EIST Zusammenfassung 2018

1. **Lecture (problem solving, Software Engineering, System, Models, View)**
   - System Integration Problems
     - Algorithmic Fault
     - Bad communication
     - Bad interface specification
     - Bad subsystem decomposition
     - Physical impossibility
   - Software Engineering is..
     - Problem solving
     - Dealing with change
     - Dealing with complexity (abstractions..)
   - Abstraction
     - Ignore unessential details
     - Ideas distanced from objects
     - With a model
   - Models
     - Object Model (Structure of system, i.e. class diagram)
     - Functional Model (functions, i.e. Use case Diagram)
     - Dynamic Model (how systems reacts to external events)
     - = System Model
   - Why System development difficult ?
     - Ambiguous, unclear requirements, complex problem, difficult to manage, flexibility, hidden surprises
     - It's a problem solving activity
   - S.E use
     - Techniques (quicksort)
     - Methodologies (Object oriented Analysis)
     - Tools (Compiler)
   - Computer Science vs. Software Engineering
     - C.S. : develops techniques, proves, designs language, infinite time
     - S.E. : works in multiple application domains, limited time, changes occur
   - Software Engineering (S.E.) Definition
     - Has: techniques, Methodologies, Tools
     - Produces: high quality software system (budget, deadline, changes)
     - Challenge: deal with complexity + change
   - Deal with Complexity:
     - Modeling
     - Notations
     - Analysis + Design
   - Deal with Change:
     - Release Management
     - Delivery
     - Software Lifecycle Model
     - Management
     -

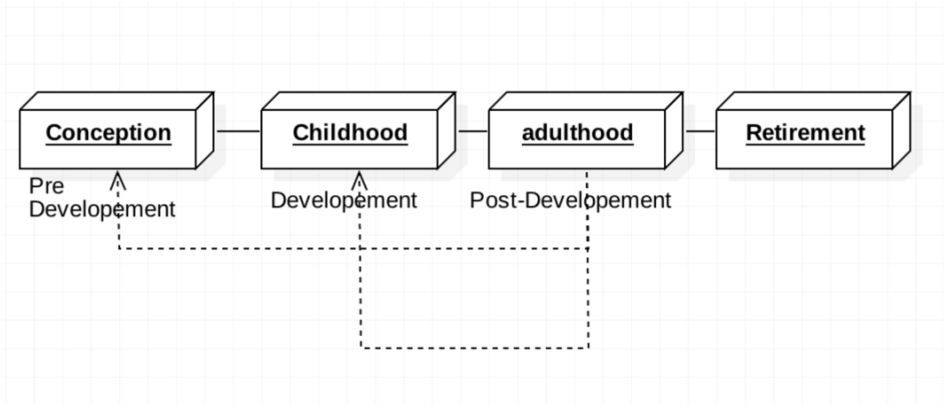
- Abstraction:
    - Concept: name (watch)+ purpose (measure time) + Members (all watches)
    - Phenomenon: An object of the abstraction
- System-Model-View
    - System: organized Set of communicating parts with a purpose
    - Model: abstraction describing a system
    - View: shows selected aspects of a model

2. Lecture (Software Lifecycle, Problem Statement, UML, Analysis/Design/Implementation Example)

- Software Lifecycle:
    - Set of activities (Analysis, System Design..) and their relationships (Testing before implementation,…)
    - S.L. Model: abstraction representing development of software
    - Activities:

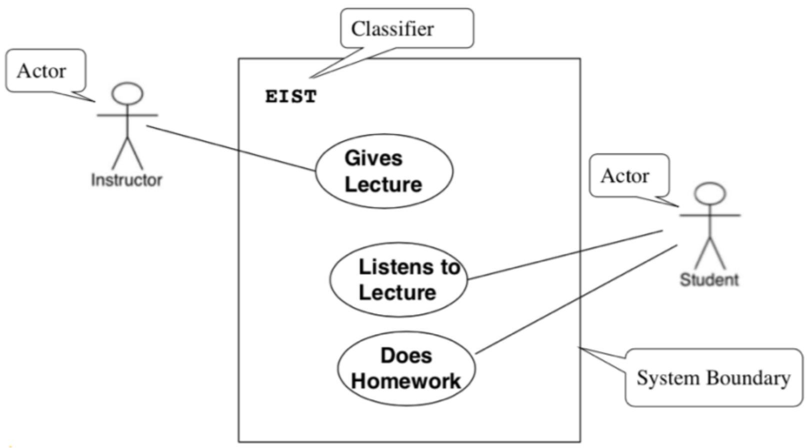
Requirement Analysis	What's the problem ?
System Design	What's the Solution ?
Detailed Design	Best Mechanism to implement
Implementation	Construction of solution
Testing	Is the Problem solved ?
Delivery	Can customer use solution ?
Maintenance	Are enhancements needed ?

    - Concept:

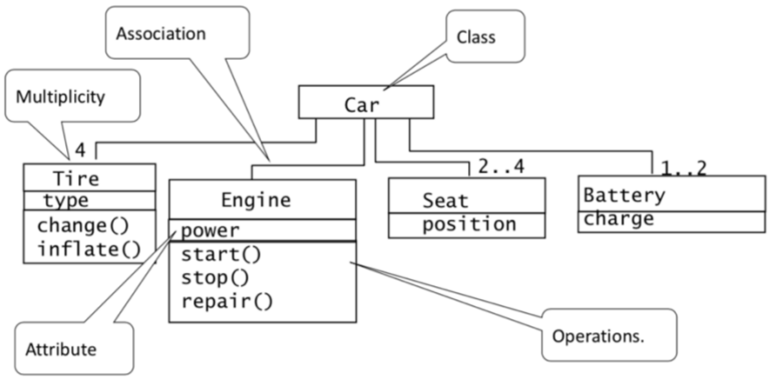


- Tailoring: adjusting Lifecycle Model to fit a project
- Controlling Software Development
    - Defined Process: Well defined inputs, all activities well defined, same output every time, Change can be ignored (don't deal with interference) – Waterfall Model
    - Empirical process: well defined inputs, changes are expected and are seen as opportunities, different outputs – Scrum Model
- Problem Statement:
    - Current situation
    - Functionality of new system
    - Environment

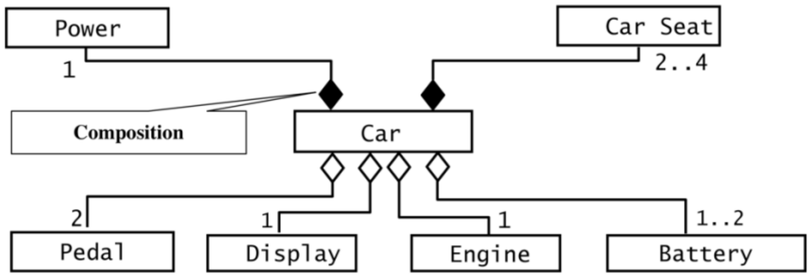
    - Deliverables
    - Delivery dates
    - Set of acceptance criteria (tests)
- UML (Unified Modeling Language)
    - Reduces complexity by abstracting
    - “high level” programming language
    - Use for communication in software project
- Application vs. Solution Domain
    - Application (Analysis): environment in which the system in operating
    - Solution (Design, Implementation): technology used to build system
- Use case Diagram:
    - Example:



    - Represents functionality of a system from users point of view
    - Actor: specific type of user
    - Use case: functionality provided by system
    - <<extends>> : additional behavior
    - <<includes>> : includes other functionalities
- Class Diagram:
    - Example:

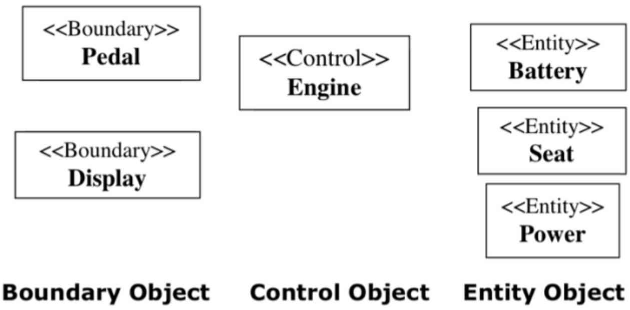


    - Represents structure of a system and relationship of its classes
    - Aggregation: Part-of Hierarchy (empty diamonds)
    - Composition: special form of aggregation, lifetime of instance controlled by aggregate (solid diamond), components don’t exist by their own

- 

3. Lecture (stereotypes, requirement elicitation, analysis)

- UML Stereotypes
  - To distinguish between different object types in a class diagram we use stereotypes
  - Boundary, control and entity object

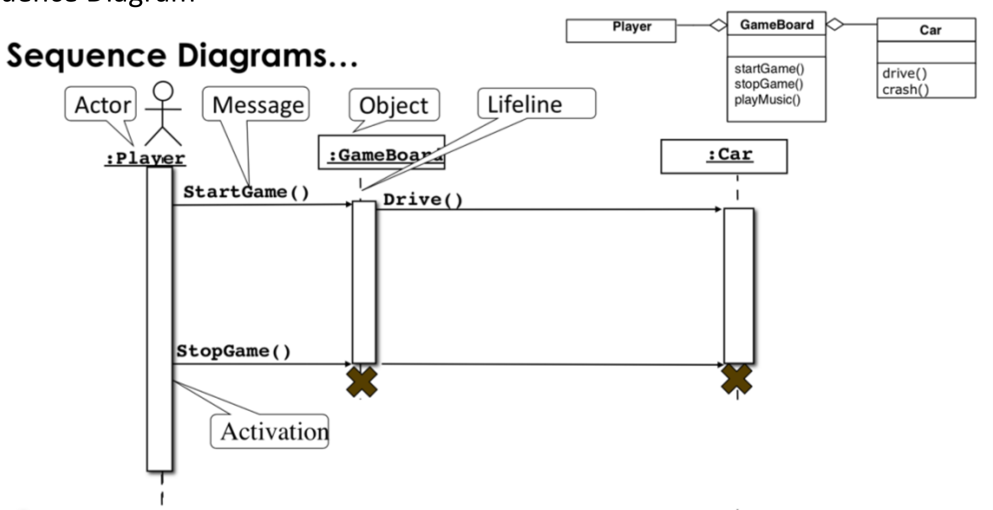


- Dynamic Modeling:
  - State Diagram: one diagram for each class with interesting dynamic behavior
    - To find new operations
  - Sequence Diagram Interaction between classes
    - To find new classes
- Requirement Engineering
  - Requirement elicitation: Definition of the system in simple terms
    - Result: Requirement specification
  - Analysis: Definition of System understood by engineer
    - Result: Analysis Model
  - Requirements Engineering: Combination of those two
- Requirement Types:
  - Constraints (Pseudo Requirements) (Non Functional)
    - Operating environment
    - Required standards
    - Legal requirements
  - Performance (Non functional)
    - Response time
    - Speed
    - Recovery time
    - Availability
  - Functional Requirements
    - Use Case model
    - Interaction with people/hardware/software
  - Quality requirements (Non functional)
    - Maintainability

