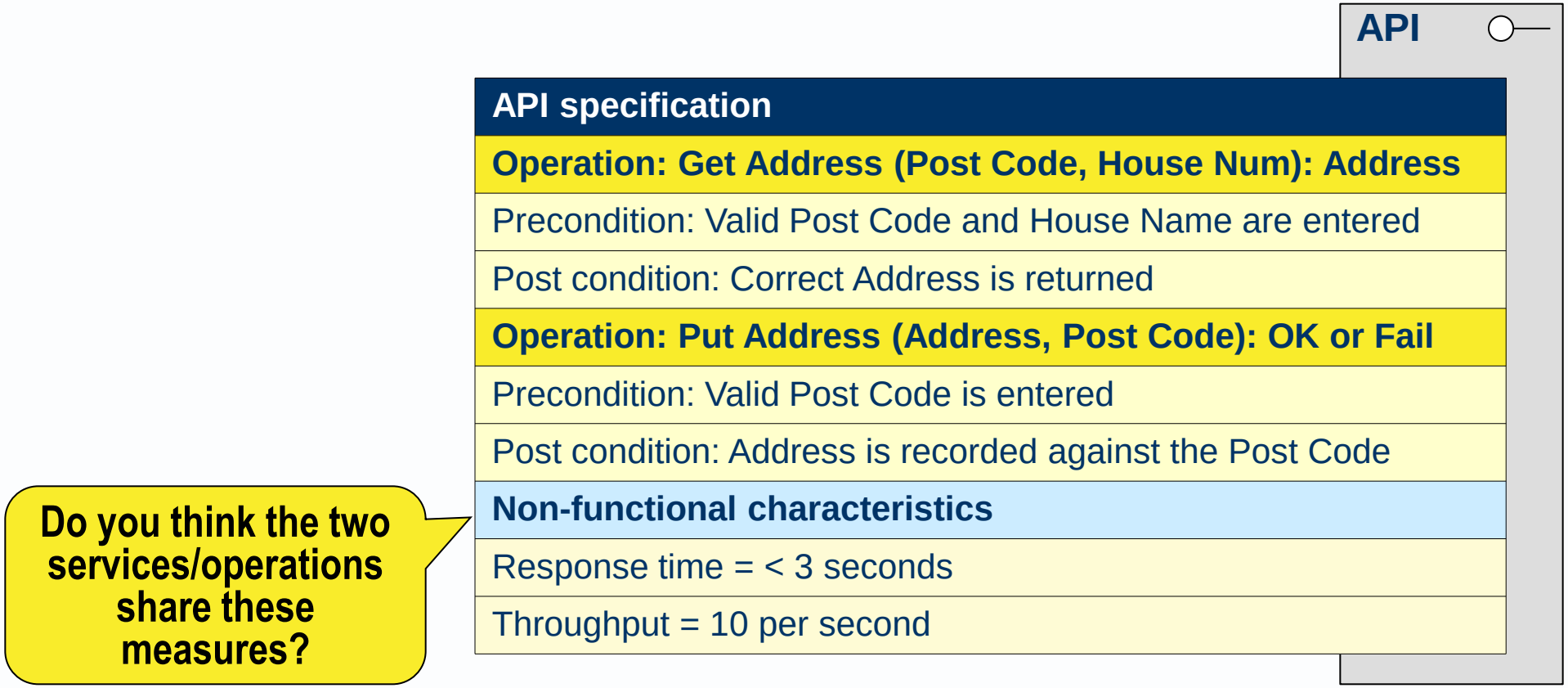
“Test-driven design” using preconditions and post conditions

- Some programming languages come with tools for specifying preconditions and post conditions, and testing them



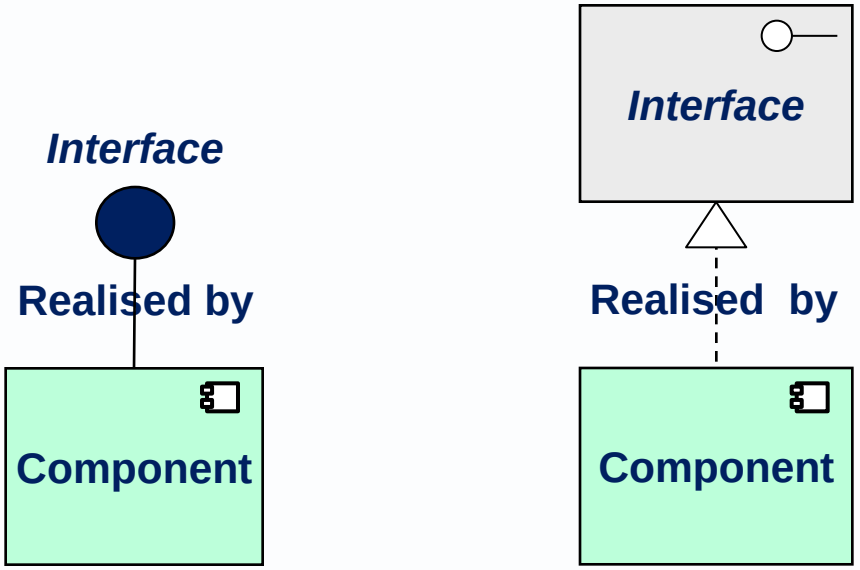
Don’t forge the numbers - and NFRs!

- Some NFRs apply to a whole component
- But also, each service/operation has its own characteristics



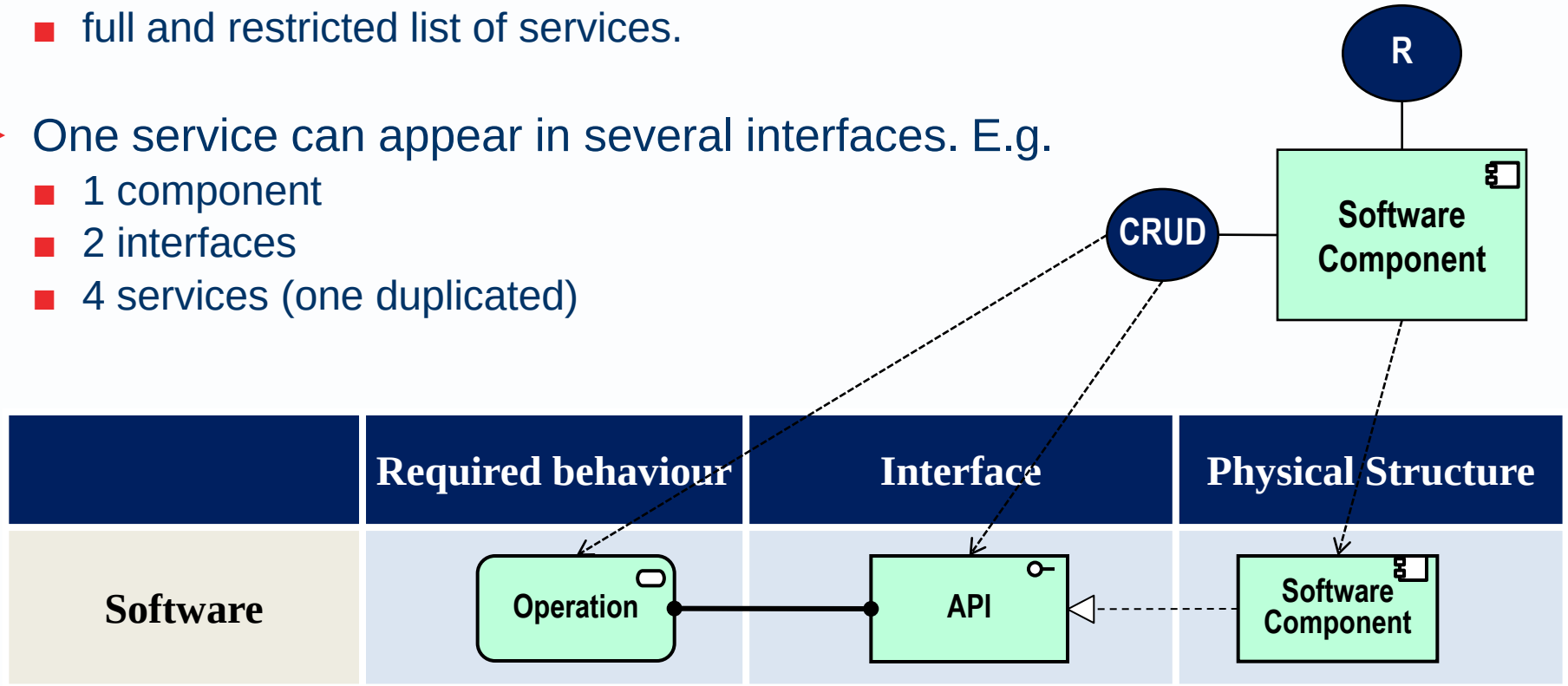
Realisation (software sense)

- Realisation = providing implementations for services in an interface.
- Representable in ArchiMate as below



Separation of component, interface and service

- One component may realise more than one interface.
  - older and newer versions, or
  - full and restricted list of services.
- One service can appear in several interfaces. E.g.
  - 1 component
  - 2 interfaces
  - 4 services (one duplicated)

