Avancier Methods

Very basic design patterns

It is illegal to copy, share or show this document
(or other document published at http://avancier.co.uk)
without the written permission of the copyright holder

Copyright Avancier Limited 2007-2016

Design patterns

- There are patterns for
  - enterprise architecture
  - solution architecture
  - software architecture

- A design pattern is a general shape that recurs in many cases
- It should
  - be a general solution to a general design problem
  - be a tried and tested design
  - speed up development
  - encourage consistency, help standardise how design is done in an enterprise.
  - reduce the risk of reliance on an individual designer.

Copyright Avancier Limited 2007-2016

Patterns as diagrams

- Some patterns are available in the form of code.
- But patterns are usually represented in diagrams

"If you think you have a pattern, you must be able to draw a diagram of it. This is a crude, but vital rule. "

"If you can't draw it, it isn't a pattern."

"A pattern defines a field of spatial relations, and it must always be possible to draw a diagram for every pattern. "
Christopher Alexander (1979) in The Timeless Way of Building

Copyright Avancier Limited 2007-2016

Encapsulation

- Many design patterns are based an assumption

- Each component in it is **encapsulated** and defined by
  - its input/output interface
  - the discrete events it can process and services it can offer.

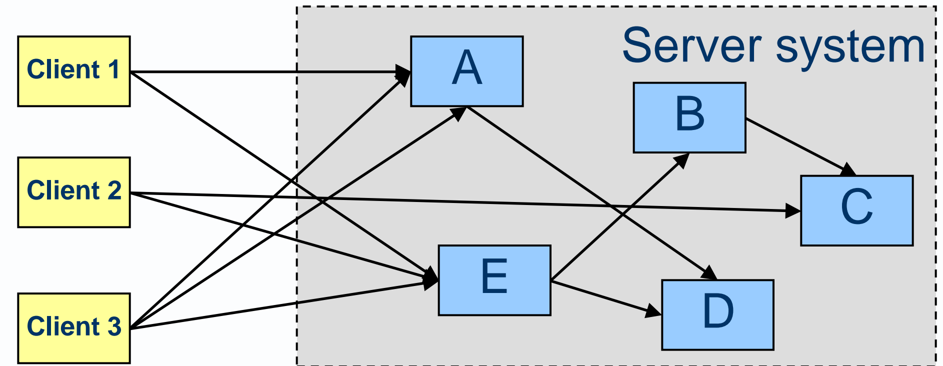
- On its own, encapsulation isn't a pattern
- But it is usually assumed in other patterns.

Copyright Avancier Limited 2007-2016

A simple and common pattern: Façade

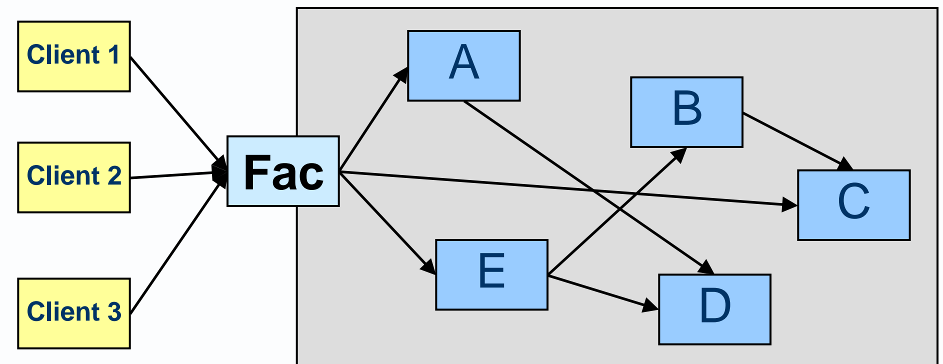
Problem:

- How can a client avoid being too tightly coupled with a server subsystem?
- How to aggregate services into a coarser-grained component?



Solution:

- Introduce a façade that provides a unified interface for different clients, and makes the subsystem easier to use
- So clients know what the subsystem does for them, but nothing else



Copyright Avancier Limited 2007-2016

Fowler example of a Facade

Avancier

- The façade simplifies a remote invocation for an address

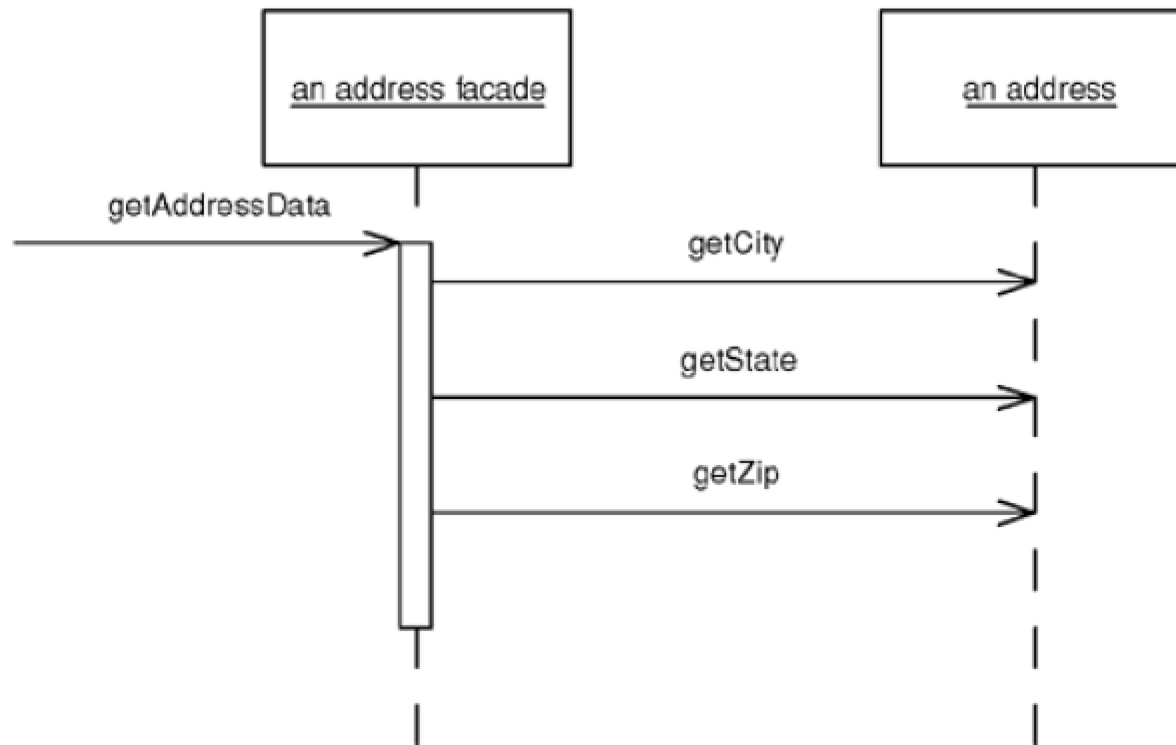


Figure 15.1. One call to a facade causes several calls from the facade to the domain object

Copyright Avancier Limited 2007-2016

Alternative patterns

- Capable architects
  - understand the available patterns
  - look to use them in the right circumstances
  - choose between **alternative** patterns by trading off their pros and cons

