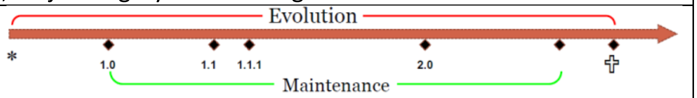
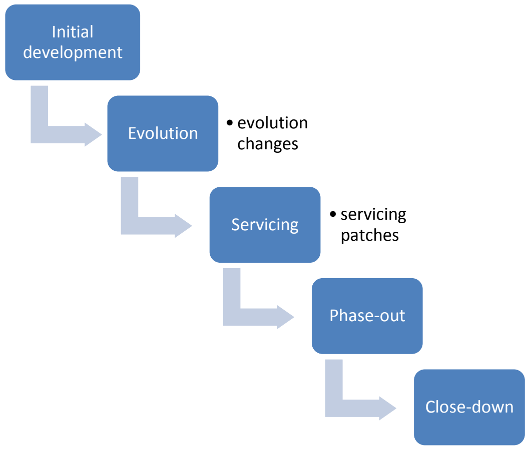
SOFTWARE ENGINEERING

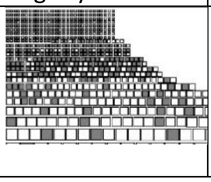
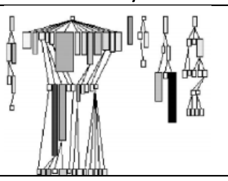
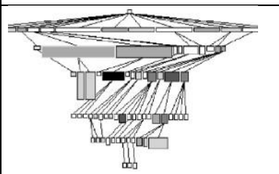
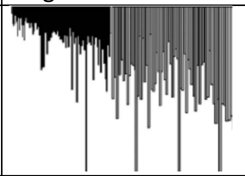
Software Evolution

Evolution	is the set of activities (technical and managerial), that ensures that software continues to meet organizational and business objects in a cost effective way over its lifetime. Driven by changes from stakeholders.
Types	Requirements, Architecture, Design, Test case, Traceability, Data, Runtime, Language,
Classical Engin.	Waterfall model: when we are done after testing, then why does maintenance cost 70-80% off all cost
Agile Engin.	Software evolution is a ingredient of agile software development (iterative development, allow change)
Software Maintenance	functionality stays the same! • **Corrective**: Errors need to be fixed (Bugfixing) • **Preventive**: Prevent problems in the future (e.g. fix design issues) • **Adaptive**: Something has changed in the environment (e.g new version) • **Perfective**: Improve system qualities (e.g. performance)
Software Aging	Causes: Lack of Movement (product owner don't see that changes are needed), e.g. DOS Ignorant Surgery (caused by changes which do not understand the original design concept)
	Cost: Inability to keep up (with the market) e.g. VMS; Reduced performance Decreasing Reliability (buggy, accidental bugging) e.g. MS-Office
	Prevent: Design and code for change; Keep records (docu); Second opinion (reviews)
	Treat old software: retroactive documentation (build afterwards); retroactive incremental modularization; amputation (remove unused code); major surgery - restructing
Maintenance vs Evolution	**Maintenance:** software is already delivered **Evolution:** from the very beginning 
Types of Programs	**S-Programs**, can be completely and formally specified (e.g. sort an array) **P-Programs**, can be completely specified, but makes an approximation of the real world (e.g. chess) **E-Programs**, mechanize a human or societal activity (e.g. ERP-system)
Lehman's Law	Software systems have to evolve, otherwise they gradually become useless.
Roadmap	Initial development stage • Research challenge: design for change • Outcome: architecture & team knowledge Evolution stage • Goal: implement changes • Research challenge: program comprehension • Management issue: keep team Servicing stage • Goal: tactical changes at minimum cost • Research challenge: program compehension Phase-out • servicing discontinued Close-down • switch-off 
Legacy Systems	old computer-based systems, which are still in use by organizations • still business critical • many changes over the years • many people involved • replacing is risky (no/incomplete docu, change at high costs, no knowledge/understanding) • difficult to modify
Reverse engin.	trying to understand the architecture or behaviour of a large software system from source code
Re-engineering	trying to re-structure a legacy system, to produce a new system that is more evolvable
Forward engin.	traditional process of moving from design to implementation
Deal with change	**Program comprehension**: Understanding the existing program in order to change it. Methods: Source code/Runtime/Performance/Design/Architecture analysis, Metrics, Visualization tools Tools: Sotograph, Metric, Checkstyle, CodeCrawler, Sonar **Change impact analysis**: Identification of parts of the system that will be affected by a proposed change. **Change propagation**: Making sure that all affected parts are changed correctly. **Restructuring/Refactoring**: Improving the software structure or architecture without changing behavior. **Regression testing**: Verifying that the change should not have an impact on the previous behavior. **Program transformation**: One or multiple modifications applied to a program • Translations (other language) e.g. Program Migration, Reverse Engineering • Rephrasing (same language) e.g. Reengineering, Refactoring

Quality Metrics/Analysis and Visualization

Goal		Quality Control, determine Quality of legacy code
Levels		Code, Design, Architecture
Metrics		a measurement scale and method to determine the value of an indicator of a certain software product
Types		Size Metrics: LOC, Number of Classes, Number of Methods, Halstead-Metric Logical Structure Metrics: Cyclomatic Complexity McCabe Data Structures Metrics: Number of Variables, Duration Style Metrics: Naming Conventions, Nesting Metrics for Cohesion and Coupling: Fan-In, Fan-Out, Lack-of-Cohesion, Number of called Methods
Pro		quick overview, Indikator SW-Quality, Timeline SW-Quality, automatisierbar, motiviert, vergleichbar
Cons		absolute Zahlen, nicht immer aussagekräftig, lange Rechenzeit, Zahlenfixiert, keine optimale Schwellwerte
Tools		Metrics (for Eclipse), Checkstyle, Emma, CMT, Sonar, SonarQube, …
Visualizations		Software visualization tools use graphical techniques to make software visible (e.g. CodeCity)
Types		Hierarchical Views: Euclidean cones, Hyperbolic trees; Bottom UP Approach: Filter
Goal		read quality, get understanding, various levels, scalable
Approach		Polymetric View (=colored rectangles for the entities and edges for the relationships)
Pro		Customizable, modifiable, simple, powerful, scalability
Cons		Visual language must be learned, can't view inside the classes and strucutres -> go to code
Tools		CodeCrawler, Evolizer, Moose, Creole / Shrimp, CodeCity, EvoSpacer, Rigi, JInsight, Sonargraph, …
Technical Dept		metaphor to help us think about doing something quick and dirty
FEAST		Feedback, Evolution And Software Technology
Code Duplication	Goal	Avoid code and data duplications / redundancy
	Problems	Increase size of code, hard to understand and maintain code, more bugs
	Types	1: is an exact copy without modifications (except for whitespace and comments) 2: is a syntactically identical copy, only variable type, or function identifiers has changed 3: is a copy with further modifications; statements have changed/added or removed
	Cause	Unknown change impact; badly, organized reuse; time pressure; Ignorance; shortsightedness
	Handling	• Preventive: activities to avoid new clones • Compensative: limit the negative impact of existing clones • corrective: remove clones from systemk
	Solutions	Code refactoring, modularization and parameterization

Polymetric View

	Layout: Checker	Layout: Tree	Layout: Tree	Layout: Stapled	Layout: Scatterplot
Target	Classes	Classes	Classes	Classes	Methods
Scope	Full system	Full system	Subsystem	Subsystem	Full system
Metrics	Width: NOA Height: NOM Color: WLOC	Width: NOA. Height: NOM. Color: WLOC	Width: NMA. Height: NMO. Color: NME	Width: NOM. Height: WLOC. Color: NOM	Position (X): LOC. Position (Y): NOS.
Sort	Width	-	-	-	-
goal	identify large and small classes scales up to very large systems	detect complexity and structure in terms of the functionality	qualifies the inheritance relationships by displaying NMA relative to NMO	relates NOM with WLOC of classes detect exceptions in height	very scalable view shows all methods using a compare LOC and NOS as position metrics
					

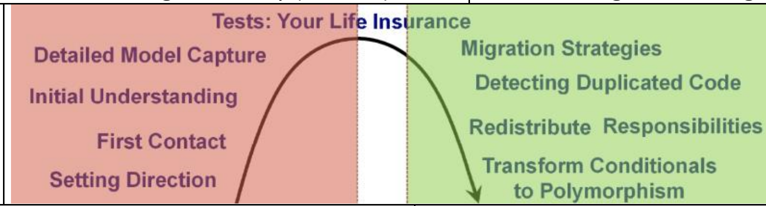
Metrics

Class Metrics	**Method Metrics**
HNL: Number of classes in superclass chain of class	LOC: Method lines of code
NME: Number of methods extended, override but use base	MSG: Number of method message sends
NMI: Number of methods inherited, defined in superclass	NOP: Number of (input) parameters
NMO: Number of methods override, redefined	NI: Number of invocations of other methods within method body
NOA: Number of attributes (= NIV + NCV)	NMAA: Number of accesses on attributes
NOC: Number of immediate subclasses of a class	NOS: Number of statements in method body
NOM: Number of methods	**Attribute Metrics**
WLOC: Sum of LOC over all methods	NAA: Number of times directly accessed (= NGA + NLA)
WNOC: Number of all descendant classes	NGA: number of direct accesses from outside of its class
	NLA: number of direct accesses from within its class