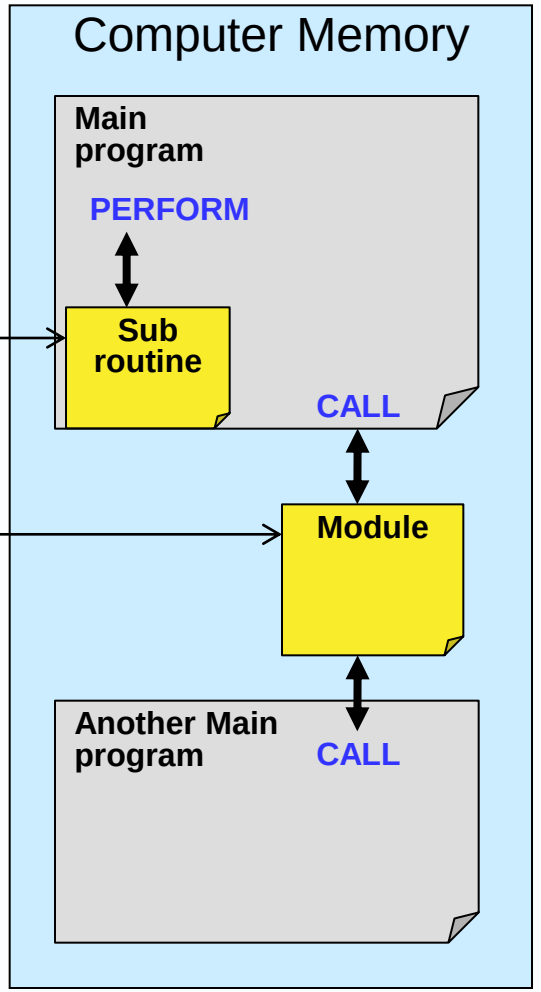


Modularisation in the 1970s

- Encapsulation
- **Modularisation in the 1970s**
- Remote Procedure Call
- The OOP revolution
- The MVC pattern

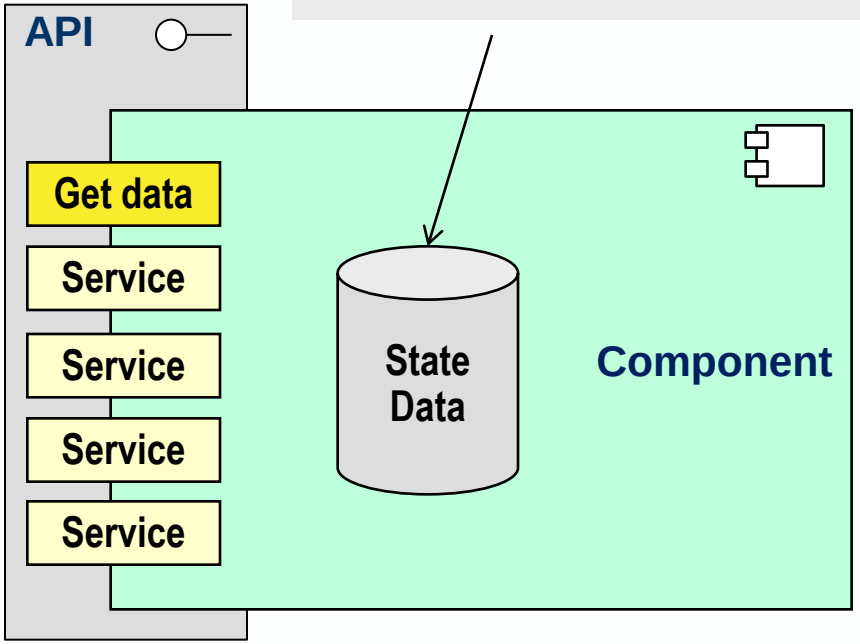
1970s Mainframe modular design – as in COBOL

- A COBOL program can delegate work to a reusable component
- A main program can **perform** an internal subroutine
  - not encapsulated, since program and subroutine can access the same state data
- Several programs can **call** a properly encapsulated external module
- Both “perform” and “call” are **local procedure calls,** little more than a GO TO and a GO BACK in the memory space of a computer



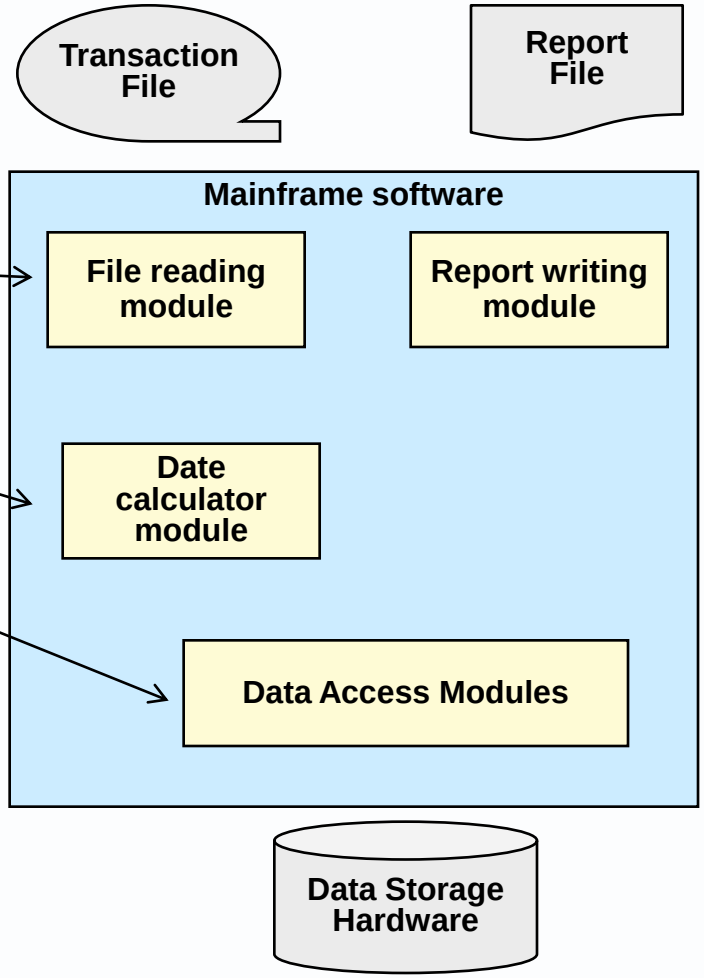
1972 “Criteria to Be Used in Decomposing Systems into Modules”

- Parnas introduced the idea of information hiding.
  - “One begins with a list of difficult design decisions.
  - Each module is designed to hide a decision from the others.
  - [e.g. hide] **a data structure,** its internal links, accessing procedures and modifying procedures.”



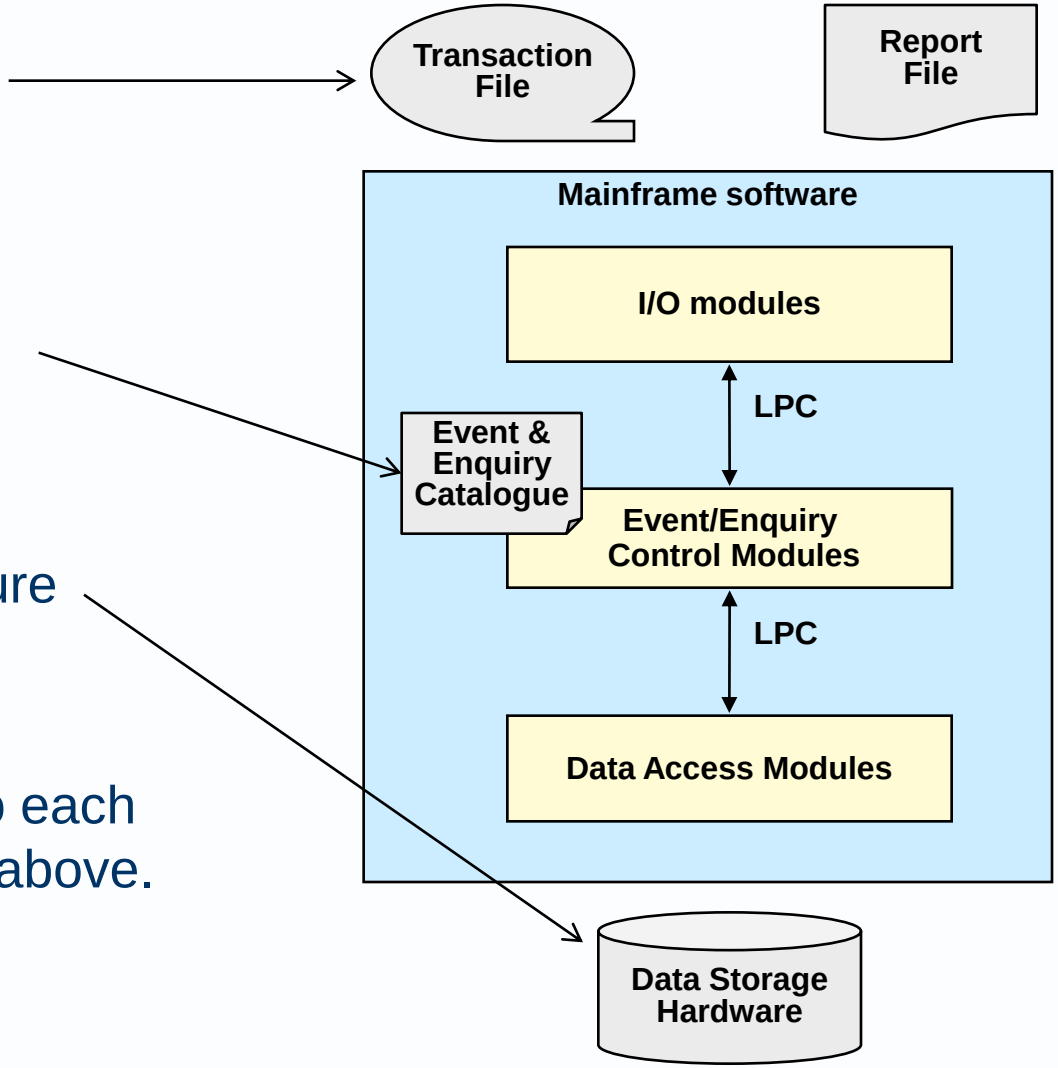
1975 Jackson's modular design patterns

- Jackson taught us to resolve "structure clashes" by designing distinct modules to handle
  - I/O data structures
  - complex data types
  - database structures



