Architektur von Anwendungssystemen – Zusammenfassung

Sommersemester 2019

Lukas Heiland feat. Yannis Blosch – last updated:

7. Mai 2019

Inhaltsverzeichnis

1 Definitions/Notions

1.1 Architecture

The architecture of an IT system is the structure or structures of the system which comprise software and hardware components, the externally visible properties of those components, and the relationships among them.

- Architecture isn't simply *good* or *bad*
- Architectue is fit or unfit *for a purpose*

1.1.1 Goals of an Architecture

- Producing a framework to support the development of software
- Creating an integration platform for future enhancements
- Producing the interface definitions for collaboration of components

1.1.2 Importance of Architecture

If the size and complexity of a software system increase, the global structure of the system becomes more important than the selection of specific algorithms and data structures.

1.1.3 Architecture vs. Design

An architecture provides a framework and a 'set of rules' for the act of designing a particular thing. So there can be many individually designed instances of each particular architectural style.

1.1.4 Architecture Levels

Conceptual Architecture

- direct attention at an appropriate decomposition of the system without delving into details
- provides a useful vehicle for communicating the architecture to non-technical audiences, such as management, marketing, and users
- consists of the Architecture Diagram (without interfaces) and an informal component specification for each component (see: ADL)

Logical Architecture

- adds precision, providing a detailed "blueprint"from which component developers and component users can work in relative independence
- incorporates the detailed Architecture Diagram (with interfaces)

Execution/Physical Architecture

- Shows the mapping of components onto the threads, processes, (virtual) machines, ...of the physical system
- created for distributed or concurrent systems

1.2 Architect

Some interesting definitions/quotations/etc. from various slides...

- The IT Architect defines (i.e. architects) solutions to client business problems through the reasoned application of information technology.
- The task of an architect is reduction of complexity to orders of magnitude that can be realistically handled.
- The definition of Vitruvius ($\approx 25B.C.$) adds, that an architect (of any kind) should have a lot of general knowledge.

The architect is the advocate of the client.

1.2.1 Where do Architects get ideas from?

Studies of work of other architects is key!
Importance of reference architectures, patterns, styles

1.2.2 Architectural thinking

Architectural thinking is based on basic architectural principles:

- Separation of concerns
- Information Hiding
- Design by interface
- Separation of interface and implementation
- Partitioning/distributing responsibilities

Architectural thinking involves

- Looking at the solution from the direction of requirements, not technology
- Understanding all aspects of the requirements (functional and non-functional)
- Understandign all aspects of the solution (functional and non-functional)
- Using reference architectures and patterns whenever appropriate
- Compromising and balancing; every solution to a requirement will cause other problems

1.3 System

Composition of parts into a new whole which represents via the collaboration of the parts more than the sum of its parts.

1.3.1 Emergence

This is a central aspect of Systems: Emergence is the appearance of properties of a system which none of its constituents has; i.e. a shelf: it is comprised of wooden planks and screws, and after you finished building it, you can put stuff on it. This is emergence: the planks and screws themselves did not offer the possibility to store things, it emerged from the system that is called *a shelf*.

1.4 Views

- Views = different models of a single system. Can be built by abstraction
- Architecture consists of multiple different model descriptions of a single building. Different model descriptions target different participants (stakeholders) of the project:

$$\text{Ground plan} \mapsto \text{Decorator}$$
$$\text{Wiring} \mapsto \text{Electrician}$$
$$\text{Plumbing} \mapsto \text{Plumber}$$
$$\ldots \mapsto \ldots$$

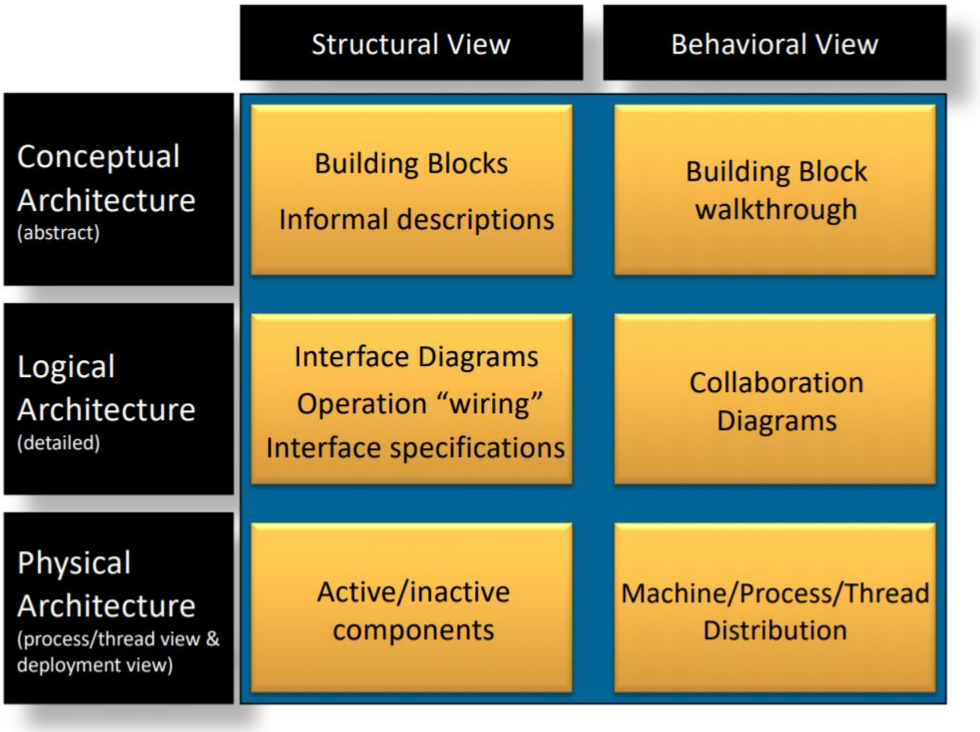
1.4.1 Structural and Behavioral Views

These are used to enhance understandability of the architecture's levels (see 1.1.4).

Structural Views consist of the Architecture Diagram, and Component and Interface Specifications

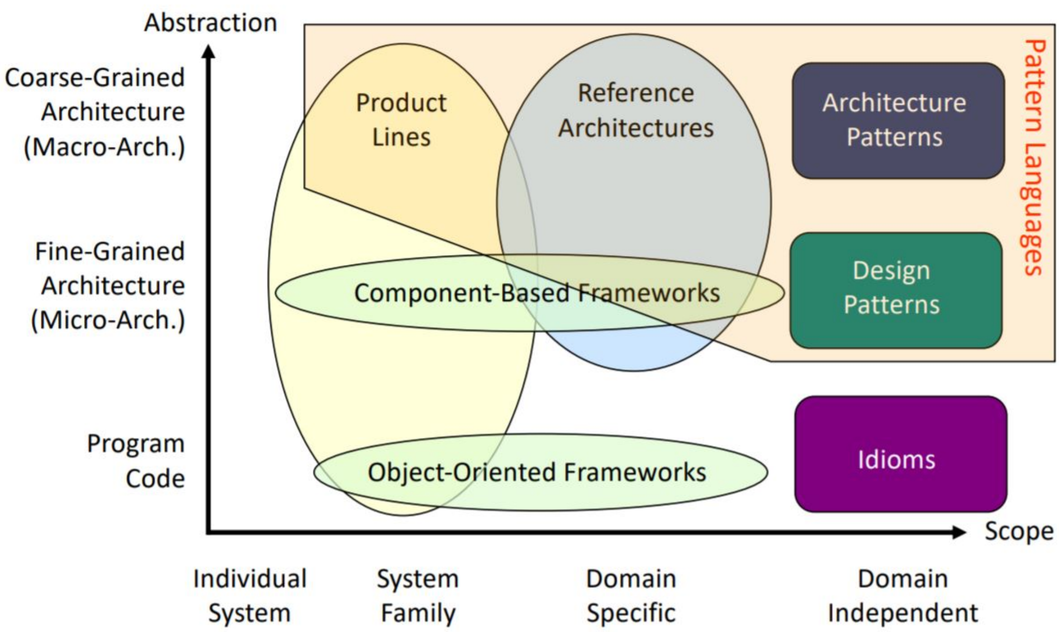
Behavioral Views Contain Component Collaboration or Sequence Diagrams; they answer the question '*How does this work?*'

1.5 Architectural Levels and Views Together



1.6 Reuse

Classification



2 Diagrams and Styles

2.1 Basic elements of an architecture

2.1.1 Components

Components are the result of decomposition of a system.

