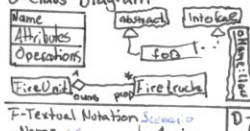
Models

- abstract, simplify, describe
- Object M: structure
- Functional M: functions
- Dynamic M: react ex. events

O-Class Diagram



F-Textual Notation

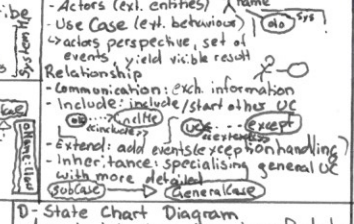
- Name ✗
- particip. Actors ✗
- Entry Conditions
- Flow of Events ✗
- Exit Conditions
- incl. extends
- Quality Requirements
- not functional!

Scenario
- As-is
- visionary
- evaluation
- test
- training
- step-by-step instruct.

F-Use Case Diagram

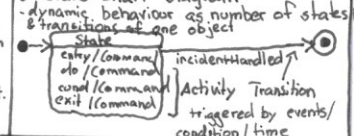
- Actors (ext. entities)
- Use Case (ext. behaviour)
  - actors perspective, set of events, yield visible result

Relationship
- Communication: exch. information
- Include: include/start other UC
- Extend: add events/exception handling
- Inheritance: specialising general UC with more detailed



D- State Chart Diagram

- dynamic behaviour as number of states & transitions at one object

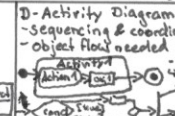


D- Sequence Diagram

- Communication among Objects by sending messages
- horizontally: participating objects, vertically: time
- Iteration noted by ‚loop' label / cond on first true
- Conditions noted by ‚alt' label - else on first false
- Creation: arrow to upper corner, Destruction: X on LL

D- Communication Diagram

- Object Diagram as base
- extended by sequence of messages (numbered)



D- Activity Diagram

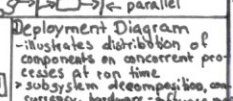
- Sequencing & coordination of low level behavior
- object flow needed for coordination

Component Diagram

- illustrates dependencies among components (subsystems) at design time/compilation/run time
- Compon. offer/require interfaces
- Ports are access-points for environment

Deployment Diagram

- illustrates distribution of components on concurrent processes at run time
  - subsystem decomposition, concurrency, hardware-software mapping



Architectural Styles

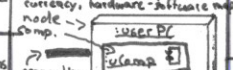
Architectural styles can be used as patterns for subsystem decomposition
↳ identification of subsystems/their services collaborations & relationships
- software architectur is instance of A. Style

Layered Architecture

- layers are subsystems providing services
  - only depend on lower layers
  - no knowledge about higher layers
- compile time: depend; run time: ‚call'
- Closed (opaque): only call Operation from THE next layer below X-Performance, V- Maintain
- Open: call Operations from any layer below ✓Performance, X high coupling, flexibility

Repository

independent subsystems work together on shared data structure (repository)
- Subsystems access & modify data in repository
- subsystems are loosely coupled



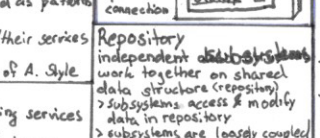
Pipes & Filters

- two types of subsystem
- Filter: perform processing steps
- pipe: connects filter (2) in/out

Client - Server

- one/more servers provide services to clients (subsystem)
  - client knows server interfaces
  - server don't knows client interfaces
- response immediate
- end user interacts with client

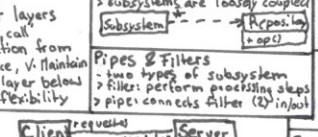
✓ Portability, Performance, scalable
X no Peer-to-Peer, request-response pool