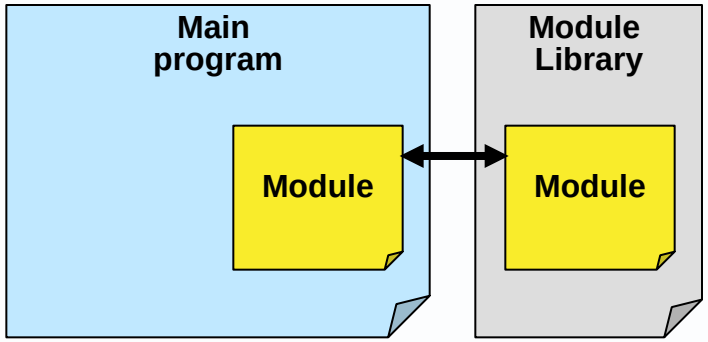
1979 Three-layer software design pattern

- Define the I/O data structures
- Define the events and enquiries contained in the input data
- Define the persistent data structure
- Structure the code into layers, so each layer offers services to the layer above.



Reuse of modules

- There were many attempts to create a shared module library
- Enabling programmers to copy a module into their own program



- Success was patchy
- Around1975, Michael A Jackson reputedly said

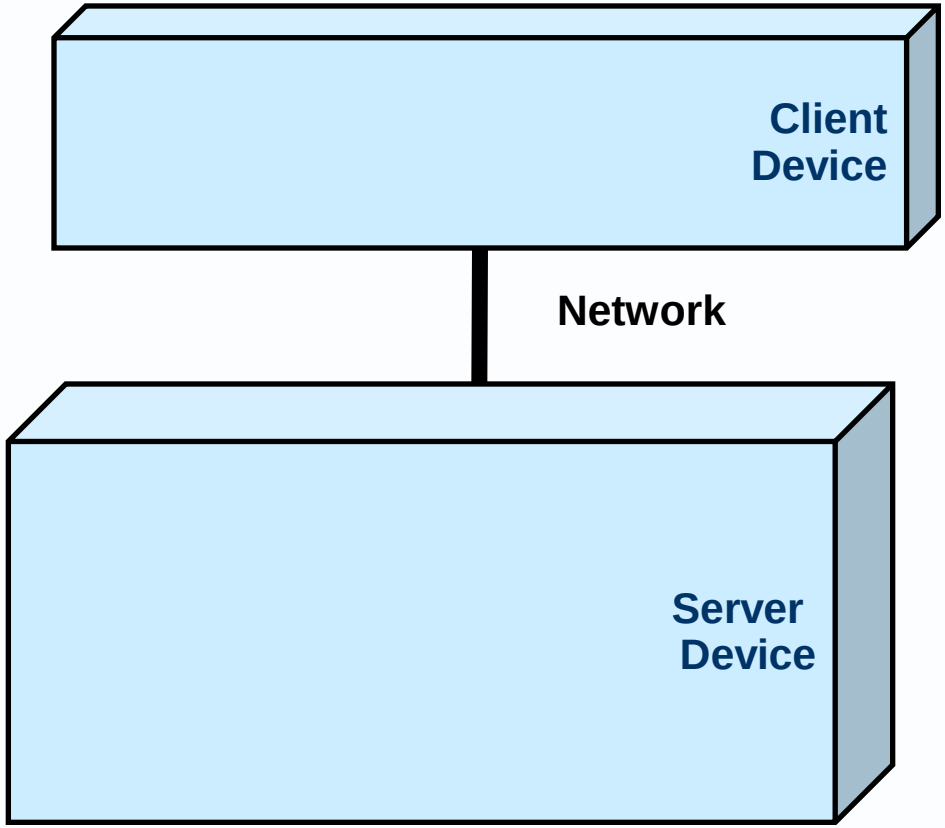
> *"A module library is the only kind of library that everybody wants to put something in, and nobody wants to take something out of".*

Remote Procedure Call

- Encapsulation
- Modularisation in the 1970s
- **Remote Procedure Call**
- The OOP revolution
- The MVC pattern

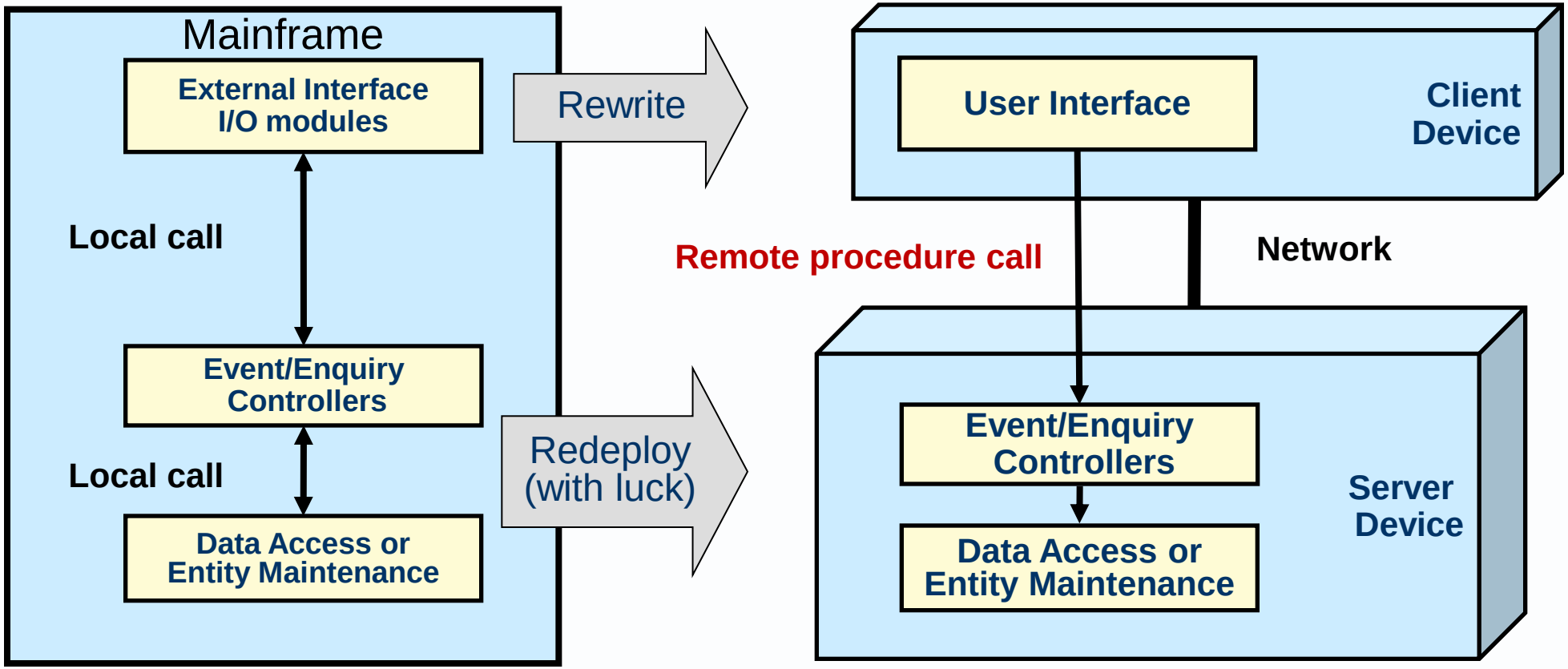
1980s: Client-server design

- The next trend was to *physically* separate code on end-user client devices from code on database servers.



When businesses gave client devices to their employees:

- Enterprise applications had to be divided between modules
  - handling user interface data structures (on client devices)
  - handling business data structures (on database servers).