Restructuring Existing Code - Evolution of Legacy Code

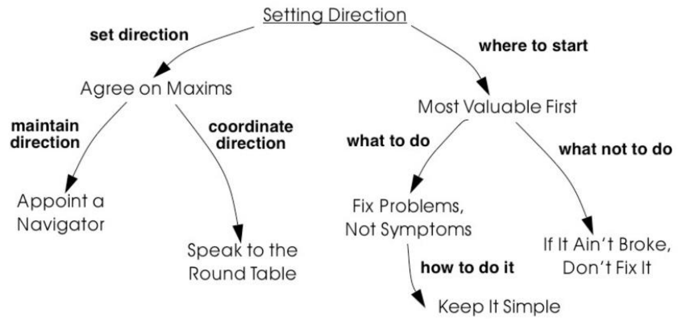
Tools	There are a lot of tools, but people must do the job.	
New Functionality	Var A) Hack: duplicated code, complex conditionals, abusive inheritance, large classes/methods -> like taking a loan on your software -> pay back via reengineering	
	Var B) No Hack: refactor, restructure, reengineer first -> investment for the future -> paid back during maintainance	
Goals	**Reverse Engineering** Cope with complexity, Recover lost information, Generate alternative views, Detect side effects, synthesize higher abstraction, Facilitate reuse	**Reengineering** Unbundling, Performance, Port to other platform, Design extraction, Exploitation of New Technology
Techniques	Redocumentation, Design recovery (metrics)	Restructuring, Data reengineering, Refactoring

Re*-Patterns



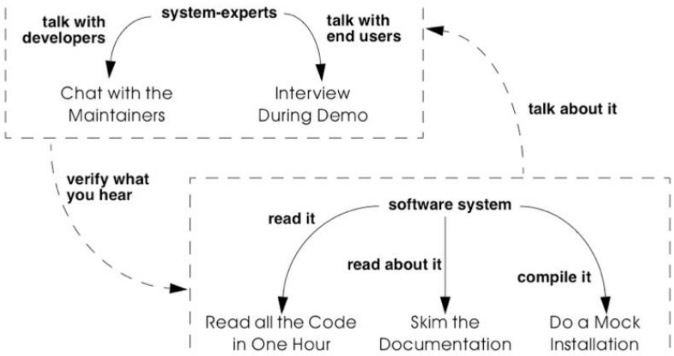
Lifecycle:
1. requirement analysis
2. model capture
3. problem detection
4. problem resolution
5. program transformation

1. Setting Direction



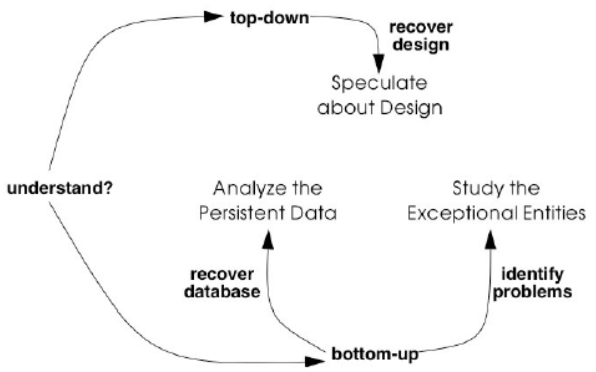
Agree on Maxims (common understanding)
Establish key priorities, Identify guiding principles
Most Valuable First (for customer)
Maximize Commitment, early results, build confidence

2. First Contact



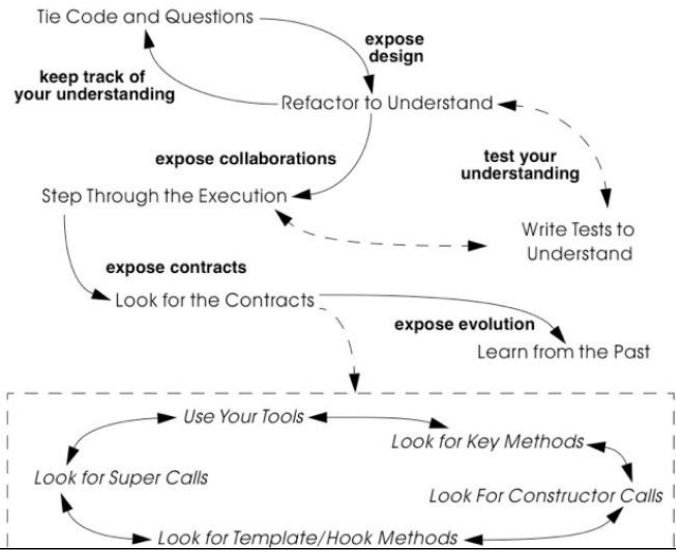
System experts
talk to maintainers to get historical and political context
talk to end users to get an initial feeling for the functionality
Software system
read it (all code in one hour), read about it, compile it

3. Initial Understanding



top-down (recover design)
bottom-up (recover database, identify problems)

4. Detailed Model Capture



Redistributed Responsibilities	The Law of Demeter (method M of obj O should invoke only methods of O, param of M, obj created by M, direct component obj of O) -> Don't talk to strangers 1. Eliminate Navigation Code (this.intermediary.provider.service(); -> remove middle man) 2. Split up God Class (to much intelligence -> split up -> easier said, than done) 3. Move Behavior Close to Data
Transform Conditionals to Polymorphism	1. Transform Self Type Checks – 2. Transform Client Checks – 3. Factor out State – 4. Factor out Strategy – 5. Introduce Null Object – 6. Transform Conditionals into Registration

Adding Tests to Legacy Code

<table>
<tr><td>Tests</td><td colspan="2">Your Life Insurance</td></tr>
<tr><td>TDD</td><td colspan="2">Test Driven Development</td></tr>
<tr><td>Process</td><td colspan="2">Write test code -> Execute test which should fail -> Write functional code until test pass -> Iterate</td></tr>
<tr><td>Pro</td><td colspan="2">better design of code -> think about its intended use, simpler code -> only program requirements,
documentation and specifications, faster iterations during impl., breaking dependencies, safely refactoring</td></tr>
<tr><td>Mock Objects</td><td colspan="2">simulated objects that mimic the behavior of real objects in controlled ways (Fake)
pro: interface discovery, consider an object's interactions with its collaborators
Need-Driven Development: guides interface design by services that an object requires, not those it provides</td></tr>
<tr><td>Legacy Code</td><td colspan="2">is code without tests -> bad code. it doesn’t matter how pretty, well written, object-oriented it is.</td></tr>
<tr><td>Why change?</td><td colspan="2">Adding a feature; fixing a bug; improve design; optimize resource usage</td></tr>
<tr><td>How do we change?</td><td colspan="2">a) Edit and Pray (work with extreme care)
plan carefully -> fully understanding of change -> make change -> run to check -> smoke tests -> pray
b) Cover and Modify (work with a safety net – a test harness)
run tests -> write new tests -> write code -> refactor -> wash/rinse/repeat -> verify by running tests</td></tr>
<tr><td>Change Alg.</td><td colspan="2">Identify change points -> Find test points -> Break dependencies -> Write tests -> Make changes and refactor</td></tr>
<tr><td>Why Breaking Dependencies</td><td colspan="2">a) Sensing (break dependencies to get visibility/understanding what the code is doing)
b) Separation (break dependencies to test in isolation)</td></tr>
<tr><td>Types of Dependencies</td><td colspan="2">• Singletons -> hope that Singleton-Class is good for your tests too
• Internal instantiations (new Class) -> hope that class runs well in tests
• Concrete Dependencies (give Class per Constructor) -> hope that class let’s you know what is happening</td></tr>
<tr><td>seam (Naht)</td><td colspan="2">a seam is a place where you can change the behavior without editing in that place
every seam has an enabling point, a place where you can make the decision to use one behavior or another</td></tr>
<tr><td>type: object seam</td><td>This method call is not a seam, no enabling point<pre>void doSomething() {
   IController c = new BombController();
   c.doAction();
}</pre></td><td>This is now a seam, we can change behaviour<pre>void doSomething(IController c) {
   c.doAction(); // change behaviour
}</pre></td></tr>
<tr><td>other types</td><td colspan="2">pre-processing seam, link seam</td></tr>
<tr><td>Problems that can occur</td><td>CUT (Class Under Test)
Object of the class can’t be created easily
Test harness won’t build with the class
Constructor we need to use has bad side effects
Significant work happens in the constructor</td><td>MUT (Method Under Test)
The method to test is not accessible
Method needs hard to construct parameters
Method has bad side effects</td></tr>
<tr><td rowspan="3">Changing Software</td><td colspan="2">Reason: I need to change a monster method and can’t write tests
Action: Introduce sensing variables, Extract what you know, Break out a method object, Skeletonize Methods, Find Sequences, Extract to the current class first, Extract small pieces, Be prepared to redo extractions</td></tr>
<tr><td colspan="2">Reason: I need to make a change, but don’t know what tests to write
Action: Characterization tests, Characterizing classes, Targeted Testing</td></tr>
<tr><td colspan="2">Reason: It takes forever to make a change
Action: Understanding, Lag Time, Breaking Dependencies, Build Dependencies</td></tr>
<tr><td>Breaking dependencies</td><td colspan="2">Extract and Override Call: Extract the call to a virtual method and then override it in a testing subclass
Extract and Override Factory Method: Extract object creation into factory method and override in tests
Replace Global Reference with Getter: Extract global reference to method and override in tests
Extract Interface: Find member functions to extract, and copy function signatures to a interface
Parameterize method: Identify dependency and make new method with arguments</td></tr>
</table>