Model-View-Controller

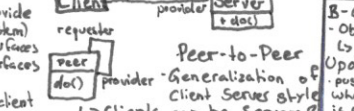
- decouples data access (entity) and data presentation (boundary) hierarchically
- View: subsystem containing boundary obj.
  - displaying information to user
- Model: subsystem containing entity objects
  - application domain knowledge
- Controller: subsystem mediating between View & Model
- Notifications
  - pull: View polls changes (in det. intervals)
  - push: Model pushes update, whenever change occurs

✓ multiple syncronized Views, exchangeables Views & Controller
X increased complexity, potential for excessive number of updates, high coupling



Mock Object Pattern

- used for testing
- mock object returns hard coded values

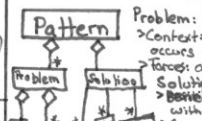


T. Sef

Patterns (Design Patterns)

Algorithm
- method for solving problems with finite sequence of well-defined instructions
- initial s. -> successives. -> final state

Pattern
- describes reoccuring problem and the core of its solution
- solution can be reused without ever doing the same thing twice



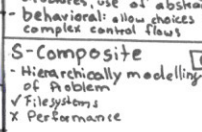
Problem: issue, which shall be handled
- Context: setting, in which the problem occurs (set of preconditions)
- forces: aspects of the problem

Solution: solution to the problem
- benefit: positive aspects associated with the solution
- Consequence: additional aspects, which need to be taken into account
- Follow-on-Problems: potential problems arising from use of the pattern

Types:
- Structural: reduce coupling, encapsulate complex structures, use of abstractions
- Behavioral: allow choices between algorithms, model complex control flows
- Creational: make system independ. from the way of object creation

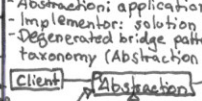
S- Composite

- Hierarchically modelling of problem
- ✓ Filesystems
- ✗ Performance



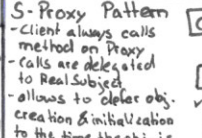
S- Bridge Pattern

- as design decisions are made at design time
  - ↳ Bridge Pattern delays decision to runtime -> delays binding between interface & subclass to start-up time
- Abstraction: application domain
- Implementor: solution domain
- Degenerated bridge pattern: no application domain taxonomy (Abstraction side)



S- Proxy Pattern

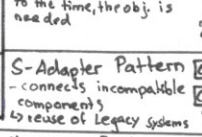
- Client always calls method on Proxy
- calls are delegated to RealSubject
- allows to defer obj. creation & initialization to the time, the obj. is needed



✓ if obj. is not used, it's not instantiated, Access control, Proxy on different nodes (hardware) decreases communication

S- Adapter Pattern

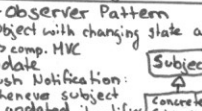
- connects incompatible components
  - ↳ reuse of legacy systems



B- Observer Pattern

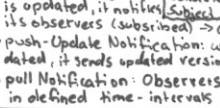
- Object with changing state and multiple Views
  - ↳ comp. MVC

Update
- push Notification: whenever subject is updated, it notifies its observers (subscribed) -> observers update their state
- push-Update Notification: whenever subject is updated, it sends updated version to all subscribed observers
- pull Notification: Observers call for current state in defined time-intervals



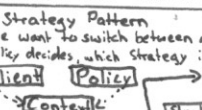
B- Strategy Pattern

- we want to switch between algorithms at run-time
- Policy decides, which Strategy is best in given context



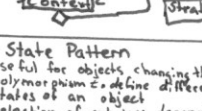
B- State Pattern

- useful for objects changing their state at runtime
- polymorphism & define different behaviors for different states of an object
- selection of subclass depends on state of object



S- Façade

- internal subsystem hidden by Façade
- Façade: set of public operation
- reduced complexity
- ✗ lower performance



Software Development

4. Requirements Elicitation
1. Analysis
3. System Design
2. Object Design
5. Implementation
6. Testing
7. Delivery & Maintenance

Waterfall Model

Application Domain

environment in which System is operating

Solution Domain: technologies used to built the System

Requirements engineering

defines requirements of system under consideration from problem statement
- Result: elicitation -> understood by customer/user -> requ. specification
- Analysis: -> understood by a developer -> analysis model, technical specification

Requirements Elicitation
- identify actors
- identify scenarios
- identify relationships among use cases
- identify nonfunctional requ.

