Distributed Systems Engineering - Summary

Author: @triggetry

October 11, 2013

Contents

1 Preamble 3

2 Chapter 1 3
- 2.1 Engineering 3
- 2.2 Software Engineering 3
- 2.3 Principles of Disign Methods 3
- 2.4 Design Methods vs. Procedure Models 3
- 2.5 Distributed System 4
  - 2.5.1 What is getting distributed? 4
  - 2.5.2 Challenges 4
  - 2.5.3 CAP Theorem 5
- 2.6 ACID vs. BASE 5
- 2.7 Distributed Systems: Middleware 5
- 2.8 Architectural Distributed Styles 6
- 2.9 Message-oriented Middleware (MOM) 6
- 2.10 Service-oriented Architecture (SOA) 7
- 2.11 Transaction Processing Monitor 7
- 2.12 Distributed Transactions 7
- 2.13 Server Components and Middleware 8
- 2.14 Other distributed Infrastructures 8

3 Chapter 2 8
- 3.1 Three dimenstion of Engineering 8
- 3.2 Design Methods and Principles 9
  - 3.2.1 Architecture/Design Methods 9
  - 3.2.2 CCD Design Principles 9
  - 3.2.3 OO Design Principles 10
- 3.3 DS design following rational OOAD 10
  - 3.3.1 Design: Tasks to complete 10
  - 3.3.2 Analysis vs Design 11
- 3.4 Modeling distribution and Parallelism 11
  - 3.4.1 Modeling Distribution 11
  - 3.4.2 Modeling Parallelism 12
- 3.5 Remoting Styles 13
  - 3.5.1 Remoting Style: RPC 13
  - 3.5.2 Remoting Style: Messaging 13
  - 3.5.3 Remoting Style: Shared Repository 14
  - 3.5.4 Remoting Style: Streaming 14
- 3.6 Remoting Patterns 14
  - 3.6.1 Software Patterns 14
  - 3.6.2 Remoting Patterns 15
  - 3.6.3 Basic Remoting Patterns 15
- 3.7 Advanced Patterns: CQRS and APP-Level Mod. 16
  - 3.7.1 CQRS - Command Query Responibility Segregation 16
  - 3.7.2 Application-level Modulatisation 18
  - 3.7.3 Domain Driven Design 18

4 Chapter 3 18
4.1 Cloud Computing . . . 18
4.1.1 Cloud Computing - Definition . . . 18
4.1.2 Why Cloud Computing? . . . 19
4.1.3 Benefits of cloud computing . . . 19
4.1.4 Infrastructure as a Service (IaaS) . . . 20
4.1.5 Plaform as a Service (PaaS) . . . 20
4.1.6 Software as a Service (SaaS) . . . 21
4.1.7 Public PaaS offerings . . . 21
4.1.8 Private PaaS offerings . . . 21

5 Chapter 4 21
5.1 Software Business undergoes Change . . . 21
5.2 Creating a new System of the world . . . 22
5.2.1 Conways Law . . . 22
5.3 Engineering for PaaS . . . 22
5.4 Application Life-Cycle Automation . . . 23
5.5 Cloud Foundry . . . 23
5.5.1 Architecture . . . 23

6 Questions 24

1 Preamble

This document is a collections of articles and informations, mainly from Wikipedia and other online resources as well as the offical slides, which cover the topics of the lecture in *Distributed Systems Engineering*. Note that this summary is **not** complete, and by no means a guarantee for passing the exam, although it should cover approx. 90 % of the material presented in the summer term of 2013. If you need the sources of the document, feel free to contact me via twitter (see title page).

2 Chapter 1

2.1 Engineering

Engineering is the discipline, art, skill, profession and technology of acquiring and applying scientific, mathematical, economic, social, and practical knowledge, in order to design and build structures, machines, devices, systems, materials and processes.

2.2 Software Engineering

Software Engineering is the application of a systematic, disciplined, quantifiable approach to the development, operation, and maintenance of software, and the study of these approaches.
That is, the application of engineering to software.

2.3 Principles of Disign Methods

- Abstraction
- Visualization
- Modeling
- Divide and Conquer
- Branch and Bound (optimized backtracking)
- Object Oriented Principles
  - Single Responsibility Principle (SRP)
  - Seperation of Concerns
  - DRY
  - Program against Interfaces
  - Dependency Injection
  - Dependency Inversion Principle

2.4 Design Methods vs. Procedure Models

Design Methods Translate the *What* into the *How*. Examples:

- Structured Design (SD)
  - Hierachical structured functional modules
  - Decomposition
  - Strictly divided view of data and functions
- Object oriented Design (OOD)
  - Class hierarchies
  - Information hiding, inheritance, polimorphism

Procedure Models Set of rules for creating a piece of software from the beginning to the delivery. Examples:

- Waterfall
- Spiral model (Barry Bohm)
- RUP/OpenUP

- V-Modell (XT)
- Agile
  - XP
  - Crystal
  - Scrum...

2.5 Distributed System

A *distributed system* consists of **multiple autonomous computers** that *communicate* throuch a computer network. The computer interact with each other to achieve a common goal.
A need for distributed systems can arise from the distributed nature of the problem (Problem-related reasons), e.g. a user in europe wants to read a webpage which is hosted on a server in the USA, the webpage has to be transported to the remote user. There are also property-related reasons for distributed Systems, e.g. performance and scalability, fault tolerance, maintainablility and deployment, security or business integration. Some application areas for distributed systems are:

- Internet
- Telecommunication Networks
- Business-to-Business collaboration systems
- International financial transactions
- Embedded Systems
- Scientific applications
- and many more...

2.5.1 What is getting distributed?

- Hardware (Memory and Storage, Computing Power)
- Software (Logic / Algorithms)
- Control

2.5.2 Challenges

Network Latency. A (remote) invocation in a distributed systems takes more time than in a non-distributed system.

Predictability. The invocation time can differ from time to time (depending on the network).

Concurrency. When developing a distributed system you are faced with real concurrency problems.

Scalability. It is not alway possible to know the communication load of each independent part of the system, so they have to be highly scalable.

Partial Failure. In a distributed system only a part of the whole system might fail. The rest of the system should still fulfill the overall system task.

Only distribute your system if distribution is really needed and carefully evaluate when and what to distribute, because distribution adds all these challenges and problems to your application that would not occure in a non-distributed system!

Don't distribute your objects

The first law of distributed objects. Martin Fowler argues that the first law of distributed objects is: Don't distribute objects! Objects are such low-level entities that the overhead of keeping track of them and maintaining them outweighs and defeats the supposed performance benefit of distributing them. Instead, an application that clusters at the component level, not the object level, is easier to design, deploy, and maintain; and the parallelism of clustered components yields reliability and performance benefits.

2.5.3 CAP Theorem

In theoretical computer science, the CAP theorem, also known as Brewer's theorem, states that it is impossible for a distributed computer system to simultaneously provide all three of the following guarantees:

- Consistency (all nodes see the same data at the same time) *Note: Consistency in CAP is not the same as Consistency in ACID! 2.6*
- Availability (a guarantee that every request receives a response about whether it was successful or failed)
- Partition tolerance (the system continues to operate despite arbitrary message loss or failure of part of the system)

According to the theorem, a distributed system cannot satisfy all three of these guarantees at the same time. In May 2012 Brewer clarified some of his positions on why the oft-used *two out of three* concept can be misleading or misapplied. [1]

2.6 ACID vs. BASE[2]

Database developers all know the ACID acronym. It says that database transactions should be:

Atomic: Everything in a transaction succeeds or the entire transaction is rolled back.
Consistent: A transaction cannot leave the database in an inconsistent state.
Isolated: Transactions cannot interfere with each other.
Durable: Completed transactions persist, even when servers restart etc.

These qualities seem indispensable, and yet they are incompatible with availability and performance in very large systems. For example, suppose you run an online book store and you proudly display how many of each book you have in your inventory. Every time someone is in the process of buying a book, you lock part of the database until they finish so that all visitors around the world will see accurate inventory numbers. That works well if you run The Shop Around the Corner but not if you run Amazon.com.

