AARHUS UNIVERSITET

Software Architecture in Practice

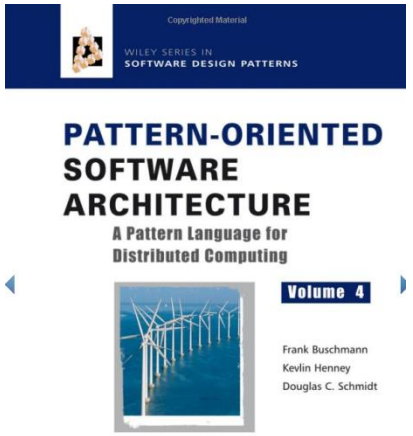
Optional Material:

POSA 4 Glimpse

AARHUS UNIVERSITET

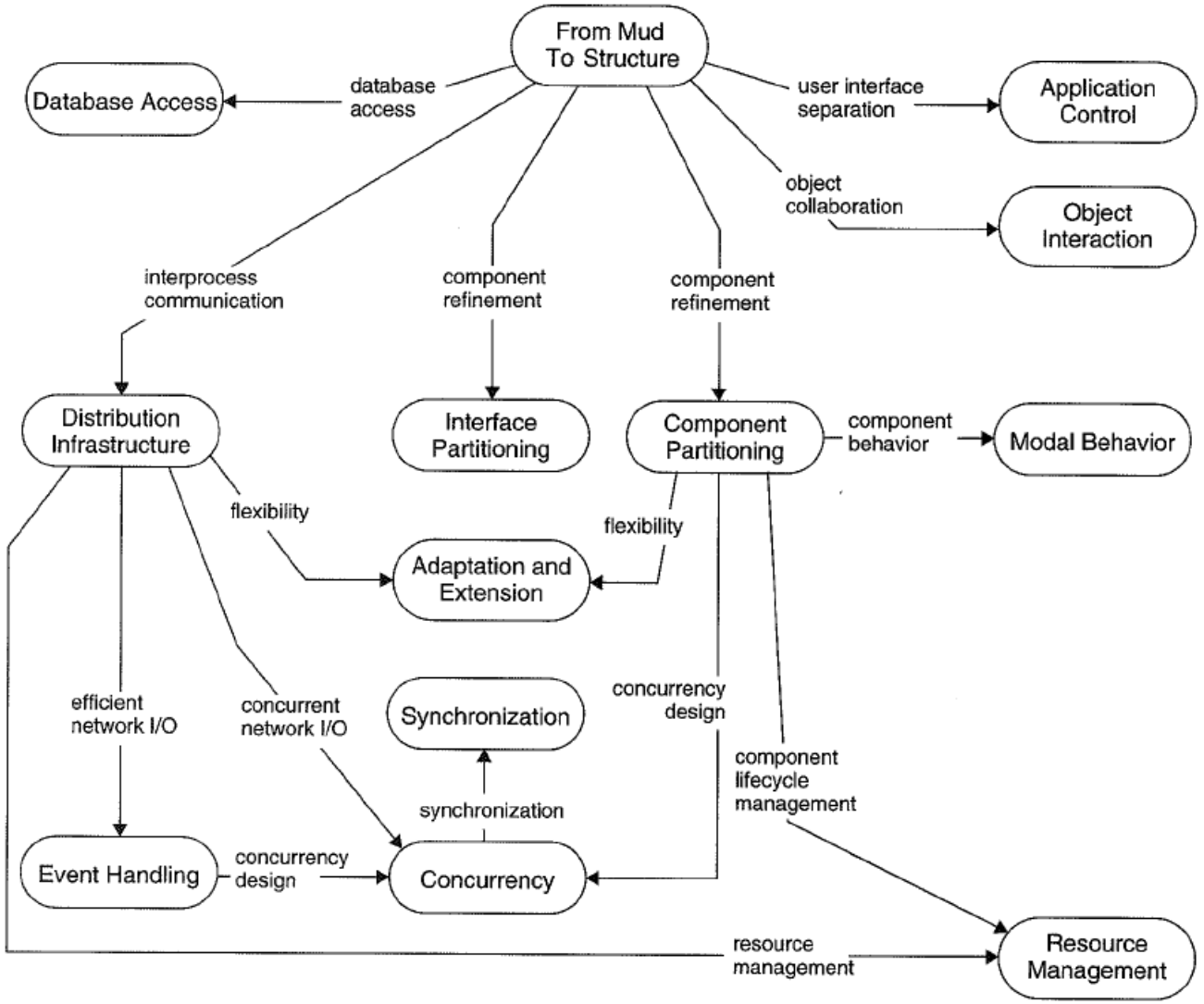
Digging in *one book*

- Observations
  - Focus on *information systems (IS)*
  - Balance between overview and detail
- POSA volume 4
  - *A Pattern Language for Distributed Computing*
  - Outline 114 patterns for IS architectures
    - Further references 150 patterns from other sources
  - Provides roadmap and overview
  - Provides enough detail to grasp idea
    - Implementation details in other volumes / sources