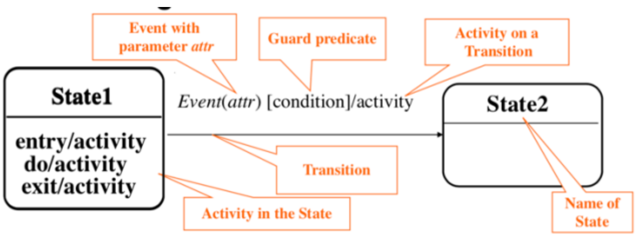
        - Security
        - Portability
        - Correctness
        - URPS (Usability, Reliability, Performance, Supportability)
- Describe Requirements:
    - Scenario:
        - a concrete description of single feature (linear event flow)
    - Use case:
        - set of scenarios of generic end users
- Requirements validation
    - Correctness (clients view)
    - Clarity (describes one system)
    - Completeness (every scenario described ?)
    - Consistency (consisting naming ?)
    - Realism (the model can be implemented ?)
    - Traceability

4. Lecture (Dynamic Modeling, System Design (1)-(2))

- Sequence Diagram

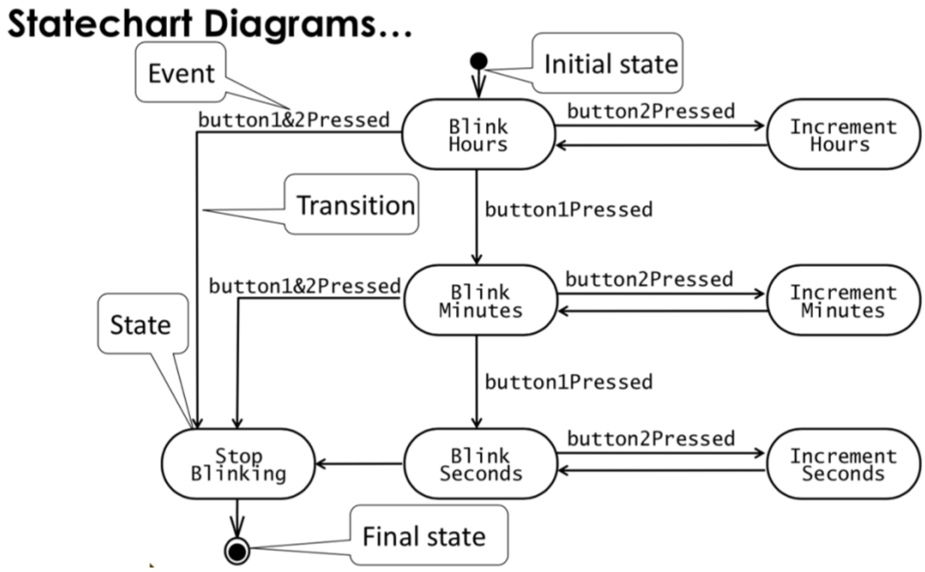


-
    - Represent behavior of a system as messages between different objects
    - Represent control flow
    - Can represent dataflow
    - Represent behavior in terms of interactions
- Statechart Diagram
    - Formal:

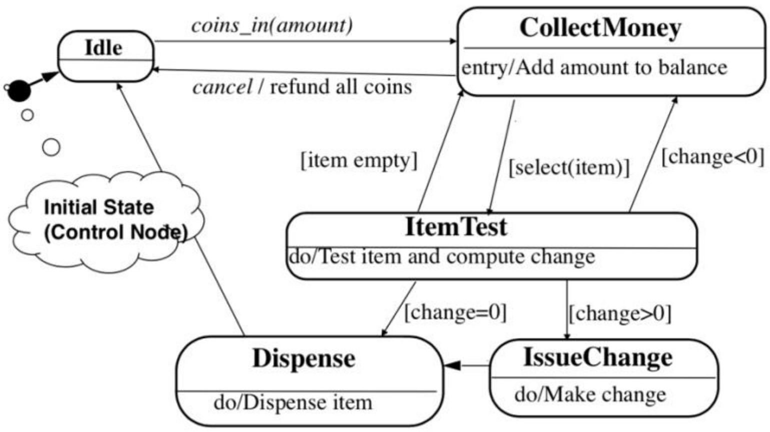


    - Represent dynamic behavior of single object
    - State: abstraction of attributes of a class

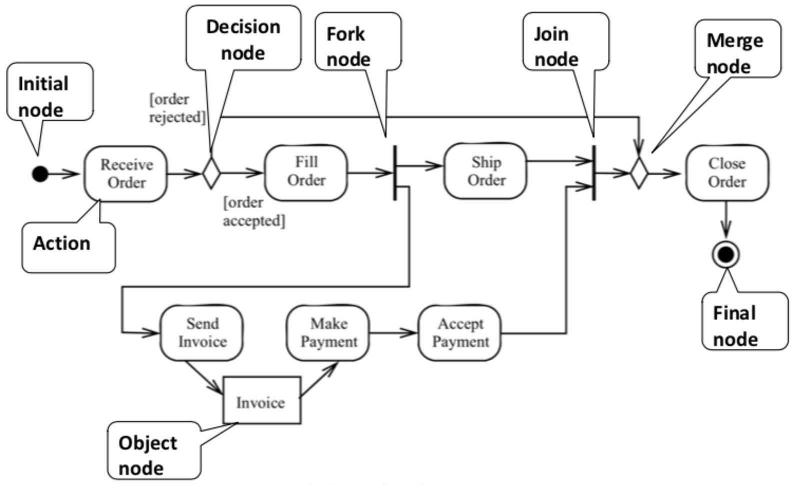
- Example watch:



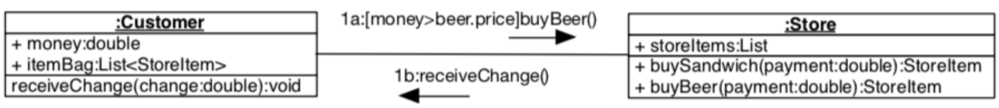
- (formal) Example Vending Machine:



- Activity Diagram:
  - Nodes describe activities (round) or objects (rectangle)
  - Control node icons: initial node, final node, fork node, join node, decision Node
  - Example:

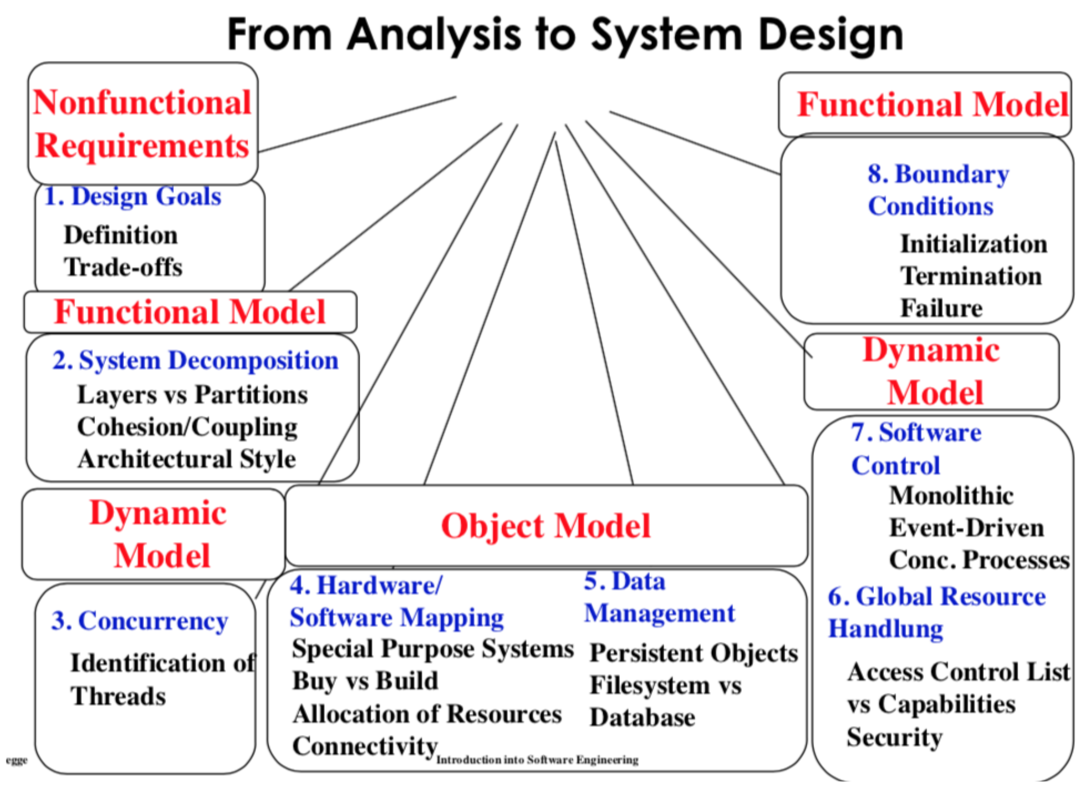


- Communication Diagram
  - Example:



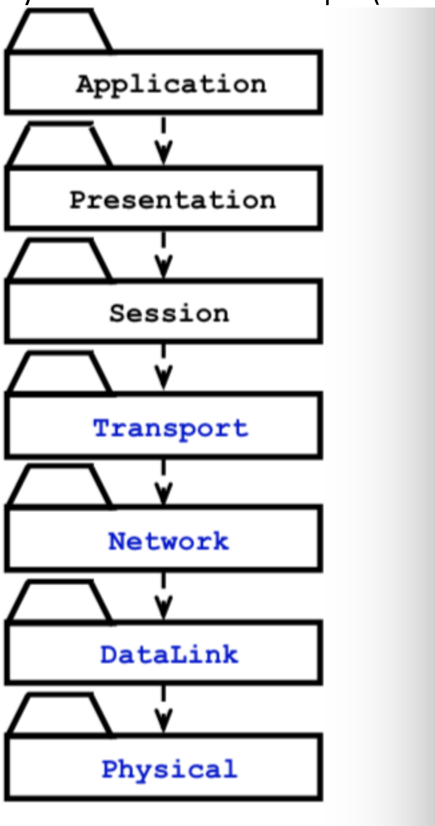
  - No association, roles, labels, multiplicities shown !
  - Structural view of communication between objects

- System Design:



  - System Design consists of 8 Issues to deal with
  - (1) Design Goals:
    - Stakeholders have different goals
    - Functionality vs. Usability
    - Cost vs. Robustness

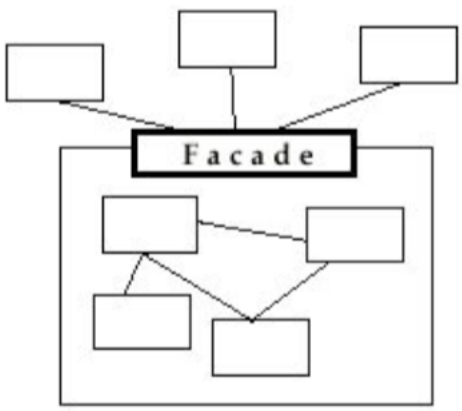
        - Efficiency vs. Portability
        - Rapid development vs. Functionality
        - Cost vs. reusability
    - (2) Subsystem Decomposition
        - Coupling vs. Cohesion:
            a. Coupling: measures dependencies among subsystems
            b. Cohesion: measures dependencies among classes
            c. Goal: high cohesion, low coupling
        - Architectural styles:
            a. A pattern for a subsystem decomposition (i.e layered, hierarchical architecture)
            b. Different layers provide services to higher/lower layers
        - Closed Architecture:
            a. Each layer can call operations from direct lower layer
            b. Goals: maintainability, flexibility
        - Open architecture
            a. Layer can call operations from any layer below
            b. Goals: high performance, high coupling = more efficient
        - 7-layerd architecture example (OSI Model Layer)



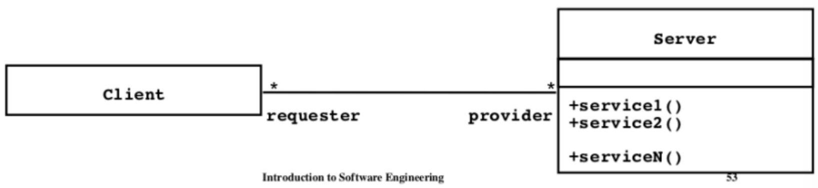
        - Decomposition:
            a. Technique to master complexity
            b. Divide and conquer
            c. Functional decomposition
                i. System decomposed into functions
            d. Object Oriented decomposition
                i. Decomposed into classes
            e. Goal: from Use case to Object Model

5. Lecture (Architectural Styles, System Design (3) – (5))

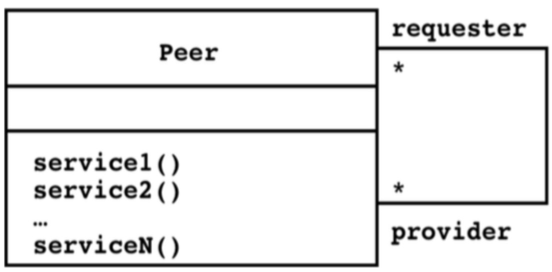
- Architectural Styles:
- Layers vs. Tiers
    - 3 Layered Architectural Style vs. 4 Tier architectural style
    - Layer is a type (i.e. OSI Model) , tier is an instance (object) (i.e. Webbrowser, Server, Database)
- The Façade (Architecture)
    - Pattern to reduce coupling
    - Unified interface for a subsystem
    - Set of public operations
    - Example:



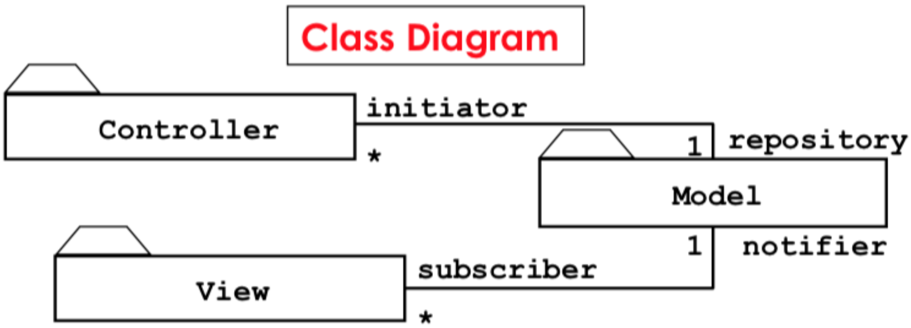
- Client Server (Architecture)
    - Often used for database systems



- Peer to Peer (Architecture)
    - Generalization of the Client/Server Architecture



- Model – View -Controller (Architecture)
    - Problem: Change to boundary objects (user interface) force change to entity object (data)
    - Solution: Decoupling
    - View: Subsystem with boundary Objects (data presentation)
    - Model: Subsystem with entity Objects (data access)
    - Controller: Subsystem mediating between both



- Model View Controller vs. 3 Tier Architecture Style
  - MVC – non hierarchical (triangular)
    a. View updates model directly
  - 3-tier – hierarchical
    a. Presentation never directly communicates with data layer

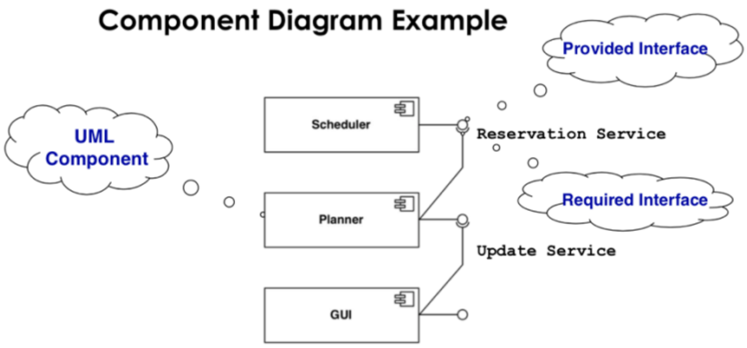
- Pipes and Filters (Architecture)
  - Filter: a subsystem with a processing step
  - Pipe: connection between two processing steps
  - Each filter: 1 input 1 output

- Components and Connectors (in Architectural Styles)
  - Components: subsystems (i.e. filters, databases, layers, objects)
    - Computations units
  - Connectors: communication (i.e. method calls, pipes, shared data)

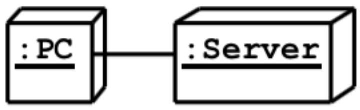
- System Design ((3) – (4))
  - (3) Concurrency
    - Identification of Threads
    - Addresses non-functional requirements
    - Two objects can receive events at the same time
    - Physical (Hardware) vs. logical (Software) concurrency
  - (4) Hardware Software Mapping
    - Problem: how to map object Model into Hardware/Software ?
    - Control Objects: Processor
    - Entity Objects: Memory
    - Boundary Object: Input/Output Devices
- Component Diagram (UML)
  - Illustrates dependencies between components at design time, compilation time and runtime
  - Components i.e.: source code, libraries, executables)
  - Connectors: edges in the graph
  - Shows how components are wired together
    - A provided interface is modeled using the lollipop notation

    - A required interface is modeled using the socket notation

- Example:



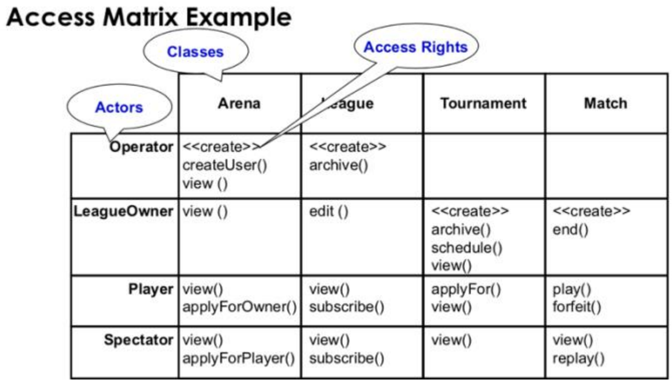
- Deployment Diagram (UML)
  - Illustrates distribution of components at runtime
  - Combination with Component Diagram possible
  - Shows design after
    - Subsystem decomposition
    - Concurrency
    - Hardware/Software Mapping



- System Design (5)
  - (5) Persistent Data Management
    - Persistency: A class is persistent if the values of attributes have lifetime beyond single execution (i.e. Filesystems, Database Systems)

