# Build

* WAR is the best archive
* Make POMs as small as possible
* Tweak builds with –T flag of Maven (concurrent builds)
* Check-in very often
* Question every dependency (keep the WAR small)
* Versioning: no build number – use the POM version
* (Check [www.semver.org](http://www.semver.org))

# CI

* Use CI
* Have a good Jenkins machine
* Configure Jenkins in Master-Slave mode
* Jenkins with Git => Hook possible that informs Jenkins about the change
* Move integration test logic to CI server
* Use [SonarSource](http://www.sonarsource.com) / [SonarQube](http://www.sonarqube.org) on the CI server to validate your architecture
* Run DB within VMWare/VirtualBox
  + Both have a Java API
  + Rollback == Reset to Snapshot
  + It’s fast
* Netbeans can integrate into Hudson-server (get status)
  + TODO: Check Jenkins-Eclipse integration

# Versioning

* Use Git
* SVN is slow / Git is fast
* Better concepts in background
* Less problems
* It’s THE new VCS (everybody moves away from SVN)

# RESTful architectures

* SOAP is not object-oriented
* SOAP calls static methods
* REST is about objects == „resources“
* REST allows to read/update the state of resources
* HTTP operations perfectly map to typical object operations
  + GET == Select
  + POST == Insert
  + PUT == Update
  + DELETE == delete
* HTTP status (e.g. 201, 202) allows well-defined return states
* Validation: use Bean Validation (no need for XSD validation)
* Security: use standard JavaEE security (HTTP basic Auth, SSL, etc.) => plain old HTTP (== Servlet)
* Practical approaches
  + Search operation
    - Define a new resource for searches
    - E.g. GET xmas/v1/searches/[your search]
  + Think in terms (== substantives), NOT operations
    - Make a resource out of every concept
    - E.g.: searches, transactions, …
  + Use ExceptionMapper instead of try/catch
    - Return a respective HTTP response
    - Use headers “x-[something]” to return additional information
* Examples:
  + Kenai (Sun Cloud API)
  + HornetQ REST API

# DRY and DIE

* **D**on’t **R**epeat **Y**ourself
* **D**uplicate **I**s **E**vil
* Ideal situation:
  + Entity == REST-Resources
  + No DTOs necessary
  + Preface:
    - The database is ours

# KISS

* **K**eep **I**t **S**imple & **S**afe
* All you need to have is the domain logic
* Simply write down the domain logic
* No extra things (a là “we might need that later”)
* Evolve business logic with iterations (“

# YAGNI

* **Y**ou **A**ren’t **G**onna **N**eed **I**t
* Question everything (also and *especially* the business logic)
* Example: Apple (do a few things, but do it good)

# Architectural and Design Patterns

* Structure the application by business – not technology
  + Example: make one package by business object – not one for all business objects
* Packages == Components
* Components are business-driven
* Component
  + Boundary: contains all interfaces (REST, local, etc.)
    - Web-Services
    - REST-Services
    - VETRO-Services
    - Business-Services
    - CDI-Producers !!!
  + Control: „helpers“ for the boundaries (are products of refactoring)
    - Rule-Services
    - DAOs
  + Entity: contains all POJOs
    - BOs
    - JPA-Entities
* Don’t allow
  + Boundary-to-Boundary calls (instead introduce a control)
  + Control-to-Boundary calls
  + Entity-to-\*
* No components within UI tier

# Testing

* Local DB testing
  + Hard-code embedded DB props in test-specific persistence.xml
  + Use non-JTA-datasource
  + Use „embedded“ AS dependency for test execution
  + Create the EM manually within @Before and set it within the tested entity
    - Better ways?
* Name the test „\*IT“ to mark it as IntegrationTest
  + TODO: check why we did not use it
* The ultimate test is the system test
* BUT: Unit tests can help you a lot on the way there

# General

* Business interfaces
  + Prefer fine-grained interfaces before course-grained ones