Software Architectures

Role of software architecture

Architecture	is a **process**: design and build; is a **role**: software architect; results in **products**: plans, models, prototypes
Definition	a software system's architecture is the set of **principal design decisions** about the system -> Taylor
Design decisions	• **structure** e.g. “The elements should be organized and composed exactly like this…” • **behavior** e.g. “Data processing, storage, and visualization will be performed in strict sequence” • **interaction** e.g. “Communication among all system elements will occur only using event notifications” • **non-functional properties** e.g. “the dependability will be ensured by replicated processing modules”
Requirements	• fulfills functional and non-functional requirements (Ressource, CPU, RAM, Responsetime, Scalability) • can be realized (political and organizational) • can be implemented (e.g. proof through prototypes)
Prove	prototype, vertical slice
Types	Enterprise architecture (defines how an enterprise uses many applications -> metaphor: city planning) Application architecture (defines the pieces that compose an application -> metaphor: building architecture)
Document	UML (Diagramme/Modelle) or Kruchten's View 4+1 (**Logical**=functional; **Development**=programmer; **Process**=dynamic; **Physical**=topology; + Scenarios=*UseCases)*
Difficulties	Multi-dimensional decisions, interdependent factors, strong impact, requirements change
Pro	communication among stakeholders (understanding/consensus/discussions/decisions) document design decisions (guideline/basis/planing/checkpoints) abstraction of the system (homogeneous systems / outsourcing or acquiring parts)
Misconceptions	Architecture is the same as design -> Architect's focus is on the boundaries and interfaces Architecture is about infrastructure -> Frameworks, application servers, and databases from a minor part of the problem space only Architecture solves technical problems -> Changes are your biggest problem, isn't technical Architecture is rigid and fixed (up front) -> Understand the impact of change -> Start with a walking skeleton -> Great software is not built, it is grown Architecture is pure science or pure art -> requires both and more
Mentorsupport	A mentor-support onboaring process can have a major **positive impact**

Role of software architect

Role	guarantee fulfillment of requirements (within budget) demonstrate achievability (with models and/or prototypes) design and construct (components, interfaces, responsibilities, structure) coach and consult developer and other stakeholders (technology, project planning, risk management)
Tasks	Decide (under uncertainty, but decide), Document (adequately), Proof feasibility, Program, Communicate, Negotiate (with stakeholders), Simplify, Standardize, Listen, Observe, Think (about the future), Lead
Mistakes	Believing the requirements; Being seduced by the technology; Majoring on your strengths and neglecting other areas; Not stopping designers from designing; Thinking you can do it all yourself.
Requirements	• Knowledge and Experience in Architecture • Technical breadth and technical understanding • Disciplined, methodical working • Experience with the whole Software Life Cycle • leadership qualities • ability to communicate
Summary	Simplicity, Abstraction, Separation, Structure, Interface, Communication, Delivery of working systems

Übung

Was ist architektur?	Framework	Programming Language
Fowler: "important stuff"	ja	ja
Taylor: principle design decisions	?	nein
Bass: Strukturen, Element, Beziehungen	ja	nein
Ford: Hard to change later	? / ja	ja

Interfaces

why?	Major aspect of good software design. It's too easy to design bad or wrong interfaces.		
types	**run-time**: call (function, method, procedure), event-callback, remote interfaces (synch/asynch) **compile-time**: inheritance, use, inclusion/import		
examples	java interface, UNIX/POSIX for files, REST		
types and styles	data interfaces (methods <-> class attributes) sequential access (iterator above list) 1-to-1 relationship interface stateless interface (no storage) minimal interface (only needed methods) with inheritance (less delegation)	vs vs vs vs vs vs	service interface (methods <-> parameters) random access (get any element in list) n-to-1 relationship of interface stateful interface (with storage) complete interface (methods for convenience/efficiency) with interfaces (delay hierarchy until usage is known)
Remote interfaces	procedural style (req-res, synch, handle failures) synchronous (immediate, blocks, not scalable) stateless (scalable server, redundancy, more data)	vs vs vs	document style (send/recv messages in document) asynchronous (scalable, parameter validation) stateful (state per client, less data, state recovery)
REST	Representational State Transfer (Fielding 2000) Client/Server, Stateless, Cachable standardisierte Operationen/Daten(JSON) resources are uniquely identified by the path WebService APIs offering REST over HTTP		**Goal**: Uniform interface with 4 Constraints: Identification of Resources in Requests Manipulation of Resources through Representations Self-descriptive messages Hypermedia as the engine of application state (?!)

HTTP Method

	Safe (no change of data)	Idempotent (multiple exec does not change the data)
GET / HEAD / OPTIONS	yes	yes
POST / PATCH	no	no
PUT / DELETE	no	yes

Design Principles

- Information Hiding
- Low coupling, high cohesion
  Coupling (2 classes): dependency between two classes
  Cohesion (1 class): low cohesion means great variety of actions, high means focus on intention
- Separate Query (get) and Action(set) (either obtain state or change state)
- Three Laws of Interfaces (Ken Pugh)
  - do what the method say it does; do not harm; notify caller if unable to perform
- Manage Dependencies – SOLID principles (Robert Martin)
  - SRP: Single Responsibility: high cohesion, only one reason to change
  - Open-Closed Principle – open for extension, closed for change
  - 'Liskov' substitution principle – subclasses fulfill interface's role
  - Interface Segregation Principle – split "fat" interfaces to increase cohesion
  - Dependency Inversion Principle:
    high-level classes should not depend upon low-level; both should depend on abstraction
    abstractions should not depend on details; details should depend on abstractions
- Simplicity

DbC (Design by Contract)

Design by Contract	a contract defines obligations of both parties (client/supplier) as their benefits Preconditions (@Requires) – what the client needs to provide as true -> client/caller is responisble Postconditions (@Ensures) – what the component promises to establish -> supplier is responsible Invariants (@Invariant) – conditions that remain true -> for supplier
	Contracts belong to the interface -> only define contracts for public interface methods
query=get command=set	1. Separate queries (get) from commands (set) 2. Separate basic queries (e.g. count) from derived queries (e.g. isEmpty) 3. For each derived query, write a postcondition that specifies what result will be returned in terms of on or more base queries (e.g. postcondition in isEmpty: return (count=0) 4. For each command, write a postcondition that specifies the value of every basic query 5. For every query and command decide on a suitable precondition 6. Write invariants to define unchanging properties of objects
problems	lack of language support, not used systematically
pro	contracts abstract from implementation; good documentation; clearly defined responsibilities between client and supplier; simpler code; helps writing better unit tests; less bugs
defensive programming	ensure the continuing function of code under unforeseen circumstances -> automatically fix failures opposite of DbC

HTTP Status Codes

This page is created from HTTP status code information found at ietf.org and Wikipedia. Click on the **category heading** or the **status code** link to read more.

1xx Informational

100 Continue
101 Switching Protocols
102 Processing (WebDAV)

2xx Success

★ 200 OK
★ 201 Created
202 Accepted
203 Non-Authoritative Information
★ 204 No Content
205 Reset Content
206 Partial Content
207 Multi-Status (WebDAV)
208 Already Reported (WebDAV)
226 IM Used

3xx Redirection

300 Multiple Choices
301 Moved Permanently
302 Found
303 See Other
★ 304 Not Modified
305 Use Proxy
306 (Unused)
307 Temporary Redirect
308 Permanent Redirect (experimental)

4xx Client Error

★ 400 Bad Request
★ 401 Unauthorized
402 Payment Required
★ 403 Forbidden
★ 404 Not Found
405 Method Not Allowed
406 Not Acceptable
407 Proxy Authentication Required
408 Request Timeout
★ 409 Conflict
410 Gone
411 Length Required
412 Precondition Failed
413 Request Entity Too Large
414 Request-URI Too Long
415 Unsupported Media Type
416 Requested Range Not Satisfiable
417 Expectation Failed
418 I'm a teapot (RFC 2324)
420 Enhance Your Calm (Twitter)
422 Unprocessable Entity (WebDAV)
423 Locked (WebDAV)
424 Failed Dependency (WebDAV)
425 Reserved for WebDAV
426 Upgrade Required
428 Precondition Required
429 Too Many Requests
431 Request Header Fields Too Large
444 No Response (Nginx)
449 Retry With (Microsoft)
450 Blocked by Windows Parental Controls (Microsoft)
451 Unavailable For Legal Reasons
499 Client Closed Request (Nginx)

5xx Server Error

★ 500 Internal Server Error
501 Not Implemented
502 Bad Gateway
503 Service Unavailable
504 Gateway Timeout
505 HTTP Version Not Supported
506 Variant Also Negotiates (Experimental)
507 Insufficient Storage (WebDAV)
508 Loop Detected (WebDAV)
509 Bandwidth Limit Exceeded (Apache)
510 Not Extended
511 Network Authentication Required
598 Network read timeout error
599 Network connect timeout error