Remember LPC *versus* RPC

Local Procedure Call

- Simple
- Fast
- Available
- Secure

Remote Procedure Call

- More complex
- Slower
- Less available
- Less secure

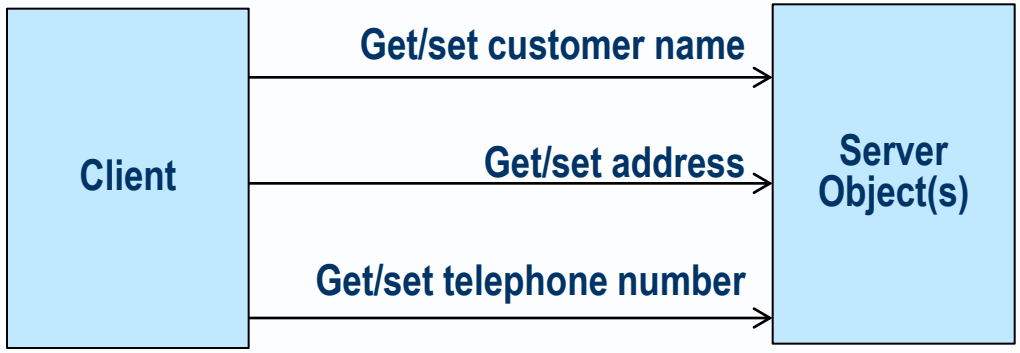
LPC *versus* RPC

Avancier

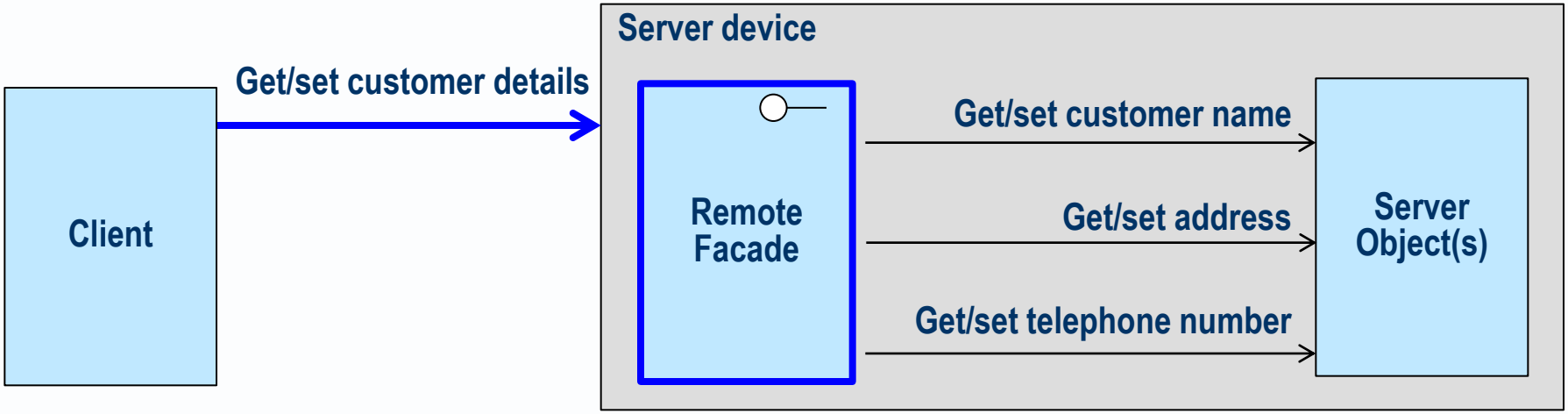
Local Procedure Call	Remote Procedure Call
1. The client makes a local procedure call to the server.	1. The client makes a local procedure call to the client stub (parameters pushed on to the stack)
	2. The client stub packs (marshalls) the parameters into a message and makes an OS call
	3. The client's local OS sends the message from to the server's machine
	4. The server's local OS passes the incoming packets to the server stub
	5. The server stub unpacks (unmarshalls) the parameters from the message
	6. The server stub calls the server procedure.
2. The server does what is requested	7. The server does what is requested
	8. The server replies to the server stub
	9. The server stub packs (marshalls) the parameters from the message
	10. The server's local OS passes the incoming packets to the server stub
	11. The server's local OS sends the message from to the client machine
	12. The client stub unpacks (unmarshalls) the parameters into a message
3. The server replies to the client	13. The client stub and replies to the client.

How physical distribution influences logical modularisation

- Local interfaces - fine-grained



- To reduce network chatter, remote interfaces need to be coarse-grained



The OOP revolution

- Encapsulation
- Modularisation in the 1970s
- Remote Procedure Call
- **The OOP revolution**
- The MVC pattern

Approaching 1990, OOP became fashionable

- OO programming is different in three ways

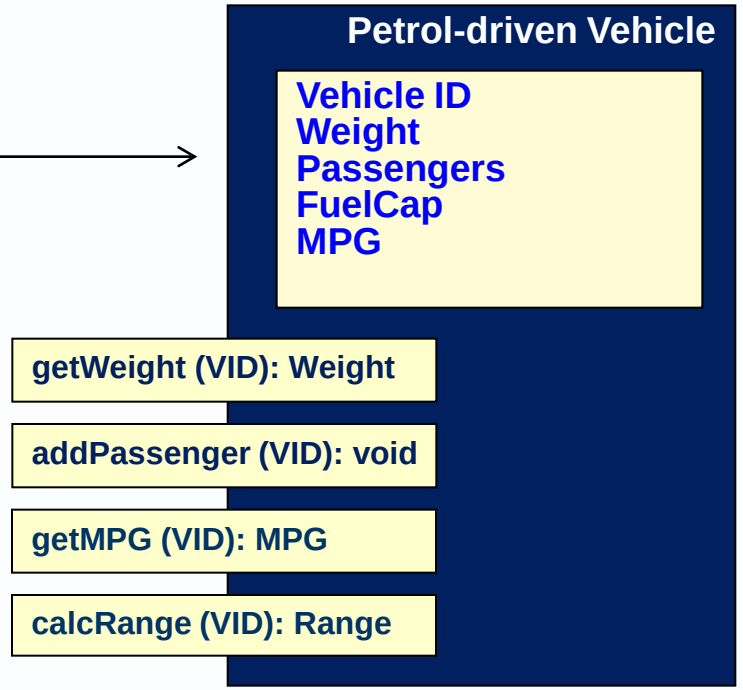
1. Many instances (**objects**) of a module type (**class**)
   - So many objects can be deployed in parallel

2. A module instance has a unique **object identifier**
   - So clients can locate an object, and remember it

3. Classes can be related in a class hierarchy
   - So objects of one class can **inherit** and **extend** the operations of another more generic class

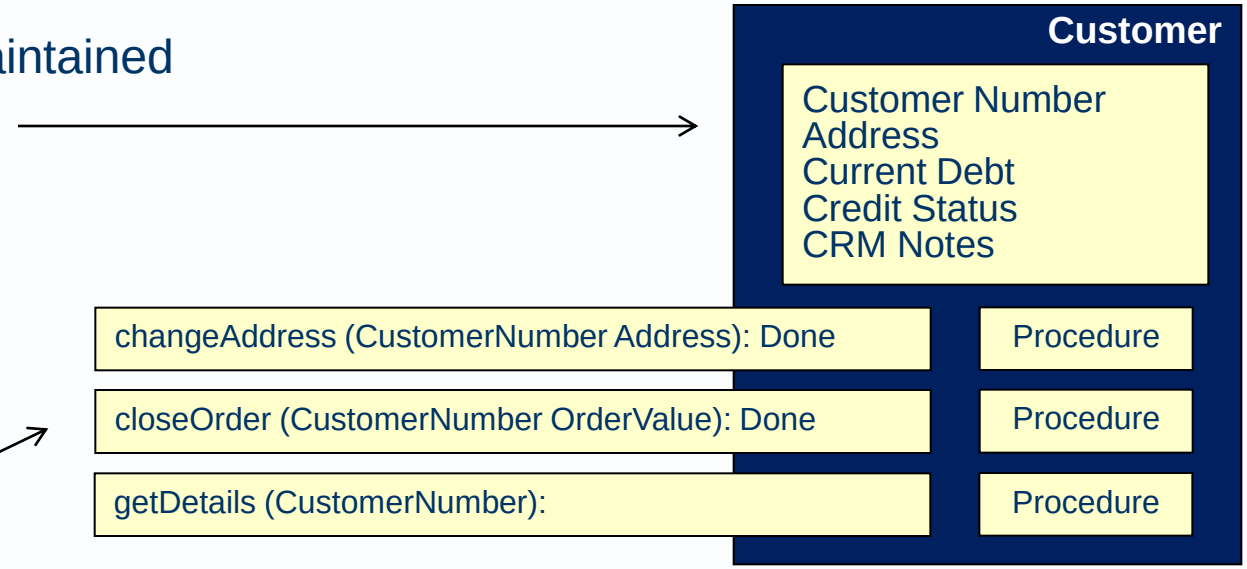
Criteria to Be Used in Decomposing Systems into Classes

- In the decade after Parnas and Jackson, OO guru Bertrand Meyer proposed classes should be **abstract data types**, which
  - encapsulate a data structure
  - define operations performable on that data structure.