Analysis: object & dynamic model

Requirements

Feature or Constraint for the system that must be satisfied to make client accept it
↳ What to achieve - not how

Types of Requirements Elicitation
- Greenfield engineering: requ. triggered by user needs
- Re-engineering: re-design of existing system, triggered by technology / new market needs
- Interface Engineering: service for existing system, triggered by technology / new market needs

Types of Requirements
- Functional: functionality + exch. interfaces
- Nonfunctional: Quality ORPS
- Constraints (pseudo requ.): legal, operating, ethical

Requ. must be defined precisely and measurable

Hardware Software mapping

- subsyst. can be realized with hardware
- object model to hard-/software
  - connol -> processor (speed!)
  - entity -> memory (enough)
  - boundary -> input/output devices (response!)

Caution to externally imposed hard- and software constraints

Component Diagram

dependencies between components (design, compilation and runtime)

Deployment Diagram

distribution of components on hardware (runtime)

Software Control

- Implicit: rule based systems, logic programming
- explicit: centralized / decentralized control
- event: dynamic behaviour is in a single object knowing all other objects (fork)
- concurrent: dynamic behaviour and knowledge is distributed (stair)

Object Design Activities

1. Reuse: identification of existing solutions
   - inheritance, patterns, off-the-shelf
2. Interface specification
   - precise description of each class interface
   - address need for reusable and extendable designs (flexible designs)
3. Object Model restructuring
   - transform object model to increase understandability & extensibility
4. Object Model optimization
   - transform to adress performance criteria

Purpose of Object Design

prepare system model/implementation by bridging object design gap based on system design decisions
↳ optimize system model

Reuse

- Types: design knowledge, existing classes, existing interfaces
  - ↳ composition / aggregation, inheritance
- Interface specification, as developers are class users / extended class / implementers
  - ↳ increase reusability by good documentation

Inheritance

Description of taxonomies by connecting hierarchically related objects
- Generalization: discovering super class
- Specialization: discovering sub class

Design Goals

- specify System Design Activities
- functionc. requ. + design methodology, design metrics, implementation goals
- design goals often conflict

↳ Design Goal Trade-Off

Subsystem Interface

- specifies interaction & information flow from and to subsyst. boundary under, object Design
- set of public operation provided by subsystem

API: specification of subsystem interface -> term should only be used during implementation

Subsystem & Service

Ssyst: collection of closely interrelated classes, associations, operations, events

Service: group of externally visible operations provided by subsystem (interface)

Coupling: dependency among subsystems

Cohesion: dependency among classes in subsyst

Goal: high cohesion, low coupl.

Persistent Data Managem.

- all classes of type entity need to be persistant
- persistency: lifetime beyond single exch. execution
- realization via file system (multi reader, single writer)
- realization via database system (concurrent readers/writers)

Concurrency

- address runtime requirements
- inherently concurrent objects: can receive events at same time without interacting
- can be assign. to separate threads of control
- Thread of Control: path through set of states where only one object is active after finance condit.!

Fault Tolerance

- exception handling
- modular redundancy

Limited black box testing knowledge on internal functionality in black-box

Horizontal Integration Risks

- higher complexity -> difficult integration
- later integration -> higher risks for occurance of unexpected failures

↳ Vertical Integration scenarios / user stories used as integration drivers
✓ always executable version available, good overview

Subsystem Decomposition

- technique to master complex
- to divide and conquer
- identific. of subsystems, services & relationships
- functional decomposition: hierarchically decomposed into functions
  - functions spread over system
- difficult code maintain:
- object-oriented decomp.:
  - decomposed into classes
  - focus on functional requ.
  - find corresponding use cases
  - identify participat. objects
  - start object model with participating objects