★ **"Top 10"** HTTP Status Code. More REST service-specific information is contained in the entry.

http://www.restapitutorial.com/httpstatuscodes.html#

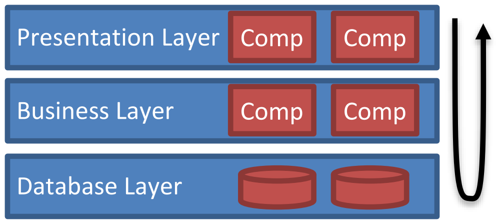
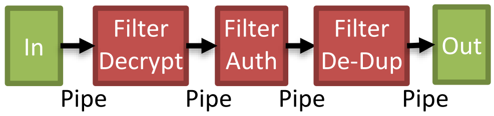
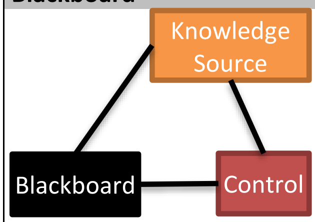
Architecture Styles and Patterns

Architecture Style	Set of established architectural organizations – components, relationships, connectors, … **Patterns**: well-known organizational structures • Descriptions of successful engineering stories • Address recurring problems • Describe generic solutions that worked **Reference Models**: Prescribes specific configurations of components and interactions
Pattern Types	**Architectural Patterns**: Express a fundamental structural organization scheme for software systems **Design Patterns**: Scheme for refining subsystems or components. **Idioms**: Low level pattern specific to a programming language
Structure of a pattern	Problem, Tension (Forces to make a problem hard), Resolution of forces (Solution), Relationship to other Patterns, Consequences (Pro/Cons)
Misconceptions	All design patterns are inherently good -> Counter-Example: Singleton "I invented that pattern" -> rule of three known uses Design patterns are blueprints -> with copy-paste copy examples NO! Let you turn off your brain -> they give you a better means to think about design Are for experts only -> good OO programming without knowing design patterns is impossible today
Dangers	Too much flexibility, too many patterns in a design, separate patterns split into separate classes, over-engineering, misunderstanding the example/diagram for the pattern
Things to know	more than 23 patterns, successful solution concept, good patterns are honest -> pro and cons pattern names give us a common vocabulary to discuss design efficiently and are targets for refactoring Beware of YAGNI (You ain't gonna need it)! Create simple code!
Summary	A pattern consists of more than the solution (diagram) but is a description of a proven engineering experience that applies in each context and solves a problem generically with stating benefits and liabilities. Some patterns are obsolete (Singleton)

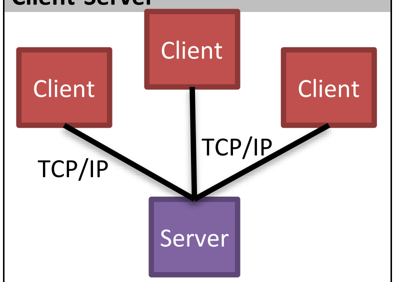
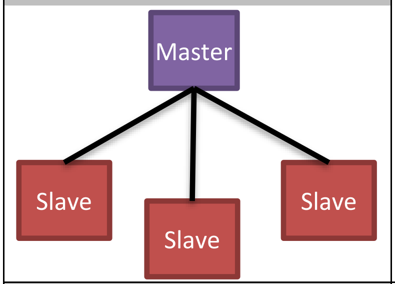
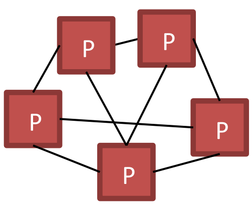
Architectural Pattern

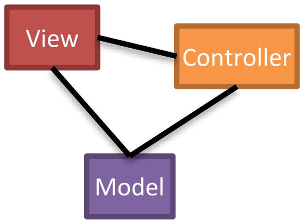
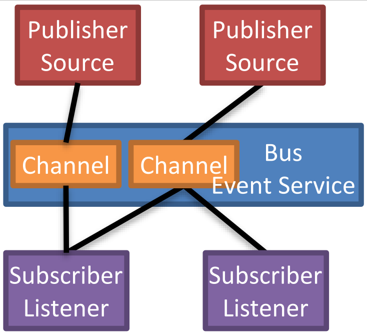
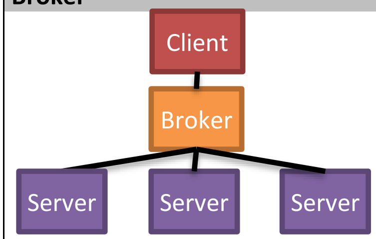
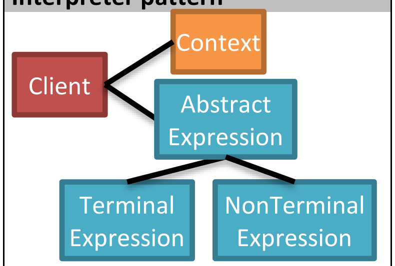
A pattern for software architecture describes a recurring design problem that arises in specific design contexts, and presents a well-proven generic scheme for its solution.

From Mud to Structure

Layers  structure the software into several layers each layer has a role and responsibility	**Intent** structure applications that can be decomposed into groups of subtasks **Solution** services and interface per layer, bottom up **Examples** 3-tier architecture, OSI 7 Layer model, TCP/IP, APIs, Virtual machines	**Pro** Reuse of layers dependencies kept local exchangeability **Con** cascades of changing behavior buffering of data unnecessary work (checksum, encrypt) difficult to find correct granularity
Pipes and Filters  Divide a large processing task into smaller, independent steps (filters) that are connected by channels (Pipes)	**Intent** Structure for systems that process a stream of data Each step is encapsulated in a filter Data is passed through pipes between adjacent filters (Buffering/Sync) **Examples** Compiler, Incoming-Order	**Pro** concurrent processing, reusable, exchangeability, scalable **Cons** efficiency is limited to slowest filter buffering of data
Blackboard 	**Intent:** Blackboard as common exchange of information. knowledge source with specialized modules and own representation control component which selects, configure and execute modules **Example** Speech recognition, vehicle identification and tracking	**Pro** Easy to add new apps, extend data space is easy **Cons** modifying the structure of data space is hard as all apps are affected need synchronization and access control

Distributed Systems

Client-Server 	**Intent:** services are centrally available on a server. clients **request** services from the server server **respond** relevant services to clients **Examples**: Web-Clients, Document-Sharing	**Pro** Good to model a set of services **Con** Thread per Request IPC can cause overhead
Master-Slave 	**Intent** master distribute work among identical slaves slaves return result to master master computes a result **Examples** huge computation operation	**Pro** Accuracy, Performance **Cons** isolated slaves -> no shared state latency due to communication only for decomposable problems
Peer-to-Peer 	**Intent:** Distributed application partitioning of tasks or work load peers are equally privileged peers can be both clients and servers **Examples** File-sharing networks (G2, Gnutella), Multimedia protocols (P2PTV), Multimedia applications (Spotify)	**Pro** supports decentralized computing robust in failure of a peer scalable in resources and power **Cons** no guarantee about quality, security performance depends on number of nodes

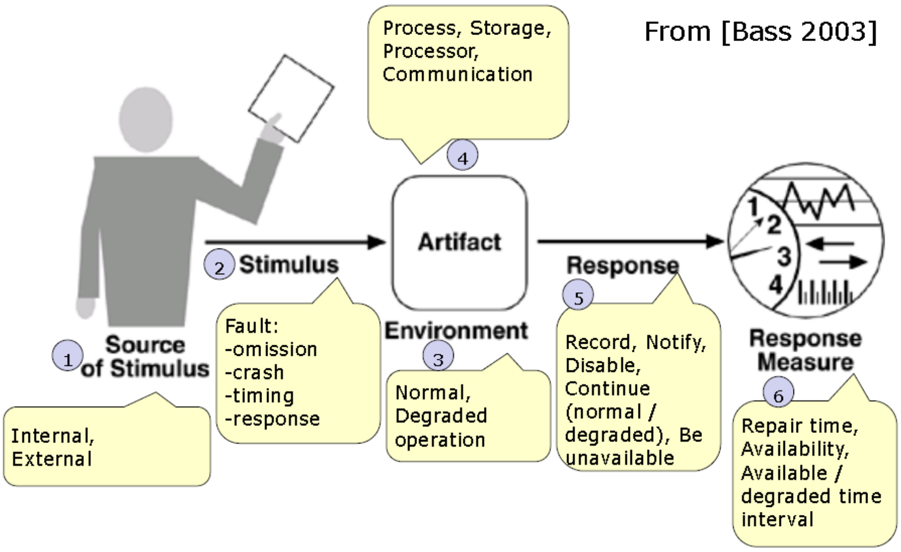
SOA (Service Oriented Architecture)	**Intent:** independent products/service/technologies can be access remotely services are implemented through messaging	**Pro:** Loose coupling, information hiding, stateless, reusable, compose services **Cons:** not scalable because of shared interface
MVC (Model View Controller) 	**Intent:** Model contains the core functionality and data. Views display information to the user. Controllers handle user input. Views and Controllers together comprise the UI. **Solution** Identify core functionality and model classes Implement change-propagation mechanism Design view/controller/relationship/init **Examples:** MFC, Swing, Web frameworks (Django, Rails)	**Variants:** merge view and controller **Pro** Multiple, synchronized, pluggable view; exchange of "look and feel", framework potential **Cons**: increased complexity potential excessive number of updates private, intimate coupling between view and controller; close coupling of views and controllers to a model
Publisher-Subscribe or Event-bus pattern 	**Intent** keep the state of components synchronized. Enables one-way propagation of changes: one publisher notifies any number of subscribers about changes to its state. **Solution** subscribers register their interest in an event and are subsequently asynchronously notified of events generated by publishers. Loosely coupled form of interaction required decoupling: nobody knows each other, scalable **Examples** Android development, Notification services	**Variants** Filtering: not all events are of interest **Pro** Decoupling, can come and go, effective for highly distributed systems **Cons** Event service may need to store events Authenticity of events: trust each other difficult to scale
Broker 	**Intent** Server publish their services to broker Client request a service from the broker Broker redirects client to a suitable server **Example** Apache ActiveMQ, Apache Kafka, RabbitMQ, JBoss Messaging	**Pro** dynamic change, addition, deletion and relocation of servers transparent distribution to developer **Cons** requires standardization of server
Interpreter pattern 	**Intent** Interprets programs written in dedicated language. class for each symbol **Example** Database query language (SQL) Communication protocols languages	**Pro** Highly dynamic behavior, good for end user programmability, flexible **Cons** interpreted language is slower than compiled one

Enterprise Integration Patterns

EAI Pattern provide solutions for the integration of systems and components.