Copyright Avancier Limited 2007-2016

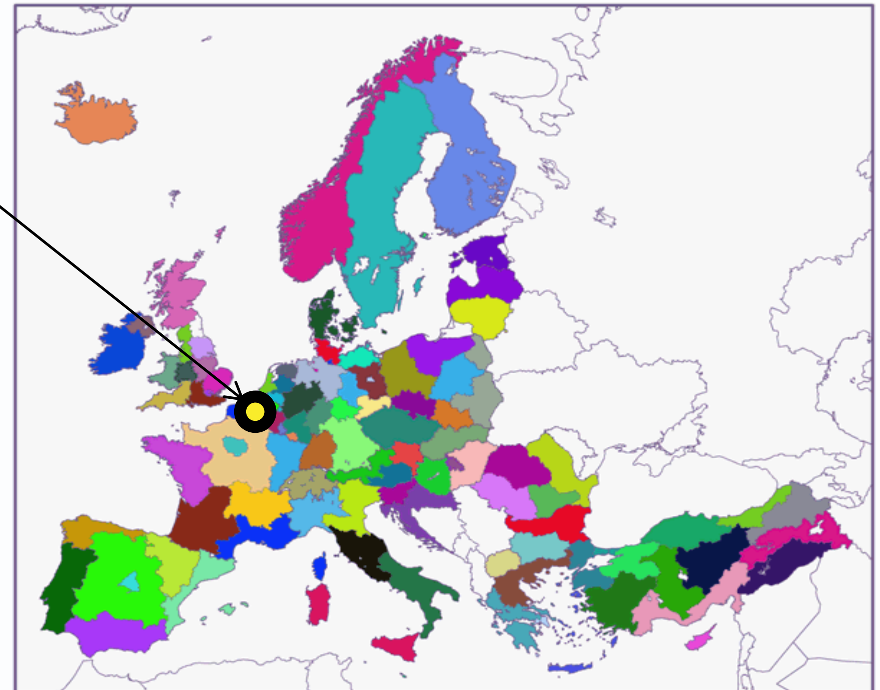
Alternative patterns: Centralisation and Distribution

Avancier

A never-ending debate

How far should powers be

- distributed to many nodes/places? → a "Europe of Regions"?
- centralised in one node/place?