2.1.2 Connectors

Connectors connect components.

2.1.3 Constraints

Components must be constrained to provide that

- the required functionality is achieved
- no functionality is duplicated
- the required performance is achieved
- the requirements are met
- modularity is realized (e.g. which modules interact with the operating system)

2.1.4 Rationales

3 Model Driven Architecture (MDA)

3.1 Origins

- There are so many (not necessarily interoperable) technologies
- These evolve and get obsolete very quickly

$\Longrightarrow$ desire to have ones business logic (processes, rules, ...) to be as independent as possible from any one technology (*future-proof* business logic)

3.2 Terminology

3.2.1 Architecture

specification of the parts and connectors of the system and the rules for the interactions of the parts using the connectors.

3.2.2 Platform

Set of subsystems/technologies that provide a coherent set of functionality through interfaces and specified usage patterns. Any subsystem that depends on the platform can use it without concern for the details of how the functionality provided by the platform is implemented.

3.2.3 Implementation

A specification which provides all the information needed to construct a system and to put it into operation.

3.3 MDA Models

3.3.1 Computation independent model (CIM)

- a.k.a. *domain model* or *business model*
- focuses on the system and its environment; details of the structure of the system are hidden or undetermined
- specified using a vocabulary that is familiar to the practitioners of the domain in question
- may hide information about the use of automated data processing systems

3.3.2 Platform Independent Model (PIM)

Exhibits platform independence and is suitable for use with a number of different platforms of similar type.

3.3.3 Platform Specific Model (PSM)

Combines the specifications in the PIM with the details that specify how that system uses a particular type of platform.

3.3.4 Platform Model (PM)

Provides a set of technical concepts, representing the different kinds of parts that make up a platform and the services provided.
Influences the way a PIM is mapped to a PSM.

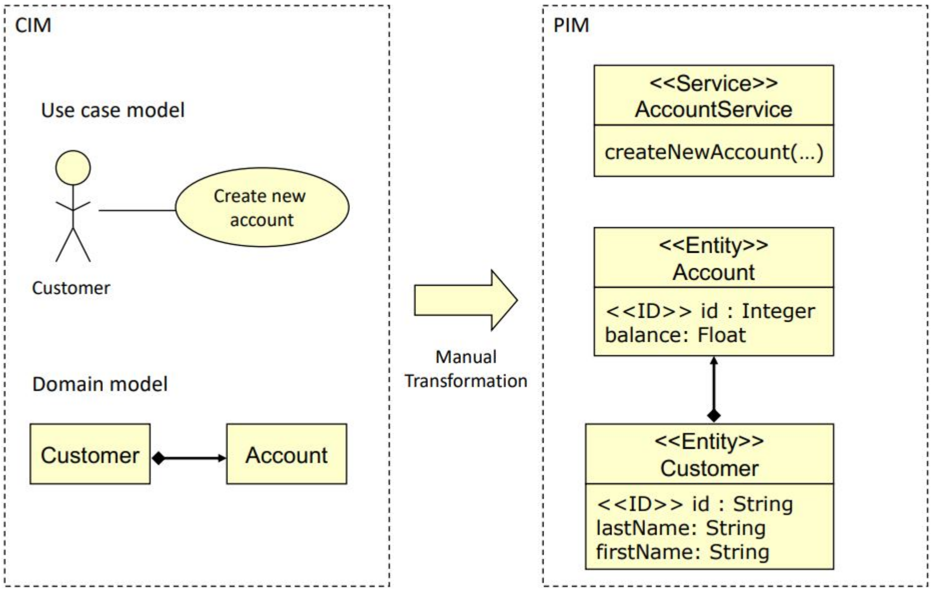


Abbildung 3.1: How to convert a CIM to a PIM

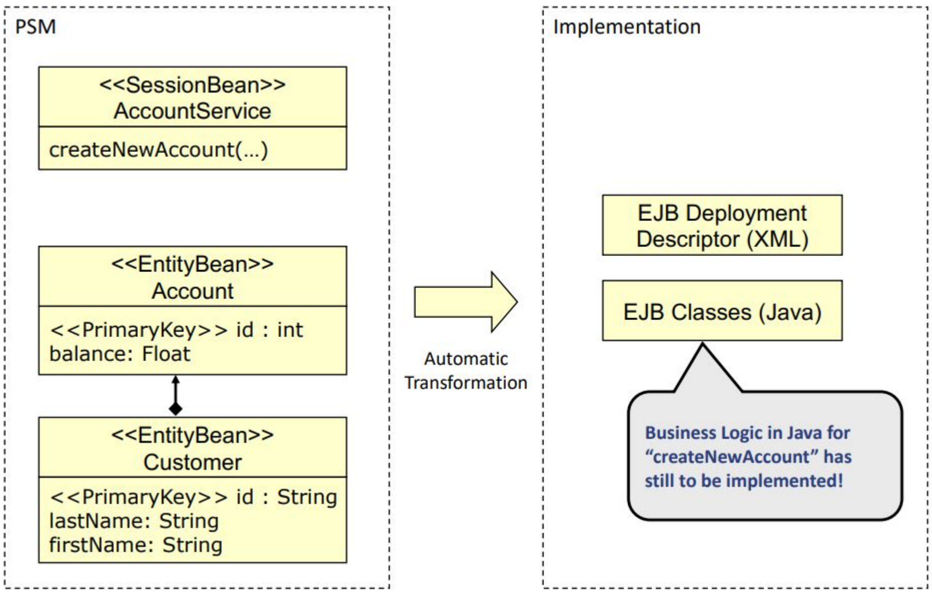


Abbildung 3.2: Conversion from a PSM to an implementation

3.4 Model transformation

This is the process of converting one model to another model of the same system. It is done by a process called **mapping**. An MDA mapping is a set of specifications for transformation of a PIM into a PSM for a particular platform. The platform model will determine the nature of the mapping.