From Analysis to System Design

1. Identify Design Goals: nonfunc. requ., trade-offs
2. nonfunctional requirements
3. Subsystem Decomposition: layers vs. partitioning, architectural styles, cohesion & coupling, functional model
4. Identify Concurrency: identification parallelism & dynamic model
5. Hardware/Software mapping: identific. of nodes, special purpose system, buy vs. build, network, object model
6. Persistent Data Management: store persis. data, data-base vs. file system, object model
7. Global ressource handling: access control, security & dynamic model
8. Software control: monolithic, event driven, concurrent processes, dynamic model
9. Boundary conditions: initialization, termination, failure, functional model

Global Resource Handling (AC)

Access (control: different actors have different rights to functions & data
- functional model (analysis)

Access-Matrix: rows = actors, columns = classes, access-rights = matrix entries
- Global access table: every entry as triple (actor, class, operation)
- Access control list: List of permissions for an affected class (invoked class)
- capabilities: List of permissions in actor (involving class)

Boundary Conditions

- Initialization: bringing the system from non-initialized to steady state
- Termination: resources are cleaned up and other systems are notified about termination
- Failure: bugs, errors, external problems

Boundary conditions are modeled using use cases (actor: admin)

Specification & Delegation

- Specification: subclassing from specification (abstract class)
- delegation: catching operations and sending to other object, where it is already implemented

✓ A calls receiver B, B delegates to delegate C

Incremental: improves your product
↳ add onto sth"
Iterative: improves your product
↳ re-do sth
Adaptive: improves reaction to changes
↳ reach to changing requirements

Software Engineering

collection of techniques, methodologies, tools to construct software

- modeling
- problem solving activity
- knowledge acquisition activity
- rationale choice activity

Problem Statement: description of problem addressed by the system
- current situation
- functionality
- dynamics
- physical data
- environment interfaces

Change
- release management
- delivery (customer)
- SLC-Model
- rationale management
- project management

Complexity
- modeling
- interdependency elimination
- analysis & design
- implementation
- testing

Systems: organized set of communicating parts
Model: abstractions describing systems
View: depicts selected aspects of a model

Requirements Validation
- Correctness, Realism
- Clarity, Completeness
- Consistency, traceability

Inheritance for code reuse to implementation
Subclassification of complete lasting to types (interchangeable) specification

Testing

- Failure: any deviation of observed behaviour from specified behaviour
  - ↳ informally, exash
- Error: state, such that further processing by the system can lead to failure
- Fault: mechanical/algorithmic cause of an error -> informally, bug
- Validation: activity of checking definitions
  - ↳ often needs of external stakeholders
  - ↳ Are you building the right thing"
- Verification: meeting internal regulations, requirements, specification & imposed conditions
  - ↳ Are you building the thing right?"

Fault Avoidance

- Methodology: to reduce complexity
- Configuration Management: prevent inconsistency
- Verification: to prevent algorithmic faults
- Reviews: to identify faults visible in design

Fault Detection

- Testing: activity to provoke failures in a planned way
- Debugging: find and remove causes (faults) of observed failure
  - ↳ Monitoring: deliver information about state & behaviour (debug - media)

System Testing

- Functional Testing: Validates functional requirements
  - ↳ test cases derived from requirements analysis, reuse of unit test -> black box
- Structure testing: validates subsystem decomposition
  - ↳ test all I/O parameters for each component (all paths in system design)
- Performance testing: validates non-functional requir.
  - ↳ try to violate nonfunc. requir. by overload, unusual order of execution, time spent in use cases

Test Model

Consolidation of all test related decisions/components into one package
- Test drivers: program executing the test cases
- Test case: function derived from UC which needs test
- Input data: data needed for the test cases
- Oracle: expected output data
- Test harness: framework that allows testing under varying conditions while monitoring behaviour & outputs of System under Test (SUT)