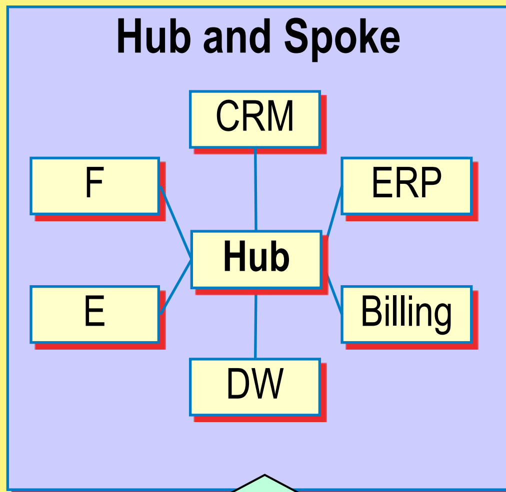


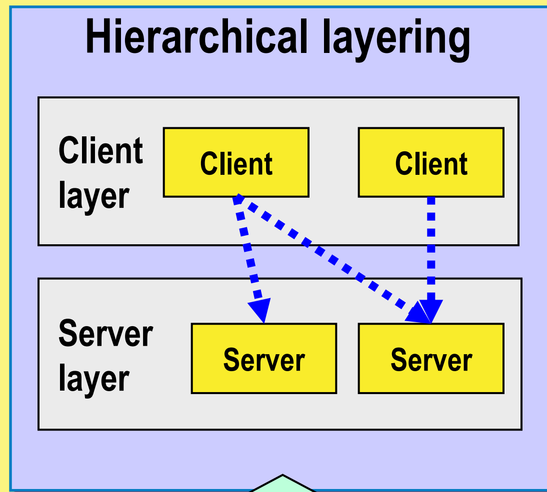
Copyright Avancier Limited 2007-2016

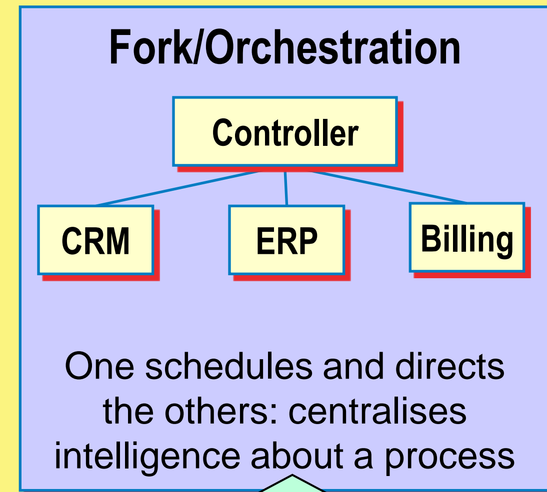
Alternative patterns to be discussed

Avancier

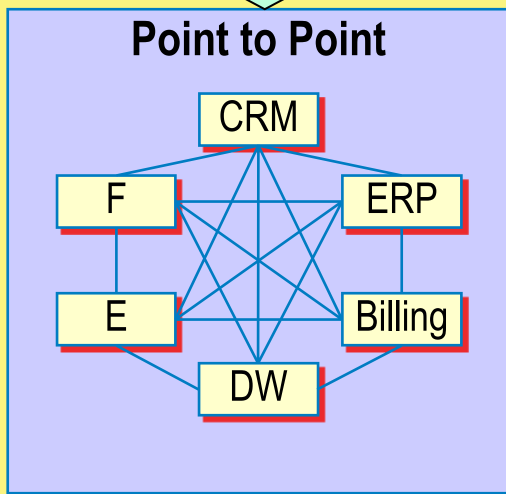
Centralising

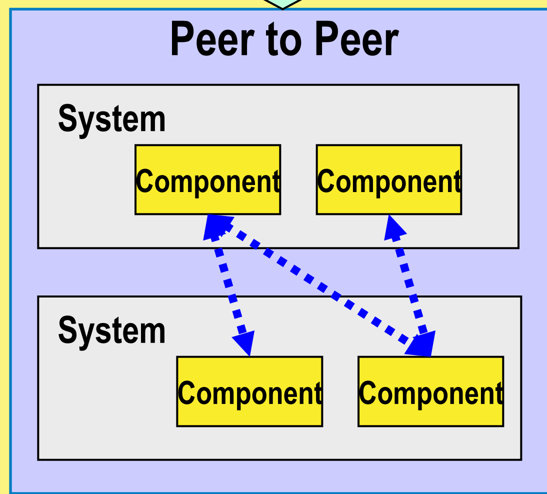


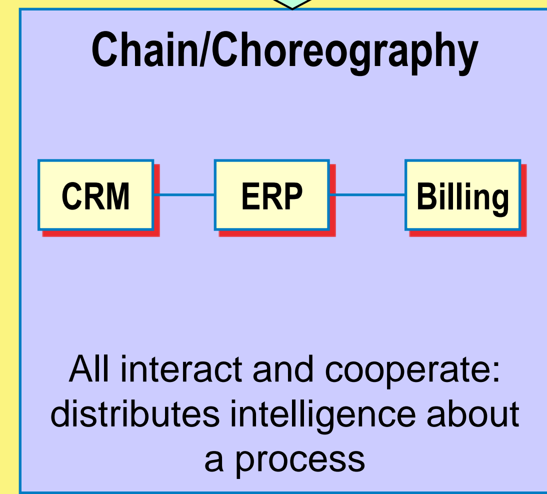




Distributing

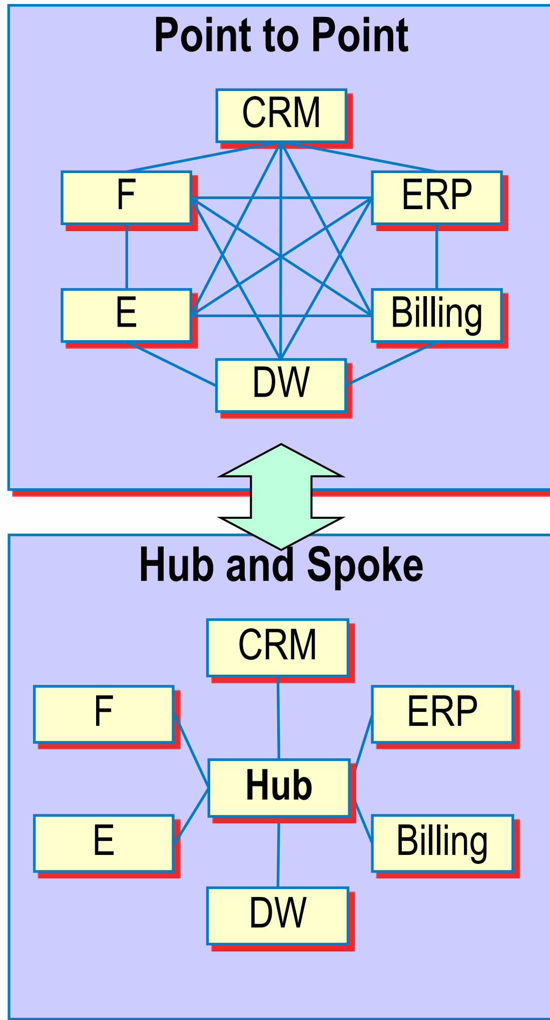






Copyright Avancier Limited 2007-2016

Point to Point v Hub and Spoke



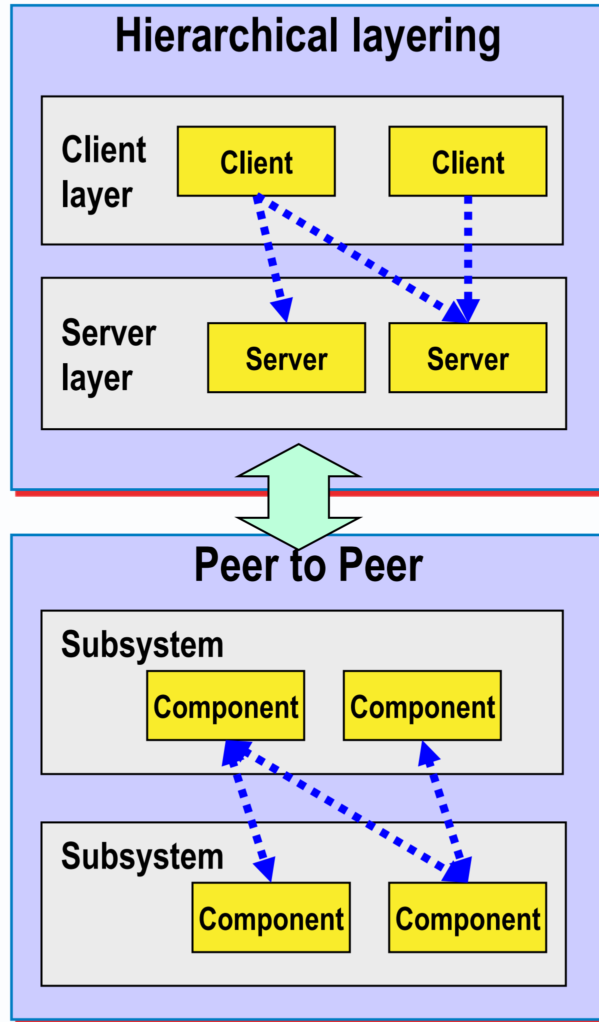
- Subsystems talk to each other directly.
- Can be faster and simpler
- OK where
  - inter-component communication is 1 to 1
  - components at either endpoint are stable.

- Subsystems communicate via some kind of mediator or middleware.
- Can be more complex and slower than point-to-point integration.
- Better where
  - inter-component communication is many to many
  - components at either endpoint are volatile.

Copyright Avancier Limited 2007-2016

Hierarchical layer v Peer to Peer

Avancier



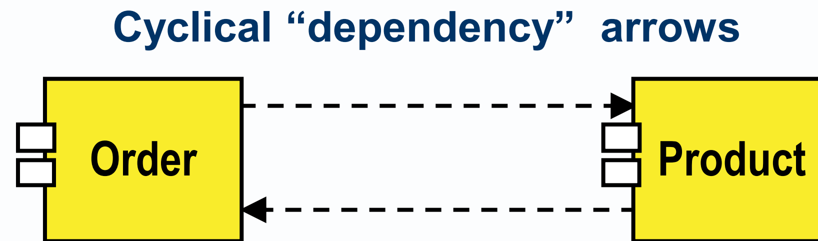
- Oft used to structure a complicated system.
  - Machine architecture (programming language, OS, device drivers and CPU instruction sets, logic gates inside chips)
  - Network architecture (FTP, TCP, IP, Ethernet).
  - Software architecture (UI, Logic, Database)

- Sometimes said to be a bad thing
  - fragile and unstable structure
  - difficult to understand and maintain
  - undermine testability, parallel development, and reuse.