File Transfer	Applications generate files for commonly used data, which are exchanged	
Shared Database	Applications read and write into a shared DB	
Remote Procedure Invocation	Applications offer interfaces that can be called	
Messaging	Applications use a shared messaging system for exchanging data	

ADD: Attribute Driven Design

ADD	**quality requirements (general or specific) -> set of tactics -> architecture**	
Qualität requirements	**observable via execution:** performance, security, availability, functionality, usability **not observable via execution:** modifiability, portability, reusability, testability	six quality attributes from Bass 2003
Quality Attribute Scenario		1. External 2. Unanticipated msg 3. Normal operation 4. Process 5. Inform operator, continue to operate 6. No downtime

Tactic: A tactic is a design decision that influences the control of a quality attribute response.

Modifiability tactics

- Localize modifications – Reduce the number of modules directly affected by a change
  - maintain semantic coherence – responsibility work together -> Layer, SRP
  - anticipate expected changes – minimize effect on change -> Adapter, Strategy, Interp, Facad
  - generalize module – broader range of functions due to its input type -> Interpreter
  - limit possible options – limit set of options -> Layer, Common Abstraction Layer
  - abstract common services – through specialized modules -> Helper/lookup service, ...
- Prevent ripple effects – limit modifications to the localized modules
  - Types of Dependencies: Syntax, Semantic, Sequence, Identity of an interface, Runtime location, Quality of service / data provided, Existence, Resource assumption
  - Information hiding – decompose and choose private-public -> Facade, Adapter, Proxy
  - Maintain existing interfaces – separate interface from implementation -> layering, adapter
  - Restrict communication paths – restrict data production and data consumption -> coupling
  - Use intermediary – introduce an intermediary to manage dependency -> MVC/PAC, mediat.
- Defer binding time – Control deployment time
  - Runtime registration – plug-and-play operation -> Lookup services, registries, plug-and-play
  - Configuration files – set parameters at start-up -> dependency injection
  - Polymorphism – late binding of method calls -> good class hierarchies, abstraction
  - Component replacement – load time binding -> dynamic loadable modules
  - Adherence to defined protocols – Runtime binding of independent processes -> e.g. SOAP

Availability Tactics

- Fault Detection -> Ping/Echo, Heartbeat, Exception
- Recovery-Preparation and Repair -> Voting, Active Redundancy, Passive Redundancy, Spare
- Recovery-Reintroduction -> Shadow, State Resynchronization, Rollback
- Prevention -> Removal from Service, Transactions, Process Monitor

Security Tactics

- Resisting Attacks -> Authenticate Users, Authorize Users, Maintain Data Confidentiality, Maintain Integrity, Limit Exposure, Limit Access
- Detecting Attacks -> Intrusion Detection
- Recovering from an Attack -> Restoration (see availability), Identification (Audit Trail)

Testability Tactics

- Manage Input / Output -> Record/Playback, Separate Interface from Implementation, Specialized Access Routines/Interfaces
- Internal Monitoring (Build-in Monitors)

Usability Tactics

- Separate User Interface
- Support User Initiative (Cancel, Undo, Aggregate)
- Support System Initiative (User Model, System Model, Task Model)

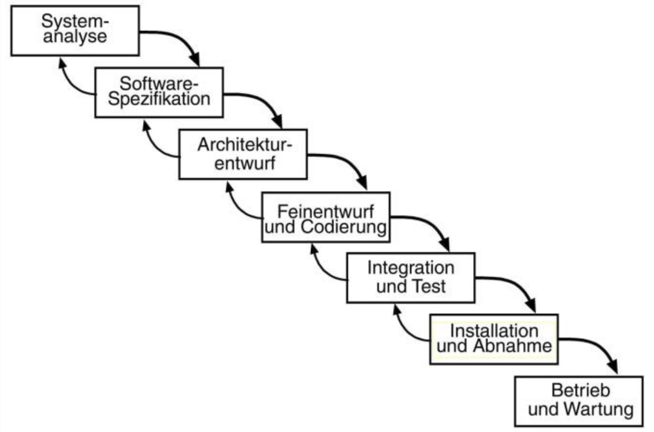
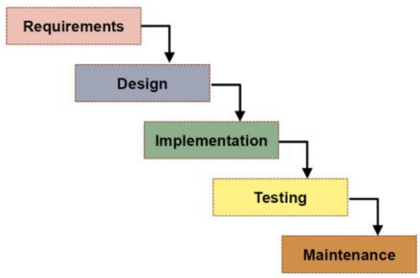
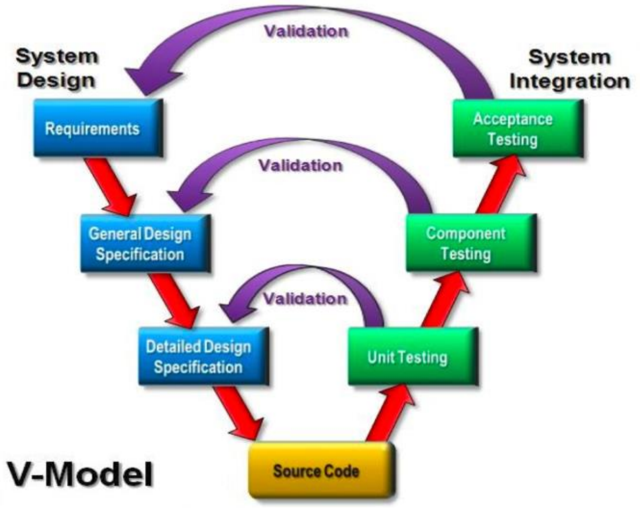
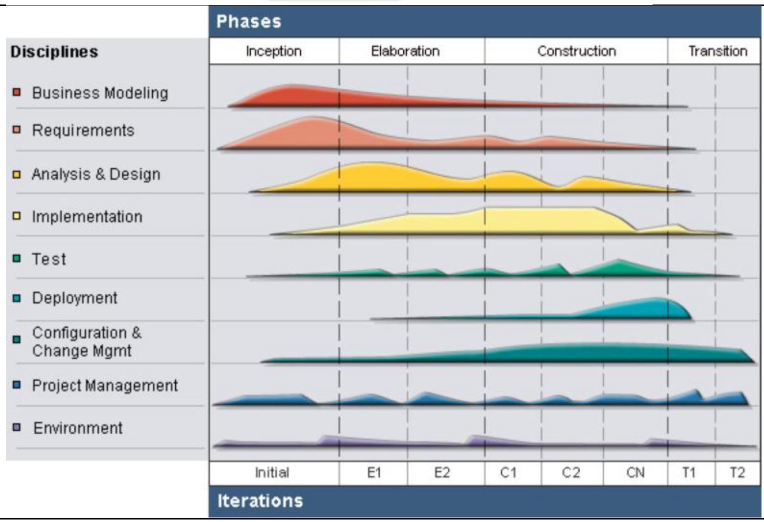
Performance Tactics	• Event: single or stream - Message arrival, time expiration, significant state change, ...
	• Latency - Time between arrival of an event and the generation of a response to it
	• Event arrives – System processes it or processing is blocked
	• Resource Demand
	  ○ Increase Computation Efficiency -> Better algorithm, cache data (Proxy)
	  ○ Reduce Computational Overhead -> Simpler protocols, data compression
	  ○ Manage Event Rate -> avoid oversampling
	  ○ Control Frequency of sampling -> perhaps by queuing requests
	• Resource Management
	  ○ Introduce Concurrency -> processes, threads, load balancing
	  ○ Maintain Multiple Copies -> copy-on-demand (proxy), caching
	  ○ Increase available resources -> faster, additional processors, more memory, faster network
	• Resource Arbitration
	  ○ Scheduling Policy -> FIFO, fixed priority, dynamic priority, static/pre-emptying scheduling

Software Architecture Documentation and Analysis

ATAM – Architecture Tradeoff Analysis Method (Architectural Evaluation)