6. Lecture (System Design (6) – (8), Object Design, Design Patterns)

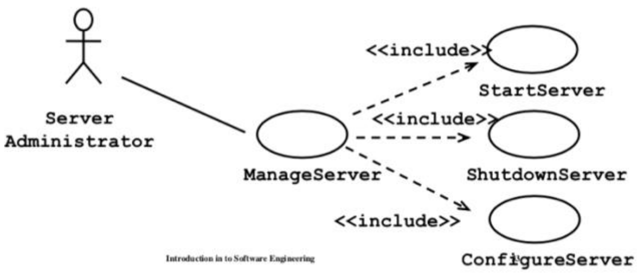
- (6) Global Resource Handling
  - Addresses access control
  - Describes access rights for different classes of actors
  - How objects can be guarded against unauthorized access
  - Access Matrix
    - Models access to actors on classes
    - Rows: actors, columns: classes
    - Access right: operation that can be used

■ 

Access Matrix Example

	Arena	League	Tournament	Match
Operator	<<create>> createUser() view ()	<<create>> archive()		
LeagueOwner	view ()	edit ()	<<create>> archive() schedule() view()	<<create>> end()
Player	view() applyForOwner()	view() subscribe()	applyFor() view()	play() forfeit()
Spectator	view() applyForPlayer()	view() subscribe()	view()	view() replay()

- (7) Software Control
    - 2 ways to control system
    - Implicit Software control (rules, logic programming)
    - Explicit software control (centralized control, decentralized control)
        - Use sequence Diagram to determine if centralized or decentralized
- (8) Boundary Conditions
    - Initialization, termination, Failure



- Object Design:
    - Purpose:
        - Prepare for implementation of system based on design decisions
        - Transform system model (+optimize)
        - Alternative ways for implementation
        - Basis of implementation
        - Closes gap between Analysis & System Design
    - 4 Main Activities
        - Reuse
            a. Identification of existing solutions
        - Interface Specification
            a. Describes each class interface
        - Object Model restriction
            a. Restructuring to improve understandability
            b. Extensibility
        - Object Model Optimization
            a. Transforms object design for performance criteria

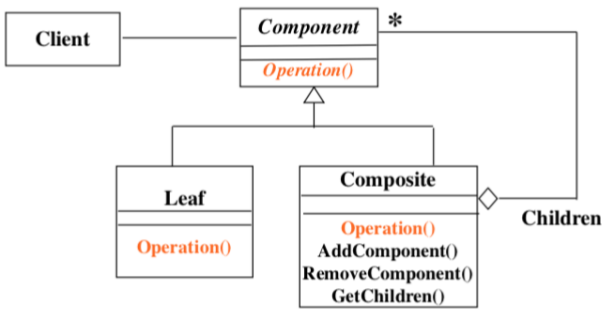
    - Reuse in Object Design:
        - Reuse of: Design Knowledge, existing classes, existing interfaces

- 2 techniques to close object design gap:
  a. Composition (black box reuse)
     i. New class created by aggregation
     ii. Interfaces
  b. Inheritance (white box reuse)
     i. New class created by subclassing
     ii. Functionality of subclass + own

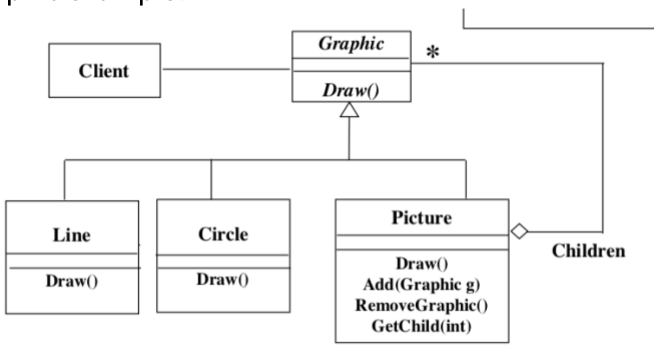
- Definitions:
  - Implementation inheritance:
    a. Subclassing from implementation
  - Delegation
    a. Copying operations
  - Specification inheritance
    a. From abstract class an operation (specified)
  - Discovering inheritance
    a. Generalization (subclass discovered first)
    b. Specialization (super class discovered first)

- Design Patterns (3 Types)
  - Structural Patterns
    a. Reduce coupling
    b. Introduce abstract class (for future extension)
    c. Encapsulate complex structure
  - Behavioral Patterns
    a. Allow choice between algorithms
    b. Assignment of responsibility
    c. Model complex control flow
  - Creational Patterns
    a. Allow to abstract
    b. Make system independent

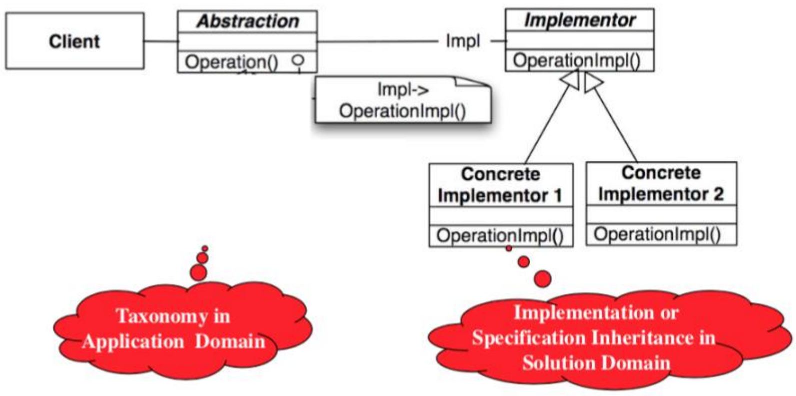
- (1) Composite Pattern
  - Hierarchical structure
  - Client has access to a component class
    a. He gets access to Leaf Classes
  - Goals:
    a. Complex structure
    b. "Must have variable depth and width"
  - Formal example:



- Graphic example:

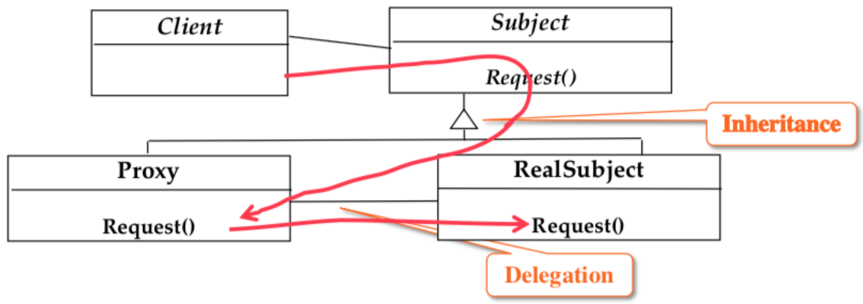


- (2) Bridge Pattern
  - Goals/Requirements:
    a. desirable to delay design decisions at design time or compile time
    b. delay design decisions until run time
    c. delay binding between interfaces and subclasses to start-up time of system
    d. user gets interface to use classes in a lower layer
    e. “Must interface to several systems”
    f. “Early prototype needed”
    g. “Backward combability”
  - “Delegation (dynamische Methodenbindung) followed by inheritance (Vererbung) “
  - Formal example:

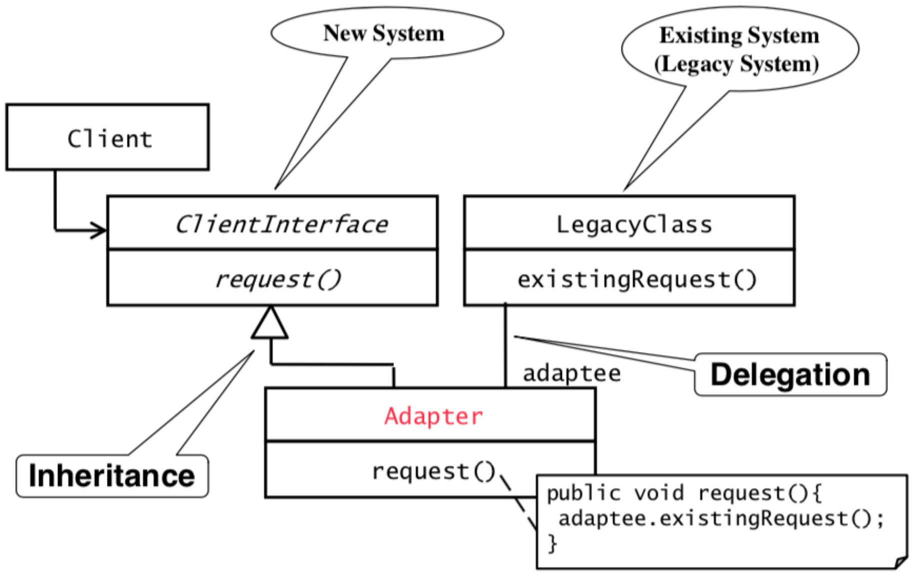


- (3) Proxy Pattern
  - Goals/Requirements:
    a. Instantiation only if needed (because its expensive)
    b. Instantiate representative (proxy) for the expensive object
    c. Provide access control to real object
    d. “High security”
    e. “Must be location transparent”
  - Client calls request() in Proxy. Proxy then uses delegation to access request() in RealSubject.