Copyright Avancier Limited 2007-2016

Peer-to-peer structures

- Given a system structure with co-dependent components

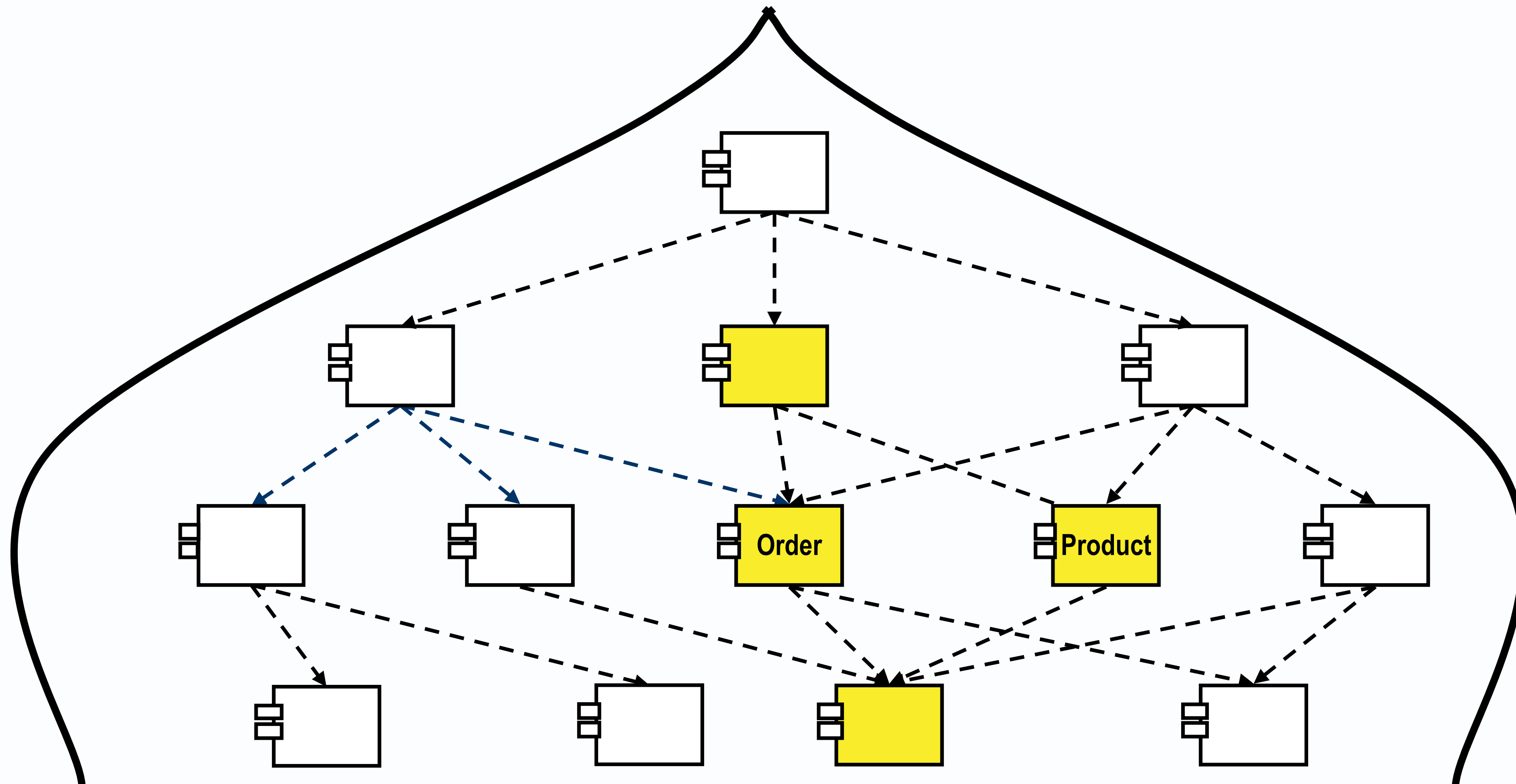


- Suppose you restructure it to eliminate all cyclic dependencies?
- The result will be a hierarchically-layered structure

Copyright Avancier Limited 2007-2016

Hierarchically layered structure

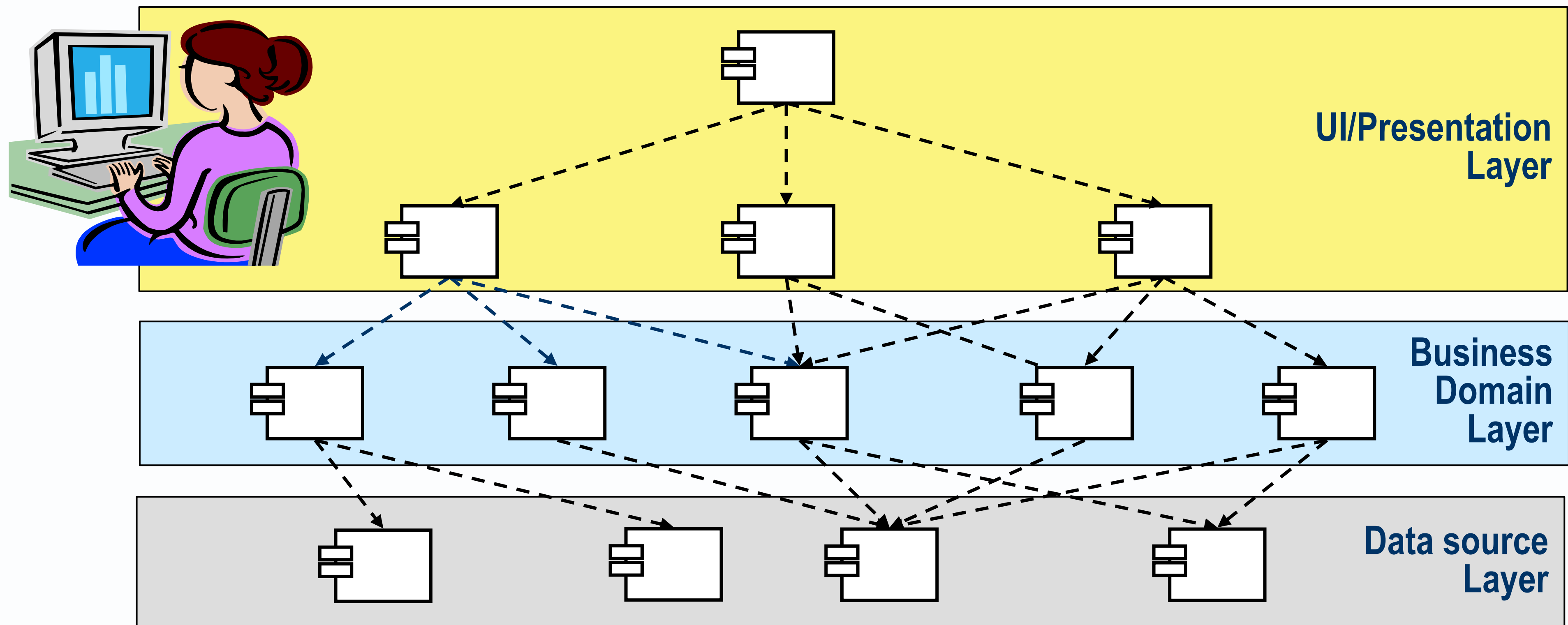
- Higher-level components depend on lower-level ones, but not vice-versa
- The structure often resembles what Barry Boehm called a Mosque Shape.



Copyright Avancier Limited 2007-2016

Hierarchical layering in enterprise applications

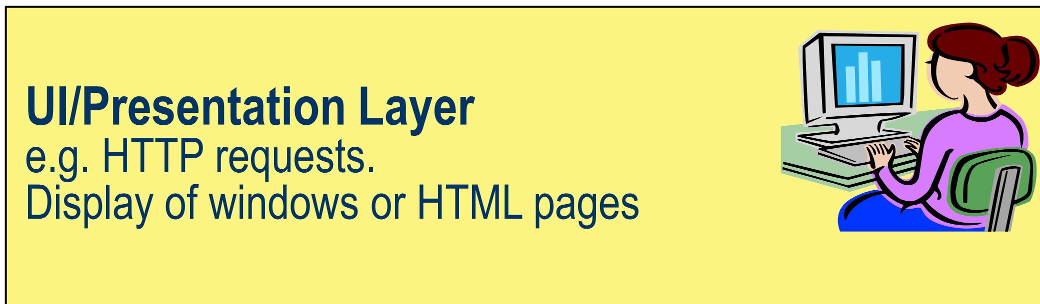
- Variations of a three-layer software architecture are common



Copyright Avancier Limited 2007-2016

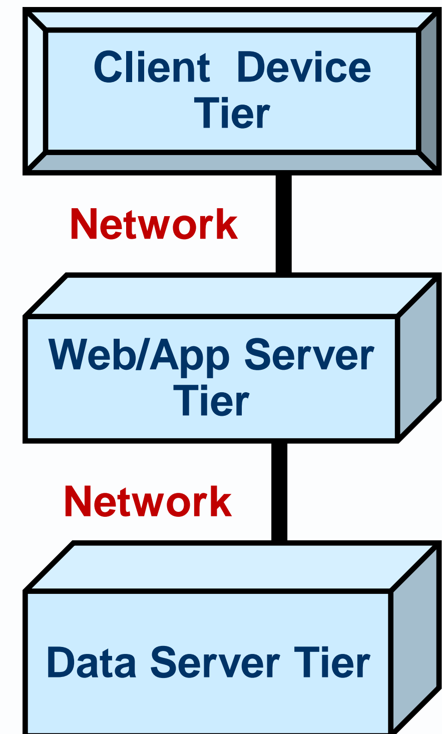
Software Layers <> Platform Tiers

- Layers and Tiers influence each other
- But don't perfectly correspond



Business Domain Layer
The business logic that is the function of the system.
Triggered by commands and queries.