Class = module type

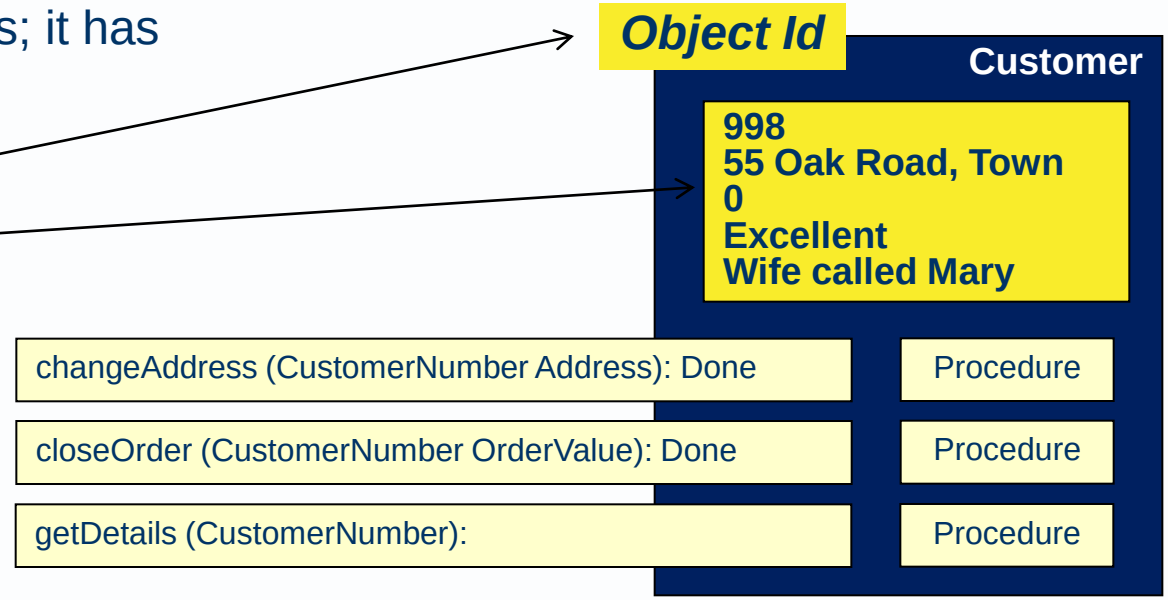
- A **class** is a template for creating objects;
- It may define **data types** maintained
  - variables, fields, attributes



- And an interface containing **operations**
  - External Service (aka Method or Operation)
  - Internal Procedure (aka Method Body)

Clients need object identifiers

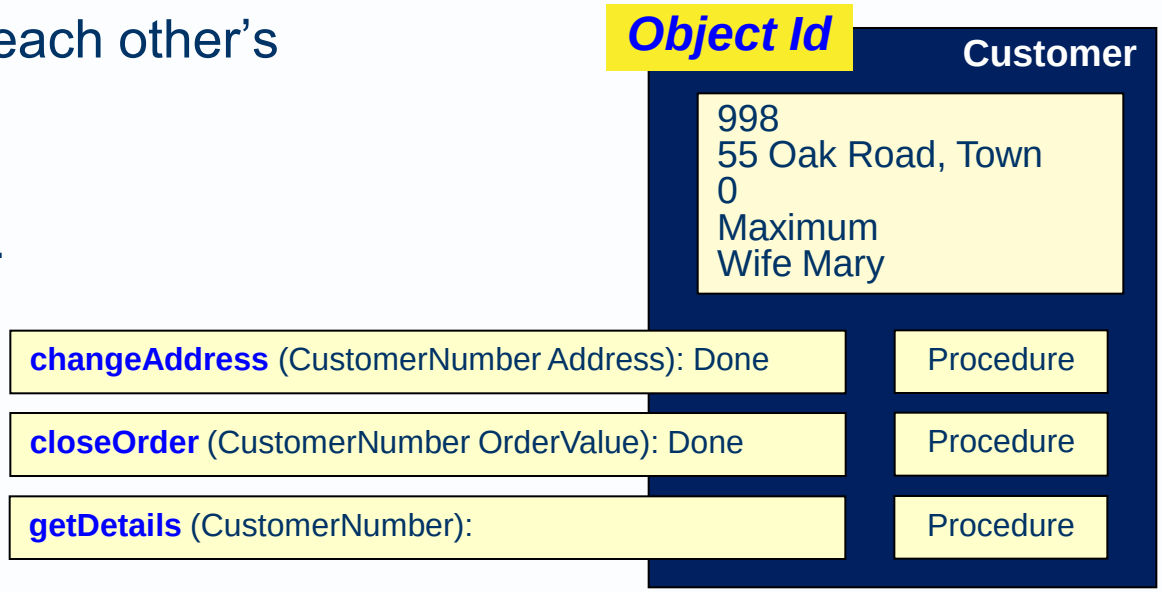
- When “instantiated” in memory an **object** is an instance of a class; it has
  - Identity
    - *unique* object Id - used by programmers
  - State
    - *unique* data values
  - Behaviour
    - *general* operations



- You call it to do work
- Using an “object/method pair”
  - E.g. 99999/closeOrder

One name space

- Client and server objects work in the *same name space*
- So they know and can use each other's
  - class names
  - object identifiers and
  - operation or method names.



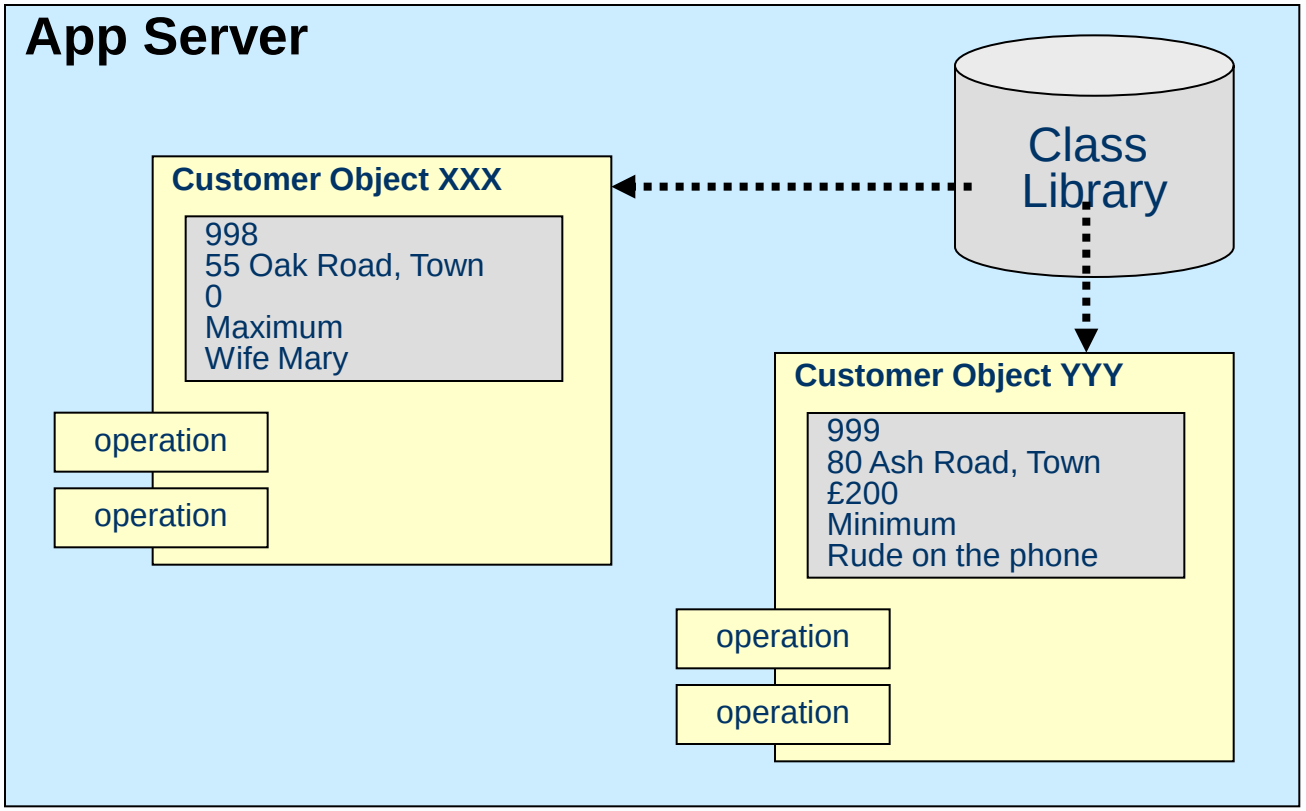
Stateful objects/modules

- Stateful objects persist, and retain their state in memory

- The earliest OO programs handled a few small stateful objects.
- The case studies were
  - real-time process control systems – objects live forever
  - graphical user interfaces - objects deleted when the UI is closed