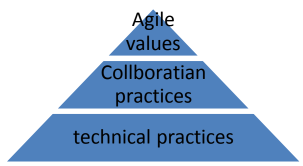
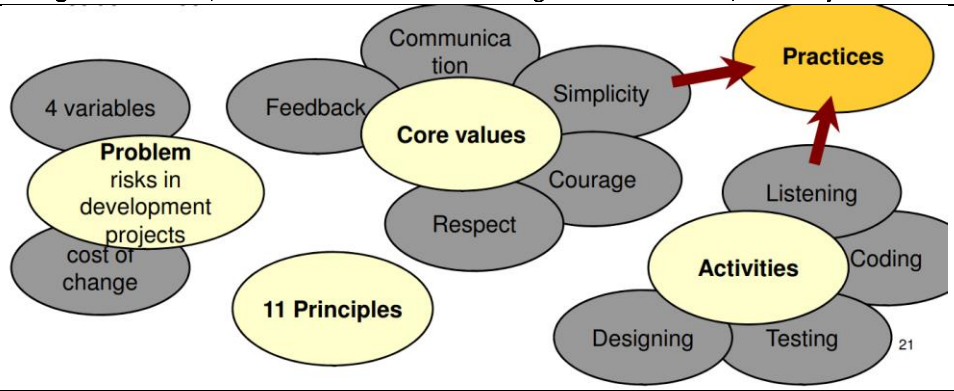
Why?	Architecture tells about system properties Architecture drives the software system Good evaluation methods
When?	Early in the lifecycle -> to be cost-effective
Costs	Staff time (accomplishments, training
Benefits	Financial, recorded rationales for decisions, early detection of problems, validation of requirements, improve
Participants	Evaluation Team (3 to 5 people, competent, unbiased, no hidden agenda) Project decision makers (architect, project manager, customer) Stakeholder (developer, tester, integrators, maintainers, performance engineer, users, system builds, …)
Outputs	Documentation, business goals, quality requirements, mapping of decisions to quality requirements, risks, non-risks, prioritization of risks, better understanding
Performance	**Phase 0: Preparation** Project representative’s brief evaluators **Phase 1 and 2: Evaluation** 1-1: present ATAM 1-2: present business drivers (functions, constraints, business goals, stakeholders, architectural drivers) 1-3: present the current architecture (1h) (context diagrams, component/behavioral/deployment views) 1-4 catalog architectural approaches (architectural patterns, style, and tactics) 1-5: generate quality attribute utility table 1-6: examine the highest ranked scenarios, evaluate architectural approaches, identify risks and non-risks -> time-out, gather more stakeholders for phase 2 2-7: brainstorm and prioritize scenarios (utility table as input) 2-8: analyze architectural approach 2-9: present results (documentation, scenarios, utility table, risks, non-risks, sensitive points) **Phase 3: Follow-up** Evaluation team produces and delivers written evaluation report
Summary	Architecture analysis method Based on evaluating quality scenarios Helps mitigate architectural risks

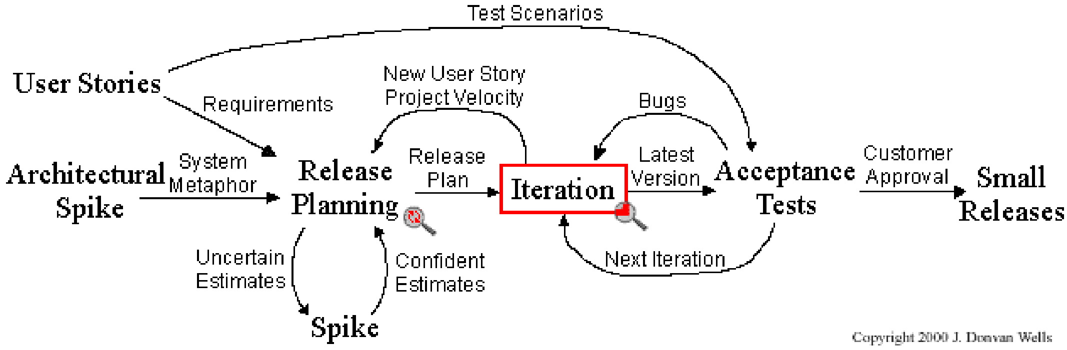
Process and Architecture

<table>
<tr><td>Process</td><td colspan="2">User needs -> Requirement -> Design -> Implement -> Test/Document -> Install/Deploy -> Check</td></tr>
<tr><td>Waterfall Model</td><td>
</td><td>sometimes no backward arrow,
but in paper of Royce are they drawn.

</td></tr>
<tr><td>V-Modell</td><td>
</td><td></td></tr>
<tr><td>RUP
Rational
Unified
Process</td><td>
</td><td>Principles
Risk as primary driver, Architecture centric, Iterative and incremental
Each phase ends with a milestone
Phases
Inception: Lifecycle objectives (scope!)
Elaboration: Lifecycle Architecture
Construction: Initial operational capability
Transition: Product Release
Issues
Heavy: Lots of documents (in UML), roles and process specification</td></tr>
<tr><td>Extreme Programming XP</td><td colspan="2">Lean => Less documentation. Delivers capabilities quickly
Belief that architecture will gradually emerge (because of YAGNI and BDUF)
YAGNI: You Ain't Gonna Need it
BDUF: No Big Design Up Front</td></tr>
<tr><td>Agile Architecture</td><td colspan="2">A system or software architecture that is versatile, easy to evolve, and easy to modify, while resilient enough not to degrade after a few changes
An agile way to define an architecture, using an iterative lifecycle, allowing the architectureal design to tactically evolve over time</td></tr>
<tr><td>The zipper model</td><td colspan="2">Architecturing design and building the system must go hand in hand
e.g. each second sprint is an architecture sprint</td></tr>
<tr><td>SODA</td><td colspan="2">Software Development Process @ HSLU</td></tr>
</table>

Agile Software Development

Agile Manifesto and eXtreme Programming

Agiles Manifest	**Individuals and interactions** over processes and tools **Working software** over comprehensive documentation **Customer collaboration** over contract negotiation **Responding to change** over following a plan While there is value on the right, we value the items on the left more.  **Scrum** **XP**
Agile Principles	Our highest priority is to **satisfy the customer** through early and **continuous delivery** of valuable software. **Welcome changing requirements**, even late in development for the customer's competitive advantage. Deliver **working software frequently** (couple of weeks/months), with a preference to the shorter timescale. **Business** people and **developers** must **work together daily** throughout the project. Build projects around motivated individuals. Give them the environment, support and trust they need. The most efficient and effective method of conveying infos to and within a team is **face-to-face conversation**. Working software is the primary **measure of progress**. Not documentation. Promote **sustainable development**. Sponsors/developers/users should maintain a **constant pace indefinitely**. **Continuous attention** to **technical excellence** and good design enhances agility. **Simplicity** - the art of maximizing the amount of work not done - is essential and elegance. The best architectures, requirements, and designs emerge from **self-organizing teams**. At **regular** intervals, the team **reflects** on how to get more effective, then adjusts its behavior accordingly.
XP eXtreme Programming	 The role of XP is to give us principles and practices in order to deal with the risks!
Problem	**4 Variables:** Time/Resources/Quality (external forces – customer/manager), Scope (our control variable) **Cost of change**: slow rate
Core Values	**Simplicity:** We will do what is needed and asked for, but no more. This will maximize the value created for the investment made to date. We will take small simple steps to our goal and mitigate failures as they happen. We will create something we are proud of and maintain it long term for reasonable costs. **Communication:** Everyone is part of the team and we communicate face to face daily. We work together on everything from requirements to code. We will create the best solution to our problem that we can together. **Feedback:** We will take every iteration commitment seriously by delivering working software. We demonstrate our software early and often then listen carefully and make any changes needed. We will talk about the project and adapt our process to it, not the other way around. **Courage:** We will tell the truth about progress and estimates. We don't document excuses for failure because we plan to succeed. We don't fear anything because no one ever works alone and. We adapt to any changes. **Respect:** Everyone gives and feels the respect they deserve as a valued team member. Everyone contributes value even if it's simply enthusiasm. Developers respect the expertise of the customers and vice versa. Management respects our right to accept responsibility and receive authority over our own work.
XP Practives	**The Planning Game:** Balance between business and technical considerations to estimate work load. Business people decide about: Scope + Priority + Composition of releases + Dates of releases Technical people decide about: Estimates + Consequences + Process + Detailed Scheduling **Small releases**: Every Releases should be as small as possible, containing the most valuable business requirements. The release has to make sense as a whole (no half-working features). **Metaphor**: Everybody on the team needs to have a common understanding for the system and a shared vocabulary. This applies for technical and non-technical people. **Simple design**: The right design for a software system is one that: runs all tests, has no duplicated logic, has the fewest possible classes/methods, “put in what need when you need it”, emergent, growing design. **Testing**: Any program feature without an automated test simply doesn’t exist. The tests become part of the system and allow the system to accept change. Development cycle (TDD) – Listen (requirements), Test (write tests), Code (simplest), Design (refactor). **Refactoring**: When implementing a feature, ask yourself if there is a way to improve the existing source code, so that implementing the feature is easier. Automated tests provide a safety-net for refactoring without fear.