Data source layer
Communication with databases, transaction managers, messaging systems etc.



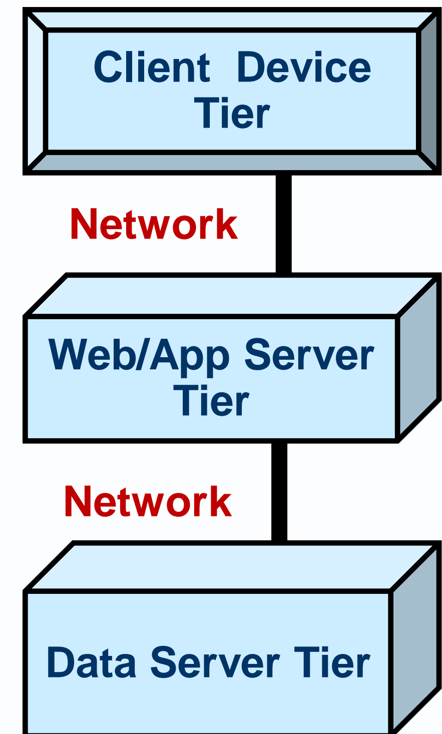
Copyright Avancier Limited 2007-2016

Where are business domain rules executed?

- A mantra of early OO design was that all business rules belong (in domain objects) in the app server tier

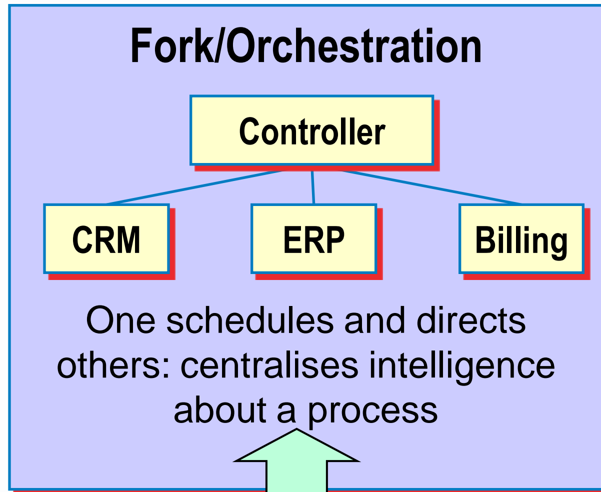
In practice

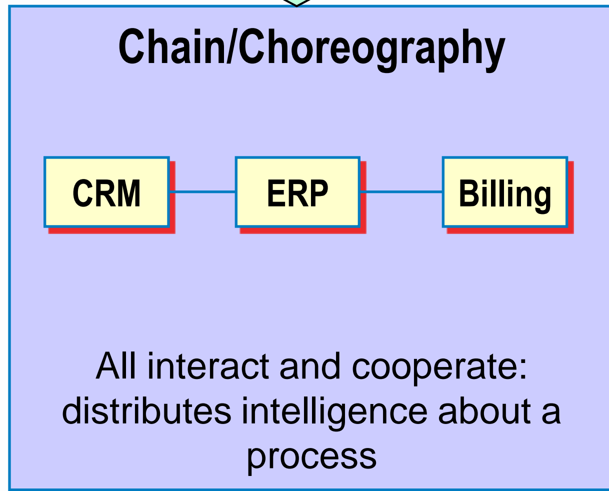
- Validation of input data items against data types ->
- Workflow logic and some business rules ->
- Data integrity and other data-bound rules ->