- Formal example:

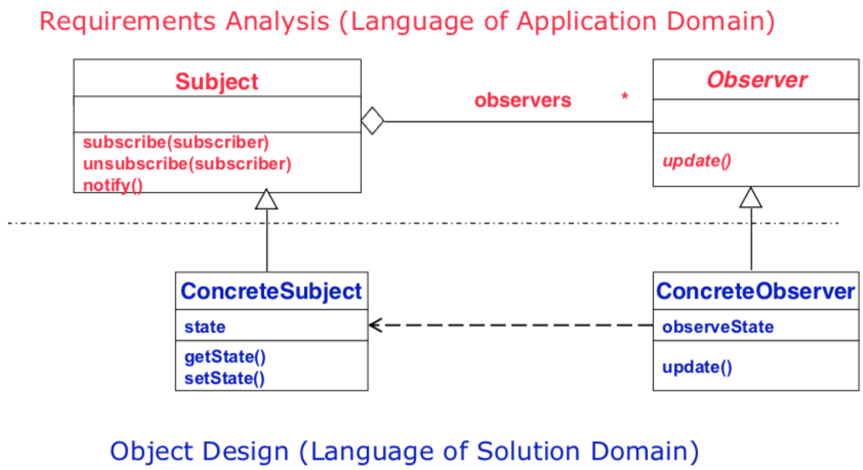


- (4) Adapter Pattern
  - Goals/Requirements:
    a. Connect incompatible components
    b. Allow reuse of existing components (legacy system)
    c. “Backward compatibility “
  - “Inheritance followed by delegation”
  - Formal example:

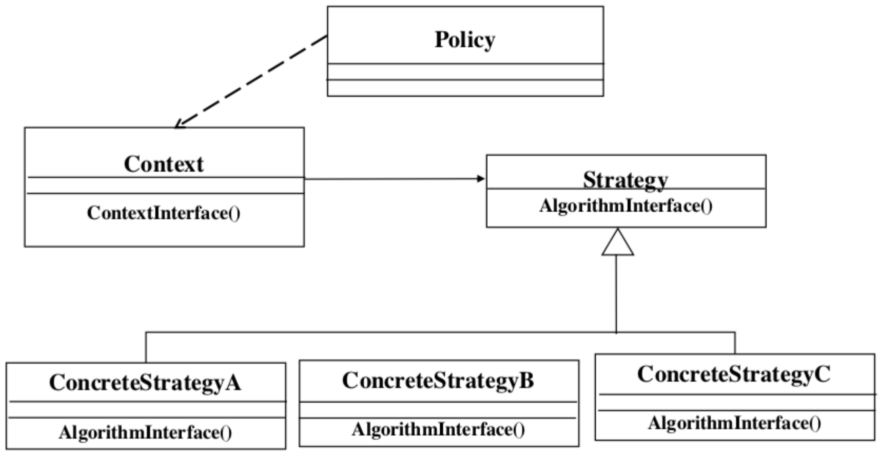


- (5) Observer Pattern
  - Problem: Object that changes its state often
  - Goals/Requirements:
    a. “System should be highly extensible”
    b. “must be scalable”
    c. Maintain consistency across redundant states

- Formal example:



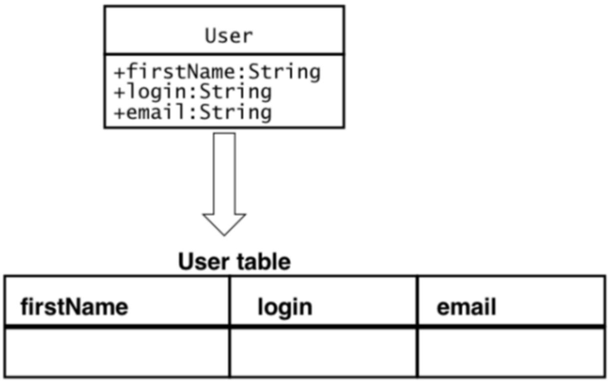
- (6) Strategy Pattern
  - Goals/requirements:
    a. Change algorithm at runtime
    b. Policy decides which algorithm
       i. Independent from the mechanism
       ii. Switch between algorithms at runtime
    c. Add new algorithms without disturbing application
  - Formal example:



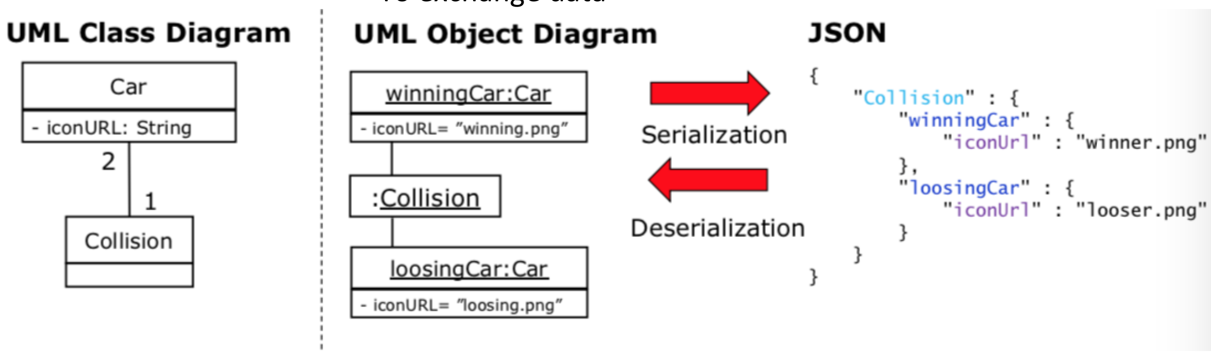
7. Lecture (Model Transformations and Refactoring)

- Definitions:
  - Model Driven Engineering:
    - Focusses on creation of domain modules
    - Goal: increase of productivity (reuse, standardization,…)
  - Model based Software Engineering
    - Application of modelling to support requirements, design, analysis, verification, validation
    - Contains several models (behavioral, functional, structural)
  - Model Transformation
    - Is also a Model
    - Input: Model, Output: other model
- 4 Types of Model Transformation:

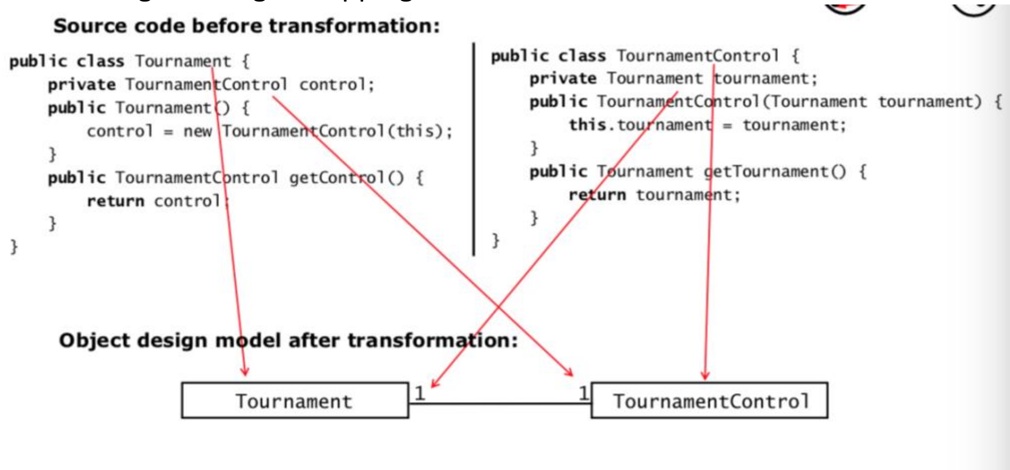
- Refactoring
    - Change made to the internal structure of source code/models

- (1) Forward Engineering: Model to Java
    - Split Model to Java interface + implementation class
    - Attributes should be non public, + getter and setter
    - Inheritance with subclasses/superclasses
    - Subtyping with interfaces
- (2) UML State Diagrams to Java Code
    - Idea:
        a. Public Method for each state
        b. Switch statement for each state
        c. If statements for state change
- (3) Mapping Contracts to Java
    - Contract: A Contract specifies constrains that the user must meet before using class
    - 3 types of constraints
        a. Precondition
        b. Postcondition
        c. Invariant
    - Check Conditions
        a. Throw Exceptions/assertions
- (4) Mapping Models to Tables
    - Convert a relational database into an object model and vice versa (Object Relational Mapping – ORM)
    - 

- Source Code Transformation
    - Mapping Objects to JSON
        - To exchange data

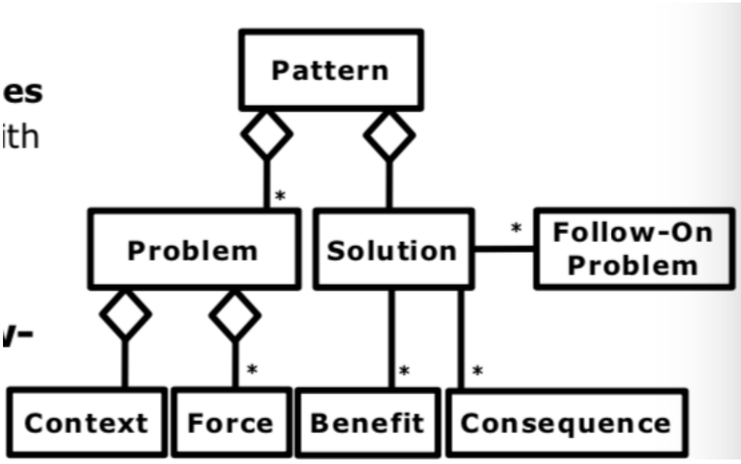


- Reverse Engeneering – Mapping Source Code to Models



8. Lecture

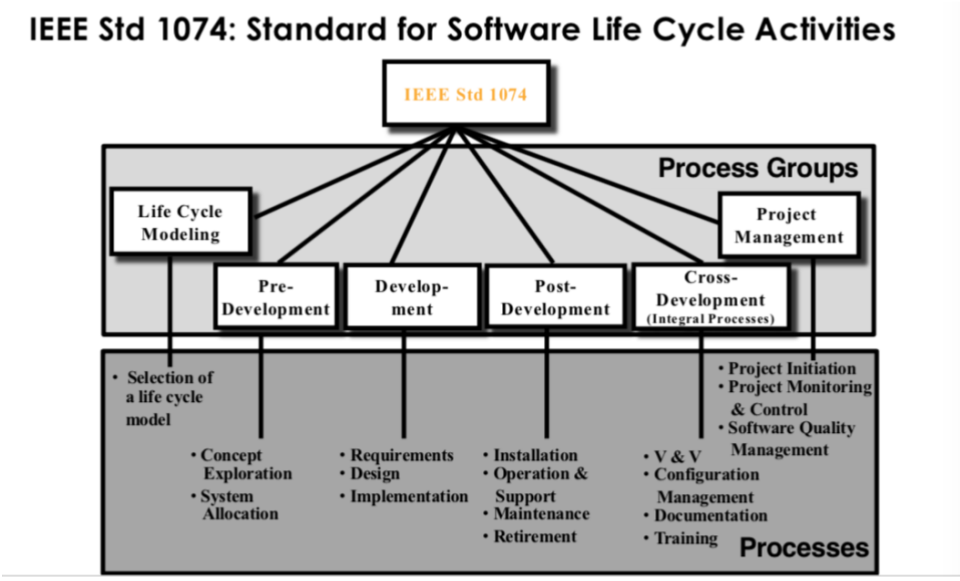
- Pattern: A Pattern is a three part rule, which expresses a relation between a certain context, a problem and a solution.