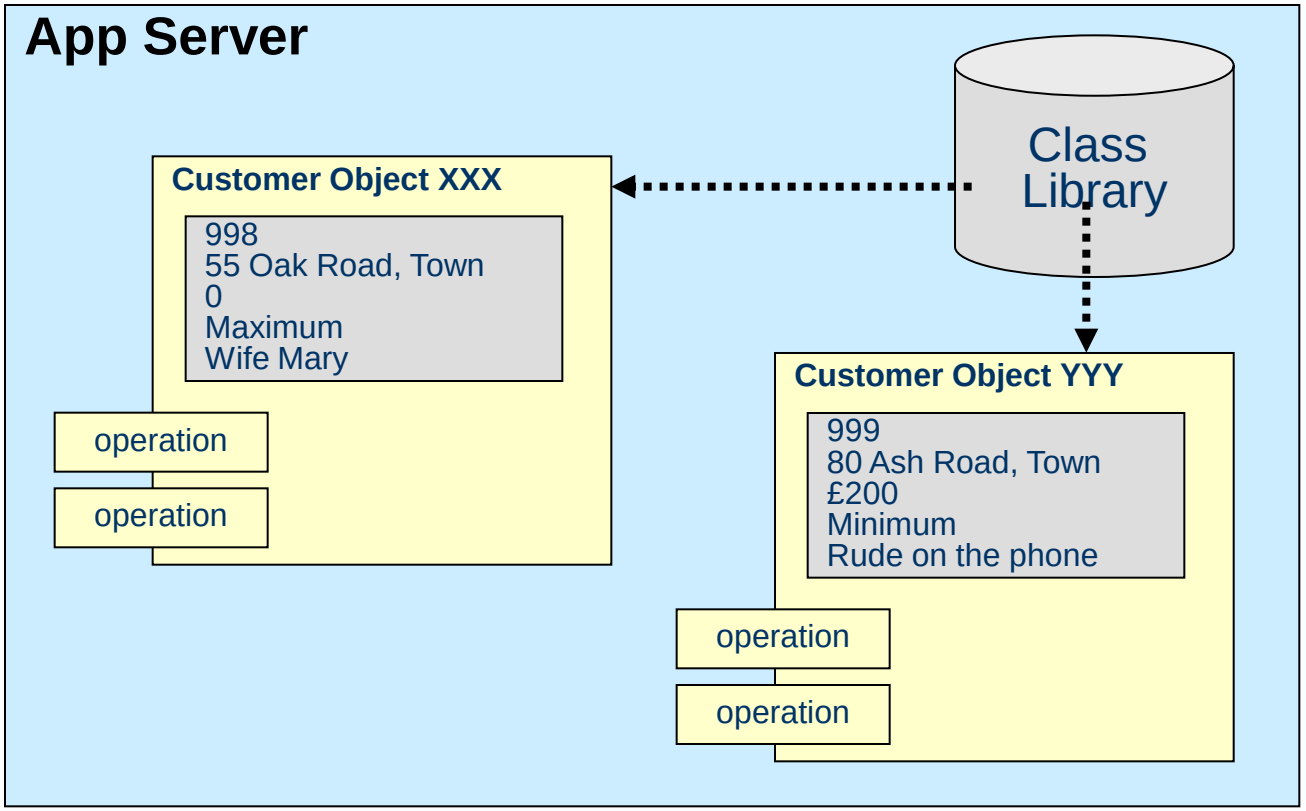
The OO envangelist view

- Objects model real world entities.
- Objects (like real world entities) are stateful and persist.
- *Databases are merely infrastructure devices and SQL is evil*!



Applying the idea to enterprise-scale applications

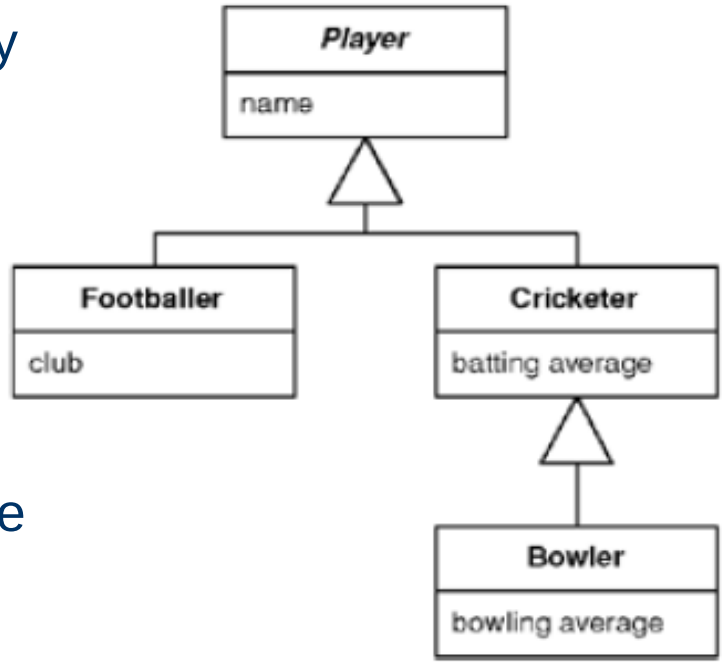
- "Pure [Distributed Object] implementations are anarchistic. Objects can appear anywhere at any time. No one knows how to get rid of them. Not good for apps w millions of run-time objects." (Client/server survival guide" 3rd edition 1999)



- So there was a need for "garbage collection"
- Objects with object identifiers that no other object remembers can be deleted

Reuse by inheritance

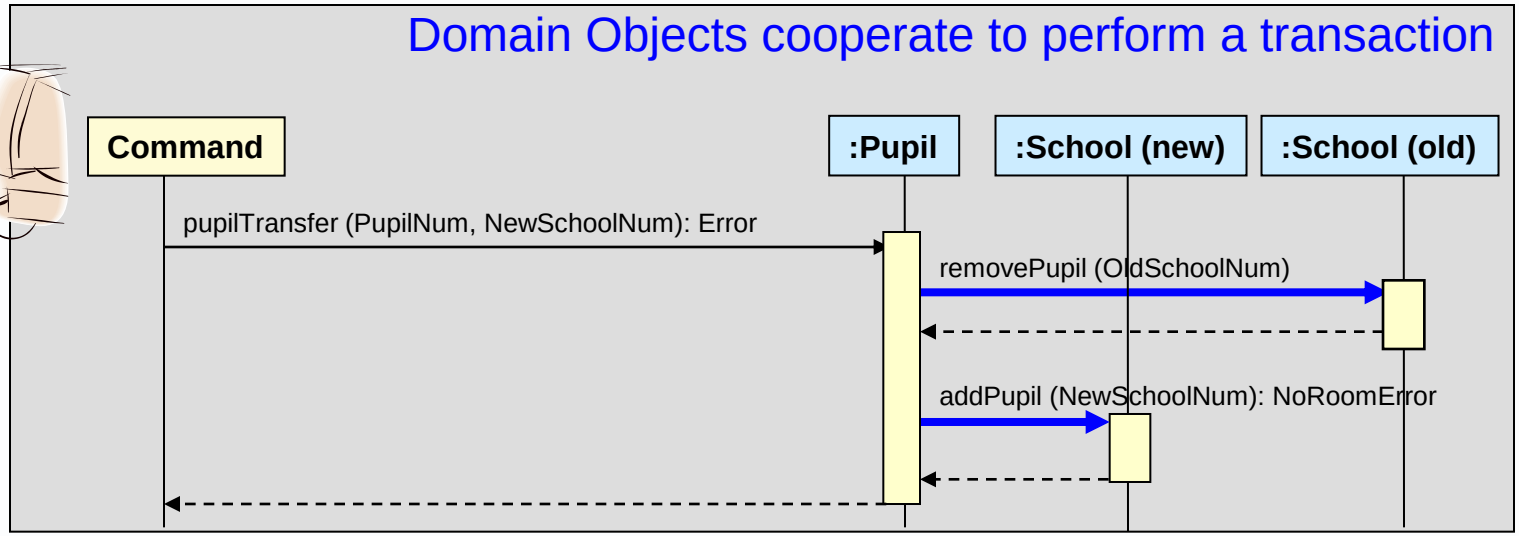
- Classes can be related in a class hierarchy



- 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

Intelligent domain objects



Synchronous request-reply invocations

- The *client* sees the communication as *synchronous*

- Client objects invoke server objects using an “object-method pair”
  - an object id and a method or operation name
- Hold a connection and wait for a reply

Blocking servers

- The *server* sees the communication as *synchronous*
- The server blocks all clients bar the current one

The OO envangelists' ambition

- Turn the world into one big OO program (one name space)
- Intelligent domain objects can run on different machines
- And cooperate as though they run on the same machine.

- Does this mean client objects must remember the network addresses of remote server objects?