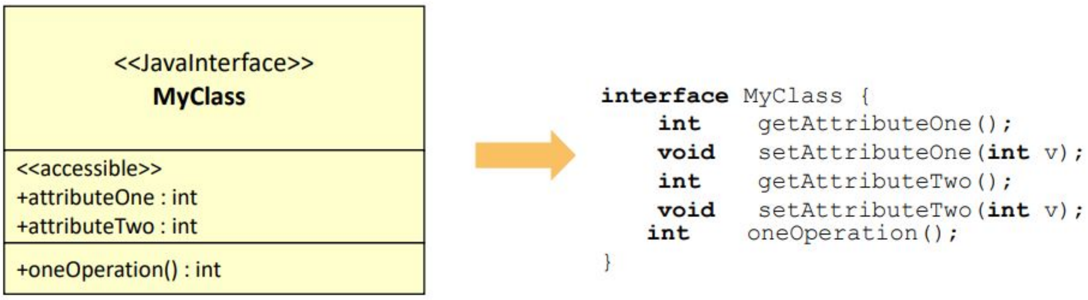


Abbildung 3.3: Model-to-code transformation – an example for a mapping

3.4.1 The MDA Pattern

The MDA pattern includes at least

- a PIM
- a PM
- a Transformation
- a PSM

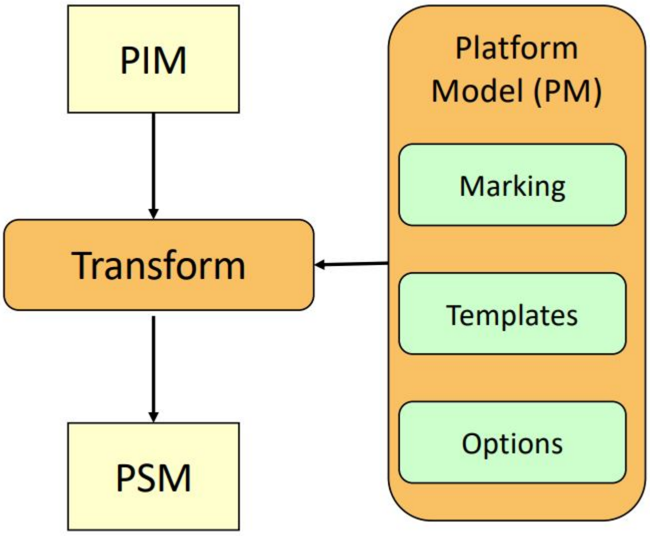


Abbildung 3.4: The MDA pattern (minimal form)

As shown in the figure, the PM influences the nature of the mapping. The PM does this through three concepts (first two described more elaborately in a later section):

1. Marking
2. Transformation Templates
3. Options: Adjust the transformation globally (similar to compiler options)

3.4.2 Mapping Concepts

Metamodel Mapping

mapping gives rules and/or algorithms how types of the PIM metamodel are to be transformed to types of the PSM metamodel. Not applicable if PSM has no metamodel specified, e.g. in Model-to-Code transformations, where the PSM is Code.

Marking

A mark represents a concept in the PSM, which can be applied to an element of the PIM to indicate how that element is to be transformed:
if more than one PSM-alternative for something in the PIM exists, the mark indicates which alternative should be taken.
Also, different platform mappings may require different markings.
Example from the lecture: In J2EE, there are two types of EJBs. Marking defines which one to use.

Transformation Templates

Parameterized models that specify particular kinds of transformations (a bit like design patterns). Typically creates groups of elements out of one element in the PIM.

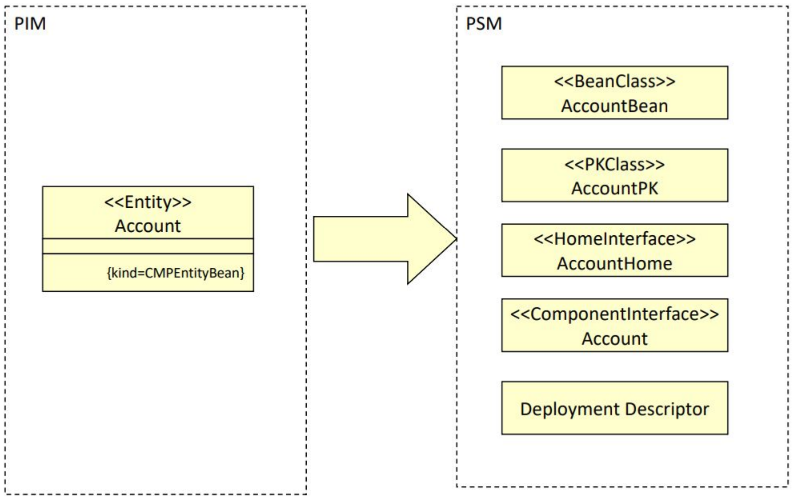


Abbildung 3.5: Example for a Transformation template

Multi-staged Transformation

= Applying MDA Pattern in a cascade. The MDA pattern can (and usually has to) be applied several times in succession; the output PSM from one iteration will then become a PIM for the next one
$\Longrightarrow$ PIM and PSM are relative concepts, they depend on the platform in use

Multi-platform Transformation

Many systems are (can be) built on more than one platform. Multi-platform Transformation means using different PMs to transform a PIM into different PSMs with parts of the system on different platforms with connections/adapters between them.

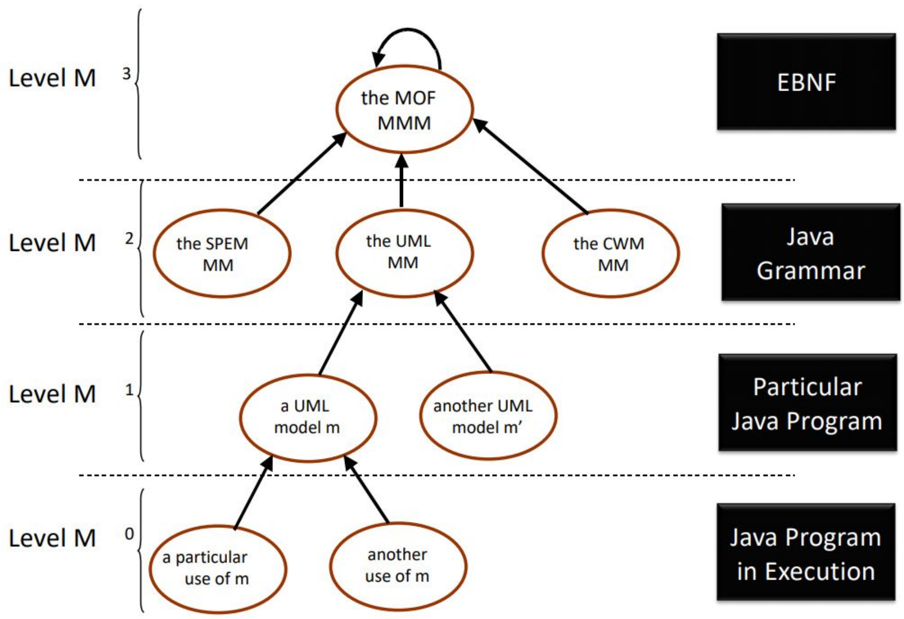
3.5 Advantages of MDA

- Each model is independent of the rest
- Software development gets increasingly closer to model transformation
- Transformations can be automated
- We gain modularity, flexibility, and facilitate evolution
- Application models capturing business logic and intellectual properties (IP) become **corporate assets**, independent from the final implementation technologies

3.6 MDA Standards

3.6.1 Meta Object Facility (MOF)

Meta-Meta-Model for the construction of metamodels in MDA



3.6.2 Unified Modeling Language (UML)

UML is central for MDA because many tools are based on UML and its extension capabilities (UML Profiles). From version 2.0 on, UML is formally defined via the MOF.