Software Lifecycle Modelling

- Software Lifecycle: Set of activities and their relationships to each other to support the development of a software system.
- SL Model: Abstraction of the process of developing software for the purpose of understanding, monitoring, controlling

Agile Models

- SCRUM Meetings
  - Project kickoff meeting: create & prioritize product backlog
  - Sprint planning meeting: create Sprint backlog
  - Daily SCRUM meeting: share status, promises, impediments - short stand-up meeting
  - Sprint review meeting: demonstrate realized sprint backlog items to Product owners
  - Sprint retrospective: evaluate previous Sprint and learn from it

Artifacts:
- Product backlog: List of requirements for the whole product
- Sprint backlog: List of requirements for one iteration (sprint)
- Potentially shippable product increment: release to product owner as result of Sprint

Roles:
- Product Owner: defines product, responsible for results
- SCRUM master: resolves impediments, monitors & controls the process
- Development Team: self-organized, cross-functional, realizing prod. increment

Unit Testing

↳ Confirm correctness & intended functionality
- individual components (classes/subsystems) are tested
- Based on knowledge
  - ↳ Analysis knowledge about functional requirements (black box)
  - ↳ Design knowledge about structure, algorithms, data (white box)
  - ↳ Implementation knowledge about algorithms & data structures (white box)

JUnit Testing -> Framework to perform automated unit-tests.

Annotations
- @Test: identifies test method
- @Test(expected=Exception.class): Test for throwing Exception
- @Test(timeout=100): Test fails, if taking longer than 100ms
- @Ignore(String): ignores test-case and prints String instead
- @Before: method performed before every test
- @After: method performed after every test
- @BeforeClass: method performed once before all tests
- @AfterClass: method performed once after all test

Assertions
- assertTrue(pred): checks if predicate evaluates to true
- assertFalse(pred): checks if predicate evaluates to False
- fail(String): let the test fail - prints String
- assertEquals(message, expected, actual [, tolerance]): checks if expected and actual are equal [with tolerance] „ alias gleiche"
- assertNull[Not Null](message, object): checks if object is [not] null
- assert[Not] Same(message, expected, actual): checks if both variables [not] refer to the same object „ das selbe"

Integration Testing

System is viewed as a collection of subsystems, which's interfaces and interactions are tested by different strategies
- Big Bang Integration: everything at the same time, one large test
- Bottom up: starting with subsystem on lowest layer & repeating integration, testing upwards
  - ✓ no doubles, useful for OO systems & strict performance req.
  - ✗ Test drivers needed, UI is tested last
- Top down: starting at top & go down (opposite Bottom up)
  - ✓ test cases = functional requirements, no test driver needed
  - ✗ doubles needed (difficult, poh. high number), some interfaces are not tested separately
  - ↳ modified top down: test each subsystem individually before
  - ✗ doubles & test drivers needed

Acceptance Testing

- Tests performed by client/end user -> majority of bugs in software is found by them
- Alpha-Test: client test at developers site with developer ready to fix bugs
- Beta-Test: client test at client site w/o developer
  - ↳ realistic workout in target environment

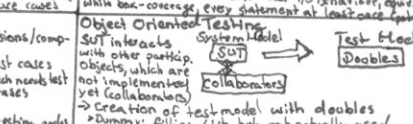
Static analysis

read source code & present to others
- ↳ compiler warnings, IDE, plugins/frameworks (Spotbugs)

Dynamic analysis: black box - test I/O behaviour, equiv. partitioning
white box - coverage: every statement at least once (testing)

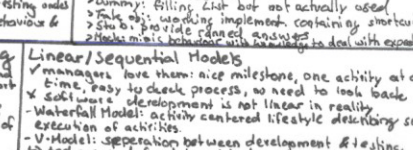
Object Oriented Testing

SUT interacts with other parts/objects, which are not implemented yet (collaborators)
- ↳ creation of test model with doubles
  - Dummy: filling list but not actually used
  - Fake obj: working implement. containing shortcut
  - Stub: provide canned answers
  - Mock: mimics behaviour with knowledge to deal with expect.