Amazon might instead use cached data. Users would not see not the inventory count at this second, but what it was say an hour ago when the last snapshot was taken. Also, Amazon might violate the "I" in ACID by tolerating a small probability that simultaneous transactions could interfere with each other. For example, two customers might both believe that they just purchased the last copy of a certain book. The company might risk having to apologize to one of the two customers (and maybe compensate them with a gift card) rather than slowing down their site and irritating myriad other customers.

There is a computer science theorem that quantifies the inevitable trade-offs. Eric Brewer's CAP theorem says that if you want consistency, availability, and partition tolerance, you have to settle for two out of three. (For a distributed system, partition tolerance means the system will continue to work unless there is a total network failure. A few nodes can fail and the system keeps going.)

An alternative to ACID is **BASE:**

- Basic Availability
- Soft-state
- Eventual consistency

Rather than requiring consistency after every transaction, it is enough for the database to eventually be in a consistent state. (Accounting systems do this all the time. It's called "closing out the books.") It's OK to use stale data, and it's OK to give approximate answers.

It's harder to develop software in the fault-tolerant BASE world compared to the fastidious ACID world, but Brewer's CAP theorem says you have no choice if you want to scale up. However, as Brewer points out in this presentation, there is a continuum between ACID and BASE. You can decide how close you want to be to one end of the continuum or the other according to your priorities.

2.7 Distributed Systems: Middleware

In general a *middleware* is used to hide low level network problems:

- not easy to scale,

[1] `http://www.infoq.com/articles/cap-twelve-years-later-how-the-rules-have-changed`
[2] `http://www.johndcook.com/blog/2009/07/06/brewer-cap-theorem-base/`

ACID vs BASE

ACID	BASE
Strong consistency for transactions highest priority	Availability and scaling highest priorities
Availability less important	Weak consistency
Pessimistic	Optimistic
Rigorous analysis	Best effort
Complex mechanisms	Simple and fast

Figure 1: Acid vs. Base

- cumbersome and error prone to use,
- hard to maintain and change,
- without a middleware we do not have transparency of the distributed communication.

Some well known Middleware solutions are:

- DCOM
- CORBA
- RPC models (SUN/RPC, RMI, .NET WCF, Spring Remoting)
- JavaSpaces / Jini
- Enterprise Service Busses
- MOM
- etc.

2.8 Architectural Distributed Styles

We distinguish between sever different architectural distributed styles:

- Client-Server
- N-Tier
- Cluster (tightly coupled)
- Peer-to-Peer
- Space-based
- Distributed Caches / Storage
- SOA
- CQRS

2.9 Message-oriented Middleware (MOM)

Message-oriented middleware (MOM) is software or hardware infrastructure supporting sending and receiving messages between distributed systems. MOM allows application modules to be distributed over heterogeneous platforms and reduces the complexity of developing applications that span multiple operating systems and network protocols. The middleware creates a distributed communications layer that insulates the application developer from the details of the various operating system and network interfaces. APIs that extend across diverse platforms and networks are typically provided by MOM.
MOM provides software elements that reside in all communicating components of a client/server architecture and typically support asynchronous calls between the client and server applications. MOM reduces the involvement of application developers with the complexity of the master-slave nature of the client/server mechanism.

2.10 Service-oriented Architecture (SOA)

Service-oriented architecture (SOA) is a software design and software architecture design pattern based on structured collections of discrete software modules, known as services, that collectively provide the complete functionality of a large software application. The purpose of SOA is to allow easy cooperation of a large number of computers that are connected over a network. Every computer can run an arbitrary number of programs - called services in this context - that are built in a way that they can exchange information with any other service within the reach of the network without human interaction and without the need to make changes to the underlying program itself. In a large network of computers SOA has the same role and duties as the traditional operating system on a single computer. Consequently SOA is designed in analogy to traditional multi-tasking operating systems like Windows, Unix, zOS etc.

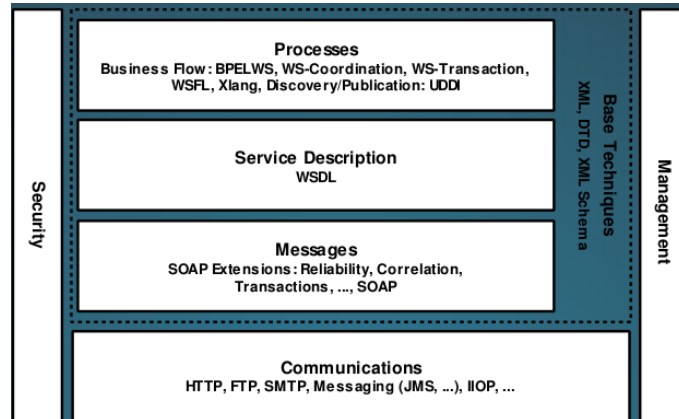

Figure 2: Web Service Stack

2.11 Transaction Processing Monitor

A transaction processing monitor (TPM) is a program that monitors transactions from one stage to the next, ensuring that each one completes successfully; if not, or if an error occurs, the TM Monitor takes the appropriate action. A transaction processing monitor's main purpose/objective is to allow resource sharing and assure optimal use of the resources by applications. This term is sometimes shortened to TP monitor.

2.12 Distributed Transactions

A distributed transaction can be seen as a database transaction that must be synchronized (or provide ACID properties) among multiple participating databases which are distributed among different physical locations.
A distributed transaction involves more than one transactional resource, such as a database, and is usually located on more than one machine.

Conventional RPC does not support the transactions. TP monitors, in contrast, allow developers to wrap a series of invocations as a transaction:

- Transactional invocations are realized typically by marking the beginning and the end of a transaction in the client code
- At the beginning of the transaction, the client contacts the TP monitor to acquire a transaction ID and context
- At the end of the transaction, the client notifies the TP monitor again
- The TP monitor then runs a commit protocol to determine the outcome of the transaction (2 phase commit, 2PC)

All transactional invocations must be marked as belonging to a specific transaction by using the transaction ID. This can be done by the invoker and requestor/client proxy. A more elegant way is to use **interceptors:**

- Invocation interceptors allow the transaction id to be added on the client side and read it on the server side transparently.
- The transaction id can be stored in an invocation context.

In a distributed transaction, a two-phase commit protocol (2PC) can be used for distributed transaction processing. [3]

Finally, the client is informed whether or not the transaction was successfully committed.

2.13 Server Components and Middleware

Component infrastructures are often built on top of distributed object middleware as an extension. Using invocation interceptors, invocation contexts, as well as suitable lifecycle managers, most of the functionality of component infrastructures can be built. Existing component infrastructures already provide useful default implementations of important, recurring features. If you need to build your own component infrastructure, starting with distributed object middleware as a basis is certainly a good idea. The current mainstream component infrastructures are built on top of distributed object middleware systems:

- EJB is based on Java RMI
- CORBA Components are based on CORBA
- COM+ uses DCOM

2.14 Other distributed Infrastructures

Peer-to-Peer systems are not based on a client/server or n-tier model, but on a set of equal peers. Peers communicate and coordinate themselves in fulfilling user services. Many P2P systems use remote objects internally for communication between peers.
Grid Computing is about sharing and aggregation of distributed resources such as processing time, storage, and information. A Grid consists of multiple computers linked to one system. Grid computing makes extensive use of remote communication, as the distributed resources need to be managed and computing requests and results need to be transmitted.
Code mobility paradigms extend the client/server paradigm:

- Remote evaluation. Code is sent to the server and executed in the server context using the data located at the server.
- Code on demand. Code is downloaded by a client and executed in the client context using data located at the client.
- Mobile agents. Code and associated data, and possibly also the execution state, migrates from host A to host B and executes in the context of host B.