Extending UML

UML can not be complete, because it's not feasible to specify every details. There are two ways to extend UML/MOF:

1. Heavyweight: completely new meta-model based on MOF (Not automatically supported by modeling tools); essentially creating a new modeling language from MOF
2. Lightweight: Extension based on the UML Metamodel or with UML Profiles

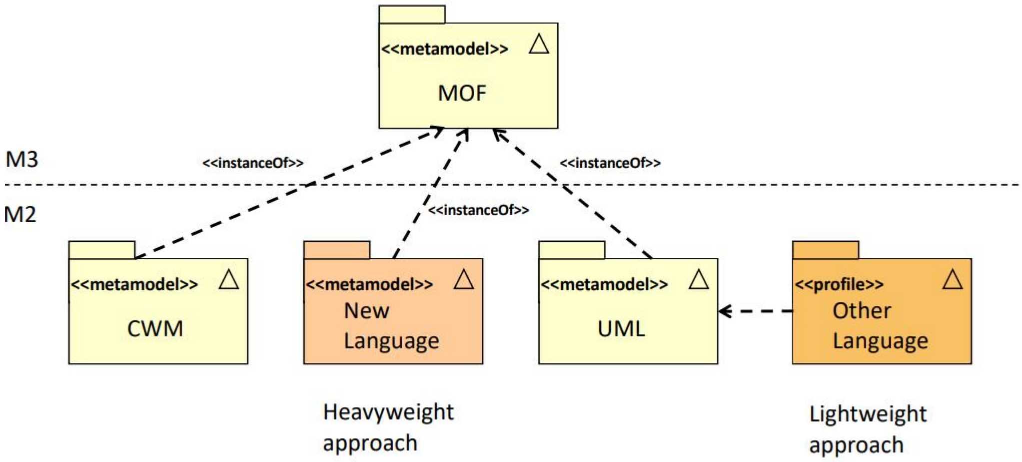
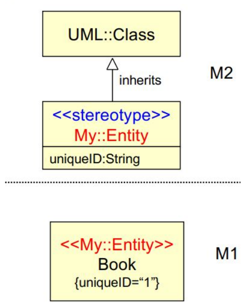


Abbildung 3.6: Comparison of the two extension approaches

Extension based on the metamodel



Two new features (can be interpreted by code generators!!!!1!!1!!!):

- Stereotype: represented by «. . . »; Specifies the metaclass
- Tagged Value: represented by {. . . }; Specifies an attribute of the metaclass

UML Profiles

Mostly used to specialize UML for specific domains, when there is no need to change UML metamodel and semantics. They are an excellent mechanism for defining MDA 'Platforms'. A UML profile consists of:

- Stereotypes: Used to refine meta-classes (or other stereotypes) by defining supplemental semantics
- Tagged values: Attributes of stereotypes with user-defined semantics; Rendered as tagged values in the model in which the stereotype is used

- OCL constraints: Predicates (e.g., OCL expressions) that reduce semantic variation; Can be attached to any metaclass or stereotype

There is a UML language construct for an extension: a filled inheritance arrow.
An extension conforming to the UML standard must not violate the standard UML semantics

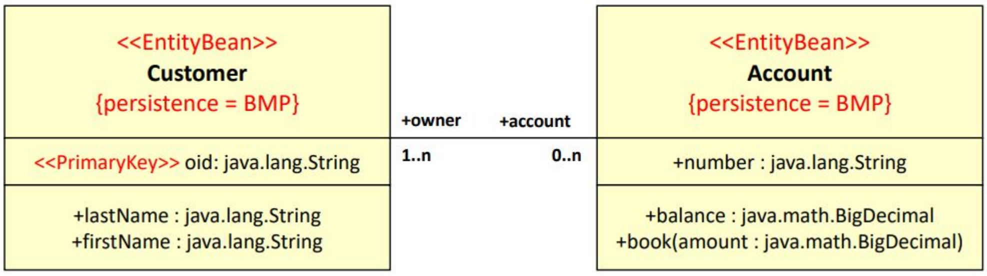


Abbildung 3.7: Usage of the EJB profile

4 Transactions

4.1 Definition

- Transaction resembles flow (cash, goods, etc.)
- Transactions are the reason for business in the first place
- Application systems must support transaction programs!
- "Transactions are the heart of economy"

4.2 Concept

- A transaction is a process, that accesses and may updates data items
  databases
  resource managers
- A transaction must see a consistent database at start
- During transaction, a database may enter a inconsistent state
- After the transaction is done, the Database must be consistent again
- There are 2 possible issues:
  Recovery (system crashes and similar)
  Concurrency Control (keeping the data consistent while multiple transactions are executed)

4.3 Concurrent Executions

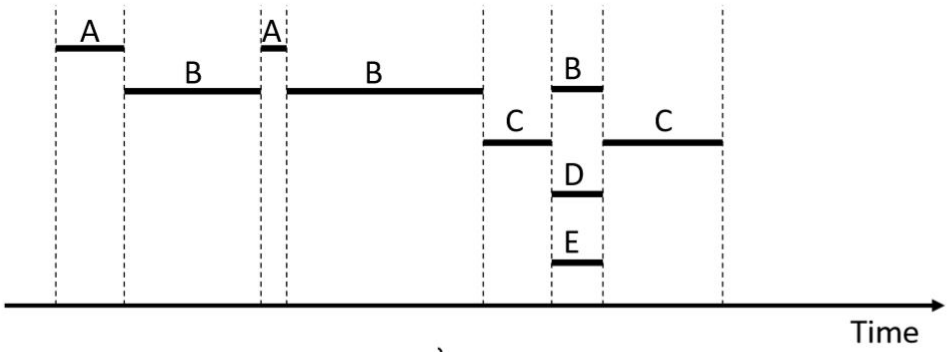


Abbildung 4.1: Types of Concurrent Transactions

- A,B,C: Interleaved
- B,D,E: Simultaneous
- all: Parallel Transactions

4.4 Benefits of Concurrency

- Higher Throughput
- More Utilization of the CPU = better value
- faster Response Time

4.5 Possible Failures

- System crash
- Transaktion Error
- Concurrency Control Enforcements (Scheduler aborts)
- Disk Failure (Data lost)
- Physikal Problems (wrong Disk hooked up, Fire etc.)

4.6 Transaction Processing

Transaction Processing is about:

- Maximum throughput
- Maximum utilization
- Maximum availability
- Maximum scalability
- Minimum downtime

4.7 ACID

- Atomicity
  Transactions are fully reflected in the Resource Manager or are not reflected
- Consistency
  The Consistency of the resources is not harmed by any Transaction
- Isolation
  Transactions made at the same Time, don't need to know from each other to achieve correctnes
- Durability
  If a transaction is completed succesfully, all changes are permanent

4.8 Transaction Operations

4.8.1 BOT = Begin of Transaction

Implicit BOT

Automatically issued on behalf of transaction of first resource manager request (after former EOT)

Explicit BOT

Issuing a BEGIN operation

4.8.2 EOT = End of Transaction

Implicit EOT

Resource manager decision based on transaction's state

- May be COMMIT or ABORT

Explicit EOT

4.8.3 COMMIT

Request to make all changes permanent

4.8.4 ABORT/Roll back

All changes must be reverted

4.9 Example in RL

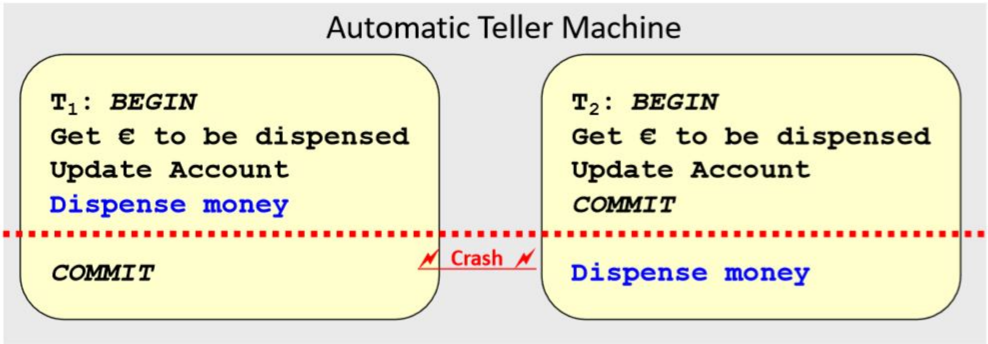


Abbildung 4.2: Real World Actions In Transactions

4.10 Transaction States

- active

  the initial state; the transaction stays in this state while it is executing
- done

  all statements have been executed
- failed

  n
- aborted normal execution can no longer be achieved
- commited

  after successful completion
- aborted

  transaction was aborted and all changes are reverted

4.11 Serializability

4.11.1 Schedule

Order sequence in which concurrent transactions are executed

4.11.2 Conflicts

A conflict occures if and only if two instructions both acces the same data at the same time and at least one of them is a write.
Intuitively, a conflict between i_k and i_j forces a (logical) temporal order between them.