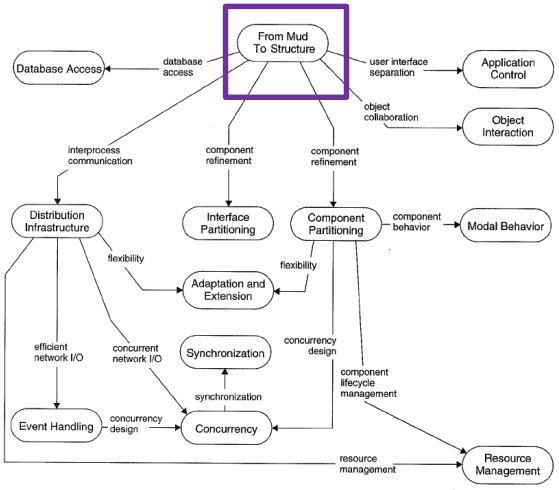
AARHUS UNIVERSITET

Posa 4 overview



AARHUS UNIVERSITET

From Mud to Structure



- *From Mud To Structure*: DOMAIN MODEL (182), LAYERS (185), MODEL-VIEW-CONTROLLER (188), PRESENTATION-ABSTRACTION-CONTROL (191), MICROKERNEL (194), REFLECTION (197), PIPES AND FILTERS (200), SHARED REPOSITORY (202), BLACKBOARD (205), and DOMAIN OBJECT (208).

- Several of these are ’styles’
  - *Shared repository* is a well known classic for IS
  - *Layers* and *MVC* are as well...

AARHUS UNIVERSITET

- Note:
  - *Functionality!*
- Roles and responsibilities

Domain Object **

When realizing a Domain Model (182), or its technical architecture in terms of Layers (185), Model-View-Controller (188), Presentation-Abstraction-Control (191), Microkernel (194), Reflection (197), Pipes and Filters (200), Shared Repository (202), or Blackboard (205) ...

... a key concern of all design work is to decouple self-contained and coherent application responsibilities from one another.

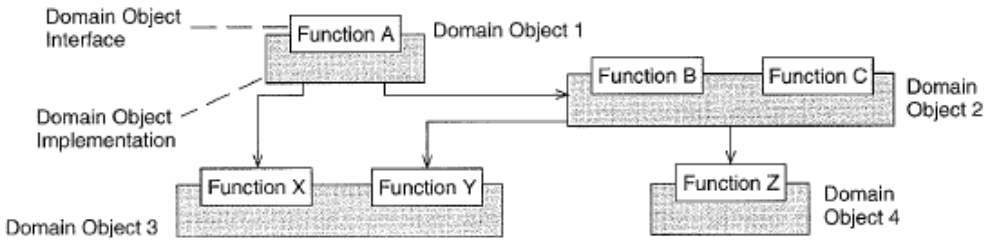
♦♦♦

The parts that make up a software system often expose manifold collaboration and containment relationships to one another. However, implementing such interrelated functionality without care can result in a design with a high structural complexity.

Separation of concerns is a key property of well-designed software. The more decoupled are the different parts of a software system, the better they can be developed and evolved independently. The fewer relationships the parts have to one another, the smaller the structural complexity of the software architecture. The looser the parts are coupled, the better they can be deployed in a computer network or composed into larger applications. In other words, a proper partitioning of a software system avoids architectural fragmentation, and developers can better maintain, evolve and reason about it. Yet despite the need for clear separation of concerns, the implementation of and collaboration between different parts in a software system must be effective and efficient for key operational qualities, such as performance, error handling, and security.

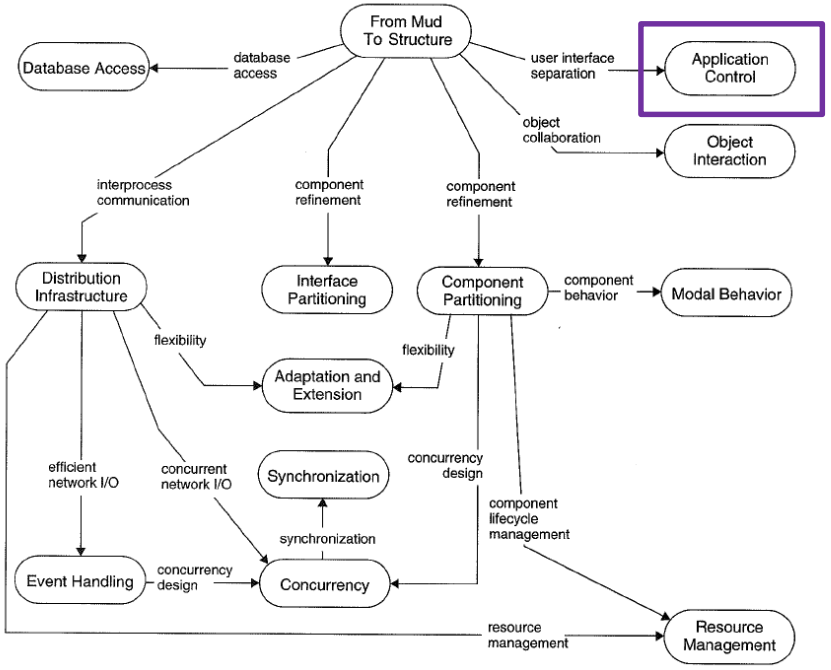
Therefore:

Encapsulate each distinct functionality of an application in a self-contained building-block—a domain object.



AARHUS UNIVERSITET

Application control



- *Application Control:* Page Controller (337), Front Controller (339), Application Controller (341), Command Processor (343), Template View (345), Transform View (347), Firewall Proxy (349), and Authorization (351).

AARHUS UNIVERSITET

- Idea:
  - Separate GUI from workflow control
- Reusable across GUIs

Application Controller **

When developing a MODEL-VIEW-CONTROLLER (188) architecture where the view is remote from the model . . .

. . . we must often provide an access point for handling user interface navigation and the workflow of an application.

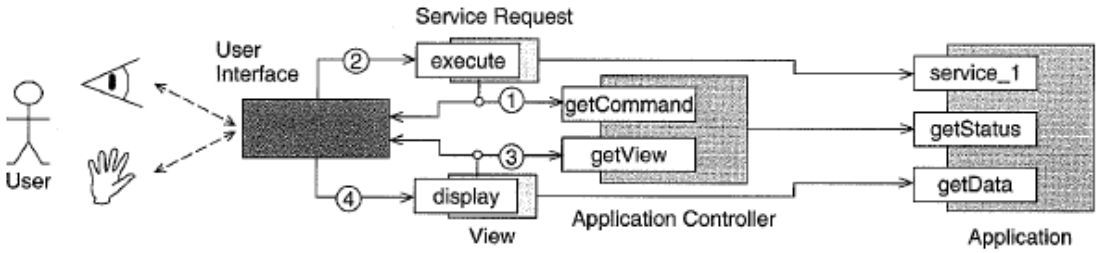
♦♦♦

Some applications lead their users through a series of screens or forms following a specific workflow, or present specific screens or forms only under certain conditions. Placing such logic in the application's controllers, however, mixes user-interface code with application-specific workflow logic.

Moreover, different controllers could instigate the same workflow, which would lead to duplicated logic that is hard to maintain and evolve. Another approach is to implement the logic of the screen or form to display next in response to a specific action directly within the application logic. This approach is not practical, however, since application components, which are generally independent of presentation aspects, would become dependent on the partitioning and screen ordering of a specific user interface.

Therefore:

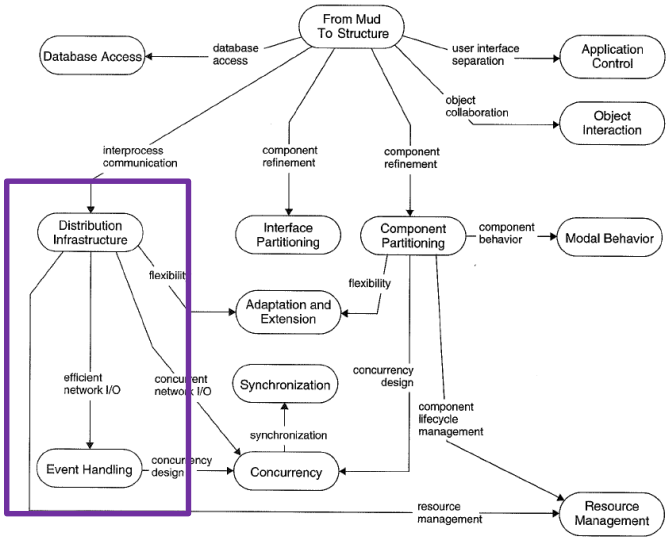
Encapsulate the application's workflow within a separate application controller. User-interface controllers use the application controller to determine the appropriate actions to invoke on application logic, as well as the correct view to display after the action has been executed.



The application controller acts as a central access point for user-interface elements, unifying access to the functionality of an

AARHUS UNIVERSITET

Distribution infrastructure



- *Distribution Infrastructure*: MESSAGING (221), MESSAGE CHANNEL (224), MESSAGE ENDPOINT (227), MESSAGE TRANSLATOR (229), MESSAGE ROUTER (231), BROKER (237), CLIENT PROXY (240), REQUESTOR (242), INVOKER (244), CLIENT REQUEST HANDLER (246), SERVER REQUEST HANDLER (249), and PUBLISHER-SUBSCRIBER (234).
- *Event Demultiplexing and Dispatching*: REACTOR (259), PROACTOR (262), ACCEPTOR-CONNECTOR (265), and ASYNCHRONOUS COMPLETION TOKEN (268).

AARHUS UNIVERSITET

Client Request Handler **

When developing a REQUESTOR (242) ...

... we must send requests to, and receive replies from, the network.

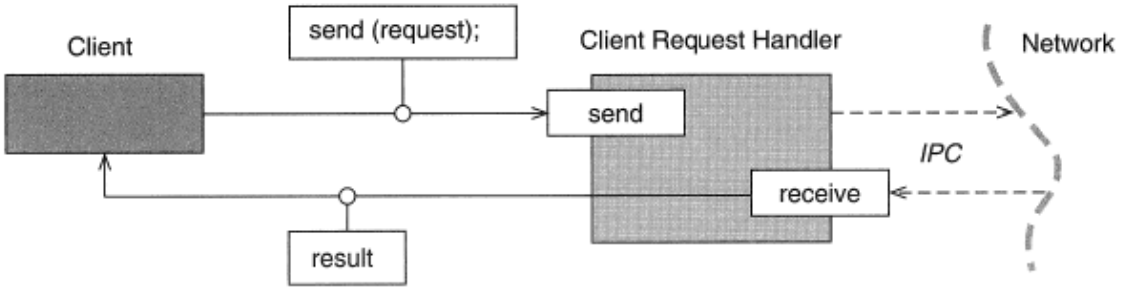
♦♦♦

Sending client requests to and receiving replies from the network involves various low-level IPC tasks, such as connection management, time-out handling, and error detection. Writing and performing these tasks separately for each client uses networking and endsystem resources ineffectively.

The more clients access the network, and the more requests and replies must be handled simultaneously, the more efficiently network resources must be managed to achieve appropriate quality of service in a distributed application. Network connections and bandwidth, for example, are limited resources and must be shared and used judiciously by all clients to ensure acceptable latency and jitter. In addition, writing error detection and time-out handling tasks separately for each client duplicates code and pollutes the application with non-portable networking code.

Therefore:

Provide a specialized client request handler that encapsulates and performs all IPC tasks on behalf of client components that send requests to and receive replies from the network.



AARHUS UNIVERSITET

From Net4Care

```
private TeleObservation teleObs1;

@Before public void setup() {
  teleObs1 = HelperMethods.createObservation120over70forNancy();

@Test public void shouldUploadTeleObservation() throws IOException {
  // Validate that we can upload the observation
  Result result = forwarder.send(teleObs1);
  assertNotNull(result);
  assertTrue( result.isSuccess() );
```

- Benefits
  - Simplify the domain code (just use ’forwarder’)
  - Easy to reconfigure forwarder for testing!
- (Liability
  - Named ’forwarder’ in POSA 1)

AARHUS UNIVERSITET

Reactor **

When developing event-driven software, or designing a CLIENT REQUEST HANDLER (246) or a SERVER REQUEST HANDLER (249) ...

... we must decouple infrastructure behavior associated with detecting, demultiplexing, and dispatching events from short-duration components that service the events.

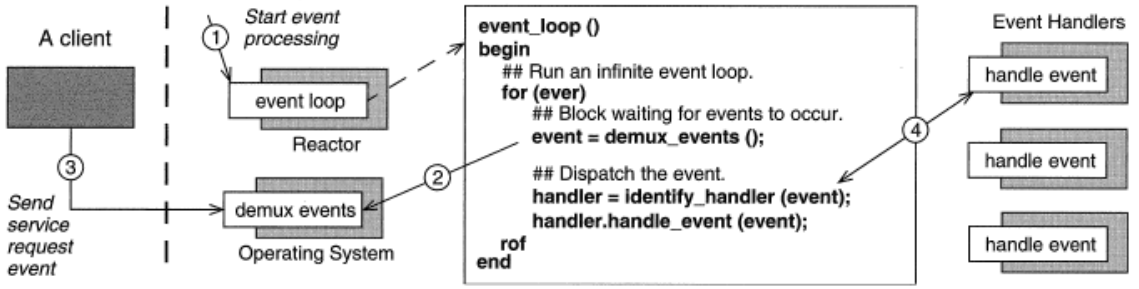
♦♦♦

Event-driven software often receives service request events from multiple event sources, which it demultiplexes and dispatches to event handlers that perform further service processing. Events can also arrive simultaneously at the event-driven application. However, to simplify software development, events should be processed sequentially and synchronously.

Efficiently and flexibly processing events that arrive concurrently from multiple sources is hard. For example, using multi-threading to wait for events to occur in a set of event sources can introduce overheads due to synchronization, context switching, and data movement. In contrast, blocking indefinitely on a single event source can prevent the servicing of other event sources, degrading the quality of service to clients. In addition, it should be easy to integrate new or improved event handlers into the event-handling infrastructure.

Therefore:

Provide an event handling infrastructure that waits on multiple event sources simultaneously for service request events to occur, but only demultiplexes and dispatches one event at a time to a corresponding event handler that performs the service.



AARHUS UNIVERSITET

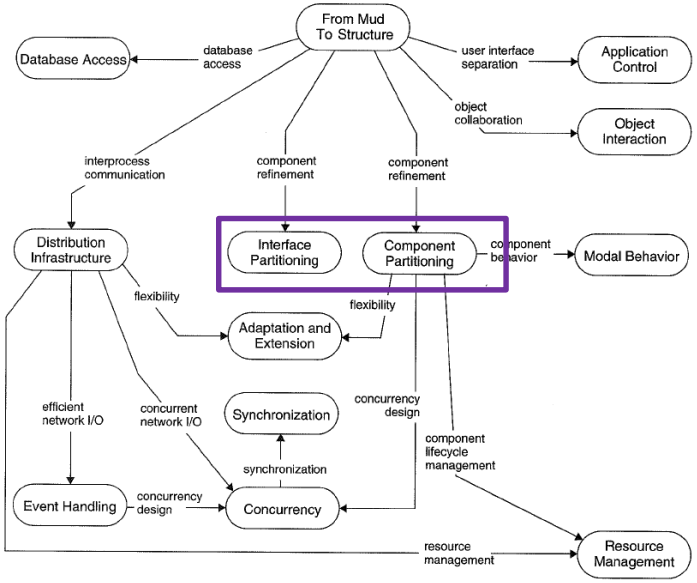
Discussion

- Classic server dispatching
  - Single threaded
  - Multi threaded (thread pr request)
  - Thread pool
- Supposed to enhance performance over these
  - Java NIO library
- But *measure it!*
  - *OS dependent*
  - *JVM version dependent*

```
public void run() {
    try {
        // Create & start the ThreadPool
        _pool = new ThreadPool(_poolSize);
        _pool.startPool();

        // Create a non-blocking server socket channel and bind to to the Reactor port
        ServerSocketChannel ssChannel = ServerSocketChannel.open();
        ssChannel.configureBlocking(false);
        ssChannel.socket().bind(new InetSocketAddress(_port));

        // Create the selector and bind the server socket to it
        Selector selector = Selector.open();
        ssChannel.register(selector, SelectionKey.OP_ACCEPT, new ConnectionAcceptor(selector, ssChannel, _pool));

        while (_shouldRun) {
            // Wait for an event
            selector.select();

            // Get list of selection keys with pending events
            Iterator it = selector.selectedKeys().iterator();

            // Process each key
            while (it.hasNext()) {
                // Get the selection key
                SelectionKey selKey = (SelectionKey)it.next();

                // Remove it from the list to indicate that it is being processed
                it.remove();

                // Check if it's a connection request
                if (selKey.isValid() && selKey.isAcceptable()) {
                    ConnectionAcceptor connectionAcceptor = (ConnectionAcceptor)selKey.attachment();
                    connectionAcceptor.accept();
                }
                // Check if a message has been sent
                if (selKey.isValid() && selKey.isReadable()) {
                    ConnectionReader connectionReader = (ConnectionReader)selKey.attachment();
                    connectionReader.read();
                }
            }
        }
    } catch (IOException e) {
        e.printStackTrace(System.err);
        stopReactor();
    }
    stopReactor();
}
```

AARHUS UNIVERSITET

Component refinement



Interface Partitioning: EXPLICIT INTERFACE (281), EXTENSION INTERFACE (284), INTROSPECTIVE INTERFACE (286), DYNAMIC INVOCATION INTERFACE (288), PROXY (290), BUSINESS DELEGATE (292), FACADE (294), COMBINED METHOD (296), ITERATOR (298), ENUMERATION METHOD (300), and BATCH METHOD (302).

- *Component Partitioning*: ENCAPSULATED IMPLEMENTATION (313), WHOLE-PART (317), COMPOSITE (319), MASTER-SLAVE (321), HALF-OBJECT PLUS PROTOCOL (324), and REPLICATED COMPONENT GROUP (326).

AARHUS UNIVERSITET

Explicit Interface

- One of the key principles from ”Gang of Four”
  - ***Program to an interface***

AARHUS UNIVERSITET

- Well supported in Java & C#
- Way to make *specialized access interface* for testing!

Extension Interface **

When specifying an EXPLICIT INTERFACE (281) . . .

. . . we may want to ensure client stability and type-safety in the face of interface evolution.

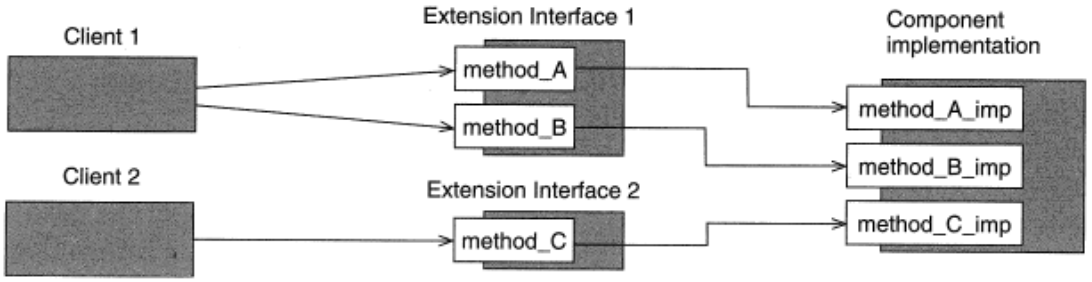
♦♦♦

Clients can use a component effectively only if it provides a stable and coherent interface. The interface of a component is often affected, however, when its functionality is modified or extended, which can break the client code—in some cases even if the new functionality is not used.

Ideally, clients of a component should not break when parts of the component's interface that they do not use change, or if they are not interested in new services added to the component. Even when interface parts that they actually do use change, clients should not break if they do not use the changes. Similarly, the existing interface of a component should remain stable when its implementation is extended with new services, or when existing service signatures are updated.

Therefore:

Let clients access a component only via specialized extension interfaces, and introduce one such interface for each role that the component provides. Introduce new extension interfaces whenever the component evolves to include new functionality or updated signatures within existing extension interfaces.



AARHUS UNIVERSITET

- Handle Iterator across network

Batch Method **

Within an EXPLICIT INTERFACE (281), ITERATOR (298), or OBJECT MANAGER (492) . . .

. . . we may need to perform bulk accesses on an aggregate component.

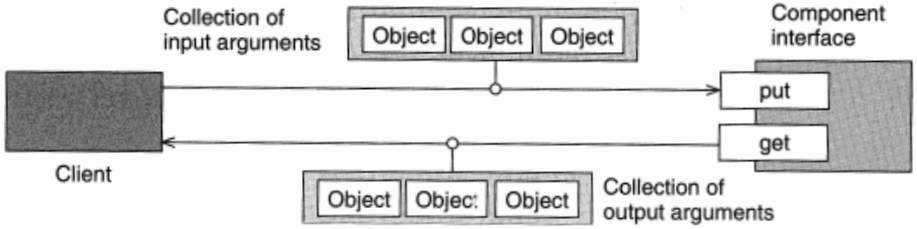
♦♦♦

Clients sometimes perform bulk accesses on an aggregate component, for example to retrieve all elements in a collection that meet certain properties. If access to the aggregate is expensive, for example because it is remote or concurrent, accessing it separately for each element can incur significant performance penalties and concurrency overhead.

If the aggregate is remote, each access incurs latency and jitter, decreases the available network bandwidth, and introduces additional points of failure. If the aggregate is a concurrent component, synchronization and thread management overhead must be added to the cost of each access. Similarly, any other per-call housekeeping code, such as for authorization, further decreases performance. Nevertheless, it must be possible to perform bulk accesses to an aggregate efficiently and without interruption.

Therefore:

Define a single batch method that performs the action on the aggregate repeatedly. The method is declared to take all the arguments for each execution of the action, for example via an array or a collection, and to return results by similar means.



AARHUS UNIVERSITET

- Good for objects that have a clear
- 'client' and 'server' side...
- WoW Hero?

Half-Object plus Protocol **

When realizing an ENCAPSULATED IMPLEMENTATION (313) or a LOOKUP (495) service ...

... at times we need to ensure reduced response time when accessing a single object from multiple address spaces.

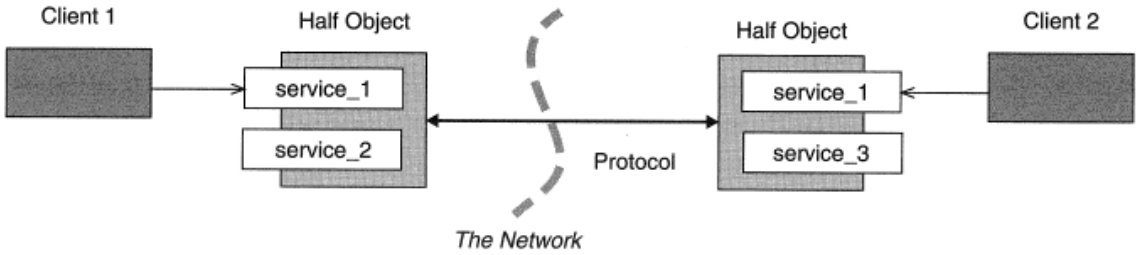
♦♦♦

Distributed systems often use designs in which clients access objects of components that reside in other address spaces. However, due to the latency and jitter incurred when exchanging requests and responses across the network, this design can be impractical when requirements demand rapid response.

Resolving the problem via replication is not always feasible. For example, a component may need access to information from multiple address spaces to carry out its behavior. Obtaining this information from each replicated component would result in significant network load and traffic, which can decrease or even eliminate the performance advantages of the component's replication. The same effect occurs for replicated components that represent a single stateful entity within the distributed system. If the state of any of the replicas is modified, the state of the other replicas must be updated accordingly across the network.

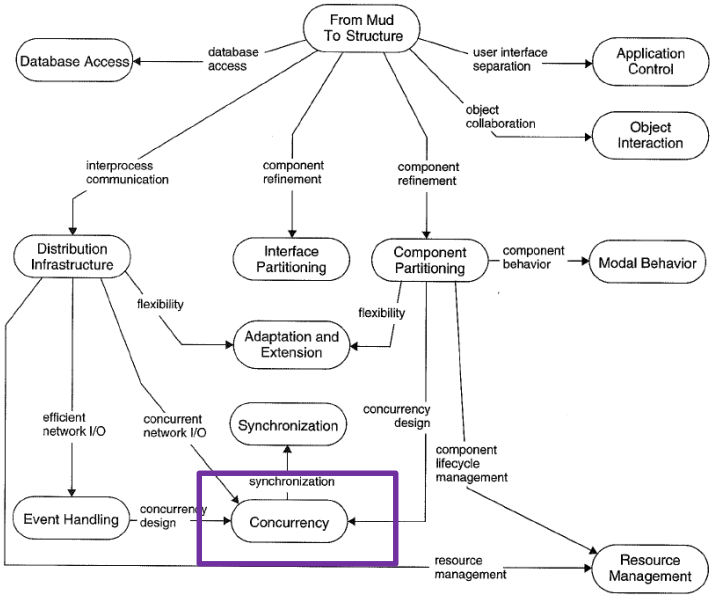
Therefore:

Divide the objects into multiple 'half objects,' one for each address space in which they is used. Each half object implements the functionality and data required by the clients that reside in 'its' address space. A protocol between the half objects helps to coordinate their activities and keep their state consistent.



AARHUS UNIVERSITET

Concurrency



- *Concurrency*: HALF-SYNC/HALF-ASYNC (359), LEADER/FOLLOWERS (362), ACTIVE OBJECT (365), MONITOR OBJECT (368).
- *Synchronization*: GUARDED SUSPENSION (380), FUTURE (382), THREAD-SAFE INTERFACE (384), DOUBLE-CHECKED LOCKING (386), STRATEGIZED LOCKING (388), SCOPED LOCKING (390), THREAD-SPECIFIC STORAGE (392), COPIED VALUE (394), and IMMUTABLE VALUE (396).

AARHUS UNIVERSITET

- Block, not on request but on reading the result...

Future **

When implementing a MASTER-SLAVE (321) arrangement, an ACTIVE OBJECT (365), PARTIAL ACQUISITION (511), or software in which client and server run concurrently and communicate via method calls ...

... at times we must access data that is computed concurrently with the control flow of a client.

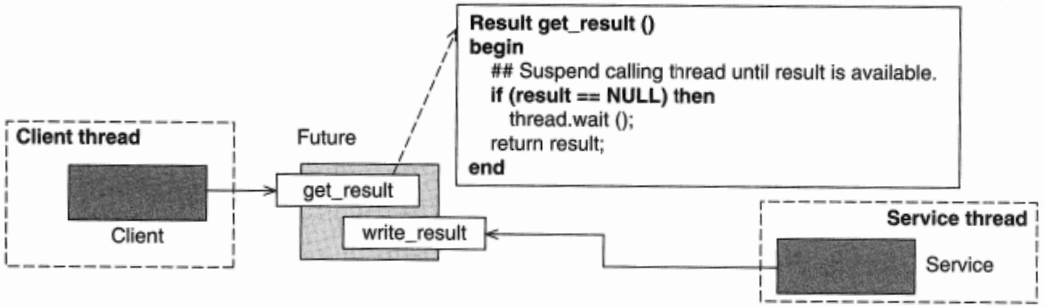
♦♦♦

Services that are invoked concurrently on a component may need to return a result to the calling client. However, if the client does not block after calling the service, continuing instead with its own computation, the service's result may not be available when the client needs to use it.

A common use of concurrency is to optimize performance by overlapping computation and communication. This optimization can be simple for one-way calls that return no results: invocation can be as simple as 'fire and forget.' A client, however, may want to invoke one or more two-way methods on one or more servers without having to wait synchronously for the server response(s). A call–return procedural model is simple to use but cannot, in this case, be used to return a result. Nevertheless, when the client needs a result to continue its processing, there must be a straightforward way to obtain it.

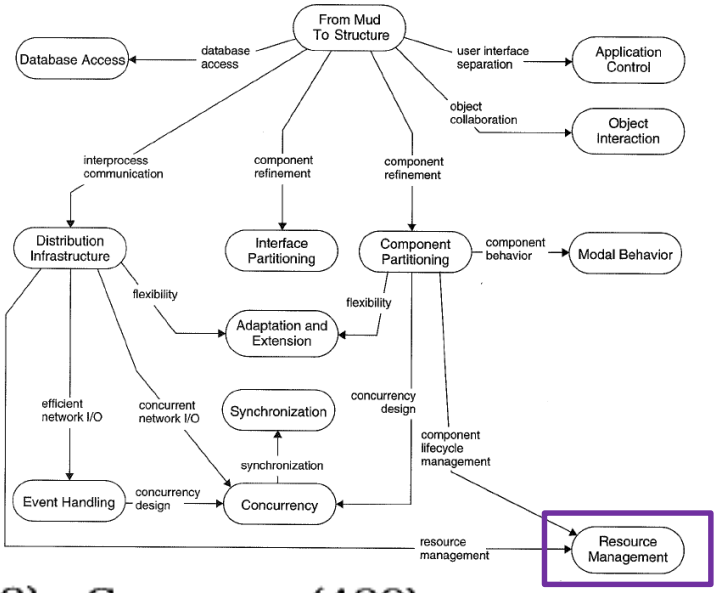
Therefore:

Immediately return a 'virtual' data object—called a future—to the client when it invokes a service. This future keeps track of the state of the service's concurrent computation and only provides a value to clients when the computation is complete.



AARHUS UNIVERSITET

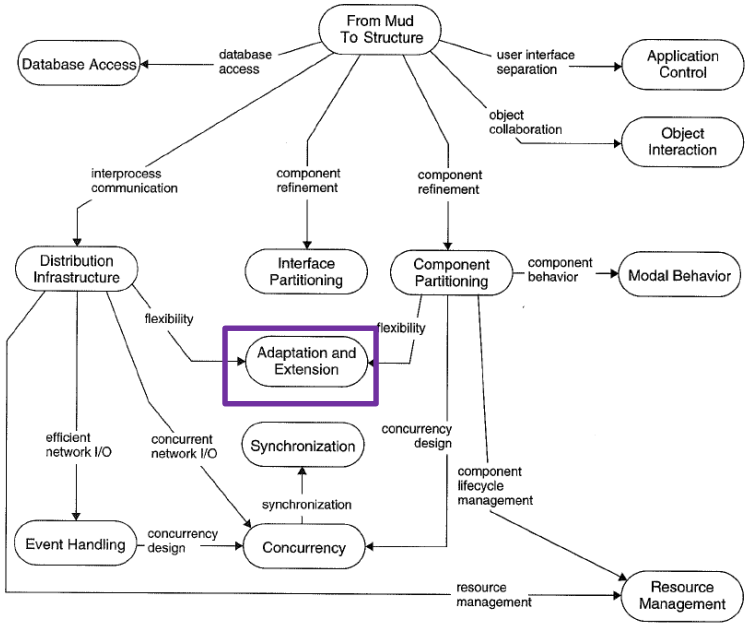
Resource management



- *Resource Management:* Object Manager (492), Container (488), Component Configurator (490), Lookup (495), Virtual Proxy (497), Lifecycle Callback (499), Task Coordinator (501), Resource Pool (503), Resource Cache (505), Lazy Acquisition (507), Eager Acquisition (509), Partial Acquisition (511), Activator (513), Evictor (515), Leasing (517), Automated Garbage Collection (519), Counting Handle (522), Abstract Factory (525), Builder (527), Factory Method (529), and Disposal Method (531).

AARHUS UNIVERSITET

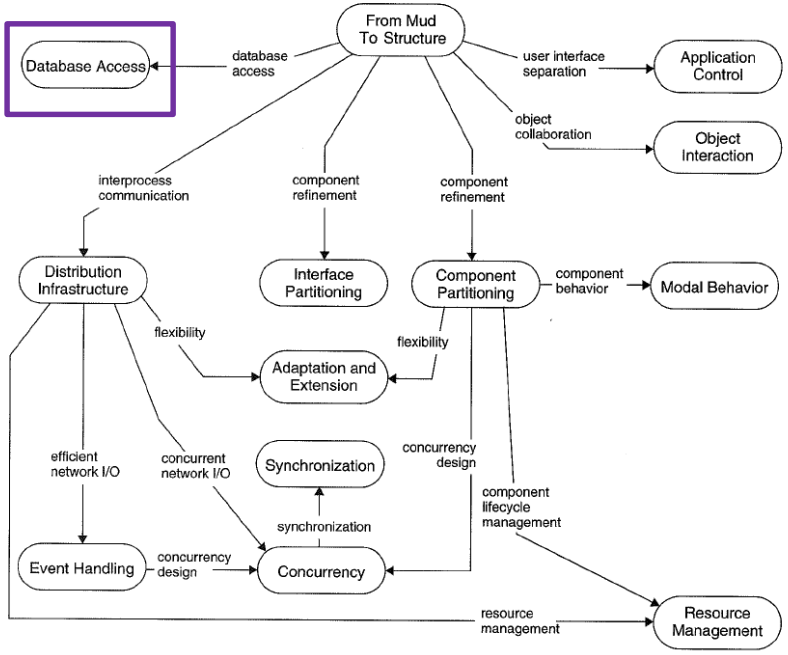
Adaption and extension



- *Adaptation and Extension:* BRIDGE (436), OBJECT ADAPTER (438), INTERCEPTOR (444), CHAIN OF RESPONSIBILITY (440), INTERPRETER (442), VISITOR (447), DECORATOR (449), TEMPLATE METHOD (453), STRATEGY (455), NULL OBJECT (457), WRAPPER FACADE (459), EXECUTE-AROUND OBJECT (451), and DECLARATIVE COMPONENT IMPLEMENTATION (461).

AARHUS UNIVERSITET

Database access



- *Database Access*: DATABASE ACCESS LAYER (538), DATA MAPPER (540), ROW DATA GATEWAY (542), TABLE DATA GATEWAY (544), and ACTIVE RECORD (546).

AARHUS UNIVERSITET

Summary

- POSA 4
  - Information Systems
  - All the way from ’front’ to ’back’ / client to storage
  - Draws upon many resources
  - Overview rather than detail