- To be continued…

Summary of early OO design presumptions

Avancier

Feature	Early OO design presumptions	Recent SOA design presumptions?
Naming	Clients need object identifiers One name space	
Paradigm	Stateful objects/modules Reuse by OO inheritance Intelligent domain objects	
Time	Request-reply invocations Blocking servers	
Location	Remember remote addresses	

COBOL modules
Java objects
CORBA

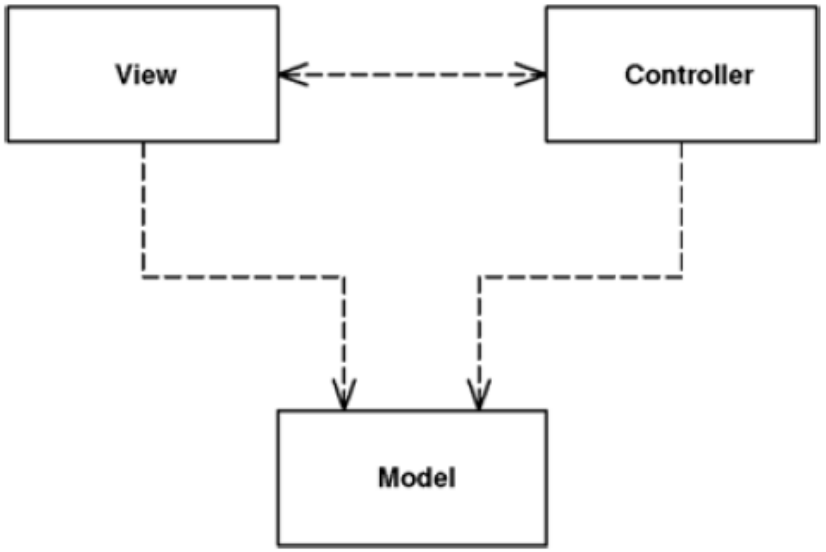
The MVC pattern

- Encapsulation
- Modularisation in the 1970s
- Remote Procedure Call
- The OOP revolution
- **The MVC pattern**

The MVC pattern: as Martin Fowler introduces it

Model View Controller

Splits user interface interaction into three distinct roles

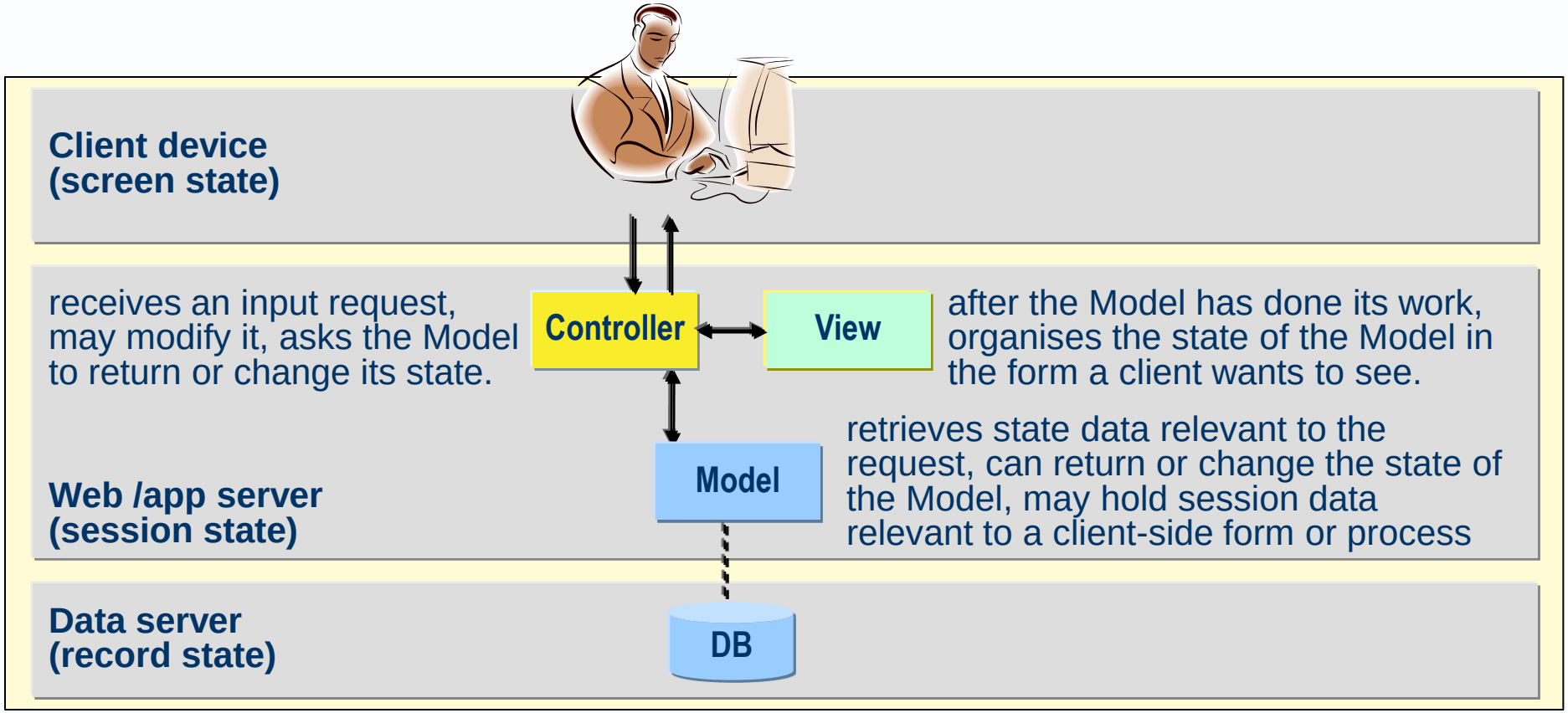


Model View Controller (MVC) is one of the most quoted (and most misquoted) patterns around. It started as a framework developed by Trygve Reenskaug for the Smalltalk platform in the late 1970s. Since then it has played an influential role in most UI frameworks and in the thinking about UI design.

- Originally developed for desktop computing in the 1970s.
- The Model contains Objects representing things that appear on the screen, regardless of the visual format

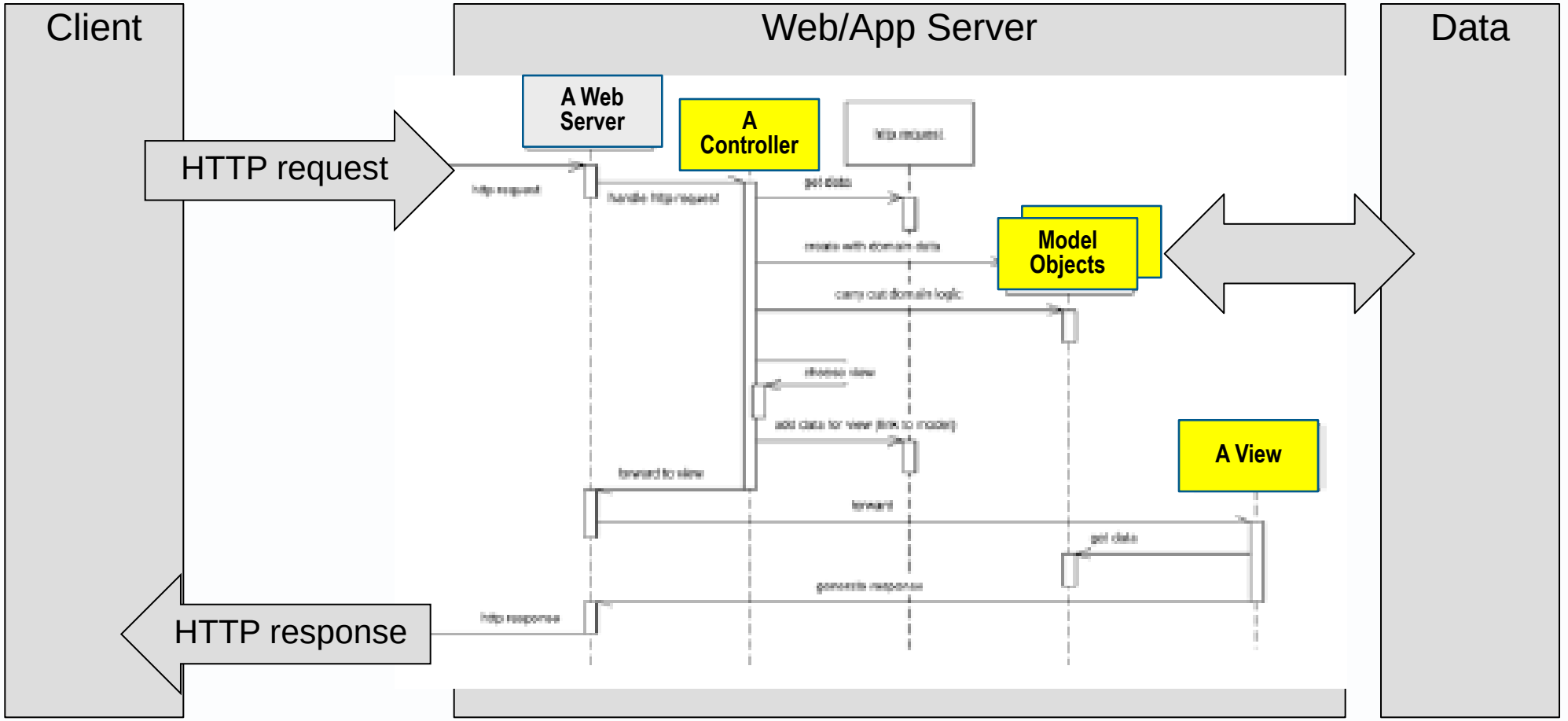
Adapting the MVC pattern for enterprise applications