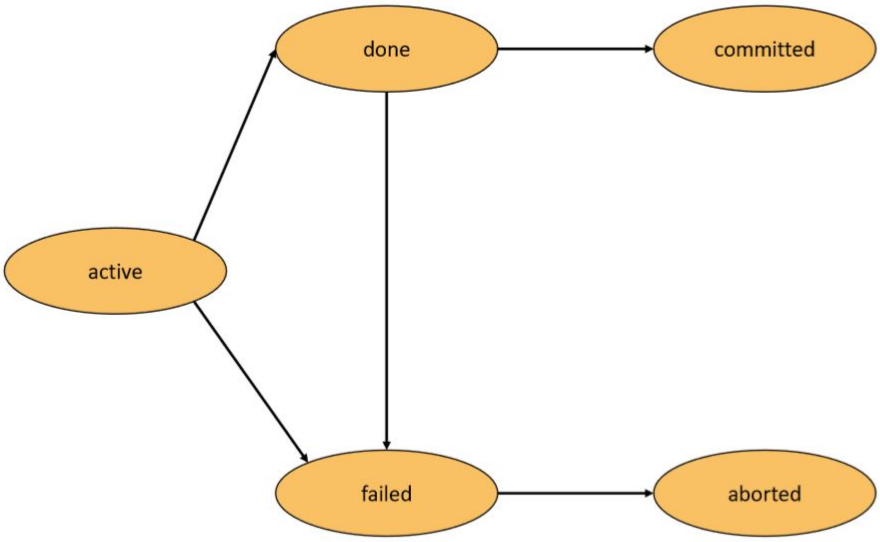


Abbildung 4.3: Transaction State Diagram

4.11.3 Swaps

If two transactions operate on different data, or don't interfere in general, they can be swaped (change their order) without changing the result.

4.11.4 Conflict Serializability

If a schedule S can be transformed into a schedule S' by a series of swaps of non-conflicting instructions, we say that S and S' are **conflict equivalent** :S ≡ S'

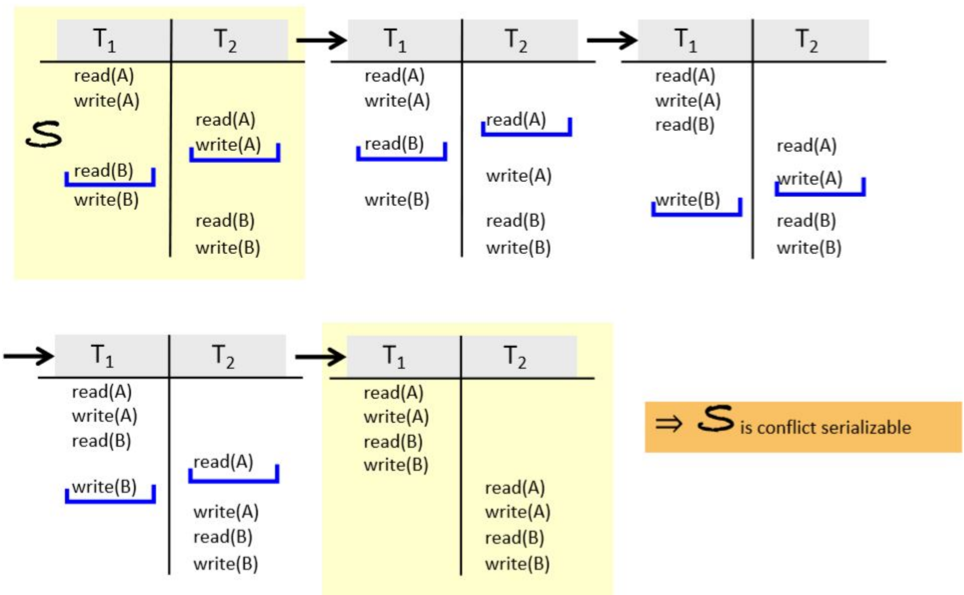


Abbildung 4.4: Conflict Serializability Example

4.11.5 Recoverability

Writer must commit before reader, so if the writer aborts, the transaction that read the uncommited changes can and must abort too.
Can lead to cascading aborts what leads to reverting a significant amount of changes

4.11.6 ACA Schedules

- prevents cascading aborts
- For each pair of transactions T_{writer} and T_{reader} such that T_{reader} reads a data item previously written by T_{writer}, the commit operation of the writing transaction T_{writer} appears before the readoperation of T_{reader}

4.11.7 Testing for Serializability – Precedence graph

transactions = nodes
conflicts = arcs between nodes
accessed item (that is causing conflict) = label of arc

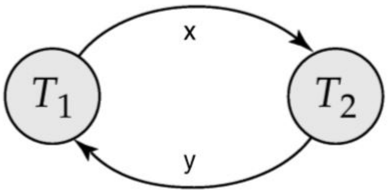


Abbildung 4.5: Precedence Graph

Theorem: A schedule is conflict serializable if and only if its precedence graph is acyclic

Theorem: If precedence graph is acyclic, a serializability order can be obtained by a topological sorting of the graph

4.12 Distributed Transactions

4.12.1 Atomicity in distributed transactions

If one System loses update, so it cannot commit the transaction when it recovers ⇒ Whole transaction must fail!

4.12.2 Transaction models

X/Open Model

Two-phase commit protocol (2PC)

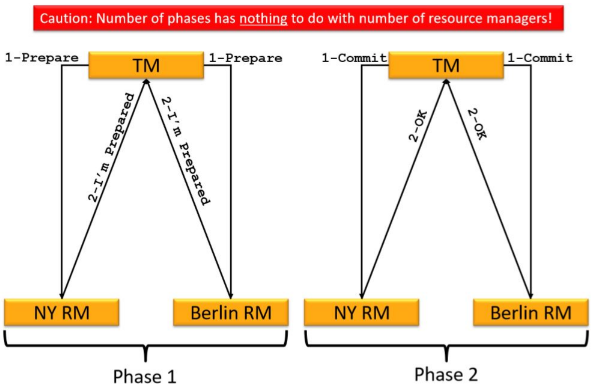


Abbildung 4.6: Two Phase Commit Protocol

Atomic Commitment Protocol (ACP)

- All participants that reach a decision reach the same one
- A participant cannot reverse its decision after it has reached one
- The COMMIT decision can only be reached if all participants voted YES
- If all participants voted YES and no failure occurred the decision will finally be COMMIT
- If all existing failures are repaired and no new failures occurred for sufficiently long, then all participants eventually reach a decision

4.12.3 Recovery

4.12.4 2PC Message Flow and Logging

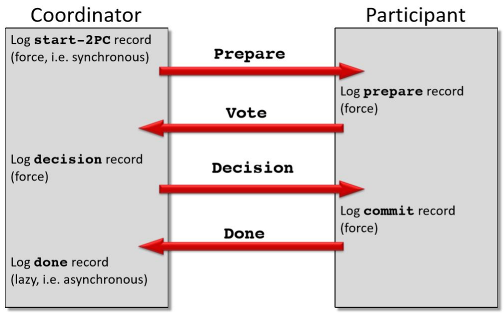


Abbildung 4.7: Message Flow and Logging

Coordinator Recovery

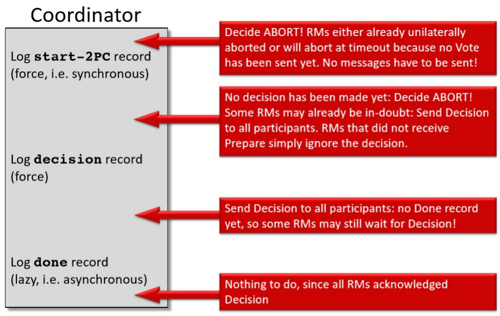


Abbildung 4.8: Coordinator Recovery

Participant Recovery

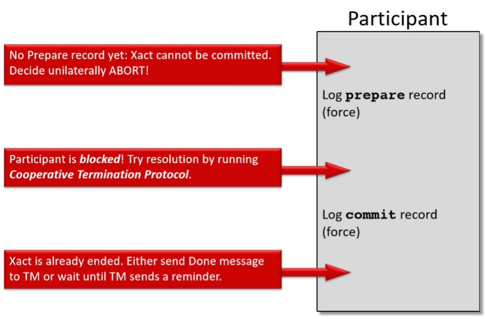


Abbildung 4.9: Participant Recovery

4.12.5 Blocking Participants

- During the time interval from having prepared until reception of the TM's decision a participant is blocked (ïn-doubt")