Linear/Sequential Models

- ✓ managers love them: nice milestone, one activity at a time, easy to check process, no need to look back
- ✗ software development is not linear in reality
- Waterfall Model: activity centered lifestyle describing sequential execution of activities.
- V-Model: sequence between development & testing
  - ↳ test cases defined in related constructive phase



Iterative Models

cheap

- ✓ simple, plannable ✗ difficulties when dealing with frequent changes

Spiral Model: iterative model with 4 major activities

1. Determine objectives, alternatives & constraints
2. evaluate alternatives & identify risks -> prototype
3. develop series of prototypes for identified risks (waterfall model)
4. evaluate results if risks are resolved and plan next iteration
   - ↳ unresolved risk -> terminate

✓ iterations: concept of operations, software requirements, software product design, detailed design, code, unit test, integration and test, acceptance test, implementation
✓ risk handling, flexibility in requirements
✗ expensive (complex), difficult to plan (unpredictable)

Unified Process: iterative & incremental SL Model

phases with several iterations

- 1. stage: engineering
  - ↳ inception, elaboration
  - ↳ small team, unpredictable
  - ↳ focus on design & synthesis
- 2. stage: production
  - ↳ construction, transition of deployment
  - ↳ larger team, predictable
  - ↳ focus on construction, test & deployment

✓ several workflows performed in parallel
✓ iterations create informal, internally controlled artifacts
✓ phases create formal, stake-holder approved artifacts

Manifesto for Agile Software Development

- Individuals & interactions > processes & tools
- working software > comprehensive documentation
- customer collaboration > contract negotiation
- responding to change > following a plan

Software Configuration Management
Manage evolving software system and control changes in the system (coordination)
- set of management disciplines within software engineering process to develop baseline
> baseline: formally reviewed and agreed product, basis for further development
- initiating, evaluating, controlling change to software products during and after a project
- contains activities & roles

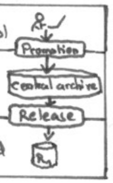
Activities
- Configuration item identification: set of evolving components, which shall be tracked
- change management: handling, approval & tracking of change (request)
  > in accordance with change policies
- promotion management: creation of internal versions for other developers
- branch management: management of concurrent development
- build and release management: creation of versions for users/clients (publish/release)
- variant management: management of coexisting versions in different variants

Roles
- Configuration manager: identifying configuration items, defining procedure for promotion/releases (policies)
- change control board member: approving/rejecting change requests
- Developer: create promotions, check changes & resolve conflicts
- Auditor: selection & evaluation of promotion for releases

Merge Conflict
Occurs when VCS is not able to merge developers push with remote repositories version
1: pull conflicted version, 2: resolve conflict,
3. push deconflicted version
- best practices: commit & push often

IEEE 828
- Programers directory: holding newly created/modified software entities
- Masters directory: manages current baseline(s), changes must be authorized
- Software Repository: archive for baselines published for general use -> release

- ,Dynamic Library'
- completly under control of one programmer
- ,controlled Library'
- central directory of all promotions
- ,static library'
- externally released baselines



Version Control Systems
- allows collaborative work on configuration items
- stores different versions of configuration items (commit history)
- versions are stored in repository and can be checked-out into working copy

Peer-to-Peer VCS
- each developer fully mirrors the repository (example GIT)
✓ work offline, work incrementally, efficient context switch
x high learning curve, scaling issues, less administrative control

GIT
local working directory | staging area | local repo | remote repo
|— add —>
|— commit —>
|— push —>
<— fetch —|
<— merge —|
pull = merge + fetch

Branch Management
- Master: external releases (e.g. SCRUM product increment)
- Development: internal releases (baseline)
- Feature: incremental development & explorations

Continous Integration
development technique, where members of a team integrate their work frequently. Integrations are verified by automated build incl. execution of tests
✓ always an executable version of the system available, good overview of the project status, automatic regression testing
x costly