Copyright Avancier Limited 2007-2016

Fork/Orchestration v Chain/Choreography

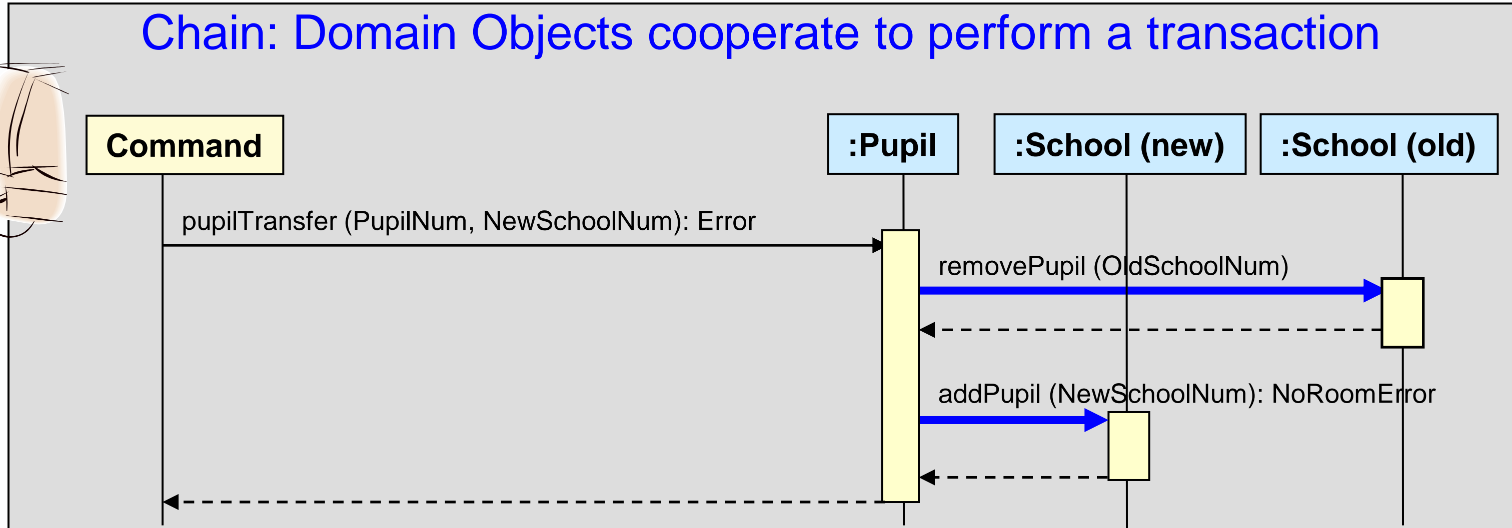


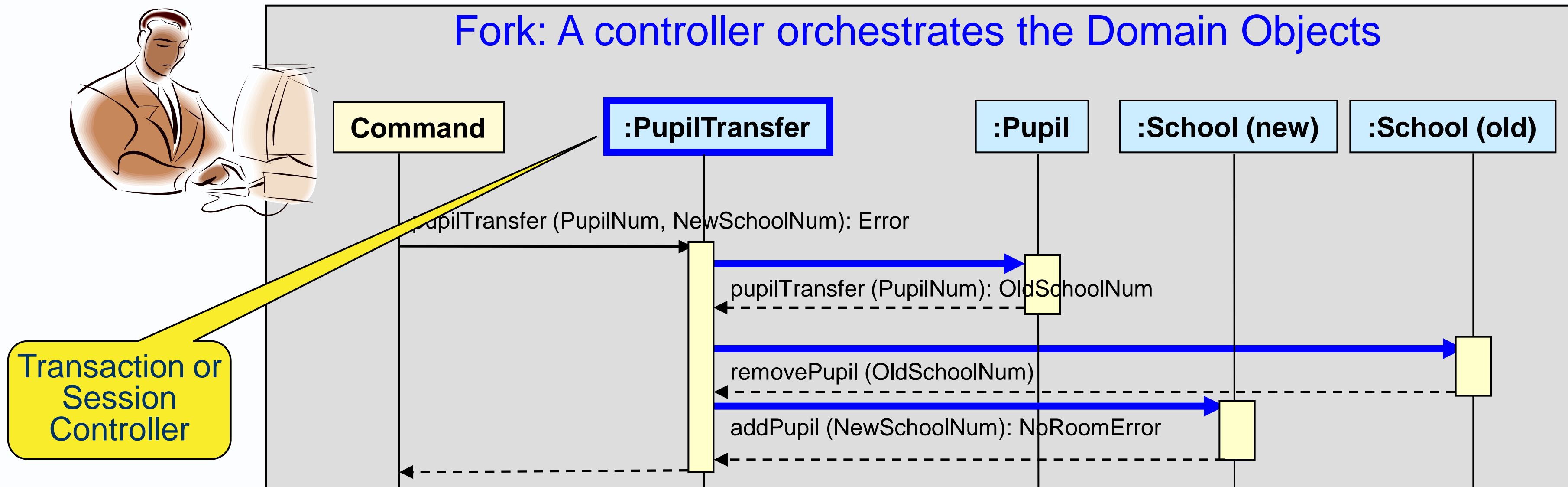


- Fork/Orchestration:
  - centralises intelligence about a process sequence in a workflow controller that supervises and orchestrates the procedure.
  - A controller directs other components to complete the process.
  - It manages the sequence of activities by invoking components in turn.
- Chain/Choreography:
  - distributes intelligence about a process sequence between several entity or domain components.
  - Components cooperate to complete the process.
  - Each component does part of the work, then calls the next component (cf. pipe and filter.)

Copyright Avancier Limited 2007-2016

Fork/Orchestration v Chain/Choreography





Copyright Avancier Limited 2007-2016

Alternative patterns

- Capable architects
  - understand the available patterns
  - look to use them in the right circumstances
  - choose between **alternative** patterns by trading off their pros and cons.
- You should understand
  - the problem you have;
  - the problem the pattern is intended to solve;
  - the benefits, costs and alternatives;
  - the trade-offs between alternative patterns
  - whether the pattern is or should be a local standard.

Copyright Avancier Limited 2007-2016

What can we learn from Software Design Patterns?

- All system design is about
  - designing required processes
  - dividing the system into actors/components.
  - organising the actors/components to perform processes

- At least some Software Design Patterns are relevant to
  - enterprise application integration
  - the design of human activity systems

Copyright Avancier Limited 2007-2016

Beware

- “Patterns are a starting point…
- “Every pattern is incomplete…
- “You have the responsibility of completing it in the context of your own system”
- Martin Fowler

Copyright Avancier Limited 2007-2016

Footnotes

- Classic patterns in enterprise application architecture
- GRASP pattern
- Demeter’s law

Copyright Avancier Limited 2007-2016

Classic patterns in enterprise application architecture

Transaction script

- Centralises intelligence about the process for an event or enquiry
- A simple procedural model – needs only a simple data source layer
- Start with opening a transaction and end with closing it

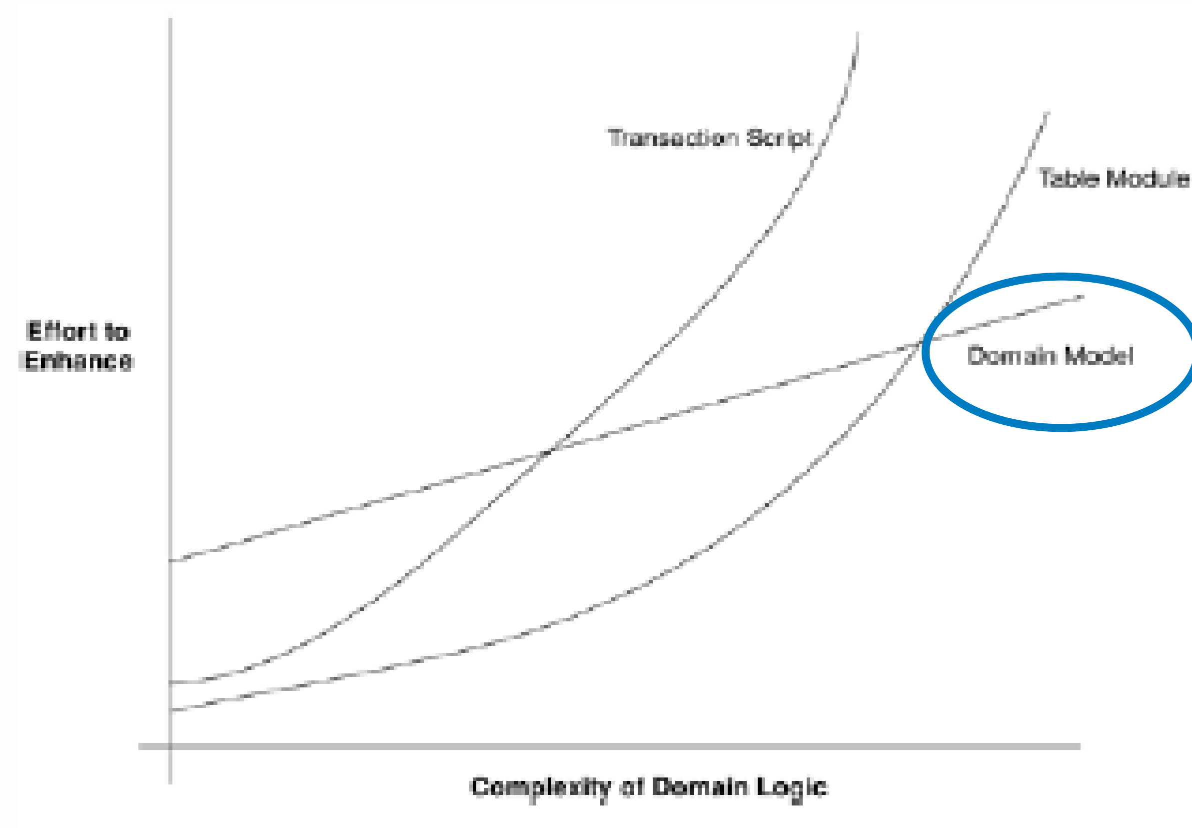
Object-oriented

- Distributes intelligence about a process between entities
- **Table module**
  - One object for each database table (record set)
- **Domain model**
  - One object for each entity instance
  - **Simple domain model:** mostly 1 OO class to 1 database table
  - **Rich domain model:** complex class to table mapping
    - "anecdotal observations put the effort of mapping to a relational database at around a third of programming effort—a cost that continues during maintenance." Fowler