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



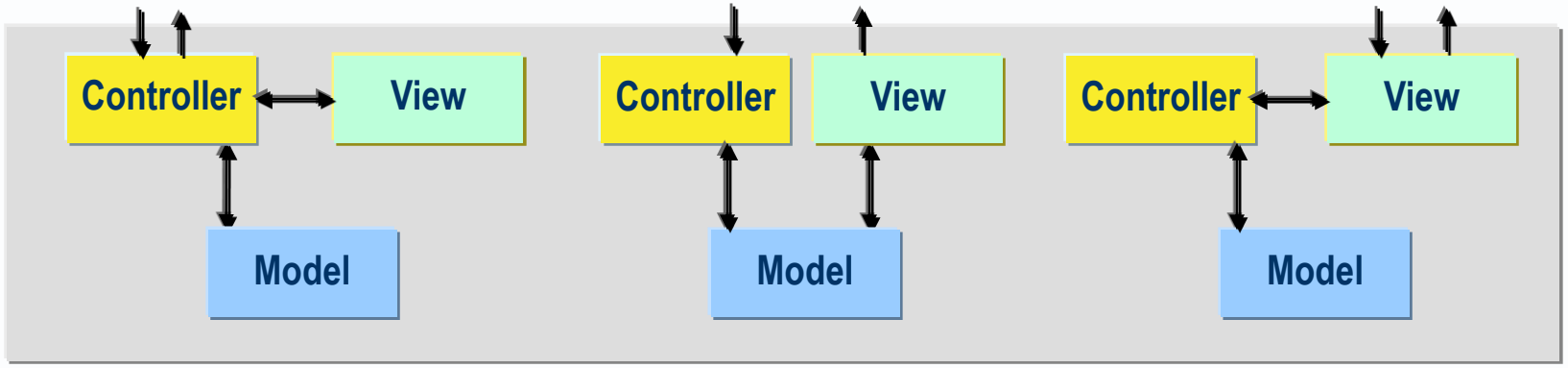
Adapting the MVC pattern for enterprise applications

- Graphic based on how Fowler represents MVC interactions



Adapting the MVC pattern for enterprise applications

- There are many variations of the pattern



- 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 (Wikipedia says)

- Web frameworks divide MVC differently between client and server tiers.

- Early web frameworks mostly put MVC components **on the server**.
  - E.g. Ruby on Rails, Django, ASP.NET MVC and Express
  - Client sends hyperlink requests or form input to the controller
  - Client receives a complete and updated web page (or other doc) from the view.

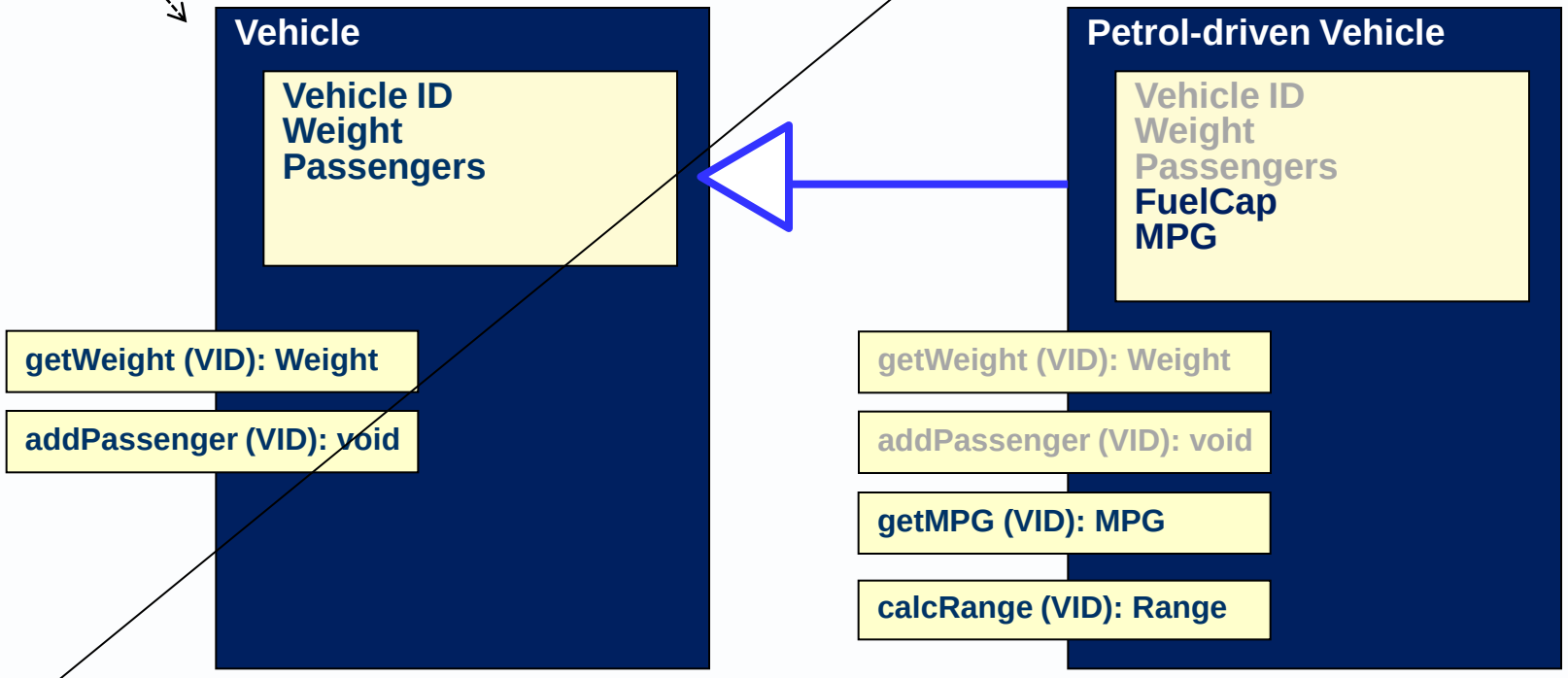
- Other frameworks allow MVC components to execute partly **on the client**
  - E.g. AngularJS, EmberJS, JavaScriptMVC and Backbone (also see Ajax).

Footnotes

- On inheritance and OO-relational mapping

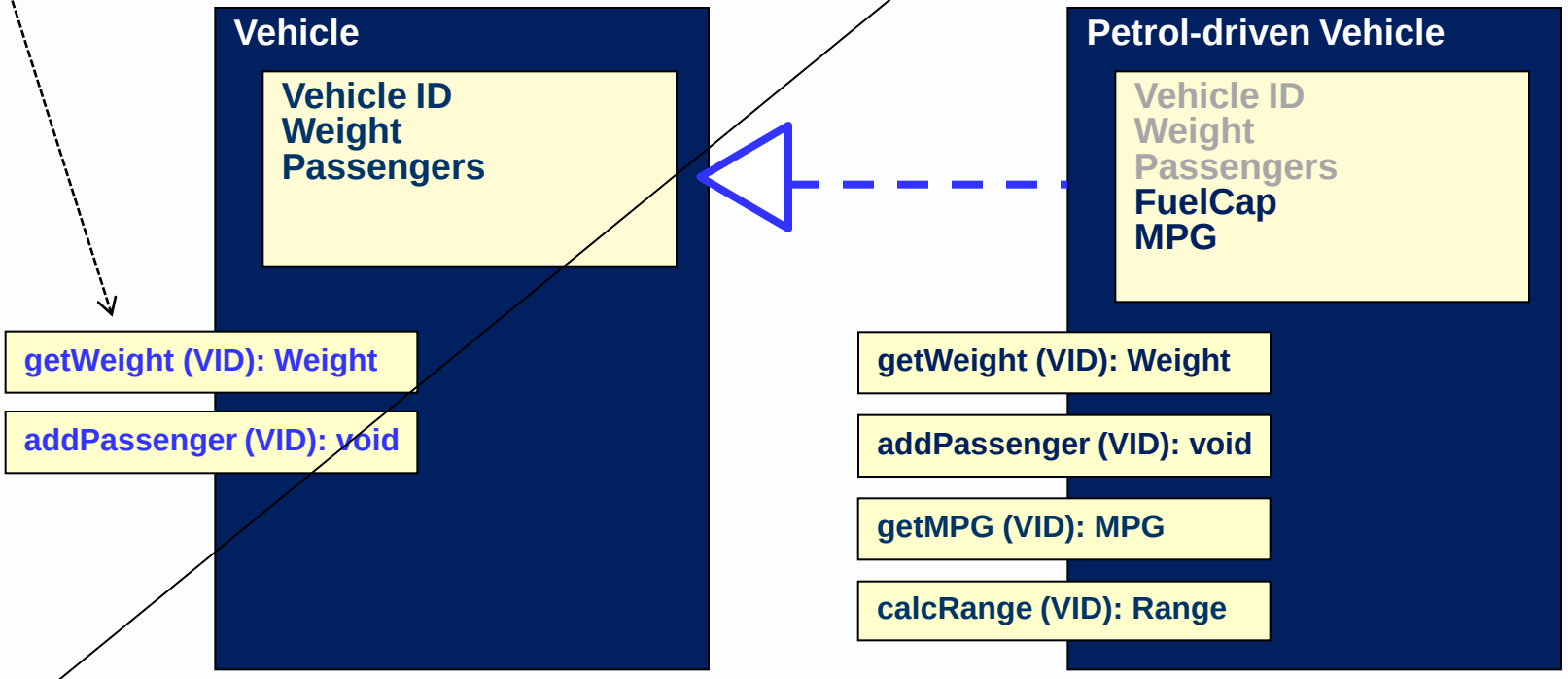
Base class

- a superclass to which designers can add subclasses that “extend”, “inherit from” or “derive from” the base class.



Abstract or virtual operation

- an operation for which the signature (name, inputs and outputs) is defined, but not the internal procedure needed to implement the operation.



Abstract or virtual class

- presents an interface containing nothing but abstract or virtual operations.