Regression Testing
- verify, that software changed or interfaced with other software still performs correctly
- Techniques: retest all, regression test selection (using dependencies), test case priorization
- When to execute?
  ↳ costly x

Build and Dependency Management
Maven: build and dependency management tool for java projects (pom.xml)
> stores libraries, frameworks, plugins in central repository -> simplified reuse

Release Management
- Continous delivery: teams keep producing valuable software in short cycles to ensure that software can be released at any time (manual)
- continuous deployment: every change passing automated test is deployed (automatic)
- continuos software engineering: capability to develop, release and learn in short cycles

Modeldriven Engineering
Methodology focused on the creation of domain models (for particular application domain)
Increase productivity through
- reuse of standardized models
- reuse of knowledge
- collaboration between developers working on the system
- standardized terminology
- identification of best practices in application domain

Model based Software Engineering
Application of modeling to support requirements, design, analysis, verification & validation

Model Transformation
- Refactoring: change made to internal structure of model (source code) to make it easier to understand & cheaper to modify
- Model Transformation: Input = model conforming to meta-model > output = an other model conforming to meta model
  ↳ e.g. improved by generalization
- XMI (XML metadata interchange) is common standard for exchange of models

JSON
serialization to JSON
```
{
  obj. "collision" : {
      "winningCar" : {
         "name" : "a"      attribute
      },
      ...
  }
}
```

Forward Engineering
- Models to Classes
  > every UML class should split into a Java interface & implementation class
  > attributes should be private
  > signatures of UML operations are mapped to signature of Java methods
  > UML inheritance can be mapped with inheritance (extends) or subtyping (implements)
- Models to StateCharts
  > public method for each event
  > variable state (enum)
  > write switch statement with cases for each of the states in UML state diagram
  > write an if statement for each guard predicate on the transitions to check if it has become true
  > update the state variable accordingly
- Models to Contracts
  > Contracts: constraints on a class to share assumption among users, implementors & extenders
  > Precondition, Postcondition, Invariant
  > OCL: Object Constraint Language
  > use assertions to ensure compliance with constraints
- Models to Table
  > UML class diagrams can be mapped to relational databases
  > Mapping of classes, attributes & associations
  - each class is mapped to a table
  - class attributes are mapped to columns
  - instance of class => row in table
  - many-to-many => new table
  - one-to-many => foreign key
  - inheritance is mapped horizontally (subclasses only) / vertically

Source Code Transformation
Change code to increase readability & simplicity

Reverse Engineering
Mapping source code to models
(see above „forward engineering")

Laws of Project Management
- Pareto principle (80:20)
- allowing frequent change may result in change exceeding progress -> limit change

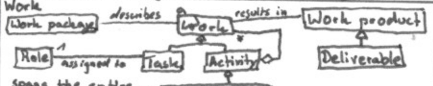
Project
- start date & duration
  ↳ limited time
- set of deliverables for the Client -> clear goal
- schedule: conception (formulate idea), definition (define problem), start (form teams), steady state (do work), termination (project finished, delivery)
- set of technical & managerial activities to produce & deliver deliverables
- resources consumed
- managed by project manager

Roles
define responsibility, assigned to people
- Project Manager: make sure goals are met, allocate/administer resources, maintains accountability
- Analyst: analyse application domain, create taxonomy of the domain abstraction
- System Architect: decompose system into subsystem, choose software architecture
- Tester: design and implement tests

Assignment of Roles
- one-to-one: ideal
- many-to-few: each project member assumes several positions, danger of over-commitment
- many-to-many: some people don't have significant roles, lack of accountability, Loosing touch with project

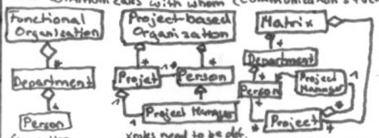
Problems in Assignment
- Incompetence
- useless roles
- increase of bureaucracy

Concepts for Mapping Roles to People
- Authority: ability to make binding decisions between people and roles
- Responsibility: commitment of a role to achive specific results
- Accountability: tracking a task performance to a specific accountable person
- Delegation: binding a tasked assigned to one person to another person,
  ↳ reasons: time management, expertise, training (Auth., not Respons.)