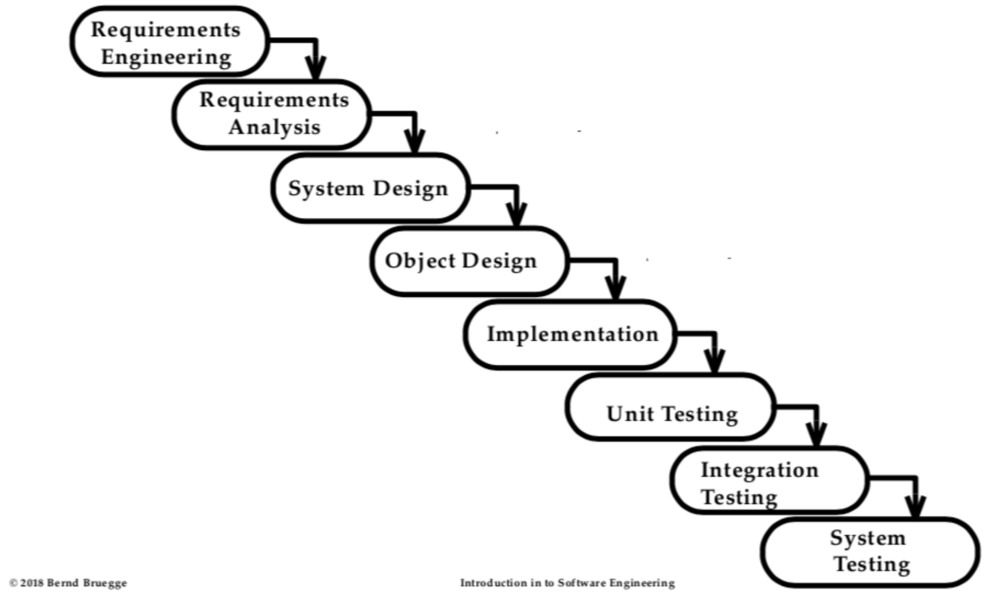
- Pattern based development
  - Use of Pattern during analysis, system design, object design and testing
  - Goal: manage complexity, reduce cost/time
- Pattern Coverage
  - Every Element in the UML Model + Source Code is covered by a pattern (Desirable: 100%)
- Pattern Scheme (example):

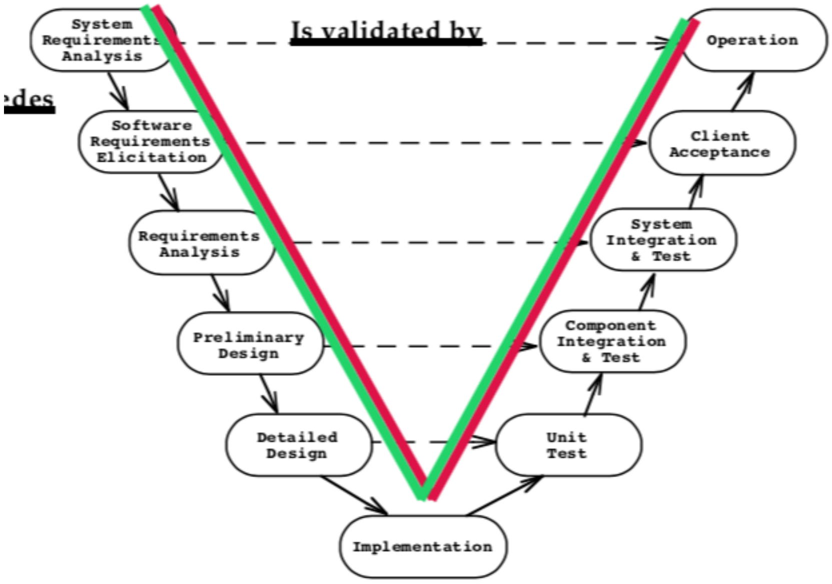
Pattern Name	Bridge
Problem	Permanent binding between an interface and its implementation
Context	Decouple interface from its implementation so that they can vary independently
Forces	Delay design decisions to start-up time
Solution	The ***Abstraction*** class defines the interface visible to the client. The ***Implementor*** is an abstract class that defines the lower-level methods available to ***Abstraction***. An ***Abstraction*** instance maintains a reference to its corresponding ***Implementor*** instance. ***Abstraction*** and ***Implementor*** can be refined independently.
Benefits	• Decoupling interface and implementation • Interfaces and implementations can be refined independently
Consequences	Information Hiding: Client does not need to know the implementor
Follow-On Problem(s)	• No switching between implementations at run-time (can be solved by additionally implementing the Strategy Pattern)

9. **Lecture (Software Lifecycle Model)**
- Model Software Lifecycle
  - Set of activities and their relationships to each other to support the development of a software system
  - An abstraction, representing the development for understanding, monitoring and controlling the software
  - Software lifecycle Activities Standart



- Sequential Model: Waterfall Model
  - Step by step process (sequential)
  - Includes verification
  - Nice milestones
  - No need to look back
  - Always one activity at a time
  - Requirements problems are identified during process
  - -> Software Development is not liner though ?!

- 

- From Waterfall to V-Model:
- 

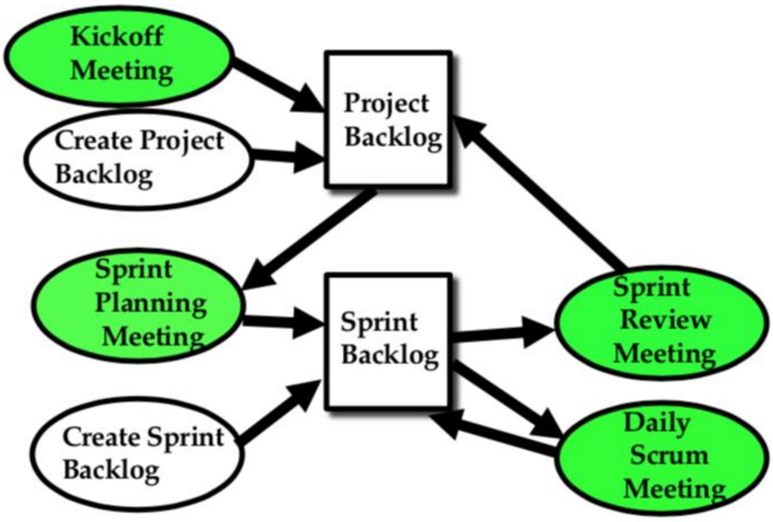
- Iterative Model: Spiral Model
    - 9 rounds with 4 activities:

1. Concept of Operations
2. Software Requirements
3. Software Product Design
4. Detailed Design
5. Code
6. Unit Test
7. Integration and Test
8. Acceptance Test
9. Implementation

- For each round go through these activities:
    1. Define objectives, alternatives, constraints
    2. Evaluate alternatives, identify and resolve risks
    3. Develop and verify a prototype
    4. Plan the next round.

- Can only deal with change within phases

- Entity-Oriented Model (Scrum)
    - Two Artifacts
        - Project Backlog
        - Sprint Backlog
    - Two Activities
        - Establish Project Backlog
        - Establish Sprint Backlog
    - Four Meeting Activities
        - Kickoff Meeting
            a. In the beginning
            b. Create Product Backlog
        - Sprint Planning Meeting
        - Daily Scrum
            a. Every day (15min) before team starts working
            b. Answer basic questions (status, issues, actions)
        - Sprint Review Meeting
    - Three Scrum Roles
        - Scrum Master (Project Manager)
        - Product Owner (Product Manager)
        - Scrum Team



    -

    -

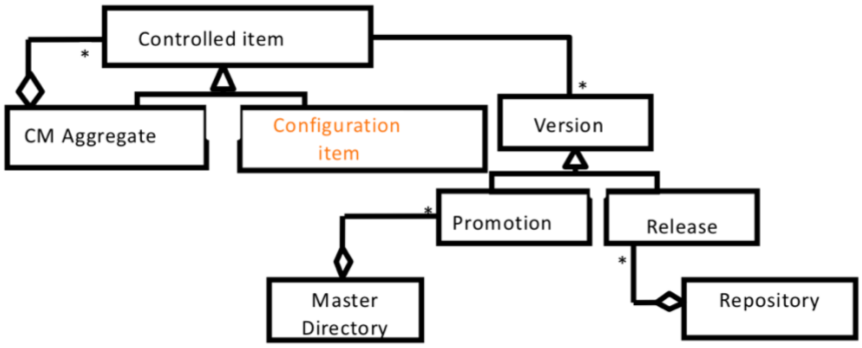
- Development types
  - Incremental
    - “add onto something”
    - Improves process
  - Iterative
    - “to re-do something”
    - Improves product
  - Adaptive
    - “react to changing requirements”
    - Improves reaction to changing customer needs

- Defined Process Control model
  - Well defined input
  - Same output every time
  - Waterfall model
  - Change can be ignored, output predictable
- Empirical Process Control Model
  - Well defined input – dfferent output
  - “Expect the unexpected”
  - Scrum