3 Chapter 2

3.1 Three dimenstion of Engineering

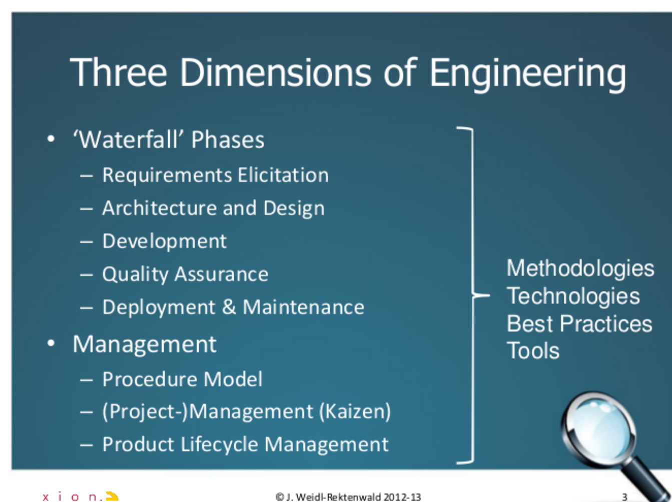

Figure 3: Three dimenstion of Engineering

[3]Two-phase commit guarantees the ACID properties and supports distributed synchronization of multiple transactional resources.

Software engineering is a rich, multi-faceted, and evolving field. It is often useful to think of it in three dimensions, each dimension being concerned with one particular aspect.
The first dimension contains all of the tools, techniques, methods, and processes required to develop software.
The second contains the management techniques required to organize software projects successfully, to monitor the effectiveness of the development, and to improve the development process.
The third addresses the way in which the non-functional attributes of the software being developed in achieved. Non-functional attributes refer not to what the software does (its function) but instead to the manner in which it does it (its dependability, security, composability, portability, interoperability ... these are sometimes referred to as the '-ilities').

3.2 Design Methods and Principles

3.2.1 Architecture/Design Methods

Architecture/Design Methods represent first idea of a system.

Big Design Up Front: The term BigDesignUpFront is commonly used to describe methods of software development where a "big" detailed design is created before coding and testing takes place. Several ExtremeProgramming (XP) advocates have said that such "big" designs are not necessary, and that most design should occur throughout the development process. While Xp does have initial design (the SystemMetaphor), it is considered to be a relatively "small" design. Much of this page disputes the amount of up-front design required for software projects. [4]

Top-Down / Bottom-Up: Top-down and bottom-up are both strategies of information processing and knowledge ordering, used in a variety of fields including software, humanistic and scientific theories (see systemics), and management and organization. In practice, they can be seen as a style of thinking and teaching.
A top-down approach (also known as stepwise design or deductive reasoning, and in many cases used as a synonym of analysis or decomposition) is essentially the breaking down of a system to gain insight into its compositional sub-systems. In a top-down approach an overview of the system is formulated, specifying but not detailing any first-level subsystems. Each subsystem is then refined in yet greater detail, sometimes in many additional subsystem levels, until the entire specification is reduced to base elements. A top-down model is often specified with the assistance of "black boxes", these make it easier to manipulate. However, black boxes may fail to elucidate elementary mechanisms or be detailed enough to realistically validate the model. Top down approach starts with the big picture. It breaks down from there into smaller segments.
A bottom-up approach (also known as inductive reasoning, and in many cases used as a synonym of synthesis) is the piecing together of systems to give rise to grander systems, thus making the original systems sub-systems of the emergent system. Bottom-up processing is a type of information processing based on incoming data from the environment to form a perception. Information enters the eyes in one direction (input), and is then turned into an image by the brain that can be interpreted and recognized as a perception (output). In a bottom-up approach the individual base elements of the system are first specified in great detail. These elements are then linked together to form larger subsystems, which then in turn are linked, sometimes in many levels, until a complete top-level system is formed. This strategy often resembles a "seed" model, whereby the beginnings are small but eventually grow in complexity and completeness. However, "organic strategies" may result in a tangle of elements and subsystems, developed in isolation and subject to local optimization as opposed to meeting a global purpose. [5]

Design by Contract: Design by contract (DbC), also known as contract programming, programming by contract and design-by-contract programming, is an approach for designing software. It prescribes that software designers should define formal, precise and verifiable interface specifications for software components, which extend the ordinary definition of abstract data types with preconditions, postconditions and invariants. These specifications are referred to as "contracts", in accordance with a conceptual metaphor with the conditions and obligations of business contracts. [6]

3.2.2 CCD Design Principles

- Separation of Concerns (SoC) - Concerns are the different aspects of software functionality. For instance, the "business logic" of software is a concern, and the interface through which a person uses this logic is another. The separation of concerns is keeping the code for each of these concerns separate. Changing the interface should not require changing the business logic code, and vice versa. (example MVC)

[4] http://c2.com/cgi/wiki?BigDesignUpFront
[5] http://en.wikipedia.org/wiki/Top-down_and_bottom-up_design
[6] https://en.wikipedia.org/wiki/Design_by_contract

- Inversion of Control (IoC) - If you follow these simple two steps, you have done inversion of control: (1) Separate what-to-do part from when-to-do part. (2) Ensure that when part knows as little as possible about what part; and vice versa.
- Don't repeat yourself (DRY) - reducing repetition of information of all kinds, especially useful in multi-tier architectures. The DRY principle is stated as "Every piece of knowledge must have a single, unambiguous, authoritative representation within a system." When the DRY principle is applied successfully, a modification of any single element of a system does not require a change in other logically unrelated elements. Additionally, elements that are logically related all change predictably and uniformly, and are thus kept in sync.
- Keep it simple, stupid (KISS) - states that most systems work best if they are kept simple rather than made complex, therefore simplicity should be a key goal in design and unnecessary complexity should be avoided.
- You Ain't Gonna Need It (YAGNI) - should not add functionality until deemed necessary. "Always implement things when you actually need them, never when you just foresee that you need them."
- Favour Composition over Inheritance (FCoI) -a technique by which classes may achieve polymorphic behavior and code reuse by containing other classes that implement the desired functionality instead of through inheritance.
- Single Level of Abstraction (SLA) - try to use only one level of abstraction in methods.
- Information Hiding (Coupling/Cohesion)

3.2.3 OO Design Principles

- **S** - Single responsibility principle - a class should have only a single responsibility.
- **O** - Open/closed principle -"software entities should be open for extension, but closed for modification".
- **L** - Liskov substitution principle - "objects in a program should be replaceable with instances of their subtypes without altering the correctness of that program". See also design by contract.
- **I** - Interface segregation principle - "many client-specific interfaces are better than one general-purpose interface."
- **D** - Dependency inversion principle - one should "Depend upon Abstractions. Do not depend upon concretions." Dependency injection is one method of following this principle.
- Common Reuse Principle - The classes in a package are reused together. If you reuse one of the classes in a package, you reuse them all.
- The Stable Dependencies Principle - The dependencies between packages should be in the direction of the stability of the packages. A package should only depend upon packages that are more stable than it is.
- Law of Demeter - "Only talk to your immediate friends." E.g. one never calls a method on an object you got from another call nor on a global object. This helps a lot later when you refactor the code.

3.3 DS design following rational OOAD

Design is a *step closer to source code.* The outcome of design and analysis are design models architecture documents and data models. The designer must know use-case modeling techniques, system requirements and software design techniques.

3.3.1 Design: Tasks to complete

- Identification of interfaces
- Identification and design of subsystems
- Identification and design of design classes
- Modeling design and implementation mechanismus
- Modeling nonfunctional requirements (e.g. concurrency and distribution)

3.3.2 Analysis vs Design

What is analysis? Is there an accepted definition? If you read through the software literature and textbooks, you will find that there are as many definitions of analysis, as there are authors. Ironically, analysis is something we know we must do; and yet have no real definition for.

One of the most common ways to differentiate analysis from design is to say that analysis is ”what” and design is ”how”. This sounds compelling at first. Clearly if we can first define ”what” we want the system to do, then it will be easier to define ”how” the system should do it. Indeed, vast quantities of man-hours have been spent attempting to separate the ”what” from the ”how”. It is not uncommon for a conference room full of people to descend into the interminable argument over whether what we are currently doing is analysis or design.

