# CAP

* **C**onsistency, **A**vailability, **P**artitioning tolerance (== Clustering)
* See on [Wikipedia](http://en.wikipedia.org/wiki/CAP_theorem)
* Proven: you can only have 2 at once
* Recommendation: don’t cluster, if there is no real need
* HTTP-Session and EJB-Session is not consistent by default in Application servers
* Nearly no distributed system (e.g. Amazon, Facebook, Banks, others) are consistent – instead they are “eventually consistent”

# FLP

* Asynchronous communication is not always the answer
  + Error handling too complex
  + When waiting for an answer – how long do you wait?
* See on [Wikipedia](http://en.wikipedia.org/wiki/Consensus_(computer_science)#Solvability_results_for_some_agreement_problems)

# Hardware costs

* Hardware does not cost so much any longer
* Use virtualization combined with automization
* New possibilities regarding deployment and testing
* No modularization effort in Java necessary any longer (just deploy everything – but keep deployments small)
* In one word: **private clouds**

# 24x7

* Check if it is really needed
* Reason: reaching real 24x7 greatly involves complexity

# BASE vs. ACID

* **A**tomicity, **C**onsistency, **I**solation, **D**urability
  + CA in CAP Theorem
  + RDBMS cannot scale
* **B**asically **A**vailable **S**oft-state services with **E**ventual-consistency
  + AP in CAP Theorem
  + NoSQL can scale
* Notes on NoSQL databases
  + Writes are fast – but data is distributed async
  + You can achieve consistent reads, if you have time
  + Consistency level can by set by operation (fast or consistent)

# Review on old J2EE Patterns

* ServiceLocator
  + What it did
    - JNDI-Lookup
    - PortableRemoteObject.narrow()
  + Now
    - @Inject and @Resource
* Business Delegate
  + What it did
    - Provide a nice facade for EJBs
  + Now
    - EJBs are POJOs
* Session Facade:
  + What it did
    - Managed relations between CMPs
    - Created DTOs for Entities
  + Now
    - Relations between JPA entities
    - No more DTOs necessary (entities are POJOs)
* Composite entity:
  + What it did
    - Needed for simulating relations
    - Could carry logic
  + Now
    - JPA entities are normal objects
* Domain store:
  + == JPA EntityManager
* Data access object:
  + No more need
  + Just encapsulates the EntityManager
  + Regarding the BCE approach, a DAO is a control – but shouldn’t be named “DAO” any longer (this implies too much)
* WebserviceBroker:
  + What it did
    - Provide access to SOAP services
  + Now
    - JAX-WS
    - Injection of Endpoint
* Service Activator:
  + What it did
    - Kind of JMS decorator
    - For calling business logic asynchronously
  + Now
    - @Asynchronous
* Transfer Object assembler:
  + What it did
    - Surprise: it assembled DTOs
  + Now
    - No DTOs necessary any longer
* Application Service:
  + This will remain – it’s the boundary from the BCE pattern
* Result
  + In Java EE there remain only some beans with pure business logic

# GoF Patterns

* Factory, AbstractFactory
  + Replaced by CDI producers
* Singleton
  + Replaced by @Singleton
* Builder
  + Still valid
* Adapter
  + Still valid
  + Cool possibilities using CDI producers that @Inject the adaptee
* Bridge
  + Still valid
  + In JavaEE: CDI stereotypes
* Façade
  + Still valid
  + But not often needed
* Flyweight
  + Still valid
  + In JavaEE: CDI scopes
* Template method
  + Still valid
  + In JavaEE: use CDI @Specializes
* Observer
  + Still valid
  + In JavaEE: CDI event mechanism

# Documentation and Test

* Document the exception, not the rule
* Javadoc
  + See [this blog post](http://www.adam-bien.com/roller/abien/entry/how_to_comment_with_javadoc)
  + Put
    - the WHAT in the name
    - the HOW in the code, and
    - the WHY in the comment
* UI Design documentation
  + Proposal: provide something executable (e.g. CSS)
* Best architecture documentation:
  + Use a Maven archetype
* Testing
  + Start with the hard stuff (optimal: system test)
  + Stress tests also should come first

# Conway’s Law

* The architecture of a system is driven by the organizational structure of the company
* See [Wikipedia](http://de.wikipedia.org/wiki/Gesetz_von_Conway)
* Team size
  + 5 or less is great – more is bad
  + See history: all great things were built by small teams
* Management should trust the developers

# Parkinson’s Law of Triviality

* The more trivial a decision is, the longer it takes to take it
* Limit the amount of meetings
* Limit the time of architectural meetings

# Deployment

* Blue-green deployment
* Two nodes with one load-balancer
  + One node is productive
  + The other one gets the new version
  + Switch from old to new after all sessions are terminated on old
  + The old node will get the next new version
* Hot deployment
  + Forget in production
* Recommendation:
  + One WAR per server

# GRASP

* [Wikipedia](http://de.wikipedia.org/wiki/GRASP)
* High Cohesion, Minimal Coupling
  + See Component model
* Protected Variations
  + Hide the concept that various
* Information Expert
  + Don’t spread the logic too much
* Polymorphism
  + Introduce more class if there are too many if’s or instanceof’s
* Pure fabrication
  + Sometimes you need objects that don’t have to do with business logic
* Focus more on objects
  + Rethink our BO idea
  + Allow methods there

# Reactive programming

* In Java this is used in JavaFX
* How it works
  + define c = a + b
  + When a or b changes, c is recomputed automatically

# Command Query Responsibility Segregation (CQRS)

* [Wikipedia](http://martinfowler.com/bliki/CQRS.html)
* Example:
  + Statically cache web contents
  + Update them (only) upon a change to the background data
  + That means: Write to the dynamic background data, read from the static cache
* This improves scalability

# Service governance and versioning

* API Versioning
  + Don’t version
  + Build all versions of the client against the latest version of the server
  + Don’t delete anything
* Theorem: stable interfaces cannot by type-safe

# Stateful vs. stateless Java EE architectures

* Where is the state?
  + If not in an HTTP or EJB session it’s most probably in the DB (which is anyway the bottleneck)
* Summary
  + Stateless architectures don’t necessarily scale very well
  + Use EJB or HTTP session if it makes sense (and data amount is okay)
* Regarding entities:
  + If you keep the entities attached within the user session, the EM can automatically write changes to the object graph back to the DB
  + TODO: can’t we use this

# Error handling

* In JavaEE
  + Unchecked Exceptions cause a rollback
  + Checked ones don’t by default!!!!
    - Check @ApplicationException.rollback
* ApplicationExceptions come from the business logic
  + No exception packages!
* All Exceptions should have IDs (not per instance but per type)
  + Enums are a good candidate here
* ValidationExceptions should be ApplicationExceptions (but might rollback)
* System exceptions should be unchecked
* Business exceptions might be unchecked

# Logging

* Log to DB
* Log what helps you
* Business departments is more interested in audit (where is my process)
* Have a look at [Splunk](http://de.splunk.com)
* Define a clear mapping between log level and exception type
* Use lowest possible log level
* Logging can make your application behave SERIAL

# Management and monitoring

* JMX is great
* But REST is easier to access for operations (easy scripting)
* It’s easy to implement both

# Continuous Integration and Continuous Deployment

* A good Jenkins scenario

1. mvn clean install (and run Sonar)
2. Jenkins Copy Artifact Plugin + mvn failsafe:integration-test
3. Setup AS
4. Deploy APP to AS
5. System Tests
6. (Fitness tests)
7. Use Promoted Build Plugin
   1. To promote to the quality server
   2. Testers push the “Done” button, when quality tests are done
   3. The same might be used for promotion to PROD
8. Use Build Pipeline Plugin to visualize all boxes
9. In the night run Performance Plugin

* Don’t allow Maven lifecycle hooks
  + Instead use Jenkins for partitioning
* Use Jenkins agents not to run into performance issues