	Pair programming: All product code is written by two people looking at one screen with one keyboard and one mouse. The programmer on the keyboard focuses on the current method, the other thinks about the broader context (refactoring, etc.). Pairs change frequently. **Collective ownership**: Anybody who sees an opportunity to add value to any portion is required to do so. Everybody takes responsibility for the whole of the system. Not everybody knows every part, but everyone knows something about every part. **Continuous integration**: Code is integrated and tested at least once a day (sometimes more), Build process must be automated, on a dedicated machine. Automated tests are run and detect problems early. **40 hours week**: Sustainable development. Effort should be spread out evenly. Extended periods of overtime have a negativ impact on productivity. Goal: Be fresh every morning, be tired and satisfied every evening. Not being in front of a computer does not mean forgetting about the system… taking a step back often leads to “Aha!” moments. **On-site customer**: A real customer must be physically with the team, available to answer their questions. Real customer = user who will use the system. The real customer does not work on the project 100% of his time, but needs to be “there” to answer questions rapidly. The real customer also help with prioritization. **Coding standards**: collective ownership + constant refactoring means that coding practices must be unified
Literatur	→ Clean Code, Clean Coder, Clean Architecture
Conclusion	There is no “magic” process that would work exactly the same way for every project, in every environment. Agile methodologies and XP describe core values and key principles that you need to integrate and customize in your particular context. Agile teams need to continuously reflect on their work. XP looks like it is less “formal” than traditional methodologies. But while there are certainly less roles, less workflows and less artifacts, XP requires a lot of discipline to work well.
Extreme Programming Project	

XP Game

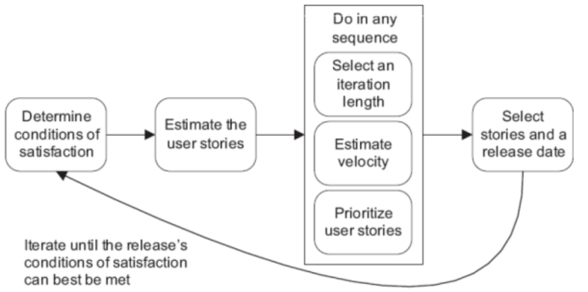
Description	The XP Game is a playful way to familiarize the players with some of the more difficult concepts of the XP Planning Game, like velocity, story estimation, yesterday’s weather and the cycle of life. Anyone can participate. The goal is to make development and business people work together in 1 team. Both will have the experience of performing the other role. It’s especially useful when a company starts adopting XP.
Outline	In real life Planning Game, development and business people are sitting on opposite sides of the table. Both participate, but in different roles. The XP Game makes the players switch between developer and customer roles, so that they understand each other’s behaviour very well. Some of the concepts in the Planning Game are difficult to grasp, for developers and for customers. This XP Game is a practical way to demonstrate how the rules of the XP Planning Game make up an environment in which it becomes possible to make predictable plans. After all, the easiest way to get a feeling for the way it works is to experience it. It differs from the classical Mousetrap or Coffeemaker Game in several ways: • The developers and customers are not separated. Everybody get to play the developer and customer role. • The stories are really very simply, everybody will understand them, • but they’re also very concrete. • We do a *real implementation*, with real, unambiguous acceptance tests, • but not a bit technical!!! (I guess everybody can inflate a balloon…) • There’s a small element of competition in it that makes it a really fun game to play.

Agile estimating and planning

Why do we plan?	Plans help us to know: Who works on the project during the period. Is the project on track to deliver the functionality the user needs. When will you be done. Organisations demand estimates (budget, marketing campaigns, product release date, training internal users).
How do we plan?	Create a coarse-grained long-term plan to know where the target is and a fine-grained short-term plan for the next week or month
Goals	Reduces risk, reduces uncertainty, supports better decision making, establishes trust, transport information)
plan vs planning	Plans are documents or figures, planning is an activity Agile planning shifts the emphasis from the plan to the planning.
Plans change	Agile plans often (and gladly) changed: During a project we learn new thinks from the customer / complexity
business value	Geld verdienen oder Geld sparen.
key idea	A project rapidly and reliably generates a flow of useful new capabilities and new knowledge. Aha Effekt.
Levels	*Strategy – Portfolio – Product – Release – Iteration – Day* (Release – Iteration – Day: *agile teams plan these levels*)
Estimate with Story poins	Story points are a relative measure of the complexity of a user story. Velocity is a measure of a team's rate of progress per iteration. Number of iterations = Total number of story points / velocity of the team.
Planning Poker	Everybody has card with number of Fibonacci, and estimate without an influence of others. Makes fun. Add cards like Coffee, infinit, or question mark.
User Stories	Describes a WHO, WHAT, and WHY scenario from user perspective. Delivers value to the user. Is small enough to estimate. Is accurate enough to be testable. A large user story is called an epic. A set of related user stories may be combined to a theme.
Deriving an Estimate	**Ask an expert**: Pro: Usually does not take long, Con: Less useful on agile projects **Analogy**: There is evidence that we are better at estimating relative size than absolute size. **Disaggregation**: Pro: Break a large story into smaller items. Cons: easy to go to fare.
	Read Reading: No Silver Bullet.

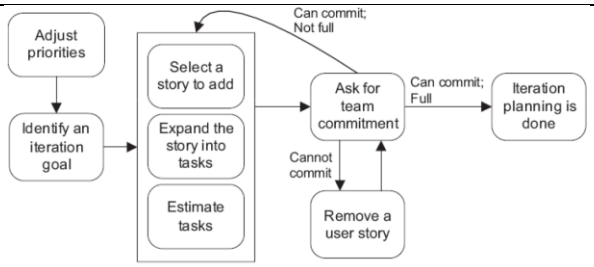
Release planning

Release planning is the process of creating a very high-level plan that covers a period longer than an iteration (3-9) months. What will be build by when.



Estimate User Stories: Let the team do the estimates (not the product owner). Don't spend too much time. Not Commitments.
Iteration length: Use 4 weeks iterations.
Estimate Velocity: Use historical values, run an iteration (or two/three). Make a forecast (with hours per day per week).
Prioritize User Stories: Product owner priorize features.
Select stories and a release date: Feature-driven project or Date-driven project.
Important: Update Release plan at start of iteration

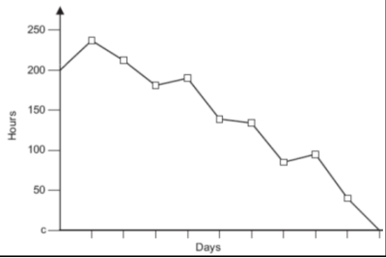
Iteration planning



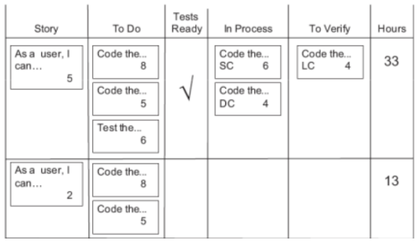
more detail than release plan
looks at the specific work of a single iteration
decompose user stories into tasks, estimate each task in terms of the number of ideal hours to complete
planning for value:
- prioritization of the User Stories
- financial value of having the features
- cost of developing (story points)
- new knowledge by developing the feature
- risk removed by developing the feature

Tracking

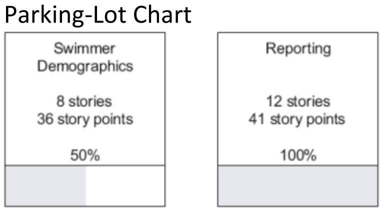
Burndown-Chart



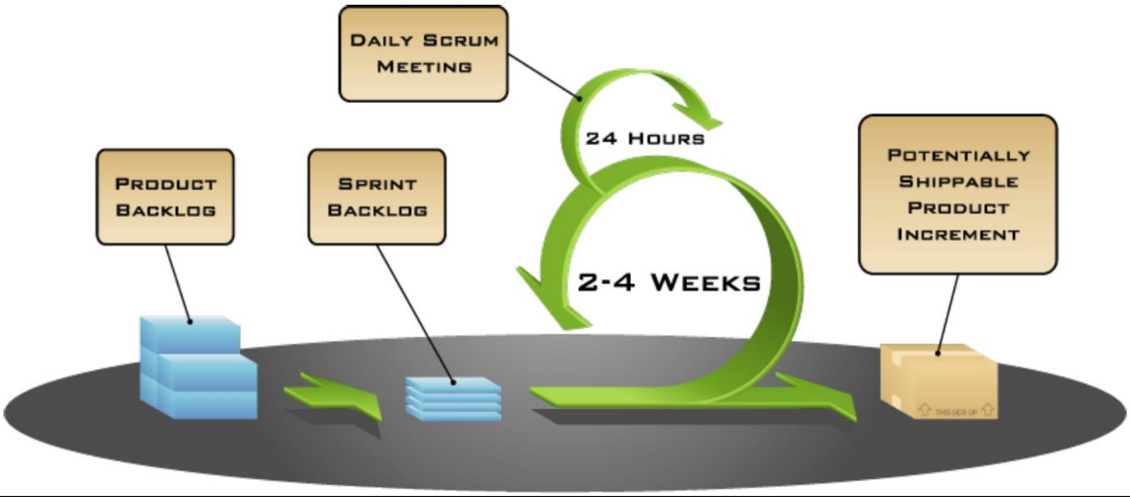
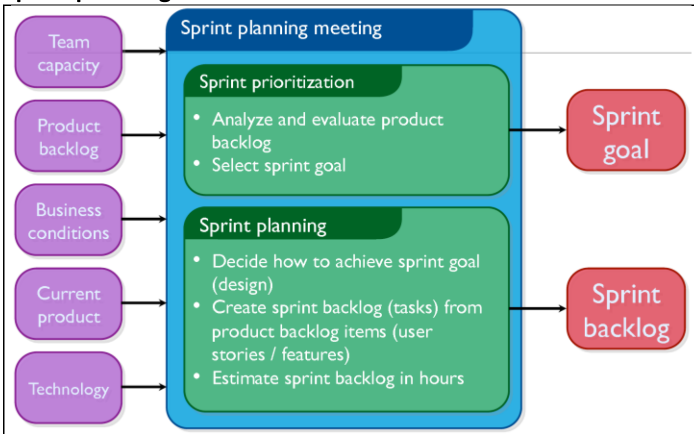
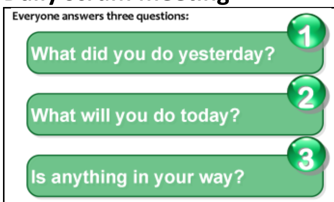
Task Board



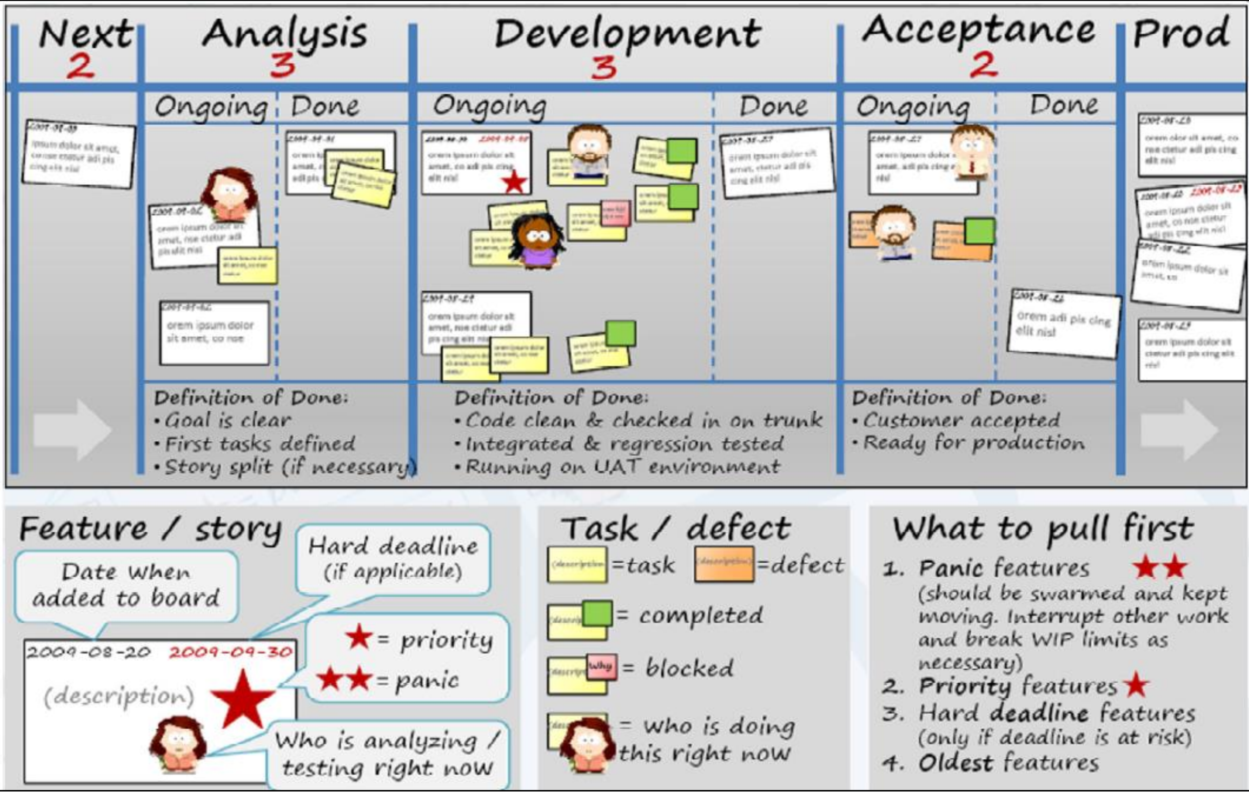
Parking-Lot Chart



Scrum

Ursprung	vom Rugby – Ball vor-/zurück hin/-her geben
Goal	Scrum is an agile process that allows us to focus on **delivering the highest business value in the shortest time**. It allows us to rapidly and repeatedly inspect actual working software (after every sprint). The business sets the priorities. Teams **self-organize** to determine the best way to deliver the highest priority features. Every two weeks to a month **anyone can see real working software** and decide to release it as is or continue to enhace it for another spring.
Characteristics	**Self-organizing** teams Product progresses in a series of month-long "**sprints**" Requirements are captured as items in a list of "**product backlog**. No specific engineering practices prescribed. Uses generative rules to create an agile environment for delivering projects. One of the "**agile processes**"
Process	
Roles	**Product owner**: defines features, decide release date and content, responsible for the ROI, prioritize features according to market value, accept or reject work results **Scrum master**: responsible for scrum values and practices, removes obstacles, ensure team functionality, enable close cooperation, shield team from external interferences **Team**: 5-9 people, cross-functional (tester/developer/designer), should be full-time, self-organizing
Ceremonies	**Sprint planning**  teams select items from the product backlog task are identified and estimated collaboratively (not by the scrum master) user stories are decomposed to tasks -> sprint backlog is created **sprint goal**: short statement what to focus **Sprint review:** whole team presents the world what it achieved during the sprint (2h preparation, no slides) **Sprint retrospective:** what is working and what is not, 15-30min, after every sprint discuss what they'd like to start/stop/continue doing **Daily scrum meeting**  15-min, stand-up, not for problem solving, invite whole world, only team members / scrum master / product owner can talk, helps avoid other unnecessary meetings these are not status for the scrum master they are commitments in front of peers
Artifacts	**Product backlog**: list of all tasks **Spring backlog:** individuals sign up for work they choose, work is never assigned, update estimations daily, any team member can add/delete/change sprint backlog **Burndown charts**: charts which indicates how well the sprint is progressing

Kanban

Origin	Original author: Taiichi Ohno (Inventor of Just-In-Time manufacturing 1995)
Principles	**Visualize the workflow**: Split the work into pieces, write each on a card and put on the wall **Limit work in progress (WIP)**: assigne explicit limits to how many items may be in progress at each state. **Measure the lead time** (average time to complete one item aka "cycle time"), optimize process
Kanban vs Scrum	Scrum is more prescriptive (more rules to follow) than Kanban
Kanban board	

Pro	Bottlenecks become clearly visible. Provides a more gradual evolution path from waterfall to agile software development. Provides a way to do agile software development without necessarily having to use time-boxed fixed-commitment iterations such as Scrum sprints. Tends to naturally spread throughout the organization to other departments.

Lean

Definition	Reduce the waste in a system and produce a higher value for the final customer
Principles	Iterative cycles, an implementation of the agile manifesto Feedback vs. Forecast
Seven Rules	**Eliminate Waste**: spend time only on what adds real customer value **Amplify Learning**: When you have tough problems, increase feedback **Decide as late as possible**: Evaluate various options, delay decisions until they can be made based on facts **Deliver as fast as possible**: Deliver value to customers as sonn as they ask for it **Empower the te**am: Let the people who add value use their full potential **Build integrity in**: Don't try to tack on integrity after the fact – built it in **See the whole**: Beware of the temptation to optimize parts at the expense of the whole

Vorträge

Evolving NoQSL Databases without downtime Nicola Lenherr und Florian Bühlmann	Problembeschreibung: Datenbankevolution: New requirements, split/merge objects, add fields, rename keys Wie bleibt die Datenbank immer verfügbar, und wie geht man mit bestehenden Daten um.	
Typen	**Relationale Datenbank** mit RDBMS (z.B. MySQL, Microsoft QSL Server, SQLite, ...) ACID, Festes Schmea	**NoSQL Datenbanken** viele verschieden Arten (z.B. Cassandra, Vertia, Duid) Ohne festes Schema, ACID nicht weit verbreitet
Ansäze	**Offline Eager Upgrade** 1) alle Applikationen herunterfahren 2) Updateskript 3) Applikationen upgraden	**Online Lazy Upgrade** 1. Applikationen updaten 2. Update einzelner Werte beim ersten Zugriff
Pro/Cons	Pro: klarer Datenbankzustand Cons: downtime	Pro: No downtime Cons: Viele if-else, performance impact
Lösung	z.B. KVolve Versionierung jedes Wertes, Funktion für das update (z.B. v1 -> v2), On-demand lazy Transformation (nur benötigte Werte updaten, ohne Abhängigkeiten) -> performance	

Paper: Enabling Agility Through Architecture

in brief	Should I take a certain action today in anticipation of increased benefit and reduced cost in the future?
Conclusion	Reliable agile software development is only possible when coupled with Architectural Agility.

Vortrag 05.04.2018

Modularity	Challenges: Cooperation, consistence, architecture
SAVI	System architecture virtual integration Architecture centric, one repository, component-based framework