Arguments like this are common because of the peculiar fact that every ”what” is also a ”how”, and every ”how” is also a ”what”. The arguments cannot be decided, because both sides are actually correct. If we use the ”what” vs. ”how” definition of analysis and design, then every concept that is an analysis concept is also a design concept and vice versa.

To demonstrate that every ”what” is a ”how” and vice versa, consider what I am currently doing.

What am I doing? I am writing a blog.
How am I doing it? I am typing on my laptop.

What am I doing? I am typing on my laptop.
How am I doing it? I am moving my fingers to hit keys that correspond to letters that are assembled into words.

What am I doing? I am moving my fingers to hit keys that correspond to letters that are assembled into words.
How am I doing it? The cognitive portions of my brain are putting words together and directing the motor portions of my brain to send signals to the muscles that control my fingers.

I could go on indefinitely. The fact that I could go on indefinitely means that ”what” and ”how” are associated by an equivalence relationship that leads to a recursive descent into detail. Every ”how” become the ”what” at the next level of recursion. Every ”what” is just the ”how” of the level above it. The number of levels is very large, and probably infinite. This means that ”what” and ”how” are separated by an infinitesimal and are virtually identical.

So, for all practical purposes, if we view analysis as ”what” and design as ”how” then they are equivalent operations that cannot be separated from each other. All analysis is design, and all design is analysis.

Analysis	Design
Focuses on understanding the problem	Focuses on understanding the solution
Idealized design	Close to real code
Behavior	Operations and attributes
Functional requirements	Nonfunctional requirements
A small model	A large model

Figure 4: Analysis vs. Design

3.4 Modeling distribution and Parallelism

Architectures are often described by **views**.

3.4.1 Modeling Distribution

- Component View
  - Static Structure
  - UML: Package-, Component-, Class-Diagram
- Runtime View
  - Dynamic behaviour
  - UML. Activity-, Sequence, Communication-, Interaction-, Timing-Diagram, State charts.
- Deployment View

- Shows (distributed) nodes and connections
- UML: Deployment diagram

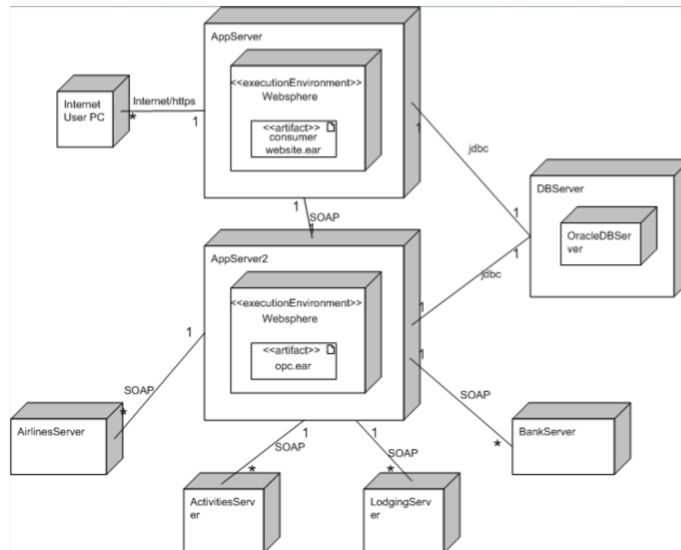

Figure 5: Deployment View

3.4.2 Modeling Parallelism

Parallelism can be modelled with UML using Class diagrams ("Active classes"), Activity Diagrams (Fork Symbols splits control flow), Sequence Diagrams, and State Diagrams.

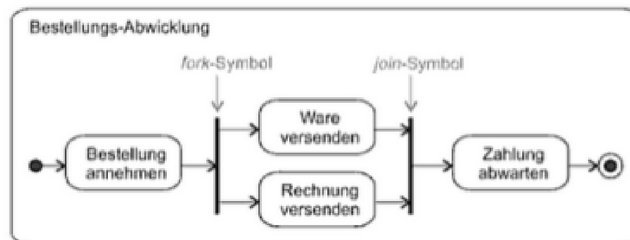

Figure 6: Activity Diagram

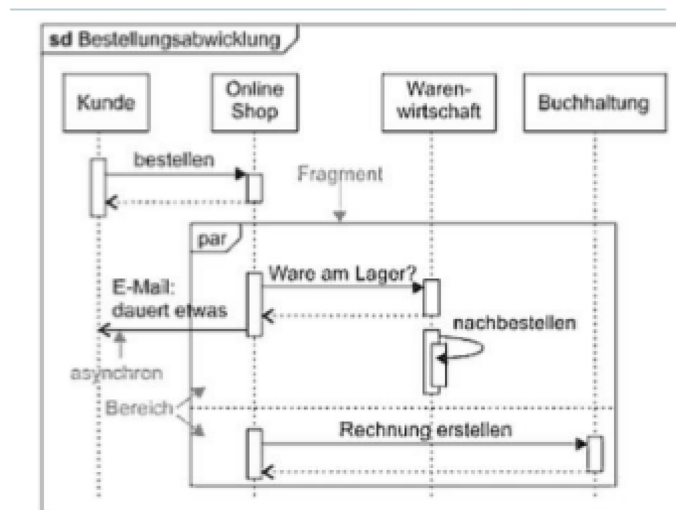

Figure 7: Sequence Diagram

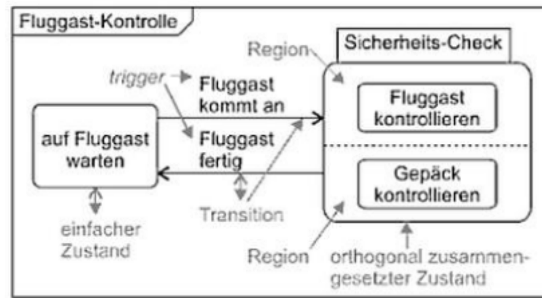

Figure 8: State Diagram

3.5 Remoting Styles

A number of different remoting styles are used in today's middleware systems There are systems that:

- use the metaphor of a remote procedure call (RPC),
- use the metaphor of posting and receiving messages,
- utilize a shared repository, or
- use continuous streams of data.

3.5.1 Remoting Style: RPC

In computer science, a remote procedure call (RPC) is an inter-process communication that allows a computer program to cause a subroutine or procedure to execute in another address space (commonly on another computer on a shared network) without the programmer explicitly coding the details for this remote interaction. That is, the programmer writes essentially the same code whether the subroutine is local to the executing program, or remote. When the software in question uses object-oriented principles, RPC is called remote invocation or remote method invocation.