10. Lecture (Software Configuration Management)

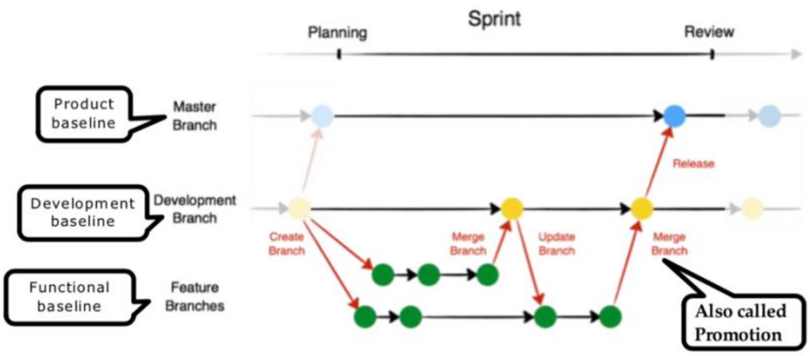
- Software Configuration Management (SCM)
  - Set of management disciplines
  - Initiating, evaluating, controlling change
- Why ?
  - Multiple people work on same project
  - More than one version has to be supported
  - Need for coordination
- Activities
  - Configuration Item identification
  - Change management
  - Promotion management
  - Branch management
  - Release Management
- Configuration Item Identification
  - Configuration Item:
    - An Item designated for configuration management (can be all types of files)
    - All files that are needed for the project
  - Baseline:
    - A specification or product that has been formally reviewed and agreed (-> Master directory in GIT)
  - Modelling the system as a set of evolving components
- Change Management
  - Management of change request
  - Promotion:
    - Internal development state is changed

- Release
  - Changed software system visible outside

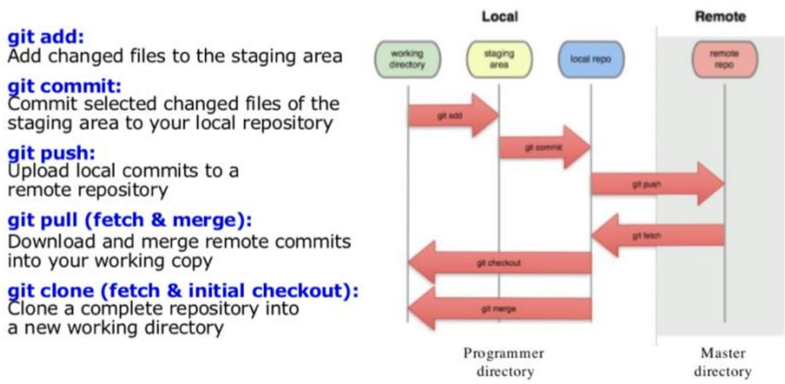


- Revision:
  - Change to a version that corrects only errors in the design
  - Doesn't affect functionality
- Change policy:
  - Guarantee that each promotion or release conforms to accepted criteria

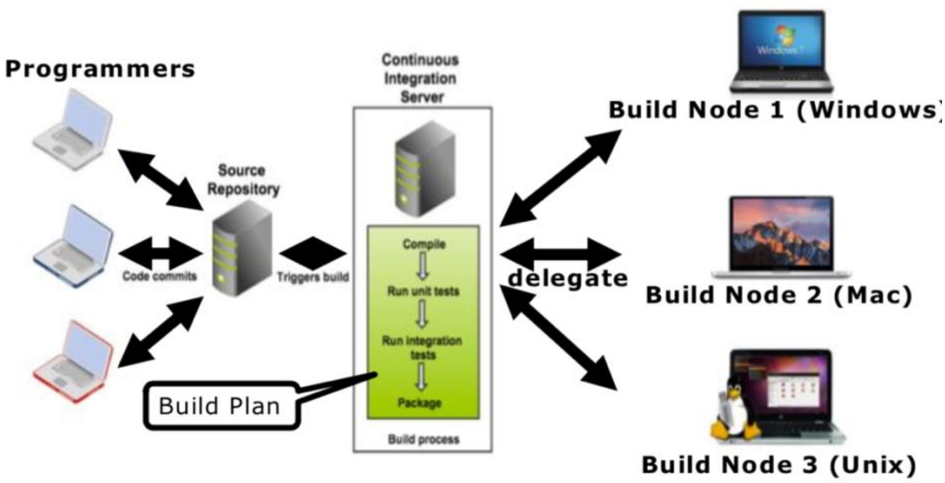
- Promotion Management
  - Creation of versions for other developers
    - With Version control systems (VCS)
  - Many developers can work on configuration items in a given project
  - VCS allow to store different versions
  - 3 different VCS architectures
    - Monotolitic (Database on one computer)
    - Repository (single server contains all versions)
    - Peer-to-Peer (All computers fully mirror masters directory)
- Branch Management
  - GIT Branch management example:
    - **Master Branch:** External Release (e.g. Product Increment)
    - **Development Branch:** Internal Release
    - **Feature Branches:** Incremental development and explorations



- GIT commands:

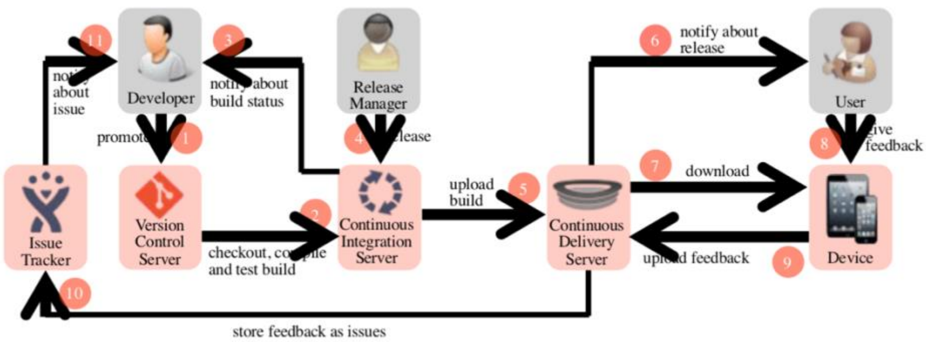


- Continuous Integration
  - The later integration happens the bigger risk of unexpected failures
  - The higher complexity – more difficult integration
  - There is always an executable version of the system
  - Example:



  - Apache Maven:
    - Open Source build and dependency management tool
    - POM – Project Object Model (main file)
    - Compile, test, package, install, deploy (releasable version)

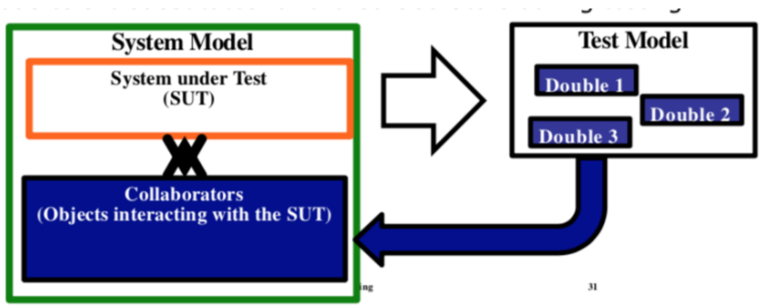
- Release Management
  - Release Management Model:



- Terminology:
    - Continuous Integration
        - Integrate work frequently
        - Each person integrates usually daily
    - Continuous delivery
        - Always a deliverable prototype reliable and ready to release
    - Continuous deployment
        - Every change that passes automated tests is deployed automatically
    - Continuous software engineering
        - Organizational development, release and learning from software in short cycles

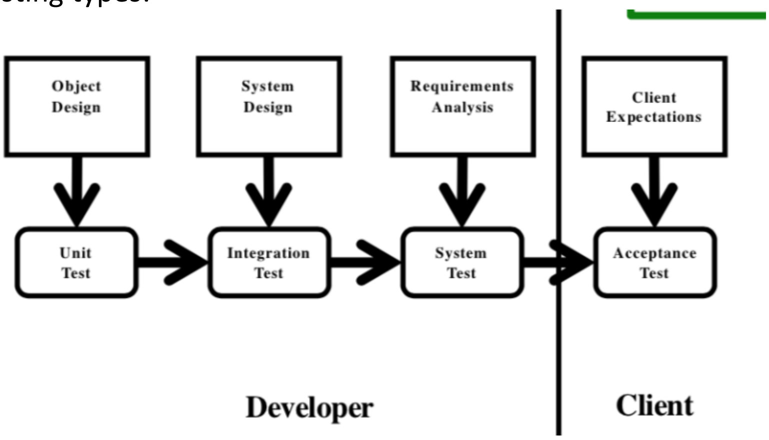
11. Lecture (Testing)

- Failure
    - Deviation from observed behavior from specified behavior
- Erroneous State (error)
    - System is in a state that can lead to a failure
- Faults:
    - Fault
        - Cause of an error
    - Fault avoidance
        - Reduce complexity
        - Configuration management
        - Verification
    - Faults detection
        - Testing
        - Debugging
        - Monitoring
    - Fault tolerance
        - Exception handling
- Validation
    - Checking for faults
- Test Model
    - Test driver (program)
    - Test case (function)
    - Input data (data)
    - Oracle (predicts output)
    - Test
- Automated Testing
    - All testcases automatically executed
- Model-Based Testing
    - System model used for generation of test model
- Object Oriented Test Modelling
    - SUT – System under Test
    - Double
        - Test Objects added to the test model (Mock objects, Stubs, dummy objects ..)

- 

- Mock objects
  - Mimic behavior of real objects (hard coded code)
- Stubs
  - Methods that return static example values
- Driver
  - Component that calls tested unit

* Testing activities
  - Unit Testing
  - Integration Testing
  - System Testing
  - Acceptance Testing
  - Dynamic analysis
    - Black Box Testing
      a. Tests input/output behavior
    - White box Testing
      a. Tests the implementation of the subsystems or class
  - Static Analysis
    - Hand execution by reading source code (syntactic, sematic errors)
    - Compiler Warnings/Errors (IDE)
  - Testing types:

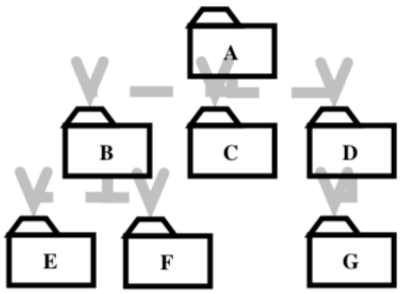


  - White Box testing:
    - Statement Testing (each statement)
    - Loop testing
    - Path testing (make sure all paths are executed)
    - Branch testing (each outcome is tested at least once)
* Unit testing
  - Individual units in a program are tested (classes/subclasses)
  - Goal: confirm component is correctly coded

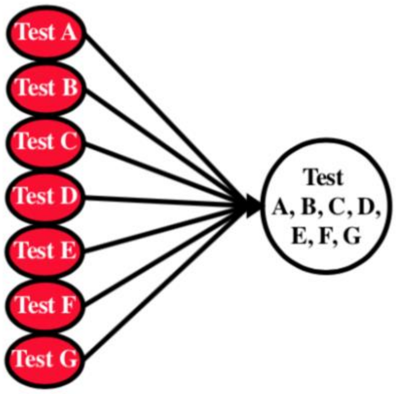
- Example:

```
public class MoneyTest {
  @Test public void simpleAdd() {
    Money m12CHF = new Money(12, "CHF");
    Money m14CHF = new Money(14, "CHF");
    Money expected = new Money(26, "CHF");
    Money observed = m12CHF.add(m14CHF);
    assertTrue(expected.equals(observed));
```

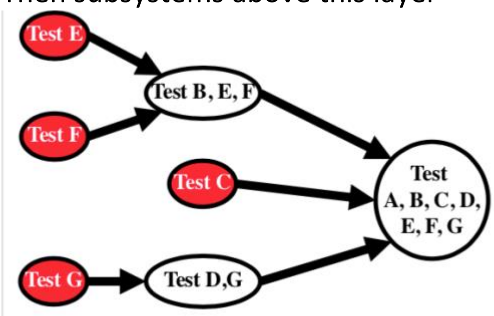
- Integration testing
  - Groups of subclasses are tested
  - Goal: test the subsystem interfaces among the subsystems
  - Strategies:
    - Example:



    - (1) Big Bang Testing
      a. Tests all classes in order (individually)



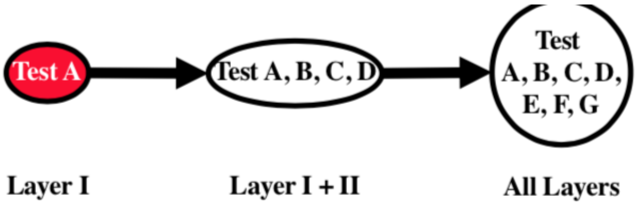
    - (2) Bottom-Up Testing
      a. Subsystems in the lowest layer are tested individually
      b. Then subsystems above this layer



      c. Good for object oriented systems
      d. Con: tests important subsystem (view) last

e. Drivers needed

- (3) Top Down Testing
    a. No drivers needed
    b. Tests subsystems in the top layer first
    c. Needs a lot of stubs



- Horizontal Integration Testing
    - Examples: Bottom up, top down
    - Risk 1: difficult if high complexity
    - Risk 2: unexpected errors if tested too late
- Vertical Integration Testing
    - Used in scenario driven examples
        a. Scrum i.e.
    - Advantages:
        a. Always an executable version
        b. All team members have good overview

- System Testing
    - Functional Testing
        - Same as black box testing
        - Test functionality of system
    - Structure Testing
        - Same as white box testing
        - Cover all paths in system design
        - All components
    - Performance Testing
        - Try to violate non-functional requirements
        - Types (examples)
            a. Stress Test
            b. Volume testing
            c. Quality Testing

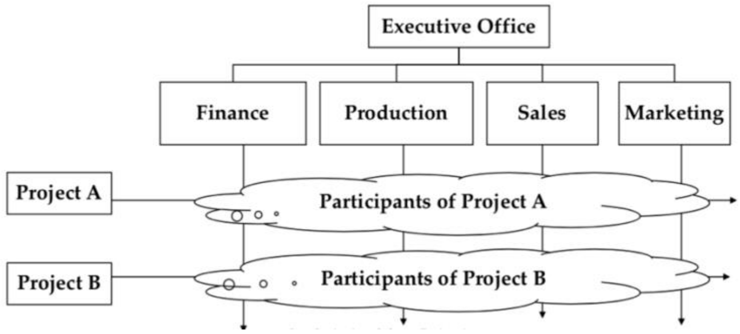
    - Acceptance Testing
        - Goal: demonstrate system is ready for operational use
        - Validates client's expectations

12. Lecture

- Softskills
    - Collaboration (Negotiate Requirements)
    - Presentation
    - Management
    - Technical writing (model system, documentation)

- Communication event
    - Information exchange with defined objectives
    - Scheduled
        - Planned Communication events
        - Problem definition
            a. goals, requirements, constraints
        - Project Review
            a. System model review
        - Client Review
            a. requirements review, changes
        - Walkthrough
            a. present subsystems
        - Inspection
            a. Demonstration of final system to customer
    - Unscheduled
        - Event driven communication
        - Unplanned communication Events
        - Participant reports problem and proposes solution
- Communication mechanism
    - Tool to transmit information
    - Synchronous/asynchronous
    - Synchronous Communication Mechanism
        - Conversation, Meeting, Email, Newsgroup, Portal
- Project Definition:
    - A project is an undertaking, limited in time, to achieve a set of goals that require concerted effort
    - A project includes
        - Deliverables to a client
        - A schedule
        - Technical and managerial activities
        - Resources
- Organization Forms
    - Defines relationship among resources (participants in a project)
    - Functional Organization
        - Groups of departments addressing an activity (function)
            a. Finance, production, sales, marketing
        - Each group has specialists
        - High change of overlap or duplication of work
        - Goal: high stability, uniformity, little communication
    - Project-based Organization
        - People assigned to project with specific problem
        - Goal: open communication, requirements change in during development
    - Matrix organization

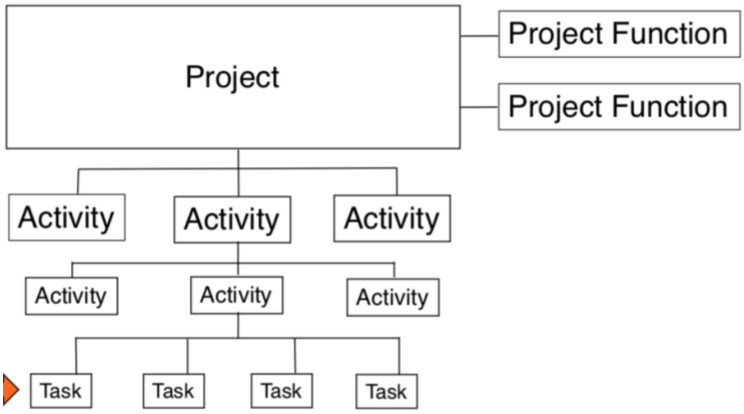
- People are assigned to one or more projects



- Participants are not used to each other
- Team members work for two bosses

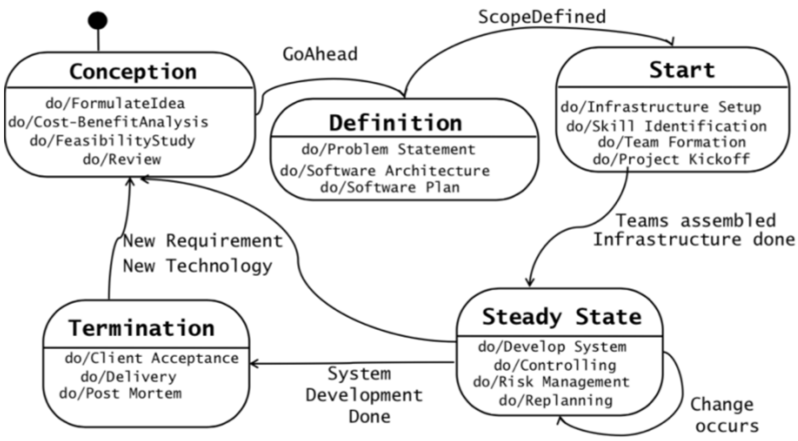
- Mapping Roles to people
  - Role: set of Responsibilities
  - Role: Tester
    - Write tests
    - Report failures
    - Check bugs
  - Role: System architect
    - Ensure design decisions
    - Define subsystems
  - Role: Liaison
    - Negotiate API with other teams
  - Responsibilities are assigned to Roles, Roles are assigned to People
  - Possible Mapping
    - One to one
    - Many to few
      a. Each project member gets several roles
    - Many to Many
      a. Some people don't have significant roles
      b. Lack of accountability
  - Key concepts for mapping:
    - Authority
      a. Make binding decisions between people and roles
    - Responsibility
      a. Commitment to achieve specific results
    - Accountability
      a. Tracking task to a specific person
    - Delegation
      a. Binding responsibility to another person

- Tasks & Activities



- Activities
  - Major unit of work
  - Grouped together into higher level activities
  - Example: Planning, Analysis, System Design, Testing …
- Task
  - Work package
  - Description of work to be done
  - Completion criteria (acceptance criteria)
- Project Functions
  - Example: Configuration Management, Testing, Documentation
  - Cross-development processes
- Work product
  - Outcome of a task
  - Example: a document, a presentation, piece of code, test report

- Model Project
  - Project has 5 states
    - Conception: The idea is born
    - Definition: plan is developed
    - Start: Teams are formed
    - Steady State: The work is being done
    - Termination: The Project is being finished



- Project Organization
    - Decision Structure (who decides?)
    - Reporting Structure (who reports status to whom?)
    - Communication Structure (Who communicates with whom?