  Blocking:= RM waits for decision message of TM

  If TM fails during this period RMs have to wait until TM recovers

  If RM fails during this period it must establish connection to TM to recover any in-doubt transaction

  Network fragmentation may separate TM from some RMs causing blocking

- Cooperative Termination Protocols: allow in certain situations to end a transaction without a connection to TM

- Heuristic termination: often used in practice

  Blocked RM assumes that its decision is the one of TM too

4.12.6 Cooperative Termination Protocol

- Together with PREPARE request TM sends list of all participants

- Participants log this list together with VOTE record

- When in-doubt participant (ïnitiator") cannot connect to TM it contacts reachable participants ("responder") from this list

  If all responders are in doubt initiator remains in-doubt too

  If at least one responder knows TM's decision (commit or abort) initiator decides accordingly

  If at least one responder has not voted yet responder unilaterally decides ABORT and initiator decides accordingly

4.12.7 Transaction branches

- When an application communicates with another application associated with another transaction manager, a new branchof the same global transaction is created

    A.k.a. "transaction infection"

    This happens based on receiving a transaction context

- A resource manager may have more than one active branch for the same global transaction

- All branches belong to the same global transaction, i.e. all branches either commit or abort

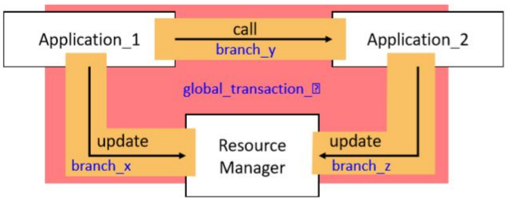


Abbildung 4.10: Transaction Branches

5 RPC & API & MOM

5.1 RPC

5.1.1 Data Conversion Problems without RPC-Middleware

- Converting data structures into messages: Data structure as processed by programs must be flattened and reconstructed for exchange ((**de-)marshalling, (de-)serialization**)
- Converting data types: Sender and receiver may be implemented in different programming languages that support different sets of data types or may use different representation for some data types

⇒ Solution: Standard data representation; e.g. CORBA: CDR, Web Services: SOAP

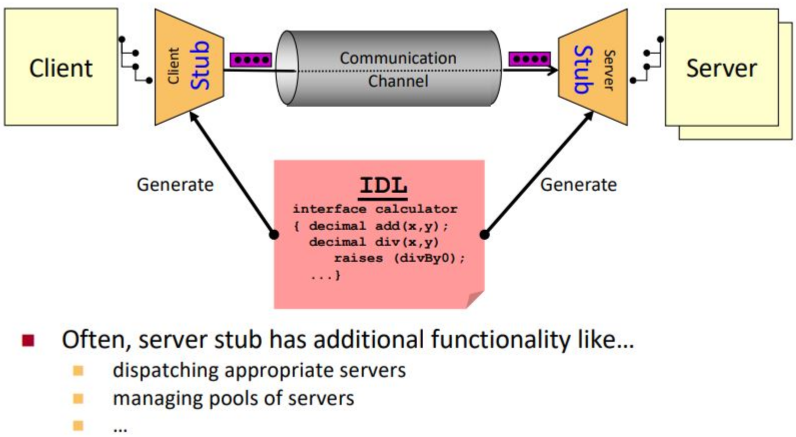
5.1.2 Other Problems

- Binding: Finding appropriate services amongst collection of services on various machines
- Fault Handling: Transparent handling of (communication, invocation,. . .) errors, e.g. Network is down, Machine is busy, Duplicated requests, . . .

⇒ Solution: Interface definition languages (IDLs)

5.1.3 IDLs

- Language to describe services in an abstract manner. Definitions are independent of the PL used to implement clients and services → Supports interoperability btw. languages
- Code is generated to be invoked by client and code that invokes services on server that deals with all these problems – so-called stubs



5.2 API

API = The set of combined interfaces making the functions of a particular application available in a coherent manner.

5.2.1 Structure of a remote API

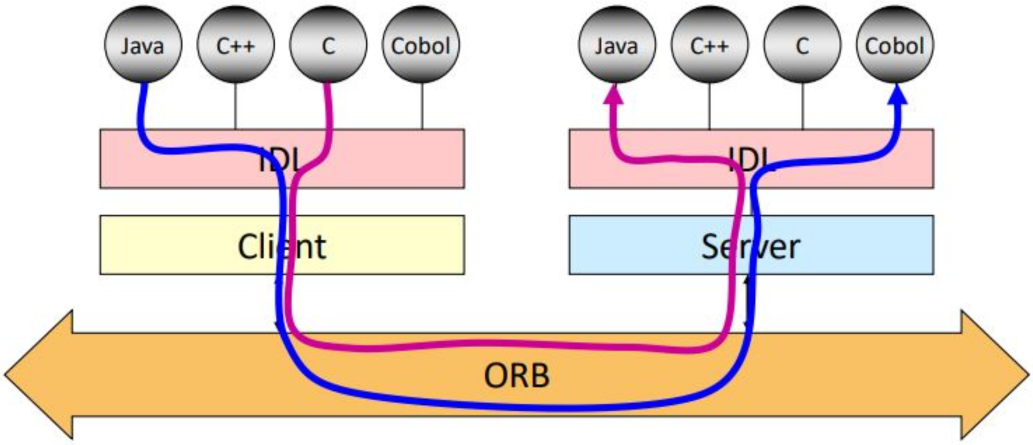
A remote API is split into two parts: The proxy on the client side and the stub on the application side. Communication logic is between the proxy and the stub.
client programs against the proxy and doesn't know whether the functions used reside on its local machine or on some remote machine → Local/remote transparency

5.2.2 CORBA – RPC for objects

CORBA is an Object Management Group specification for a **C**ommon **ORB** **A**rchitecture (ORB explained later).
Defines a metamodel for (distributed) objects and a corresponding IDL (CORBA IDL)
Bindings of this IDL to different programming languages are specified

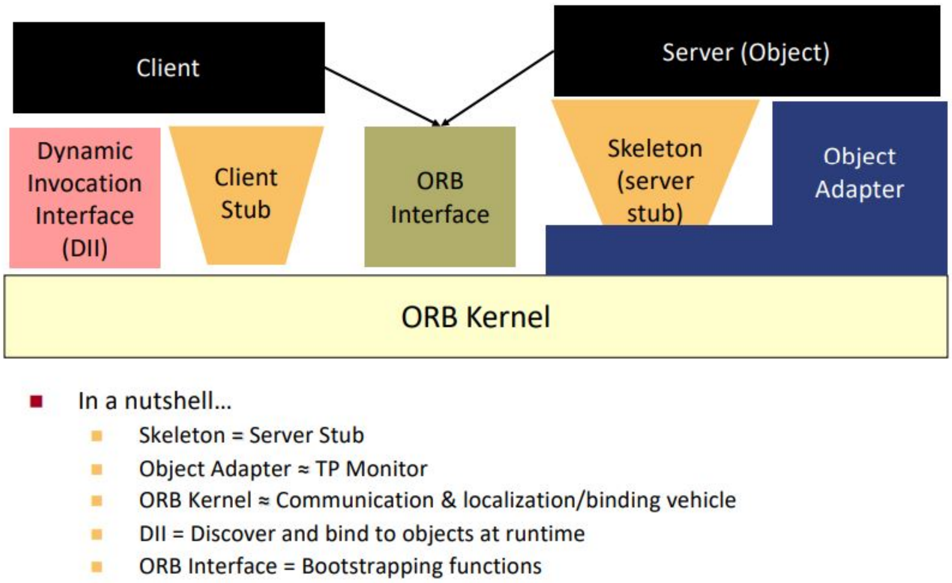
Object Request Broker (ORB)

- mediates remote method invocations in (remote) distributed object systems
- supports communication between executables implemented in different programming languages



CORBA Architecture

.