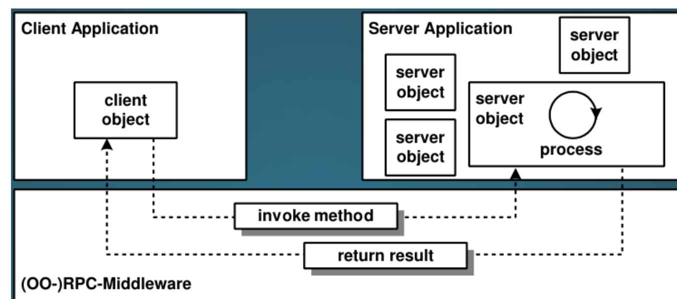

Figure 9: Remote Procedure Call

3.5.2 Remoting Style: Messaging

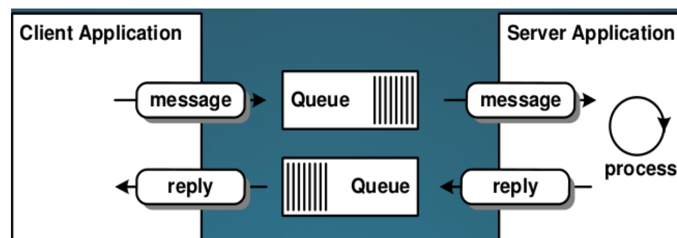

Figure 10: Messaging

3.5.3 Remoting Style: Shared Repository

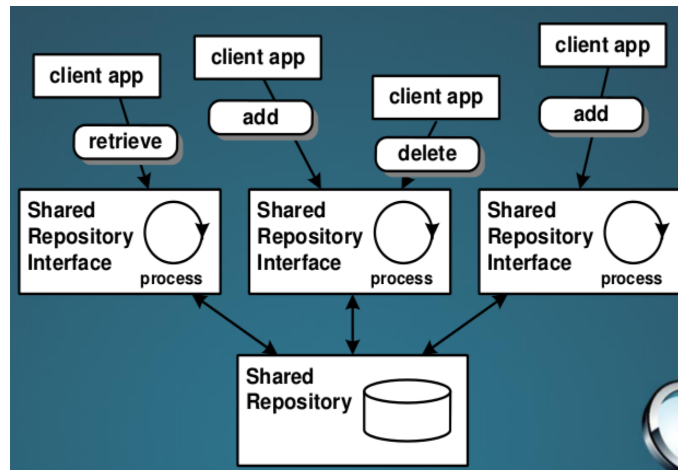

Figure 11: Shared Repository

3.5.4 Remoting Style: Streaming

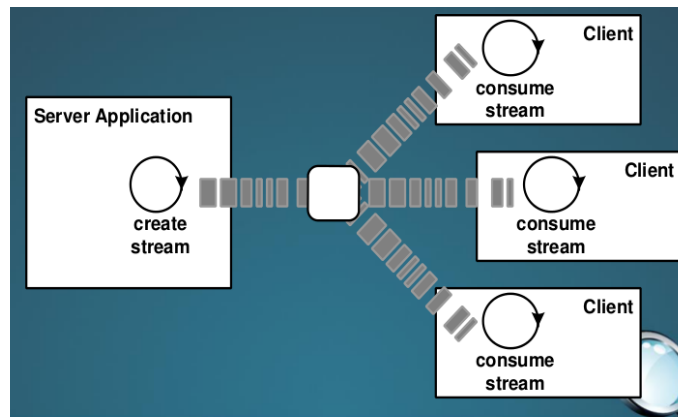

Figure 12: Streaming

3.6 Remoting Patterns

The goal of remoting patterns is to illustrate the general, recurring architecture of successful distributed object middleware and to illustrate more concrete design and implementation strategies.

3.6.1 Software Patterns

In software engineering, a design pattern is a general reusable solution to a commonly occurring problem within a given context in software design. A design pattern is not a finished design that can be transformed directly into source or machine code. It is a description or template for how to solve a problem that can be used in many different situations. Patterns are formalized best practices that the programmer must implement themselves in the application.[1] Object-oriented design patterns typically show relationships and interactions between classes or objects, without specifying the final application classes or objects that are involved. Patterns that imply object-orientation or more generally mutable state, are not as applicable in functional programming languages.

Different kinds of patterns:

- Design patterns (GoF)
- Software architecture patterns (POSA, POSA2)
- Analysis patterns (Fowler, Hay)

- Organizational patterns (Coplien)
- Pedagogical patterns (PPP)
- Many others

3.6.2 Remoting Patterns

Remoting Patterns should illustrate the general, recurring architecture of successful distributed object middleware and illustrate more concrete design and implementation strategies.

3.6.3 Basic Remoting Patterns

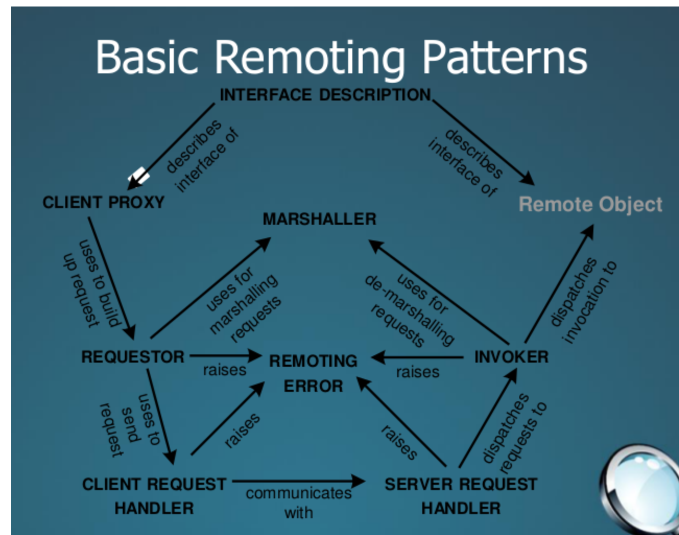

Figure 13: Basic Remoting Patterns

The pattern BROKER is, from our perspective, a compound pattern that is typically implemented using a number of patterns from the Remoting Pattern language. The BROKER pattern addresses the problem that distributed software system developers face many challenges that do not arise in single-process software. One main challenge is the unreliable communication across networks. Other challenges are the integration of heterogeneous components into coherent applications, as well as the efficient usage of networking resources. If developers of distributed systems must master all these challenges within their application code, they might loose their primary focus, to develop distributed applications that solve the application problems well. The BROKER pattern separates the communication functionality of a distributed system from its application functionality by isolating all communication related concerns in a BROKER. A BROKER hides and mediates all communication between the objects or components of a system. Local BROKERS on client side and server side enable the exchange of requests and responses between the client and the remote object. A BROKER consists of a client-side REQUESTOR to construct and forward invocations, as well as a server-side INVOKER that is responsible for invoking the operations of the target remote object. A MARSHALLER on each side of the communications path handles the transformation of requests and responses - from programming-language native data types into a byte array that can be sent over the wire.

In addition to the core patterns consisting of REQUESTOR, INVOKER, and MARSHALLER, the BROKER typically relies on the following patterns:

- A CLIENT PROXY is a placeholder for the remote object in the client process, offering the same interface as the remote object. A CLIENT PROXY lets remote operation invocations look like local operation invocations from a client's perspective. Internally, the CLIENT PROXY transforms invocations of its operations into REQUESTOR invocations. The REQUESTOR is then responsible for constructing the request and for forwarding it to the target remote object.
- An INTERFACE DESCRIPTION is used to make the remote object's interface known to the clients. Thus the INTERFACE DESCRIPTION can be used to construct a CLIENT PROXY for a particular remote object type.
- The CLIENT REQUEST HANDLER and SERVER REQUEST HANDLER handle efficient sending, receiving, and dispatching of requests. These two patterns act at the layer beneath the REQUESTOR and the INVOKER. The request handlers forward and receive request and response messages from and to the REQUESTOR and the INVOKER, respectively.

- A remote invocation introduce new kinds of errors, compared to local invocations, as for instance, technical failures in the network communication infrastructure or problems within the server infrastructure. REMOTING ERRORS are used to signal these new errors types to the client side. The REQUESTOR and INVOKER are responsible for forwarding REMOTING ERRORS to the client, if they cannot handle the REMOTING ERROR on their own.

3.7 Advanced Patterns: CQRS and APP-Level Mod.

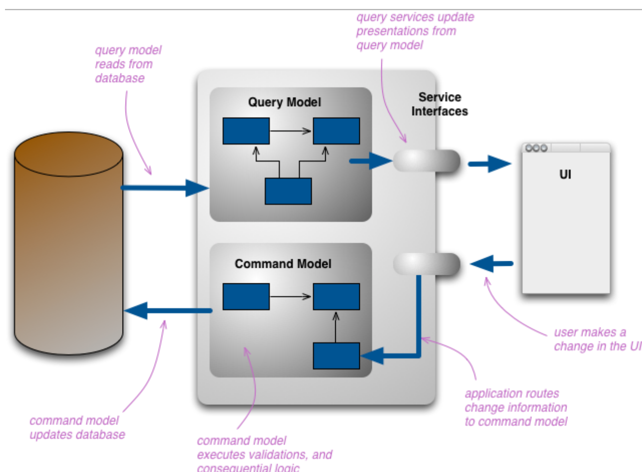

Figure 14: CQRS

3.7.1 CQRS - Command Query Responibility Segregation

Most applications read data more frequently than they write data. The CQRS pattern seperates rich domain models from view models.

Definition: Command Query Responsibility Segregation or CQRS is the creation of two objects where there was previously one. The separation occurs based upon whether the methods are a command or a query.

Higher-Level Definition: Command Query Responsibility Segregation or CQRS is the recognition that there are differing architectural properties when looking at the paths for reads and writes of a system. CQRS allows the specialization of the paths to better provide an optimal solution.

You should use the CQRS Pattern if the processing of commands and queries is fundamentally asymmetrical, and scaling the services symmetrically does not make a lot of sense.

What is CQRS? [7]

CQRS stands for Command Query Responsibility Segregation. It's a pattern that I first heard described by Greg Young. At its heart is a simple notion that you can use a different model to update information than the model you use to read information. This simple notion leads to some profound consequences for the design of information systems.

The mainstream approach people use for interacting with an information system is to treat it as a CRUD datastore. By this I mean that we have mental model of some record structure where we can create new records, read records, update existing records, and delete records when we're done with them. In the simplest case, our interactions are all about storing and retrieving these records.

As our needs become more sophisticated we steadily move away from that model. We may want to look at the information in a different way to the record store, perhaps collapsing multiple records into one, or forming virtual records by combining information for different places. On the update side we may find validation rules that only allow certain combinations of data to be stored, or may even infer data to be stored that's different from that we provide.

[7] `http://martinfowler.com/bliki/CQRS.html`

As this occurs we begin to see multiple representations of information. When users interact with the information they use various presentations of this information, each of which is a different representation. Developers typically build their own conceptual model which they use to manipulate the core elements of the model. If you're using a Domain Model, then this is usually the conceptual representation of the domain. You typically also make the persistent storage as close to the conceptual model as you can.

This structure of multiple layers of representation can get quite complicated, but when people do this they still resolve it down to a single conceptual representation which acts as a conceptual integration point between all the presentations.

The change that CQRS introduces is to split that conceptual model into separate models for update and display, which it refers to as Command and Query respectively following the vocabulary of CommandQuerySeparation. The rationale is that for many problems, particularly in more complicated domains, having the same conceptual model for commands and queries leads to a more complex model that does neither well.

By separate models we most commonly mean different object models, probably running in different logical processes, perhaps on separate hardware. A web example would see a user looking at a web page that's rendered using the query model. If they initiate a change that change is routed to the separate command model for processing, the resulting change is communicated to the query model to render the updated state.

There's room for considerable variation here. The in-memory models may share the same database, in which case the database acts as the communication between the two models. However they may also use separate databases, effectively making the query-side's database by a real-time ReportingDatabase. In this case there needs to be some communication mechanism between the two models or their databases.

The two models might not be separate object models, it could be that the same objects have different interfaces for their command side and their query side, rather like views in relational databases. But usually when I hear of CQRS, they are clearly separate models.

CQRS naturally fits with some other architectural patterns.

- As we move away from a single representation that we interact with via CRUD, we can easily move to a task-based UI.
- Interacting with the command-model naturally falls into commands or events, which meshes well with Event Sourcing.
- Having separate models raises questions about how hard to keep those models consistent, which raises the likelihood of using eventual consistency.
- For many domains, much of the logic is needed when you're updating, so it may make sense to use Eager-ReadDerivation to simplify your query-side models.
- CQRS is suited to complex domains, the kind that also benefit from Domain-Driven Design.

What is Event Sourcing? Event sourcing means that you capture changes to an application state as a sequence of events (definition by Fowler) . Imagine most systems your worked with up to now, If you had a customer, and they moved, you would probably go to the user interface (web or whatever), go to some sort of customer edit screen, change the details, and thats it, most likely this would trigger an update operation in a normalized database, and unless there was a specific requirement, you've not only lost the previous value of the address, also, you dont know if the customer actually moved or if the address was incorrect, so you are loosing history and intent, two very valuable assets.

An important detail about this way of storing data is that its an append only model , because you are only adding, the write operation its simpler, this means it takes less time. Another consequence of an append only model is that to change something, we would apply compensating actions. Interestingly, once the events are stored they are immutable.

If this is such a great way to store data then how come we are not using it everywhere? you might ask. Well there is the problem of having to replay all events to build an aggregate root each time you need to work on it, and hey, what if it's a very busy one, and there are millions of events to replay. For that, we would use snapshots, i.e. every so often we would record the current state of the aggregate root as is and then when we need to get the aggregate root, we would have to find the latest snapshot, and then replay any outstanding events to get to up to date. The real culprit is the fact that selecting the top 3 orders where the customer name is Bob, is kind of hard. However, since we are using Event sourcing on the write side and we might just get away with it .

Event Sourcing is a big theme, the Event store is one of the most complex parts of using CQRS/ES. There are a few open source implementations out there. Olivers comes to mind

3.7.2 Application-level Modulatisation

The idea is to use design components as single deployment units. This is a prerequisite for elastic scalability in cloud environments. This facilitates the localisation of errors. "Always-on" deployments.

3.7.3 Domain Driven Design

Domain-driven design is not a technology or a methodology. DDD provides a structure of practices and terminology for making design decisions that focus and accelerate software projects dealing with complicated domains. In DDD logic is moved to the domain model (smart domain logic). An important part of DDD is the so called ubiquitous language; i.e. speak the same language as the business experts. And make your code / architecture so that it reflects this language to avoid impedance problems. DDD can be seen as an Object-orientated analysis and design (OOAD) method and constists of

- Architectural techniques
- Design techniques
- Procedures

Elements of the Domain Layer

- Entities
- Value Objects
- Aggregates
- Associations
- Services
- Domain Events
- Modules
- Factories
- Repositories

4 Chapter 3

4.1 Cloud Computing

4.1.1 Cloud Computing - Definition

Cloud computing refers to the delivery of computing and storage capacity as a service to a heterogeneous community of end-recipients. The name comes from the use of clouds as an abstraction for the complex infrastructure it contains in system diagrams.

Public Cloud

A cloud is called a 'Public cloud' when the services are rendered over a network that is open for public use. Technically there is no difference between public and private cloud architecture, however, security consideration may be substantially different for services (applications, storage, and other resources) that are made available by a service provider for a public audience and when communication is effected over a non-trusted network. Generally, public cloud service providers like Amazon AWS, Microsoft and Google own and operate the infrastructure and offer access only via Internet (direct connectivity is not offered).

Private Cloud

Private cloud is cloud infrastructure operated solely for a single organization, whether managed internally or by a third-party and hosted internally or externally. Undertaking a private cloud project requires a significant level and degree of engagement to virtualize t he business environment, and requires the organization to reevaluate decisions about existing resources. When done right, it can improve business, but every step in the project raises security issues that must be addressed to prevent serious vulnerabilities.

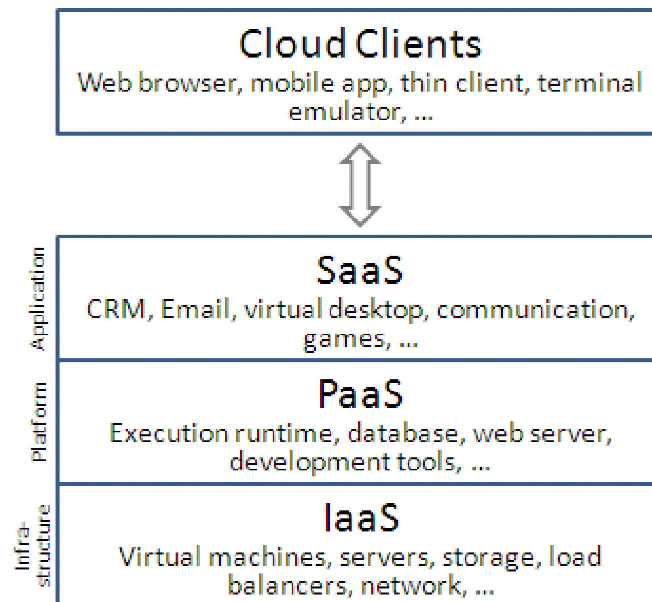

Figure 15: Cloud Computing

Hybrid Cloud

Hybrid cloud is a composition of two or more clouds (private, community or public) that remain unique entities but are bound together, offering the benefits of multiple deployment models. Such composition expands deployment options for cloud services, allowing IT organizations to use public cloud computing resources to meet temporary needs. This capability enables hybrid clouds to employ cloud bursting for scaling across clouds.