Copyright Avancier Limited 2007-2016

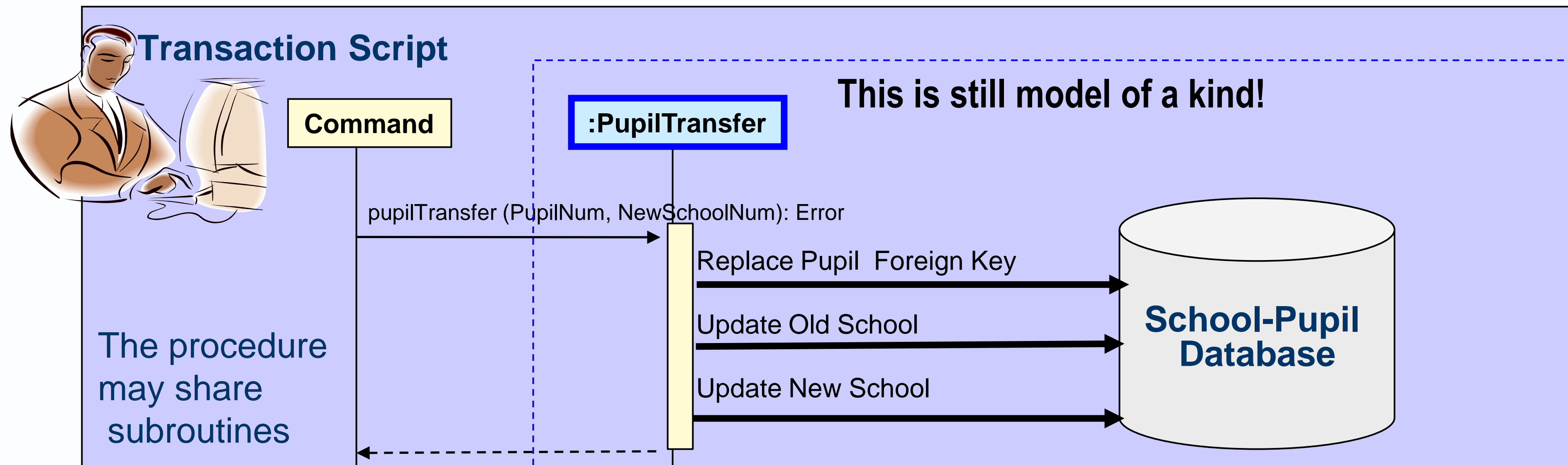
Choosing between patterns

- Fowler drew this graph
- Limiting the use of rich domain models to highly complex situations
- OO book authors like to write about those situations



Transaction script

- "However much of an object bigot you become, don't rule out Transaction Script.
- There are a lot of simple problems, and a simple solution will get you up and running much faster."…
- "Many… scripts act directly on the database, putting SQL into the procedure."
- "The simplest Transaction Scripts contain their own database logic"



Copyright Avancier Limited 2007-2016

By the way

- Fowler drew these diagrams to illustrate how each pattern works for “calculating revenue recoginations” in a case study

Figure 2.1. *A* Transaction Script's *(110) way of calculating revenue recognitions.*

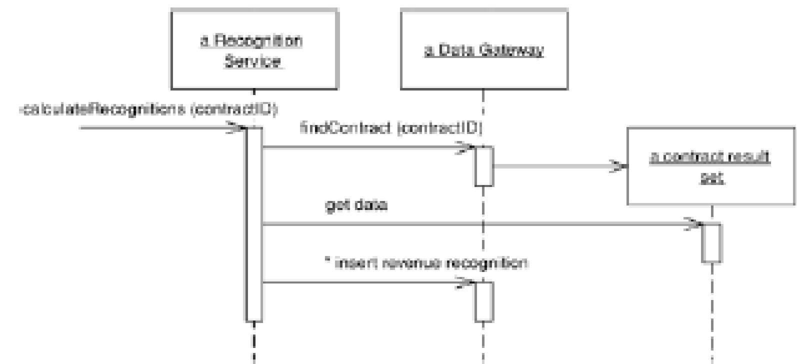


Figure 2.3. *Calculating revenue recognitions with a* Table Module *(125).*

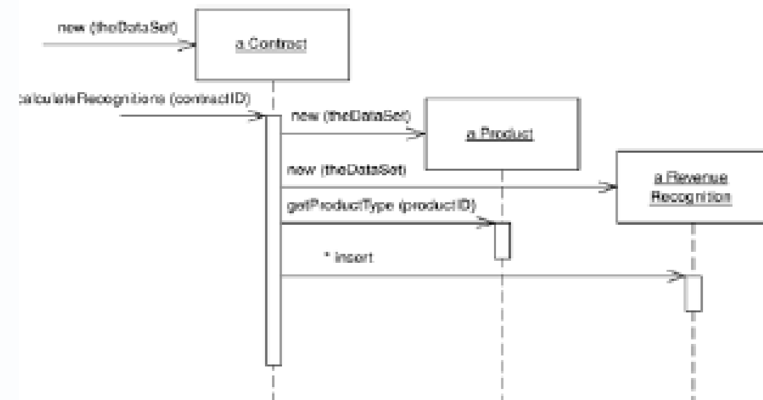
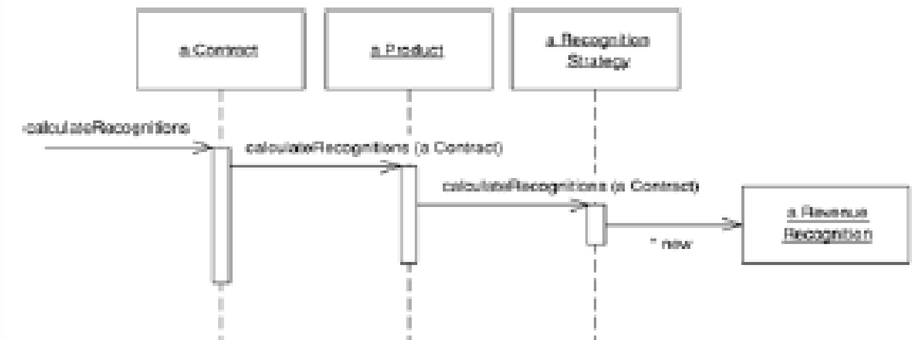


Figure 2.2. *A* Domain Model's *(116) way of calculating revenue recognitions.*



The GRASP pattern

- A good designer can mix alternative patterns
- Use each pattern where it is appropriate.

- The GRASP pattern can be used to design a structure than compromises between
  - Fork/Orchestration
  - Chain/Choreography styles.

Copyright Avancier Limited 2007-2016

The General Responsibility Assignment Software Pattern

- Craig Larman uses it to advance various principles for designing the interactions between components, including these five

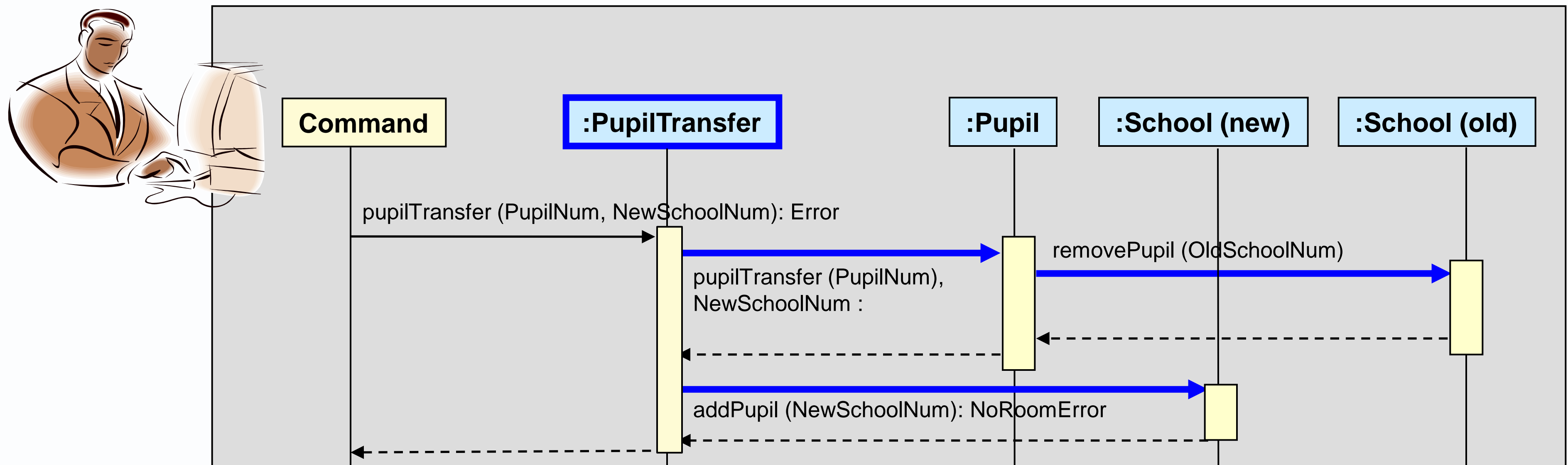
GRASP	Meaning
Expert	Assign a responsibility to the expert component, which has the data to fulfil the responsibility
Creator	Assign a responsibility for creating an object (or entity instance) to the component that collects or holds the object’s initial data
Low-Coupling	Ensure coupling between components remains low.
High-Cohesion	Ensure cohesion within a component remains high.
Controller	Create components to handle events in the end-to-end process