ORB interoperability

CORBA ORBs of different vendors have to interoperate $\Rightarrow$ GIOP (General Inter-ORB Protocol)
IIOP specifies how GIOP is done over TCP
Two ways of connecting ORBs:

1. Full Bridge (Request-Level Bridge): Bilateral non-standard bridge for interoperation between two environments

2. Half Bridge (Inline Bridge): Mapping from/to vendor specifics to IIOP

Half bridges are way more common than full bridges.

5.3 MOM (Message Buses)

5.3.1 Basics

Distributed functions can use messaging to communicate and transfer data. But sending data to another computer is a lot more complicated and requires data to be copied from one computer to another $\rightarrow$ data has to be serializable
When connecting multiple computer systems via remote communication, these systems likely use different languages, technologies and platforms. Messaging system can be a universal translator between applications, allow them to communicate through a common messaging paradigm.
This is called a message bus or MOM

5.3.2 Advantages

Asynchronous Communication

The sender does not have to wait for the receiver to receive and process the message

Variable Timing

the sender can batch requests to the receiver at its own pace, and the receiver can consume them at its own (probably different) pace

Avoid Throttling

Too many RPC calls at a time can overload the receiver and even cause it to crash. Asynchronous communication enables the receiver to control the rate at which it consumes requests. Effect on thee caller is minimal because it does not have to wait for the receiver.

Reliability

Messaging provides reliable delivery through the

- Store and forward

- guaranteed delivery

approaches.

5.3.3 Disadvantage: Complex Programming Model

ogic is split up into a number of event handlers that respond to incoming messages. Such a system is more complex and harder to develop and debug.
Moreover, Transaction model is most often compensation based, more complex than ACID and 2PC.

5.3.4 Message Queuing – MQM (Message Queue Manager)

Provides environment for queuing applications.

- provides reliable storage for queued messages
- manages concurrent access to data
- ensures security and authorization
- provides special queuing functions (like triggering)

Applications connect to exactly one MQM which is then called the *local MQM* and then use the Message Queuing interface (MQI).

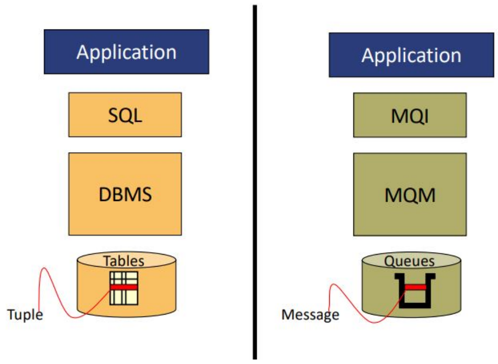


Abbildung 5.1: Analogy for people familiar with Databases

Delivering Remote Messages – The Mover

- processes messages in a transmission queue jointly with its partner mover at the other end of the channel, thus ensuring reliable transmission to remote MQM
  ⇒ realization of the guaranteed delivery approach
- Failures are tolerated: The only effect of a channel failure is a delay of the message transmission until channel becomes available again (≈ 2PC protocol)

5.3.5 Message Queuing – MQI (Message Queuing Interface)

Target of communication is queue, not program.
Two important concepts:

1. **send-and-forget**: Program simply puts message to queue and continues processing
2. **store-and-forward** (if queue is remote): local MQM ensures delivery to remote queue

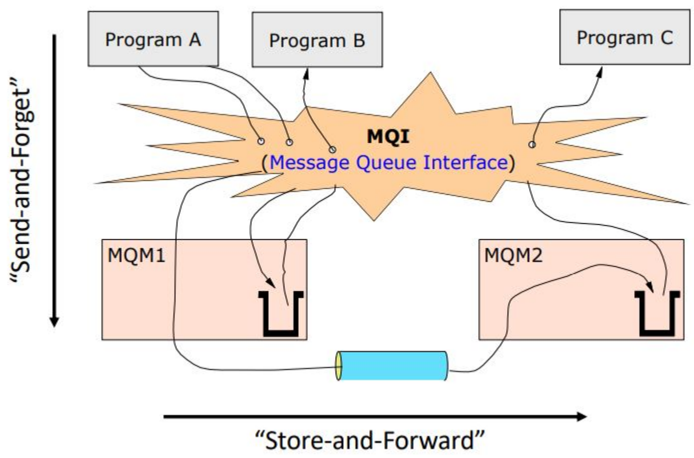


Abbildung 5.2: The MQI

MQI vocabulary

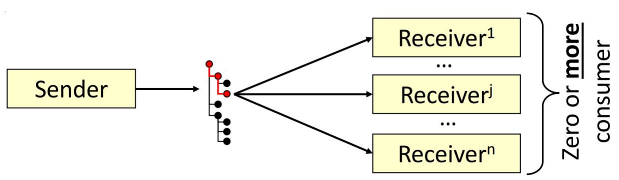
- **MQCONN**: Connects a program to a MQM; A particular MQM can be specified by name
- **MQDISC**: Tells the MQM to release resources required for supporting the program; All open queues are implicitly closed
- **MQOPEN**: Gives a program access to a particular queue (a program can have concurrent access to multiple queues)
- **MQCLOSE**: Tells the MQM to release the resources needed to support the programs operations on the associated queue
- **MQINQ**: Retrieves attributes of a particular queue and the MQM connected to it (example: Number of messages in queue, number of programs which opened the queue, ...)
- **MQSET**: Alters the current values of attributes of an object (example: Number messages required in queue before triggering occurs)
- **MQPUT**: Puts message in a queue (local or remote)
- **MQGET**: Retrieve a message from a specified queue (FIFO or selectively); Specified queue must be local!

(More in the slides...)

5.3.6 Principles

PubSub (PublishSubscribe)

A sender publishes a message to a *topic*; Zero or more *subscribers* to that topic get this message



Topics can have 0 or more topics as children; topic without parent is called *root topic*. The hierarchy consisting of a root topic and all its descendants is called a *topic tree*. A subscriber of a topic automatically subscribes to all of its children.

Loose coupling

Reduce number of assumptions two parties make about each other when they exchange information → more tolerance to changes at a partner's side
But more assumptions increase efficiency → in high-performance environments, coupling is tight. Loose Coupling also affords

- platform
- time
- reference
- format

autonomy.

6 TP Monitors

7 N-Tier Structures & Application Servers

7.1 Application Layers

7.1.1 Presentation layer

Channel between the user and the application. What it does:

- formats and protocols (FAPs): Browser, email, EDI, API, ...
- Renders data
- Reacts to device events (mouse, keyboard, etc.)
- Communication/conversation logic: 'Session' with requestor

7.1.2 Application Logic Layer

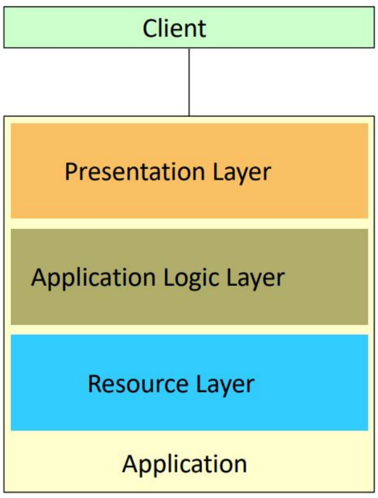
Here resides the proper logic performed by the application. The functions offered via the presentation layer are here.

7.1.3 Resource Layer

The logic to access the data needed by the application logic is in this layer as well as all DBs, files, queues, ...

7.2 One Tier – Monoliths (or Mainframes)

Client and presentation layer are not the same!!! 'Application' can also be called 'host' or 'server'.



Advantages

- Everything is centralized, managing and controlling resources is 'easy'
- Design can be highly optimized by *blurring the layers*

Disadvantages

- Ignores computing power available on clients
- Code is hard to maintain
- Reuse of functions very difficult

7.3 Two tiers – Client-Server

7.3.1 Client/Server cuts

A client-server system is a consequence of cutting an application horizontally into two parts. The upper part is the *cliet*, the lower part becomes the *server*. Depending on the location of the cut, different topologies (with pros and cons) result. The problem is always **where to put which functionality** based on nonfunctional goals (Performance, resuability,. . .)

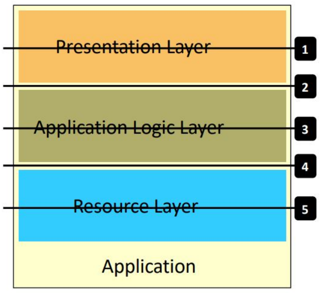


Abbildung 7.1: Different topologies result when cutting at different positions

7.3.2 The different topologies

Number in the list represents the corresponding cut from the figure above

1. Distributed presentation
2. Remote Presentation
3. Distributed Application
4. Remote Data
5. Distributed Data

7.3.3 Limitations

Often, a client needs to access many servers on many different machines ('islands of information')
⇒ Client must integrate different functions, e.g. Transforming data, Transactions across applications, . . .
⇒ Client must deal with heterogeneity: (a-)synchronous APIs, different PLs (Java C ♯ . . . API, . . .) and auth. mechanisms
Resulting problems include:

- Complexity: client design becomes increasingly difficult
- Fragility: if one API changes, client has to be changed too
- Performance issues: client machine has limited capabilities

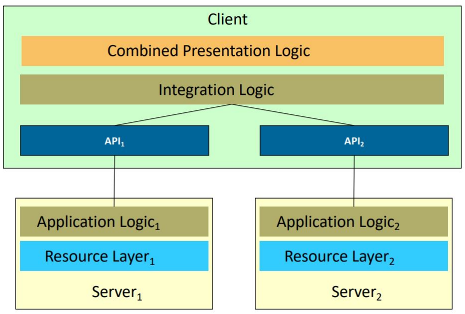


Abbildung 7.2: client as integration point – a bad idea and a limitation of two-tier systems

7.4 Three tiers – Middleware

7.4.1 What is Middleware?

- Middleware runs the integration logic
- Client (presentation logic) simply invokes integration logic to access functionality
  ⇒ from the client's perspective, the middleware **is** the appl. logic
- Typically, the layers are distributed across different machines taking advantage of the complete modularity of the design; The middleware runs on the middle-tier

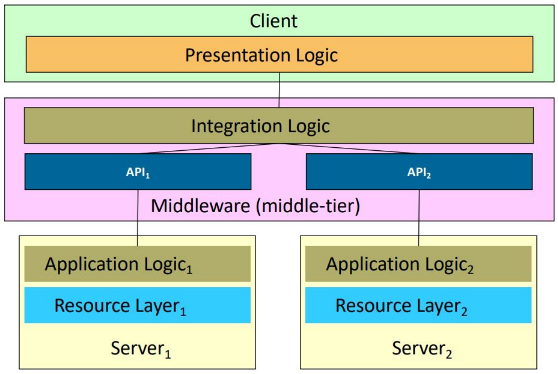
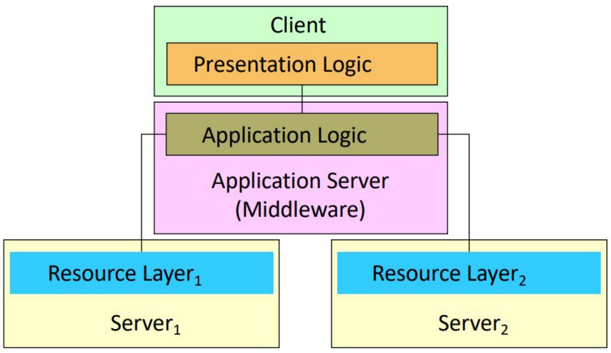


Abbildung 7.3: Middleware: Pushing out Integration Logic

7.4.2 Application server

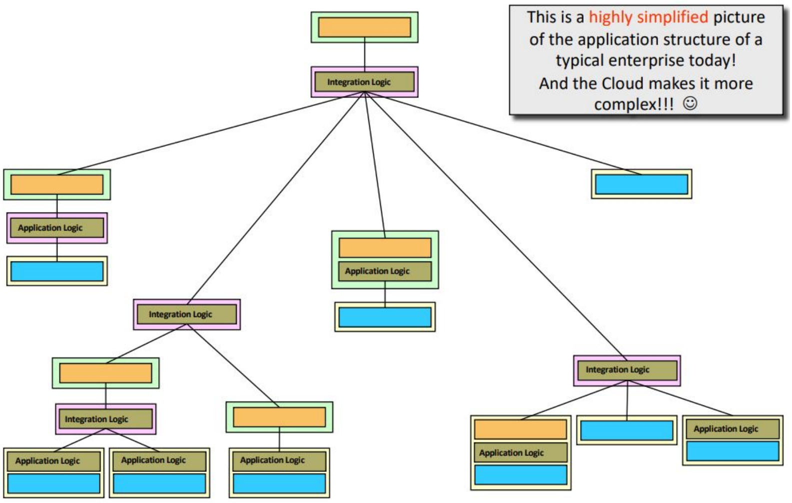
From a client perspective, there is no real difference between integration logic and appl. logic, so it makes sense to consolidate these two, using middleware to iplement the entire application logic, not only that which deals with integration ⇒ **Application server**



7.5 N-tier structure

In the real world, clients need integrated access to all kind of applications which

- may not have an application server with integration logic ⇒ we need an integration layer again, this time to manage connecting to all these different applications ⇒ fourth tier
- could in turn use other applications and so on ⇒ lots of tiers



7.6 Role of the application server

7.7 What is a component?

7.7.1 Reusability

7.7.2 Technical properties

7.7.3 Component model

...

7.8 JEE fundamentals

7.8.1 Enterprise Java Beans (EJBs)

EJBs = POJOs implementing POJIs
There are three different types of EJBs:

- Session Beans: Reflect transaction programs in a TP Monitor; two variants exist:

  stateless (SLSBs): 'one time shot' interaction only

  stateful (SFSBs): 'conversational' interaction

- Message Beans: Reflect a message-based application that communicates via queues or publish/subscribe with its clients
- Entity Beans: Reflects a persistent entity; optional to support

7.8.2 JEE Examples

Note: in the lecture, there have been many images from Oracle's JEE 7 tutorial. These will not be shown here, but are available at: `https://docs.oracle.com/javaee/7/JEETT.pdf`

7.8.3 Implications of reusability requirement

7.8.4 Annotations

7.8.5 Deployment descriptors vs. annotations

7.8.6 Notion of a container

7.9 JEE packaging and deployment