4.1.2 Why Cloud Computing?

- CapEx (Investitionskosten) to OpExGet (Betriebskosten) and better resource utilization
- Flexibility (Self-service, Elastic Scalability)
- Easier to maintain / run (Homogeneous environments, auto-updated)

4.1.3 Benefits of cloud computing

Decreased Costs: The Cloud eliminates the need for each user to invest in stand-alone servers or software that is capital intensive, but under-utilized most of the time. As technological innovations take place, these resources become obsolete and must be replaced with the latest in order to ensure operational efficiency – requiring more capital investment – and the cycle repeats. The Cloud eliminates the need for such 'replacement' capital expenditure. Many users share a Cloud leading to distributed costs and economies of scale as resources including real estate, bandwidth, and power, are centralized. The enterprise also saves on overheads such as management costs, data storage costs, costs of software updates, and quality control and is able to use Cloud services at economical rates.

Scalability and Speed: Enterprises no longer have to invest time in buying and setting up the hardware, software and other resources necessary for a new application. They can quickly scale up or scale down their usage of services on the Cloud as per market demands, during hours of maximum activity, while launching sales campaigns, etc. Cloud services are most usually reliable, since many service providers have data centers in multiple locations for keeping the processing near users.

Innovation: Enterprises can focus on innovation, as they do not have to own or manage resources. Cloud computing facilitates faster prototype development, testing and validation. Research and development projects or activities where users have to collaborate for a task/project are especially benefited.

Convenience: Sharing of infrastructure and costs ensures low overheads and immediate availability of services. Payments are billed on the basis of actual consumption only. Details of billing are made available by the service

provider also serves to check costs.

Other than an Internet-connected device, special equipment or specially-trained manpower is not needed. One-off tasks can be performed on the Cloud. High-speed bandwidth ensures real-time response from infrastructure located at different sites.

Location Independence: Service providers can set up infrastructure in areas with lower overheads and pass on the benefit. They can set up multiple redundant sites to facilitate business continuity and disaster recovery. This helps the enterprise cut costs further.

Optimal Resource Utilization: Servers, storage and network resources are better utilized as the Cloud is shared by multiple users, thus cutting down on waste at a global level. Cloud computing is more environment-friendly and energy efficient. Down-time is cut and optimization of resources across enterprises on the Cloud is achieved.

Flexibility: Users can opt out at will and thus gain a high level of operational flexibility. The services are covered by service level agreements and the service provider is required to pay a penalty if the quality agreed to is not provided.

Device Independence: Applications provided through the Cloud can be accessed from any device – a computer, a smartphone, an iPad, etc. Any device that has access to the Internet can leverage the power of the Cloud.

4.1.4 Infrastructure as a Service (IaaS)

Idea: "Configure your Datacenter via a browser."

In the most basic cloud-service model, providers of IaaS offer computers - physical or (more often) virtual machines - and other resources. (A hypervisor, such as Xen or KVM, runs the virtual machines as guests. Pools of hypervisors within the cloud operational support-system can support large numbers of virtual machines and the ability to scale services up and down according to customers' varying requirements.) IaaS clouds often offer additional resources such as a virtual-machine disk image library, raw (block) and file-based storage, firewalls, load balancers, IP addresses, virtual local area networks (VLANs), and software bundles. IaaS-cloud providers supply these resources on-demand from their large pools installed in data centers. For wide-area connectivity, customers can use either the Internet or carrier clouds (dedicated virtual private networks). To deploy their applications, cloud users install operating-system images and their application software on the cloud infrastructure. In this model, the cloud user patches and maintains the operating systems and the application software. Cloud providers typically bill IaaS services on a utility computing basis: cost reflects the amount of resources allocated and consumed. Examples of IaaS providers include: Amazon EC2, AirVM, Azure Services Platform, DynDNS, Google Compute Engine, HP Cloud, iland, Joyent, LeaseWeb, Linode, NaviSite, Oracle Infrastructure as a Service, Rackspace, ReadySpace Cloud Services, ReliaCloud, SAVVIS, SingleHop, and Terremark Cloud communications and cloud telephony, rather than replacing local computing infrastructure, replace local telecommunications infrastructure with Voice over IP and other off-site Internet services. [8]

4.1.5 Plaform as a Service (PaaS)

Idea: "Deploy into an standardized, automated and self-maintaining application run-time"

In the PaaS model, cloud providers deliver a computing platform typically including operating system, programming language execution environment, database, and web server. Application developers can develop and run their software solutions on a cloud platform without the cost and complexity of buying and managing the underlying hardware and software layers. With some PaaS offers, the underlying computer and storage resources scale automatically to match application demand such that cloud user does not have to allocate resources manually. Examples of PaaS include: AWS Elastic Beanstalk, Cloud Foundry, Heroku, Force.com, EngineYard, Mendix, OpenShift, Google App Engine, AppScale, Windows Azure Cloud Services and OrangeScape. [9]

[8] http://en.wikipedia.org/wiki/Cloud_computing
[9] http://en.wikipedia.org/wiki/Cloud_computing

Why PaaS

The "Old School" model: Peak loads are evil, because we cannot scale easily. RDBMS scale-up - not out. (Can we afford it?) Run-time gets heterogeneous with time and no one can stop it. Consistency rules! Sacrificing consistency is "no-way"!

With PaaS:

- Handling Peak-Loads is built-in (elastic scalability, pay per use)
- Big/fast data via scaling out and map / reduce (NoSQL)
- Highlander Pinciple - "There only can be one" - and it magically stays young (aka auto-updated)
- Consistency is weakened, availability rules - you can trade it

Drawbacks

- Runtime is "as-is"
  - We cannot workaround it
  - We cannot debug it
  - We cannot fix it
- Programming models changes
  - Mind-set change necessary
  - Existing software could be hard to port
  - We need to educate developers (costly)

4.1.6 Software as a Service (SaaS)

Idea: "Applications not important any more – we see the world as services accessible anywhere by anyone"

In the business model using software as a service (SaaS), users are provided access to application software and databases. Cloud providers manage the infrastructure and platforms that run the applications. SaaS is sometimes referred to as "on-demand software" and is usually priced on a pay-per-use basis. SaaS providers generally price applications using a subscription fee. In the SaaS model, cloud providers install and operate application software in the cloud and cloud users access the software from cloud clients. Cloud users do not manage the cloud infrastructure and platform where the application runs. This eliminates the need to install and run the application on the cloud user's own computers, which simplifies maintenance and support. Cloud applications are different from other applications in their scalability—which can be achieved by cloning tasks onto multiple virtual machines at run-time to meet changing work demand. Load balancers distribute the work over the set of virtual machines. This process is transparent to the cloud user, who sees only a single access point. To accommodate a large number of cloud users, cloud applications can be multitenant, that is, any machine serves more than one cloud user organization. It is common to refer to special types of cloud based application software with a similar naming convention: desktop as a service, business process as a service, test environment as a service, communication as a service. The pricing model for SaaS applications is typically a monthly or yearly flat fee per user, so price is scalable and adjustable if users are added or removed at any point. Examples of SaaS include: Google Apps, Microsoft Office 365, Petrosoft, Onlive, GT Nexus, Marketo, Casengo, TradeCard and CallidusCloud. Proponents claim SaaS allows a business the potential to reduce IT operational costs by outsourcing hardware and software maintenance and support to the cloud provider. This enables the business to reallocate IT operations costs away from hardware/software spending and personnel expenses, towards meeting other goals. In addition, with applications hosted centrally, updates can be released without the need for users to install new software. One drawback of SaaS is that the users' data are stored on the cloud provider's server. As a result, there could be unauthorized access to the data. [10]

4.1.7 Public PaaS offerings

4.1.8 Private PaaS offerings

5 Chapter 4

5.1 Software Business undergoes Change

- New Frameworks

[10] http://en.wikipedia.org/wiki/Cloud_computing

  - Spring, Grails, Rails, Node.js etc.

- New application granularity
  - Smart Device Apps, In-Browser Apps, Browser OSes
  - "One app per task" concept
- Big Data, fast data, unstructured data
  - NoSQL, In-memory DBs
- Cloud infrastructures
  - On-tht-Fly provisioning of VMs, transparent failover, Dynamic resource scheduling, etc.

It seems that infrastructure innovation drives software business innovation again. One has to scale or die, be simple/fancy or die, adopt big/fast or die and be flexible or die.

5.2 Creating a new System of the world

5.2.1 Conways Law

When a team makes a product the product ends up resembling the team that made it - e.g. 3 person teams does a 3-pass compiler.

Example: Consider a large system that a government wants to build. The government hires a company to build the system. Say the company has three engineering groups, E1, E2, and E3, that participate in the project. Conway's law suggests that it is likely that the resultant system will consist of 3 major subsystems (S1, S2, S3), each built by one of the engineering groups. More importantly, the resultant interfaces between the subsystems (S1-S2, S1-S3, etc.) will reflect the quality and nature of the real-world interpersonal communications between the respective engineering groups (E1-E2, E1-E3, etc.).

5.3 Engineering for PaaS

Is the PaaS approach a solution for the software crisis [11], or a "more of the same" medicine?

Being ready for the PaaS age means:

- knowing the software engineering basics
- knowing the distribution basics (CAP, scaling, fault tolerance, etc.)
- knowing the PaaS principles and charactaristics (Also non technical: vondor-lock-in, prices, SLAs, Data trafic routes)

In Engineering for PaaS new Patterns apply:

Continuous delivery

Continuous Delivery (CD) is a pattern language used in software development to automate and improve the process of software delivery. Techniques such as automated testing, continuous integration and continuous deployment allow software to be developed to a high standard and easily packaged and deployed to test environments, resulting in the ability to rapidly, reliably and repeatedly push out enhancements and bug fixes to customers at low risk and with minimal manual overhead. The technique was one of the assumptions of extreme programming but at an enterprise level has developed into a discipline of its own, with job descriptions for roles such as "buildmaster" calling for CD skills as mandatory.

DevOps

DevOps (a portmanteau of development and operations) is a software development method that stresses communication, collaboration and integration between software developers and information technology (IT) professionals.[1] DevOps is a response to the interdependence of software development and IT operations. It aims to help an organization rapidly produce software products and services. Companies with very frequent releases may require a DevOps awareness or orientation. Flickr developed DevOps capability to support a business requirement of ten deployments per day;[7] this daily deployment cycle would be much higher at organizations producing multi-focus

[11] Software crisis was a term used in the early days of computing science. The term was used to describe the impact of rapid increases in computer power and the complexity of the problems that could be tackled. In essence, it refers to the difficulty of writing correct, understandable, and verifiable computer programs. The roots of the software crisis are complexity, expectations, and change

or multi-function applications. This is referred to as continuous deployment[8] or continuous delivery [9] and is frequently associated with the lean startup methodology.[10] Working groups, professional associations and blogs have formed on the topic since 2009. DevOps aids in release management for a company by standardizing development environments. Events can be more easily tracked as well as resolving documented process control and granular reporting issues. Companies with release/deployment automation problems usually have existing automation but want to more flexibly manage and drive this automation - without needing to enter everything manually at the command-line. Ideally, this automation can be invoked by non-operations resources in specific non-production environments. Developers are given more environment control, giving infrastructure more application-centric understanding. Simple processes become clearly articulating, simple processes under DevOps. The goal is to automate as much as possible different operational processes. DevOps integration targets product delivery, quality testing, feature development and maintenance releases in order to improve reliability and security and faster development and deployment cycles. Many of the ideas (and people) involved in DevOps came from the Enterprise Systems Management and Agile software development movements.

Big Data

Big data is a buzzword, or catch-phrase, used to describe a massive volume of both structured and unstructured data that is so large that it's difficult to process using traditional database and software techniques. While the term may seem to reference the volume of data, that isn't always the case. The term big data – especially when used by vendors – may refer to the technology (which includes tools and processes) that an organization requires to handle the large amounts of data and storage facilities. The term big data is believed to have originated with Web search companies who had to query very large distributed aggregations of loosely-structured data.

Fast Data

"Fast Data" is a term coined by computer/social science expert John Furrier. Fast Data is a 'cousin' of so-called Big Data and implies the ability to make near real-time decisions and enable orders of magnitude improvements in elapsed time to decisions for businesses.

Scalability vs. Elasticity

Scalability is the ability of the system to accomodate larger loads jut by adding resources either making hardware stronger (scale up) or adding additional nodes (scale out).

Elasticity is the ability to fit the resources needed to cope with loads dynamically usually in relation to scale out. So that when load increase you scale by adding more resources and when demand wanes you shrink back and remove unneeded resources. Elasticity is mostly important in Cloud environment where you pay-per-use and don't want to pay for resources you do not currently need on the one hand, and want to meet rising demand when needed on the other hand.

5.4 Application Life-Cycle Automation

Automation increases speed and preserves quality. Continuous build, test and even delivery is the key to improved time-to-market. The goal is to automate manual tasks as far as possible. The Maintainance effort of automation infrastructure has to be minimized.

5.5 Cloud Foundry

5.5.1 Architecture

- Kernel (Core PaaS system)
- Kernel and Orchestrateor Shell (Layer on top of IaaS)
- Orchestrator (IaaS creation, management, and orchestration)

6 Questions

Beschreiben Sie Herausforderungen, die beim Bau von VSys entstehen können

- Network Latency
- Predictability
- Concurrency
- Scalability
- Partial Failure

Was sind die Bestandteile eines Deployment-Diagramms?

A deployment diagram in the Unified Modeling Language models the physical deployment of artifacts on nodes. To describe a web site, for example, a deployment diagram would show what hardware components ("nodes") exist (e.g., a web server, an application server, and a database server), what software components ("artifacts") run on each node (e.g., web application, database), and how the different pieces are connected (e.g. JDBC, REST, RMI). The nodes appear as boxes, and the artifacts allocated to each node appear as rectangles within the boxes. Nodes may have subnodes, which appear as nested boxes. A single node in a deployment diagram may conceptually represent multiple physical nodes, such as a cluster of database servers. There are two types of Nodes.

- Device Node
- Execution Environment Node

Device nodes are physically computing resources with processing memory and services to execute software, such as typical computer or mobile phones. An execution environment node (EEN) is a software computing resource that runs within an outer node and which itself provides a service to host and execute other executable software elements.

Vorteile von PaaS (eine bestimmte Anzahl nennen)

- handling peak loads is built in (Elastic scalability , pay-per-use)
- NoSQL scales out (Big/fast data scaling out and map/reduce
- Highlander principle - "there can only be one" and it magically stays young (auto-updated)
- Consistency is weakened, availability rules

Vorteile von Cloud-Computing (bestimmte Anzahl, Achtung: ist nicht unbedingt dasselbe wie PaaS)

- Get better resource utilization
- Flexibility (Self-service, elastic scalability)
- Easier to maintain / run (Homogenous environment, Auto-updated)

Irgendeine Frage, wo man anhand eines Praxisbeispiels das CAP-Theorem anwenden musste

Was ist das Ergebnis eines Designs? (Das war die "angekündigte" Frage. Hier konnte man etwas ausholen. Akzeptiert wurde relativ viel von Aufzählungen, Pseudo-Code, UML-Diagramme über generellere Beschreibungen. Definitv falsche Antwort: Ausführbarer Code.)

- Identification of interfaces
- Identification and design of subsystems
- Identification and design of design classes
- Modeling design and implementation mechanismus
- Modeling nonfunctional requirements (e.g.concurrency and distribution)