Copyright Avancier Limited 2007-2016

Application of the GRASP design pattern

Avancier

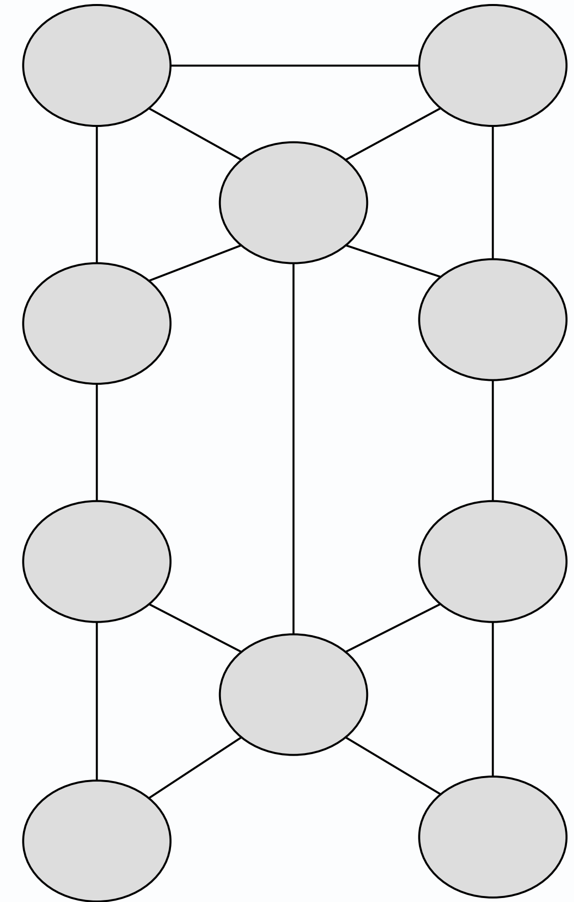
- The system remembers
  - the names of Schools
  - the Pupils currently registered in each School
  - the PupilTotal for each School.
- If the new School's maximum number of pupils if not exceeded, and the Pupil Transfer event completes, then a Pupil will be moved from his/her current School to his/her new School,



Copyright Avancier Limited 2007-2016

Demeter's Law

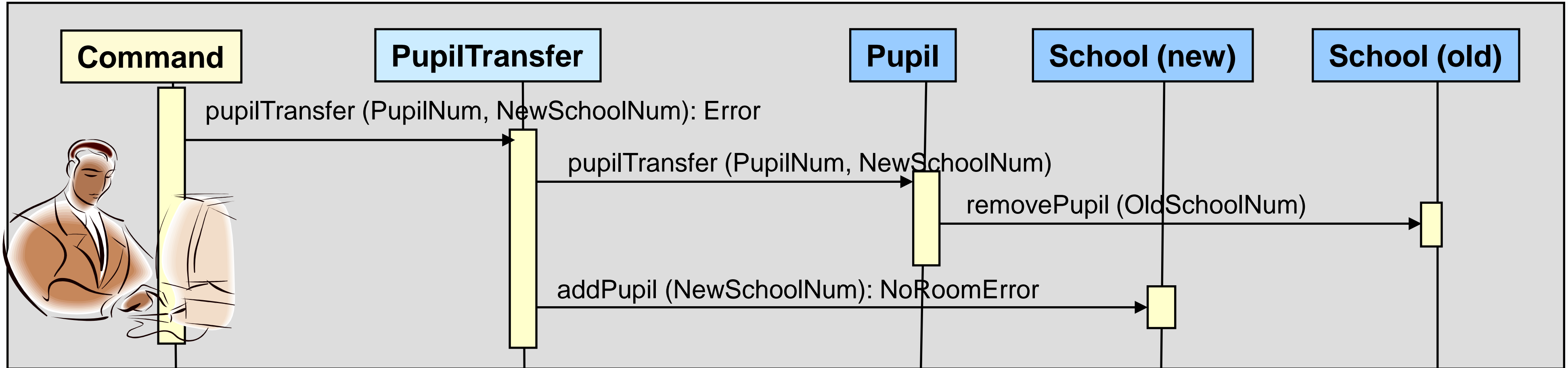
- (1) A component should know only a few (<6?) other closely-related components.
- (2) A component should talk only to its immediate relatives; not 'reach through' them to talk to components the relatives know.
- Paradox
- To enforce (1), the designer may have to add intermediate components (containers, controllers or brokers or **facades**).
- This tends to contravene (2)



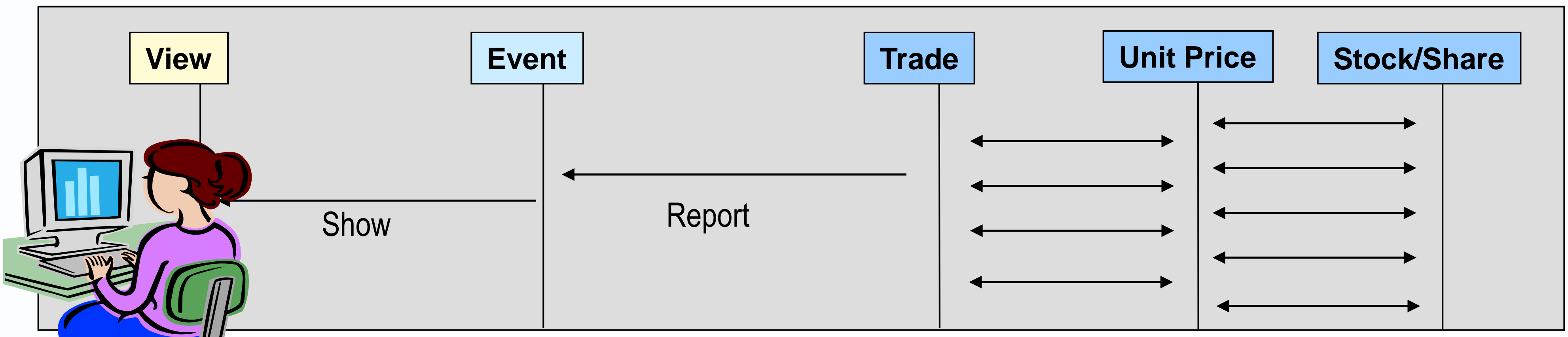
Copyright Avancier Limited 2007-2016

Input-driven v Model-driven

- Inputs feed state change events to the data servers



- A data server publishes state change events to UI views



Copyright Avancier Limited 2007-2016