Work


spans the entire duration of project -> Project function

Project Organization
- who decides (decision structure)
- who reports to whom (reporting structure)
- who communicates with whom (communication structure)



Functional Organization:
✓ expertise
x bad communic.
↳ overlaping work probable
x major investment difficult

Project-based Organization:
x roles need to be def. during project start
x slow team assembling
✓ responsive, tailored for problem

Matrix:
x Double bass
x persons not used to each other
✓ rapid team assembly
✓ expertise as needed

Staffing
- Flat Staffing: all participants assigned at start of project
- Gradual Staffing: hiring people as needed (saving resources in early stage

- Reporting Structure: supports management in tracking project status > hierarchically
- Communication Structure: communication between teams along hierarchy not sufficient
  ↳ additional com. structure
  ↳ Team member who is interface for other teams wanting to communicate = liaison

Communication Skill
- Collaboration: negotiate requirements with ~~defined objectives & scope~~ client & other team members
- Presentation: present major parts of a system during review
- Technical Writing: write part of proposal/documentation
- Management: facilitate a team meeting, find compromises, negotiate conflicting demands

Communic. Event: information exchange with def. objective & scope
↳ scheduled, unscheduled
Communic. Mechanism: tool/procedure to deal with communication event
↳ synchronous, asynchronous

Functional Requirements
- relationship I\O
- response to abnormal situations
- exact sequence of operation
- validity checks on inputs

Quality Requirements
- Usability: ease of using system
- Reliability: Robustness & Safety
- Performance: resp. time, availability
- Supportability: Adaptability, Maintain.

Constraints
- Compliance to standards
  ↳ report format, audit tracing
- implementation requirements
  ↳ tools, programming languages
- Operations requirements
  ↳ administration & management of system
- legal requirements
  ↳ licensing, regulation, certification
> packaging, interfaces, laws

Examples
- Bridge: abstractions Transp. Mech (Bus, Car, Train), implementation Power Source (Gasoline, Electro)
- Proxy: access control
- Adapter: USB-C to USB-A
- State: State of country (emergency, normal), lawmaking
- Observer: Displays at fast-food self-service & kitchen gas station

IEEE 1074

Lifecycle Modelling	Pre-devel.	develop.	post-dev.	cross devel.	project management
Selection of Lifecycle model	concept exploration; system alloc.	requirements; design; implement.	installation; ope. & support; maintenance; retirement	V&V; config. manag.; documentation; training	project initiation; project monitoring; software quality manag.

Bridge / Strategy / State

Bridge	Strategy	State
start-up	runtime	runtime
Developer specifies implementation to be used in constructor	Policy	State

Multiple solutions/implementations of common ~~system~~ problem

Façade / Proxy

Façade	Proxy
possibility to hide operation & to change signatures	same operation as real object
Façade Classe always visible/available	Real subject doesn't have to be instantiated yet

Both: delegation, interface to existing implementation, layer on top

State / Strategy

State	Strategy
different result in terms of application domain	same results in terms of application domain
current state decides about implementation (int.)	external Policy decides about implementation

Implementation change at run-time
Both use delegation

```
@RunWith(EasyMockRunner.class)
public class Test {
  @TestSubject
  Discussion td = new Discussion();
  @Mock
  Comment c;  <- interface
  @Test
  public void tltC(){
    expect(c.save(td)
      .andReturn(true);
    replay(c);
    // do sth
    assert(sth)
}
```
import org.easymock.*;
import org.junit.*
import static org.
// das gleiche wie oben

Communication Events
- Problem Definition
- Project Review
- Client Review

formal vs. informal
event: scheduled vs. unscheduled
mechanism: