# Introduction

Software Developments Models:

• Object model: What is the structure of the system?

• Functional model: What are the functions of the system?

• Dynamic model: How does the system react to external events?

• System Model: Object model + functional model + dynamic model

Software Engineering Definition:

Software Engineering is a collection of techniques, methodologies and tools that help with the production of.

Phenomenon **vs** Concept: **vs**

|  |  |
| --- | --- |
| Phenomenon: -an object in the world of a domain as you perceive it | Concept:  -describes the common properties of phenomena  -3-tuple: name,purpose and members |

Definitions

* Abstraction: classification of phenomena into concepts
* Modeling: development of abstraction to answer specific questions about a set of phenomena

Systems, Models and Views

* model: an abstraction describing a system
* view: despicts selected aspects of a model

# Model-Based Software Engineering

Definition

* Software life-cycle: set of activities and their relationships to each other (activities: Analysis, System Design)
* Software life-cycle model: abstraction representing the development if software
* defined process: given-set of inputs, same outputs are generated every time, all activities and tasks are defined

UML:

* reduces complexity by focusing on abstraction
* use:
  + Communication
  + Analysis and Design
  + Archival

Application **vs** Solution Domain

|  |  |
| --- | --- |
| Application(Analysis):  -the environment in which the system is operating | Solution Domain(Design, Implementation):  -the technologies used to build the system |

Use Case

Use Case Diagram

* [[1]](#footnote-2)Textual Use Case: {[1.Name], [2.Participating actors], [3. Entry conditions], [4. Exit Conditions], [5. Flow of events], [6. Special requirements]}
* Use Case D.: describe the functional behavior of the system as seen by the user
  + use case represents a functionality provided by the system

|  |  |
| --- | --- |
| <<extends>> Relationship [[2]](#footnote-3)  -To model rarely invoked use cases or exceptional  functionality. The extension use case is not meaningful by its own  -The direction of the arrow in the  <<extends>> association points to base use case | <<includes>> Relationship[[3]](#footnote-4)  -To model functional behavior that is common to more  than one use case. Represents functionality needed in more than one use case  -Arrows in <<includes>>  associations point from the base use case to the  inclusion use case |

Class digram

* Class D.: describe the static structure of the system

|  |  |
| --- | --- |
| Aggregation:  -case of an association denoting a “part of” hierarchy | Composition:  -is used when the life time of component are controlled by the aggregate(parts do not exist on their own) |

UML

* Entity Objects: -represents the persistent information tracked by the system
* Boundary Objects: -represents the interaction between the user and the system
* Control Objects: -represents the control tasks to be performed by the system

Actor **vs** Class **vs** Object

|  |  |  |
| --- | --- | --- |
| Actor:  -an entity outside the system. Interacting with the system | Class:  -concept from the application/solution domain  -part of the system model | Object:  -specific instance of a class |

Requirements Engineering

* Requirements Elicitation: -def. of the system understood by a customer or user →”Requirements specification”
* Analysis: -def. of the system understoodby a developmer → “Analysis model” [[4]](#footnote-5)
* Requirements Engineering: -combination of these two. **Activity that defines the requirements**

Requirements Specification **vs** Analysis Model

|  |  |
| --- | --- |
| Requ.Specification:  -uses natural language | Analysis model:  -uses a formal language |

Requirements: describes the **users view** (identify the **what** of the system, **not how**)

Types of Requirement Elicitation:

|  |  |  |
| --- | --- | --- |
| Greenfield Engineering:  -dev. From scratch  -requirements are **extracted** from **client** and **user** | Re-engineering:  -requirements **trigged** by **new** **technology** | Interface Engineering:  -provide services of existing systems in new environment  -req. triggered by technology or new market needs |

Types of Requirements

|  |  |
| --- | --- |
| Functional Req.:  -functionality: what should the software do?  -external interfaces (actors) | Nonfunctional Req.:  -Usability, Performance(response time, availability, accuracy), Reliability(robustness, safety), Supportability(Adaptability, Maintainability),... |

Techniques to Describe Requirements

|  |  |  |
| --- | --- | --- |
| Scenario:  -a **concrete**, **focused**, informal description of a single feature of the system by a **single** **actor(→** written from end user’s point) | Use case:  -concept that describes a set of scenarios by an actor with the system | User Story:  -a functional requirement from perspective of end user |

Types of Scenarios

|  |  |
| --- | --- |
| As-is scenario:  -describes a current situation. It describes the usage of an existing system(used in re-engineering projects) | Visionary scenario:  -describes a future system  (used in all types of projects) |
| Evaluation scenario:  -describes user tasks against which the system will be evaluated(used in demos & acceptance tests) | Training scenario:  -guide a novice user through a system  (used in System delivery) |

Requirements Validation:

* Correctness:
* Clarity: req. can only be interpreted in one way
* completeness
* consistency: no req. that contradict each other
* realism: req. can be implemented and delivered
* Traceability: system components and behavior can be traced to the functional req.

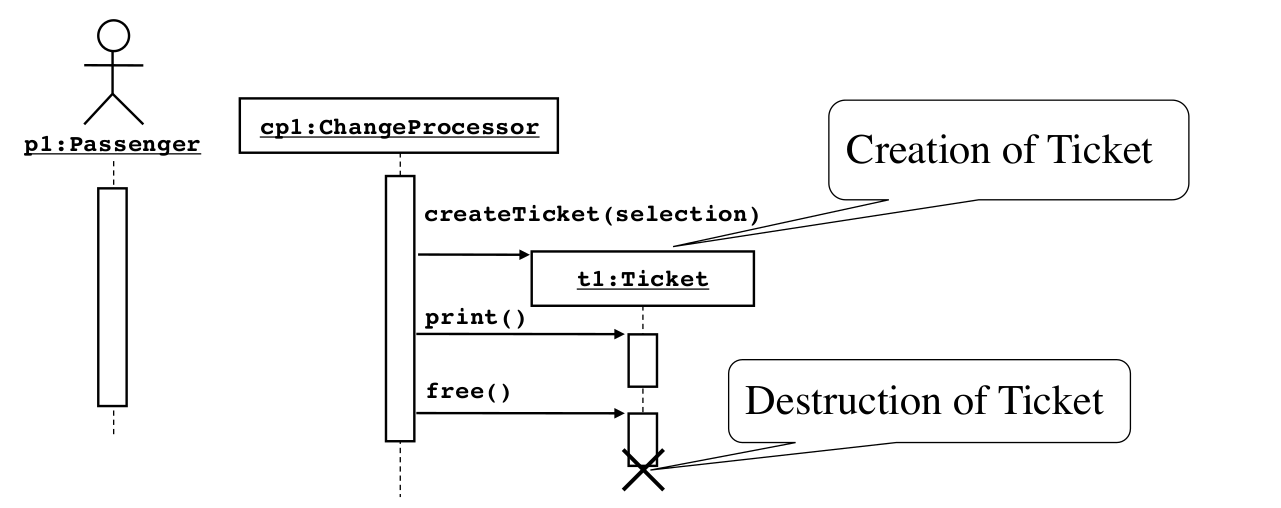
# System Design

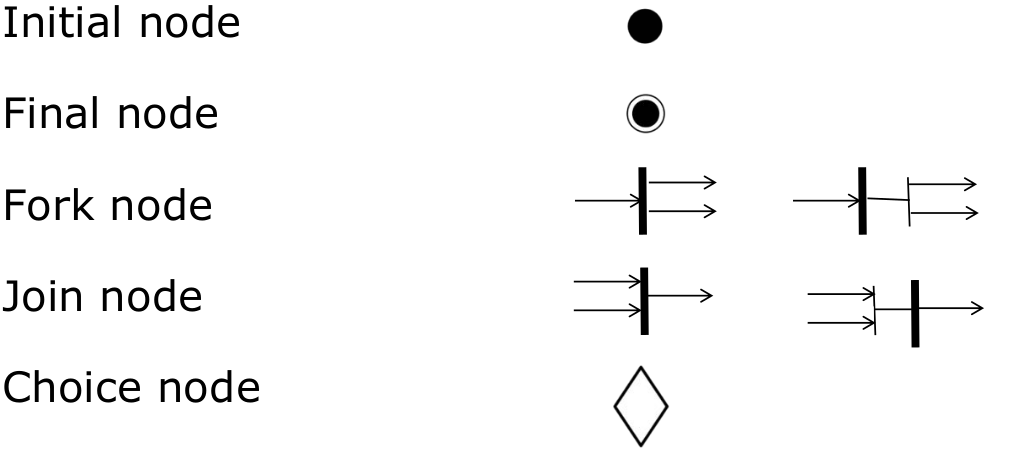
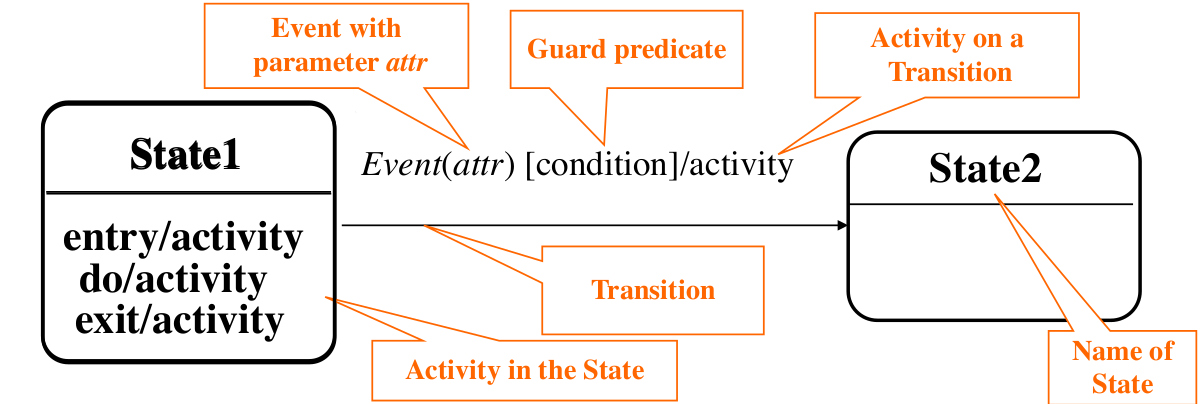
instance

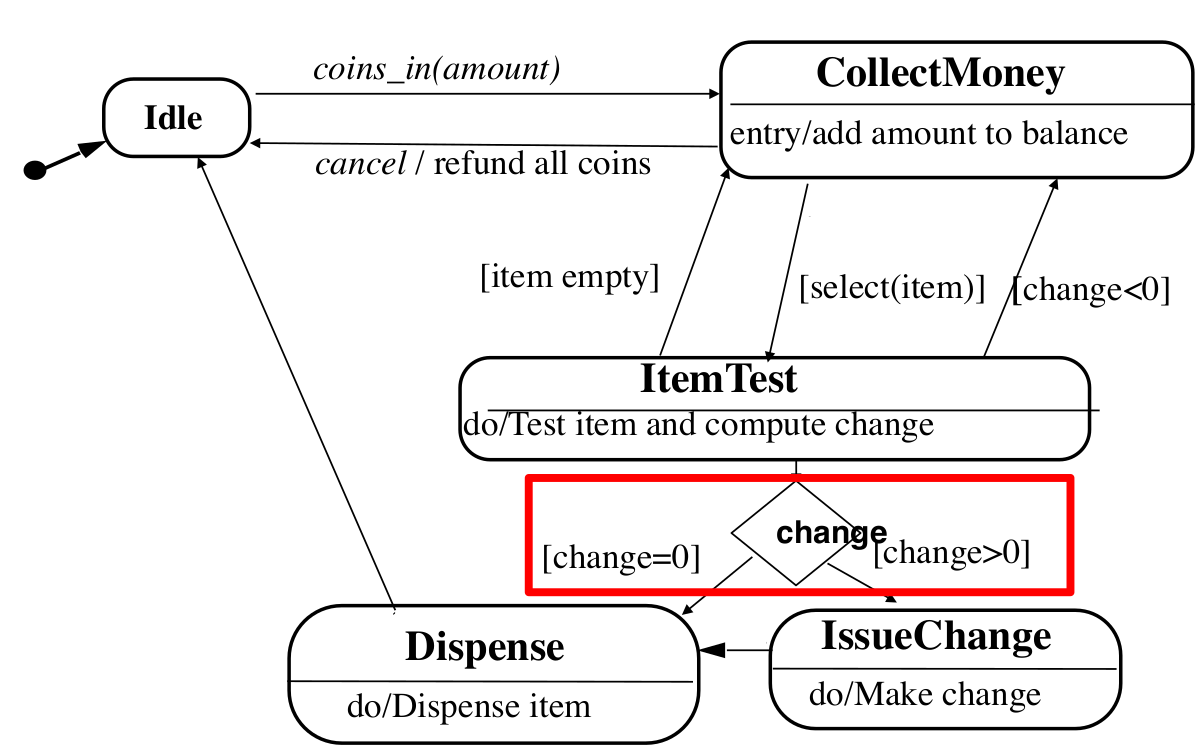
dataflow

Diagrams:

Sequenz Diagrams:

* represent control flow and behavior in terms of interaction
* all messages to an object **must be public methods**
* **-**time consuming to build +useful to identify missing objects
* +useful to identify public methods of a class/subsystem

State Chart Diagrams:

* **State:** abstraction of the attributes of a class, has a duration
* event /activity, [guard predicate]

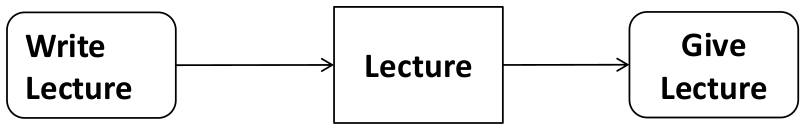
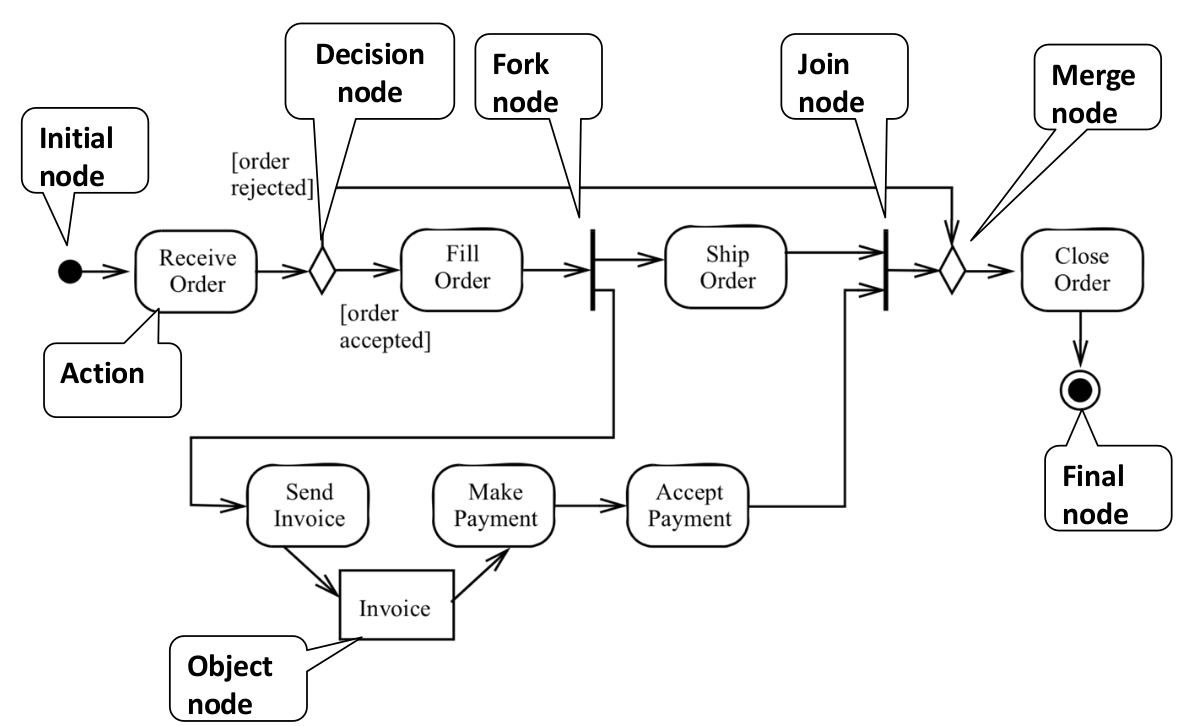
State-chart **vs** Sequence Diagram

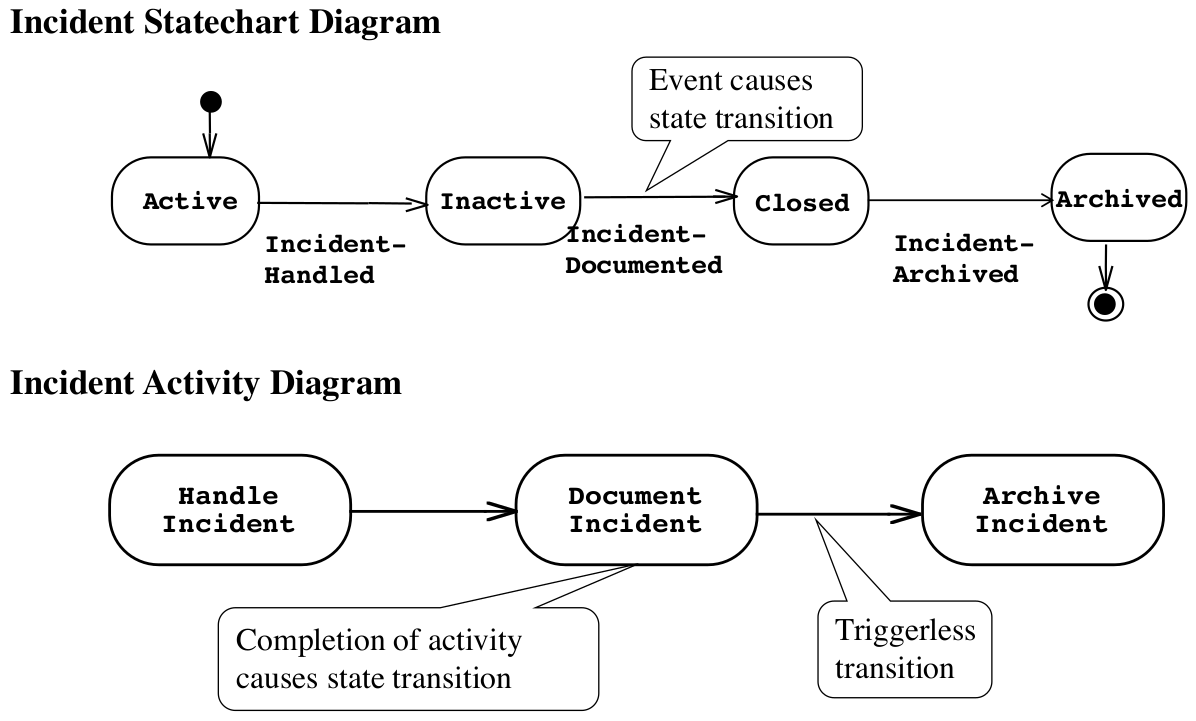
|  |  |
| --- | --- |
| State-chart:  -helps to identify changes to an individual object over time | Sequence:  -helps to identify temporal relationship between objects over time  -sequence of operations as a response to one or more events |

Activity Diagrams:

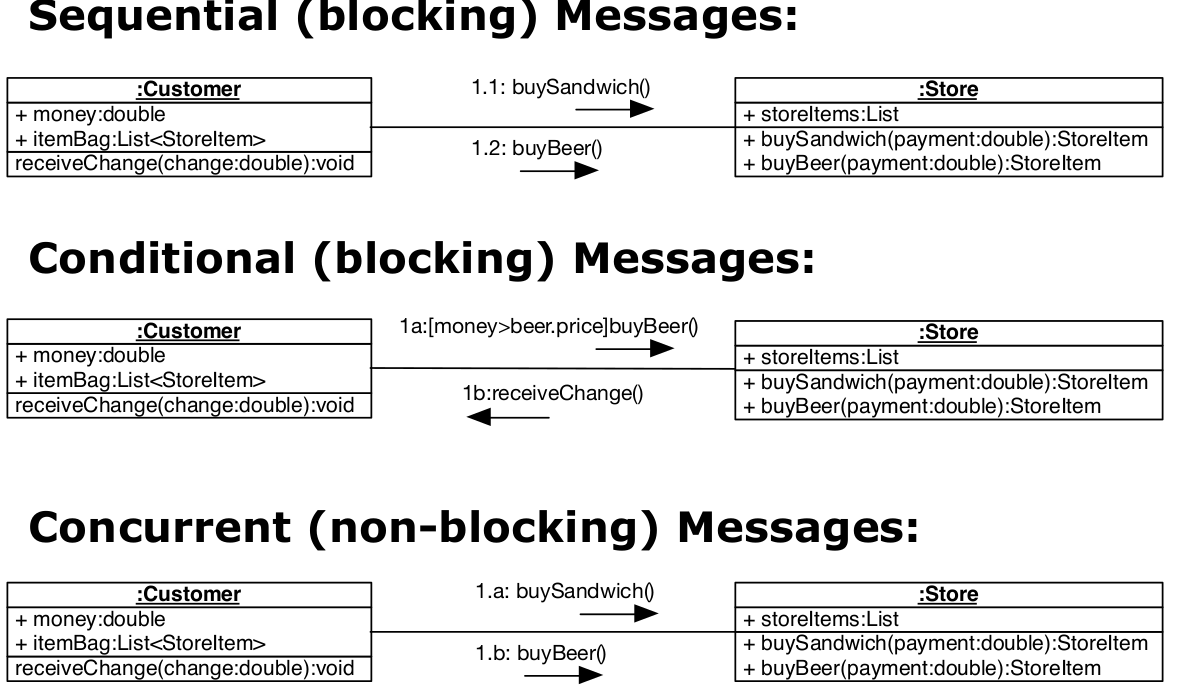
* **nodes** can describe activities and objects (Control nodes, Executable nodes(→ action node), object nodes

Object Node

* **edge** directed connection between nodes
* decision with [ ]
* fork node for splitting and synch.

State-chart **vs** Activity Digram

Communication Diagram:

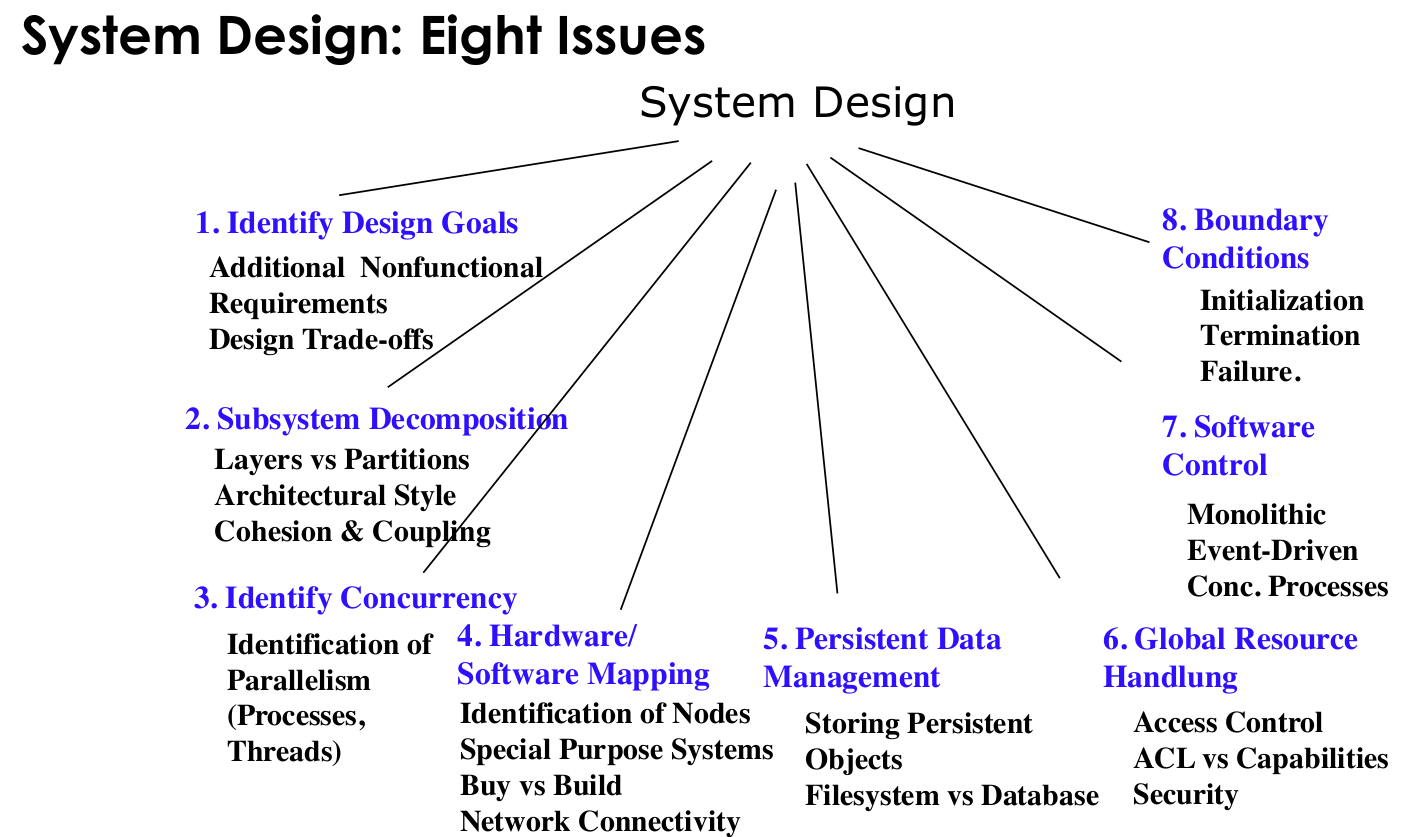
* visualizes interactions between objects as a flow of messages. Messages can be events or calls to operations
* describes the **static structure** of a system(obtained from class diagram→communication diagrams reuse layout of classes and associations)
* describes the **dynamic behavior** of a system(obtained from dynamic model(sequence & statechart diagram)
  + messages between objects are labeled with a number
* Message types:

Communication **vs**  Class Diagrams

* Association labels, roles and multiplicities are not shown in communication diagrams. Associations between objects denote messages depicted as a labeled arrows that indicate the direction of the message, using a notation similar to that used on sequence diagrams

Communication vs Sequence Diagrams

* Both focus on the message flow between objects Sequence diagrams are good at illustrating the event flow overtime. They can show temporal relationships such as causality and temporal concurrencies
* Communication diagrams focus on the structural view of the communication between objects, not the timing.



Nonfunctional Req.

Functional Model

Functional Model

Dynamic Model

Object Model

Dynamic Model

1Design Goal

* any Nonfunctional requirement is a design goal
* design goals often conflict with each other → design goal trade-offs

Design Goal Trade-offs

* functionality vs usability
* cost vs robustness
* efficiency vs portability
* rapid development vs functionality
* cost vs re-usability
* backward compatibility vs readability

2Subsystem Decomposition

Subsystems: collection of classes, associations, operations, events that are closely interrelated with each other

Service: group of externally visible operations provided by a subsystem(→**subsytem interface**)

Coupling vs Cohesion

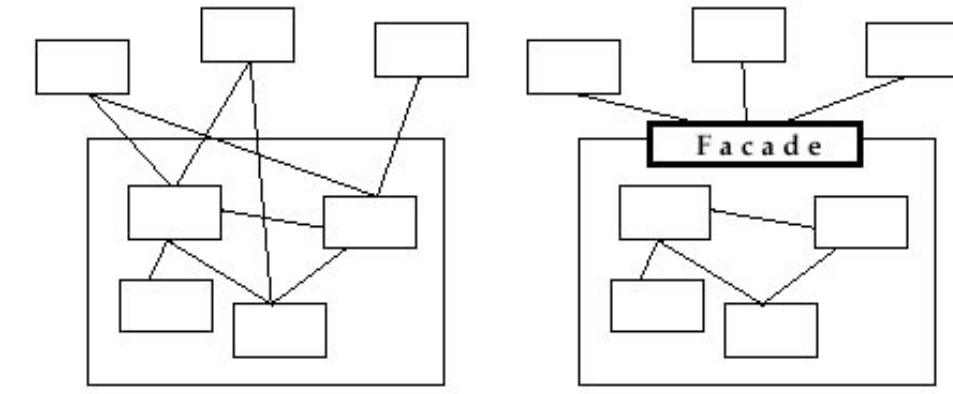
|  |  |
| --- | --- |
| Cohesion: measures dependency among classes  -**high cohesion**: the classes in the subsystem perform sir tasks and are related to each other via many associations  -**low cohesion**: lots of miscellaneous and auxiliary classes, almost no associations  Achieve high cohesion: operations work on same attributes | Coupling: measures dependency among subsystems  -**high coupling**: changes to one subsystem will have high impact on the other subsystem  -**low coupling**: a change in one subsystem does not affect any other subsystem  Achieve low coupling: smal interfaces,information hiding, no global data |

Architectural Style vs Architecture

* Subsystem decomposition: -identification of subsystems, services and their relationship
* Architectural Style: -pattern fir a subsystem decomposition(e.g layered architecture: closed=can only call from layer below, open=can call from any layer below)
* Software Architecture: instance of an architectural style

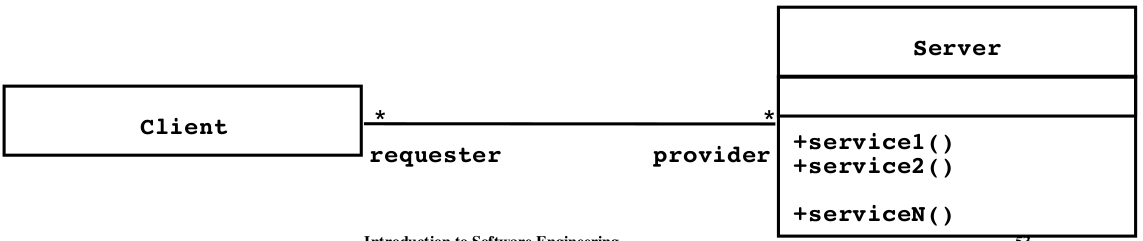
# System Design II

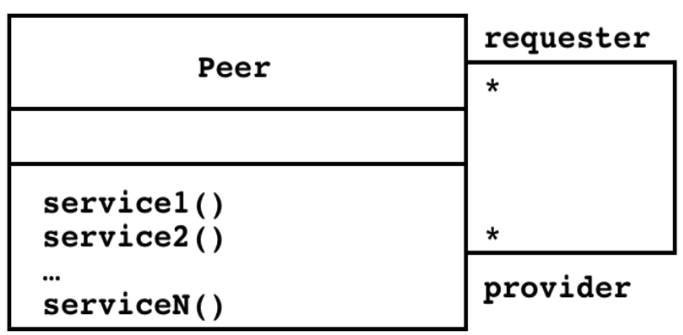
Patterns

The Facade(to reduce Coupling)

* a facade defines a higher-level interface that makes the subsystem easier to use

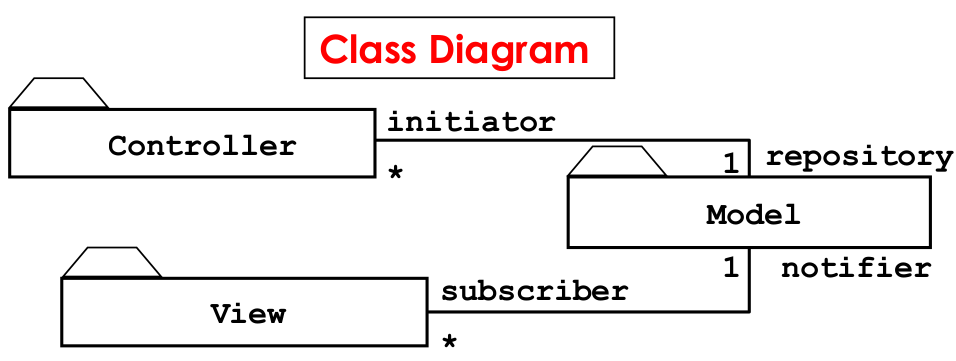
Client/Sever Architecture(special case of layerd architectural style)

* often used in database systems
* client: input from user, front-end processing of input
* server: centralized data manegement
* problems: client/server use a req-response protocol→ peer-to-peer is often needed

Peer-to-peer Architectural Style

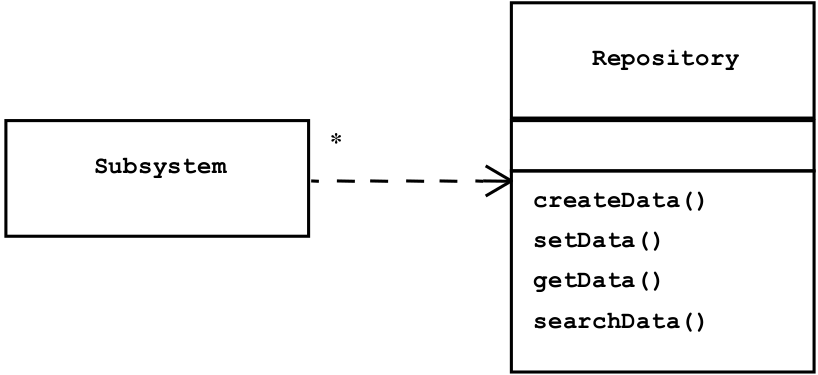
* generalization of client/server Architecture but clients can be servers and servers can be clients

Model-View-Controller

* decouples data access(entity objects) and data presentation(boundary objects)
* model: -a subsystem containing boundary objects
* view: -a subsystem containing entity objects
* controller: -a subsystem mediating between the view and the models

MVC vs. 3-Tier Architectural Style

|  |  |
| --- | --- |
| MVC:  -is nonhierachical (triangular)  -v sends updates to c, c updates m, v is directly updated from m | 3-tier:  -is hierarchical(linear)  -presentation layer never communicates directly with data layer  -all communication must pass through middleware layer |

Repository Architectural Style

* support a collection of independent programs(=Subsystems) that work cooperatively on common data structure (=Repo.)

Elements of an Architectural Style

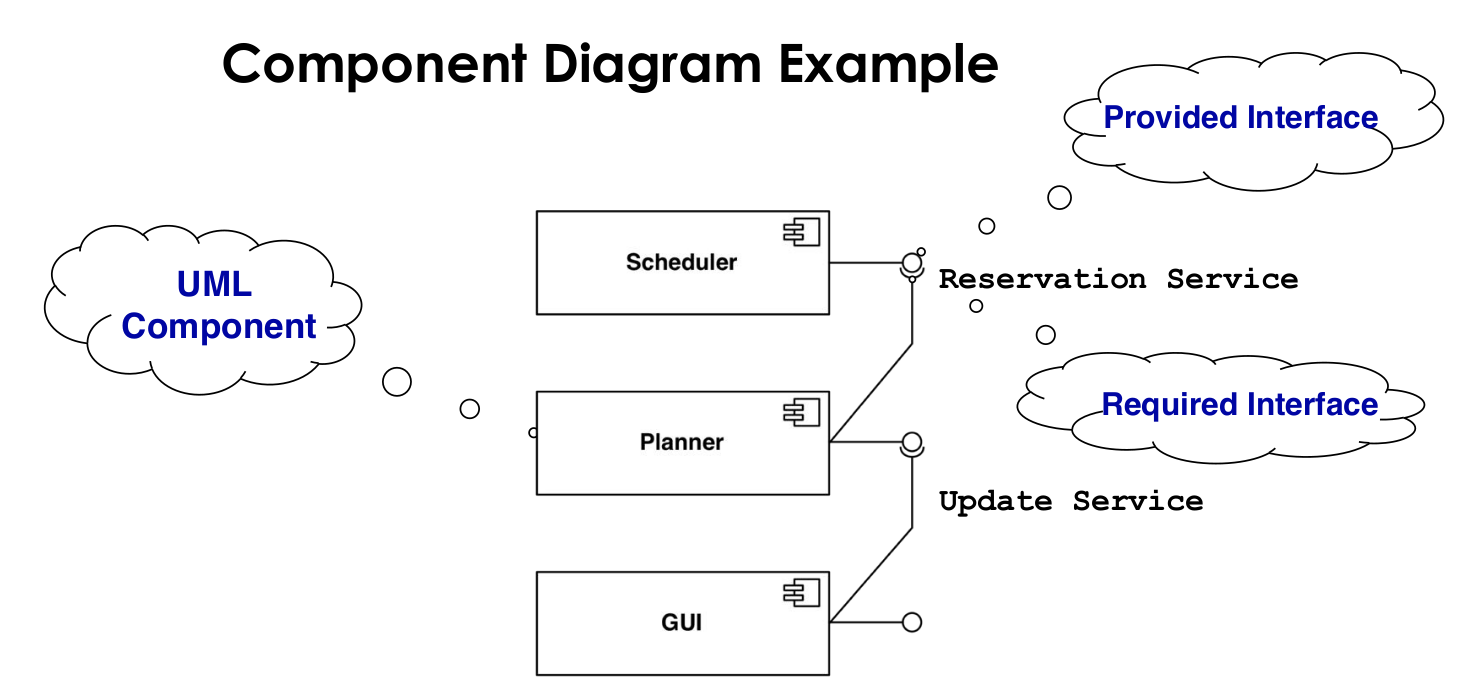
* Components(Subsystems): units with a specified interfaces
* Connectors(Communication): interactions between the components

3Concurrency

* used to address nonfunctional req. such as: performance, response time… using threads

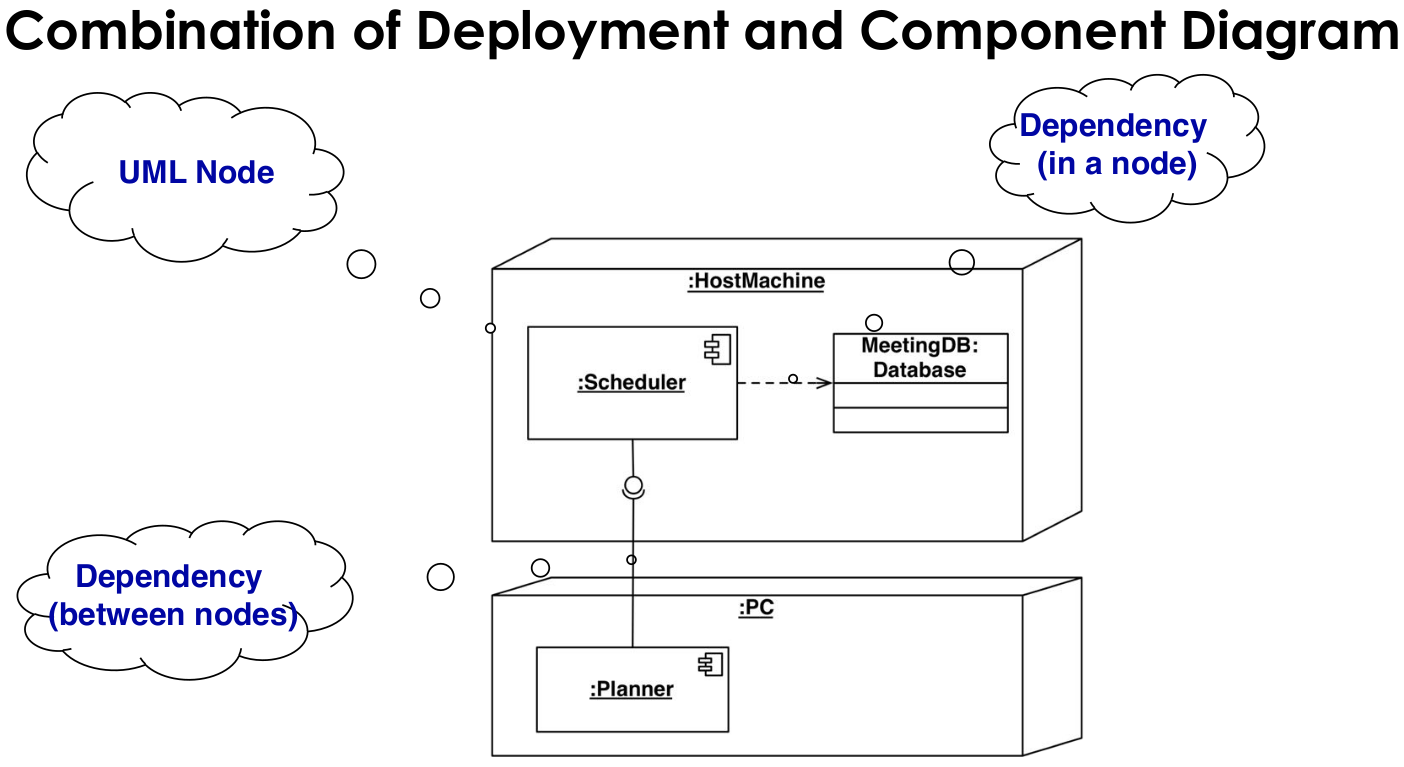
4Hardware Software Mapping

* Control Objects → Processor
* Entity Objects → Memory
* Boundary Objects → I/O Devices

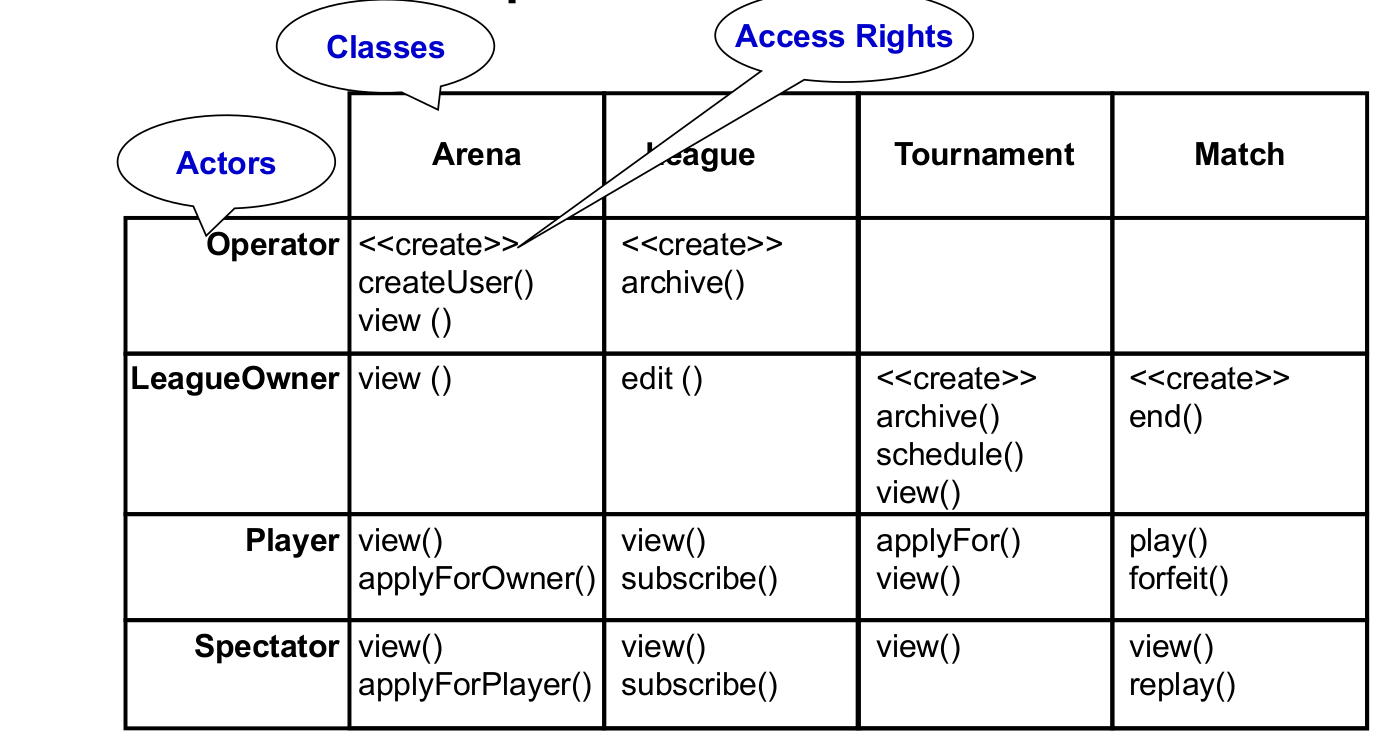
Component Diagram

* shows dependencies between components at design, compilation and run-time
* UML interface describes a group of operations provided or required by a component

Deployment Diagram

* shows the distribution of components at run-time
* uses nodes and connections to depict the physical resources in the system

5Persistent Data Management

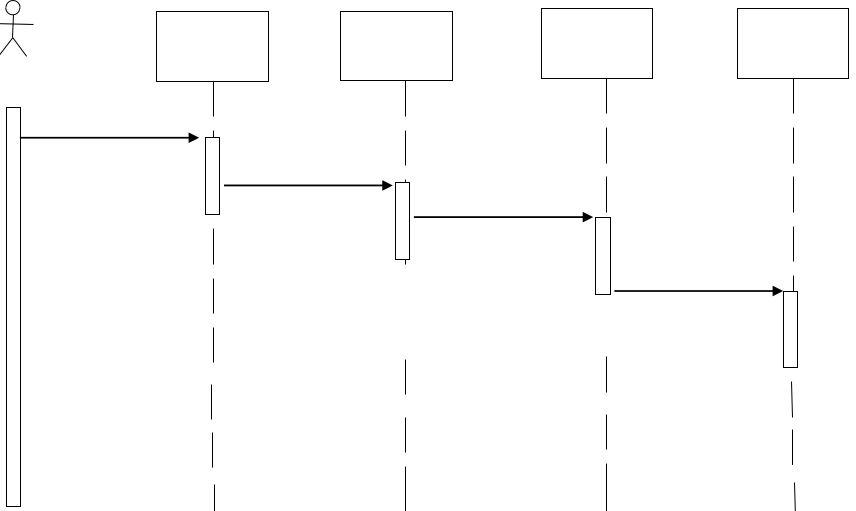
* **persistency**: a class is persistent, if the values of their attributes have a lifetime beyond a single execution(can be done with a **file system** or a **database**

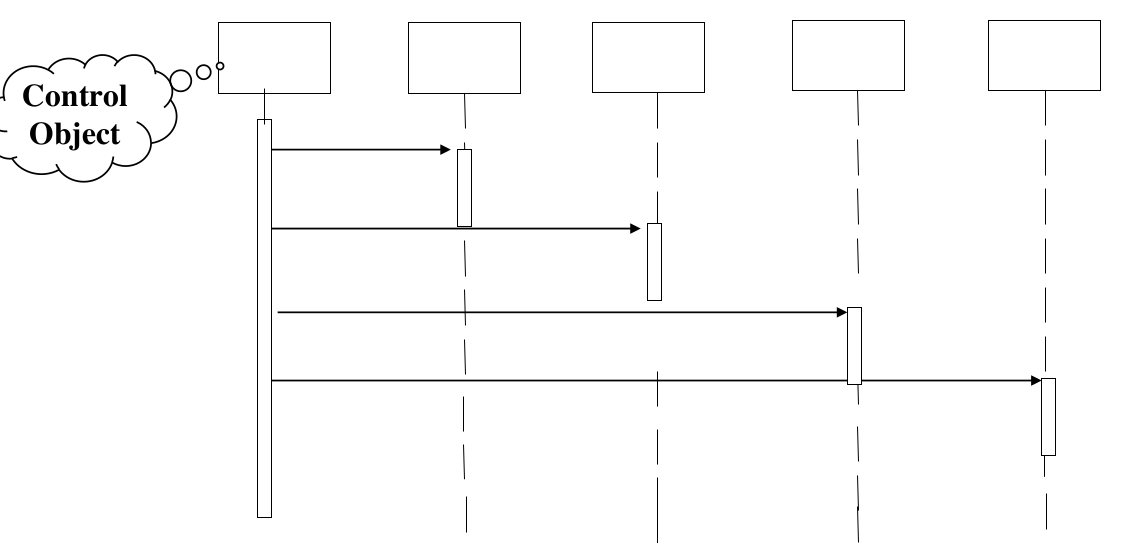
6Global Resource Handling

* addresses access control→ done with an access matrix(rows=actors, column=classes)

7Software Control

* implicit software control(=role-based systems, logic programming)
* explicit software control(=centralized control,decentralized control)
* 2 sequence diagram structures to determine decentrilasation

  
Illustration 2: Stair Diagram

  
Illustration 1: Fork Diagram

8Boundary Conditions

* Initialization: -system is brought from a non-initial state to a steady-state
* Termination
* Failure

# Object Design

* Purpose: -prepare for the implementation, transform the system model

Consists of 4 Activities:

* Reuse: Identification of existing solutions: -use of inheritance, use of design patterns
* Interface specification: -describes precisely each class interfaces
* Object model restructuring
* Object model optimization

Reuse in Object Design

* Composition (also called black box reuse) = A new class is created by the aggregation of the existing classes. The new class offers the aggregated functionality of the existing classes
* Inheritance (also called white box reuse)= A new class is created by subclassing. The new class reuses the functionality of the superclass and may offer new functionality.

Interface Specification

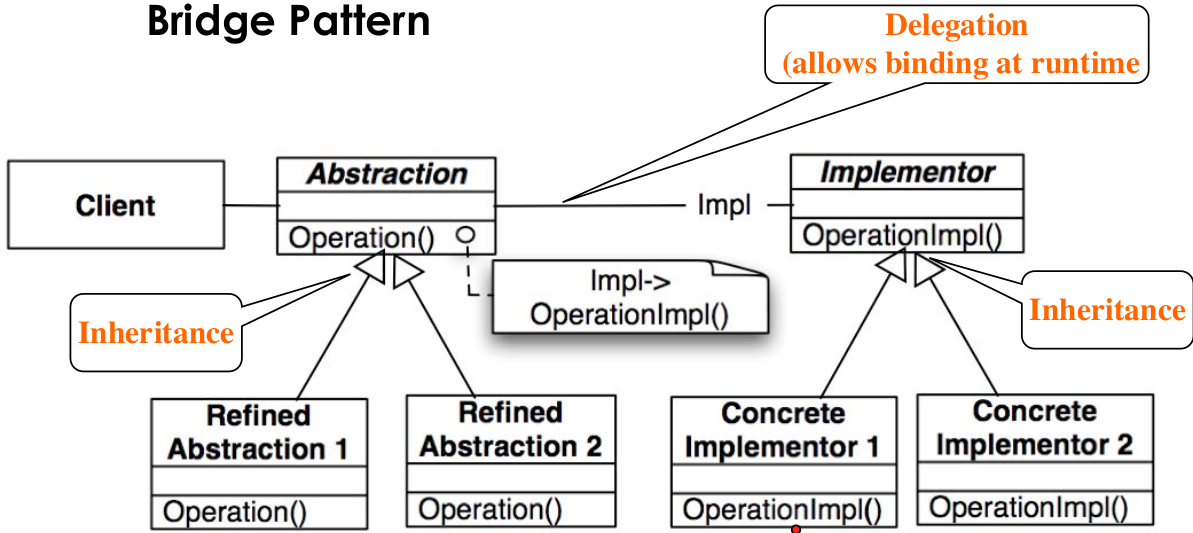
* Implementation Inheritance: reuse: implemented functionality in the **super class**
* Delegation: reuse: implemented functionality in an existing object
* Specification Inheritance: reuse: specified functionality in the superclass, but functionality in the object

Implementation Inheritance vs Delegation

|  |  |
| --- | --- |
| Implementation Inheritance:  -extends a superclass with a subclass containing a new operation or overriding an existing operation | Delegation:  -catches an operation and sens it to another object |

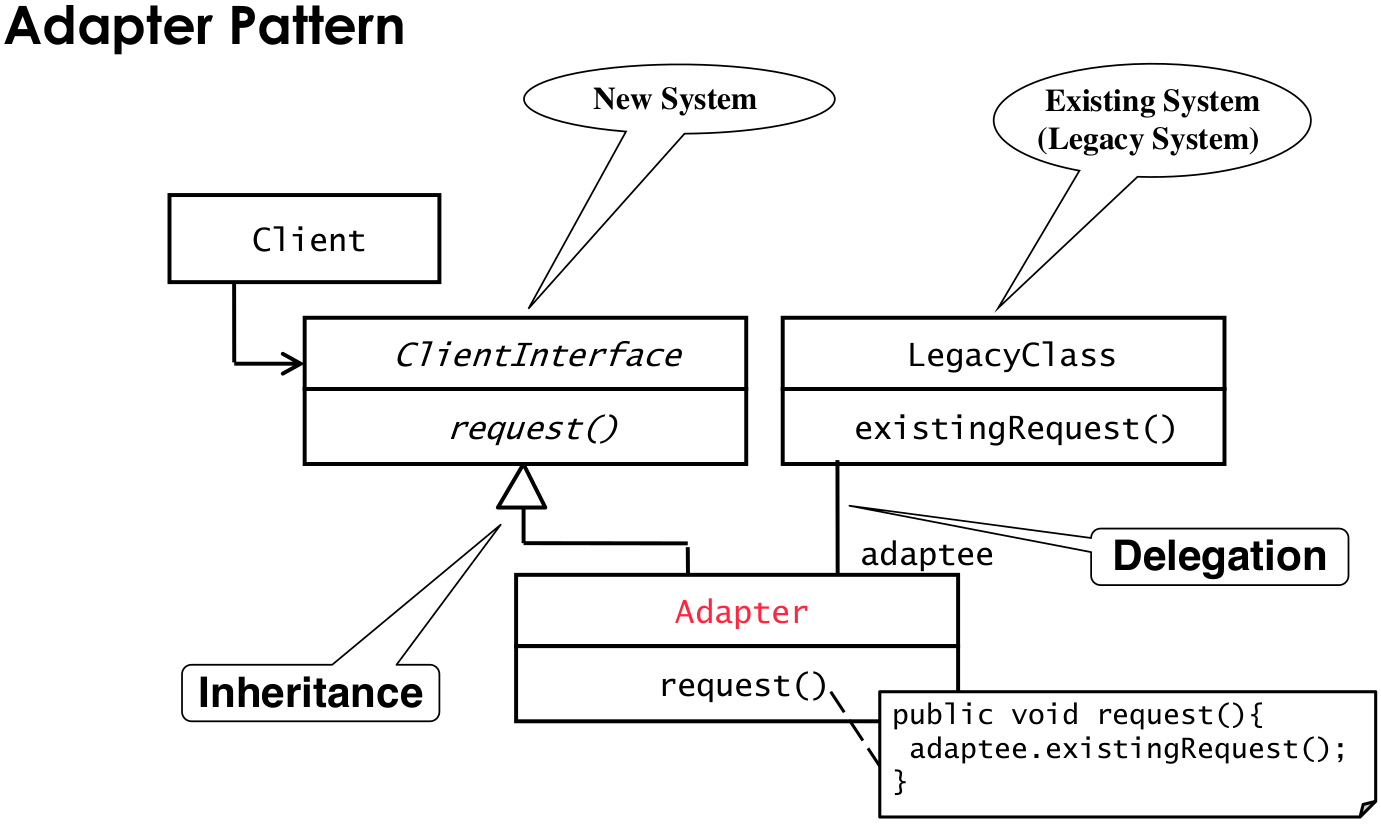
Design Patterns

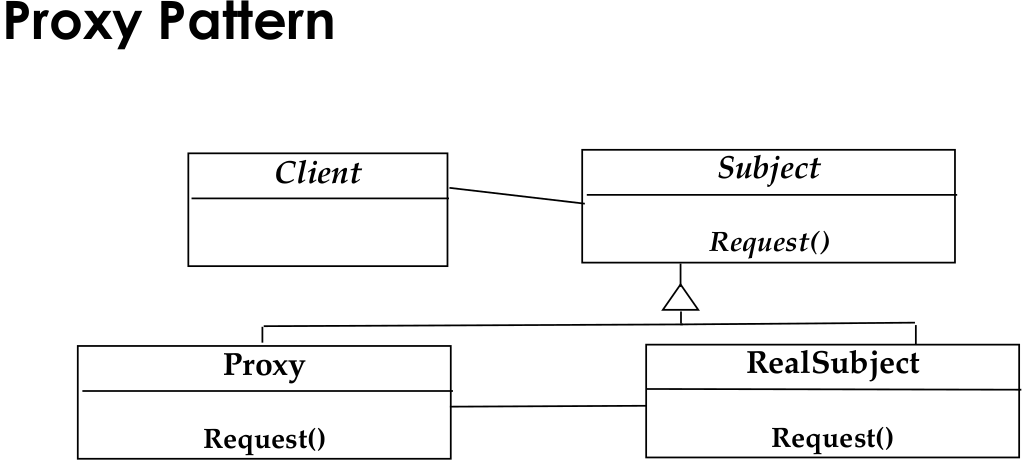
* Structural Patterns: -reduce coupling between, -introduce an abstract class for future extensions, -encapsulate complex structures
* Behavioral Pattern: -allow choice between algorithms
* Creational Patterns: -allow to abstract from complex instantiation processes



Bridge Pattern

* Problem: many design decisions are made final at design or compile time
* Bridge pattern allows to delay the binding between an interface and its subclass to the start-up time of the system

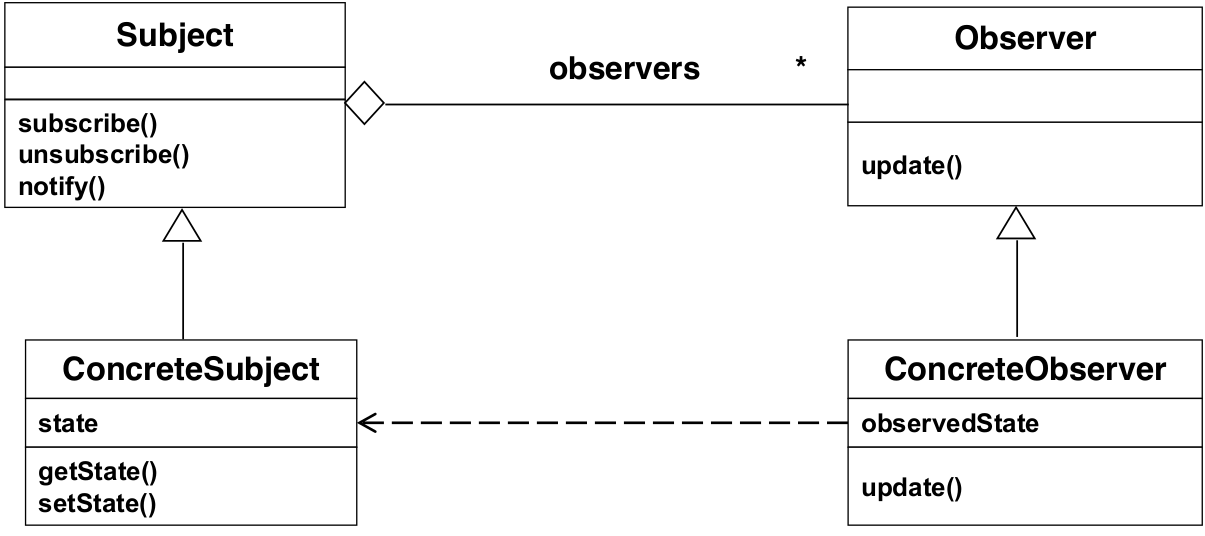
Adapter Pattern

* connects incompatible components(=wrapper)
* Proxy Pattern
* real subject is expensive to acces or to create→ proxy-object acts as a representative for the remote object and access the remote object only if really needed
* applicability: -caching(**remote proxy**), -substitute(**virtual proxy**)=object is expensive to create or to download, -access control(**protection proxy**)

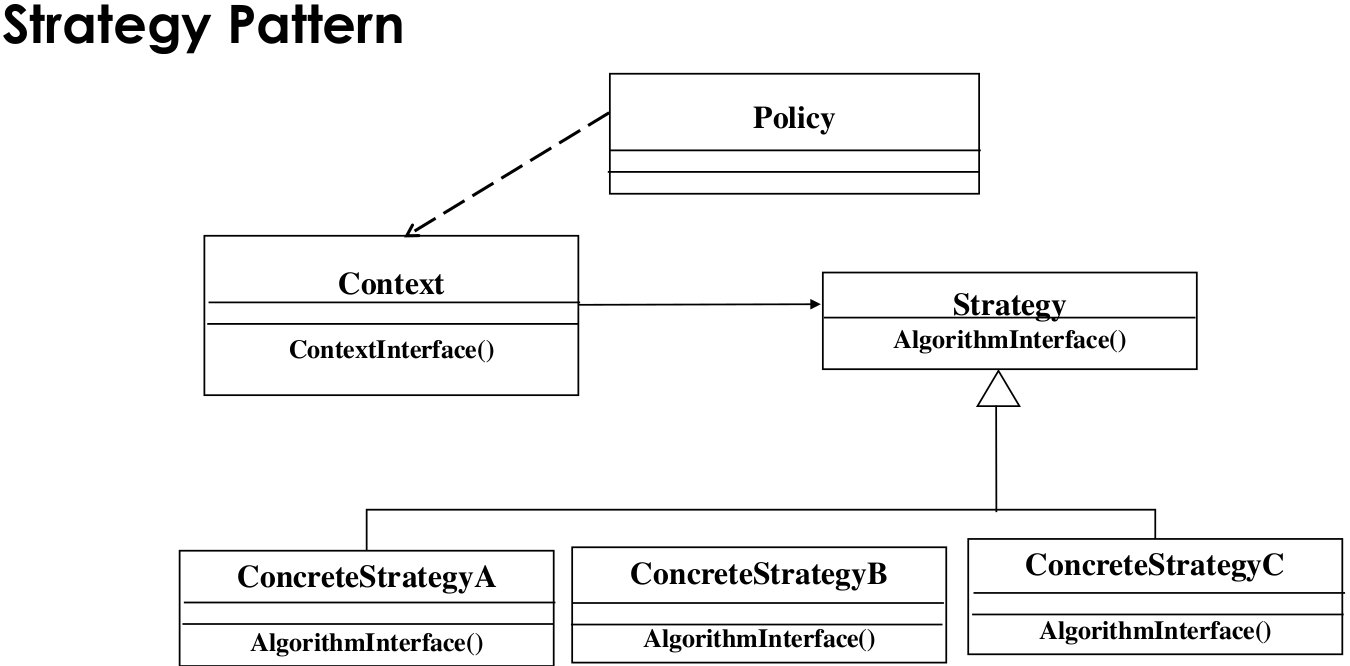
Adapter vs Bridge

-both hide the details of the underlying implementation

|  |  |
| --- | --- |
| Adapter:  -adapter pattern making unrelated components work together  -applied to systems that are already designed  →inheritance followed by delegation | Bridge  -is used up-front in a design to let abstractions and implementations vary independently  →delegation followed by inheritance |

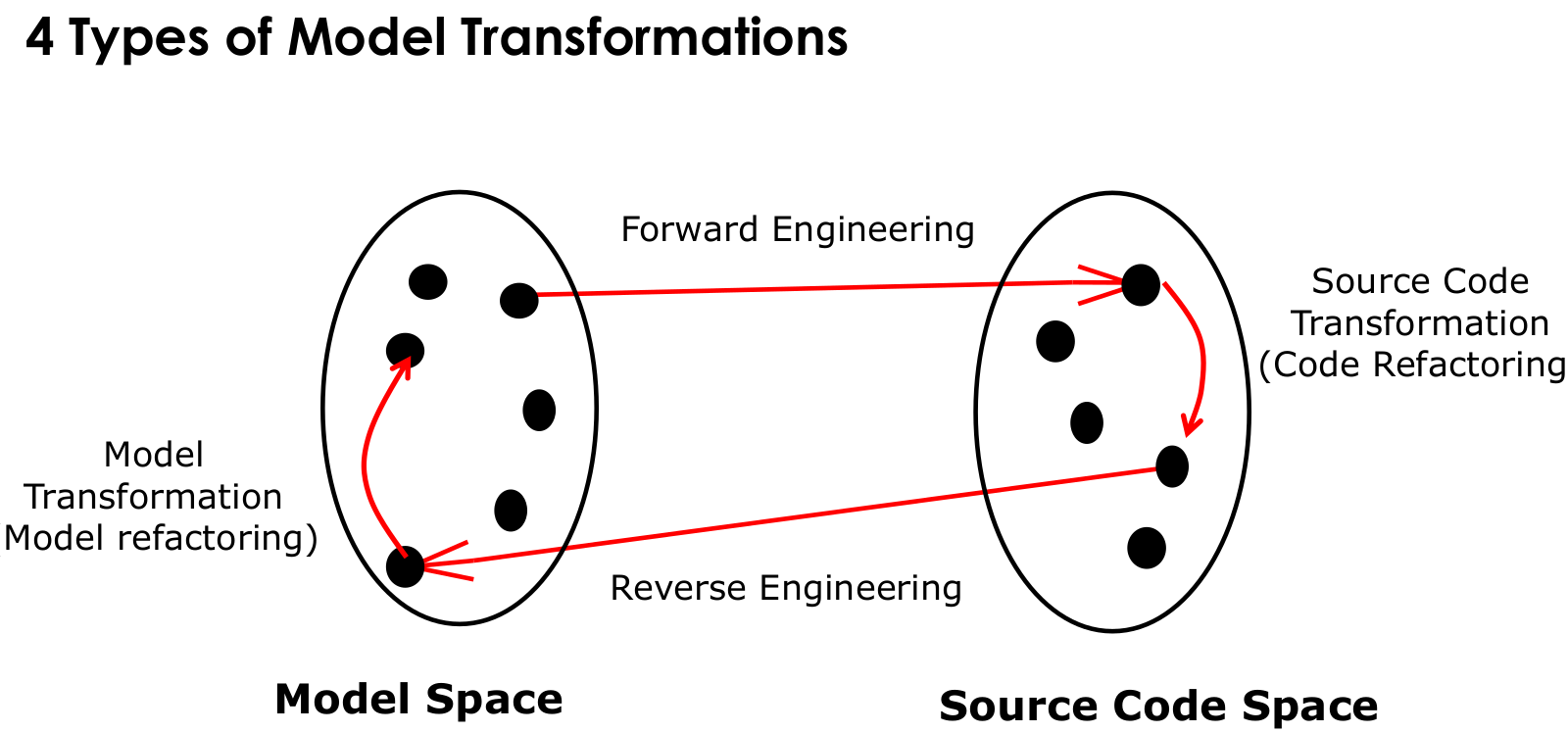
Observer Pattern (Publish and Subscribe)

* problem: object that changes its state quite often
* models a 1-to-many dependency between objects: -connects the state of an observed object(=**subject**) with many observing objects(=observers)
* usage: -maintaining consistency across redundant states

Strategy Pattern

* problem: different algorithms exist for a specific task → switch between algorithms at run time
* policy decides which “ConcreteStrategy” is best in a given context

# Model Transformations and Refactoring

Model Based Software Engineering

* application of modeling to support req. , design, analysis

Refactoring

* a change made to the internal structure of source code to make it: -easier to understand, -without changing its observable behavior

Refactor

* to restructure source code by applying a series of refactoring

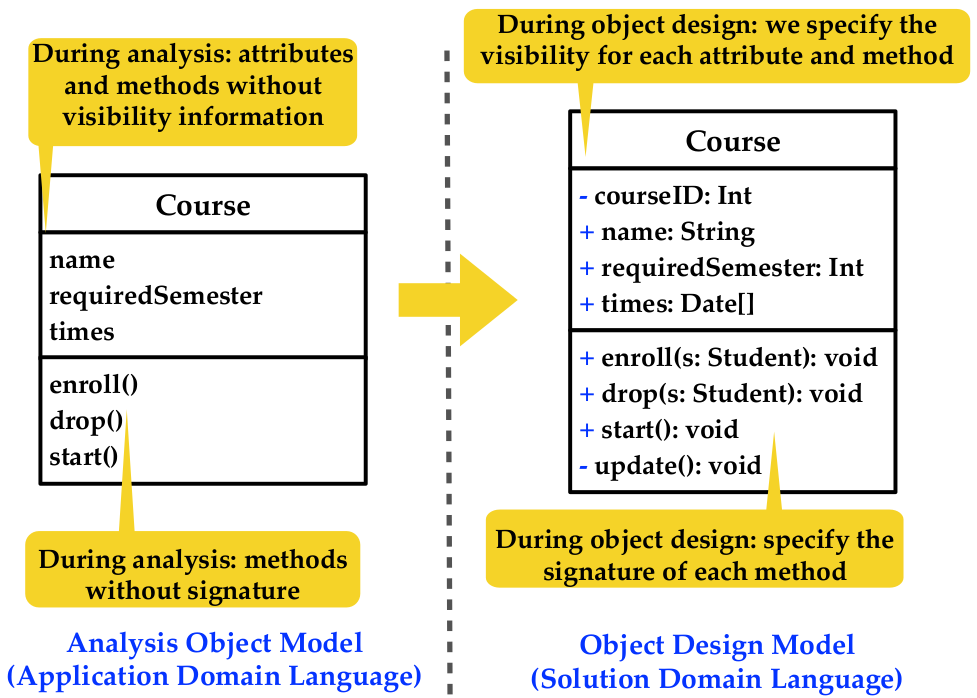
# Pattern Based Development

UML

Pattern in UML

* Context: sets stage where pattern takes place
* Forces: describes why problem is difficult to solve

Examples

* Text: “must interface with an existing object” → Adapter Pattern
* Text: “must interface to several systems, some of them to be developed in the future”, “an early prototype must be demonstrated”, “must provide backward compatibility” → Bridge Pattern
* Text: “must interface to existing set of objects”, “must interface to existing API”, “must interface to existing service” →Façade Pattern
* Text: “complex structure”, “must have variable depth and width” → Composite Pattern
* Text: “must provide a policy independent from the mechanism”, “must allow to change algorithms at runtime” →Strategy Pattern
* Text: “must be location transparent” → Proxy Pattern
* Text: “must be extensible”, “must be scalable” →Observer Pattern (MVC Architectural Pattern)

# Software Life Cycle Models

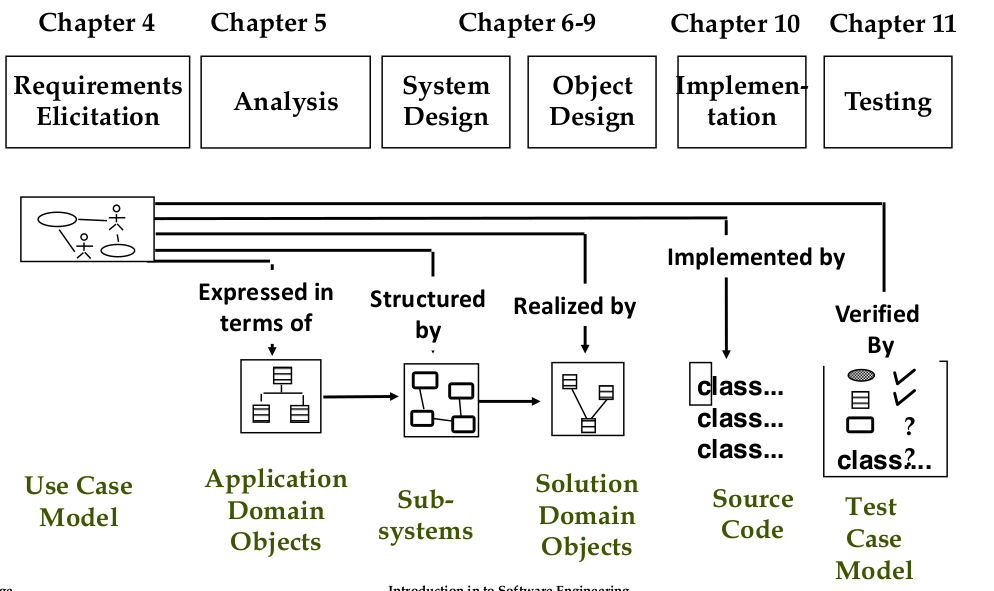
Software life cycle:

Analysis Object Model vs Object Design Model

* set of activities and their relationships to each other

Software life cycle model:

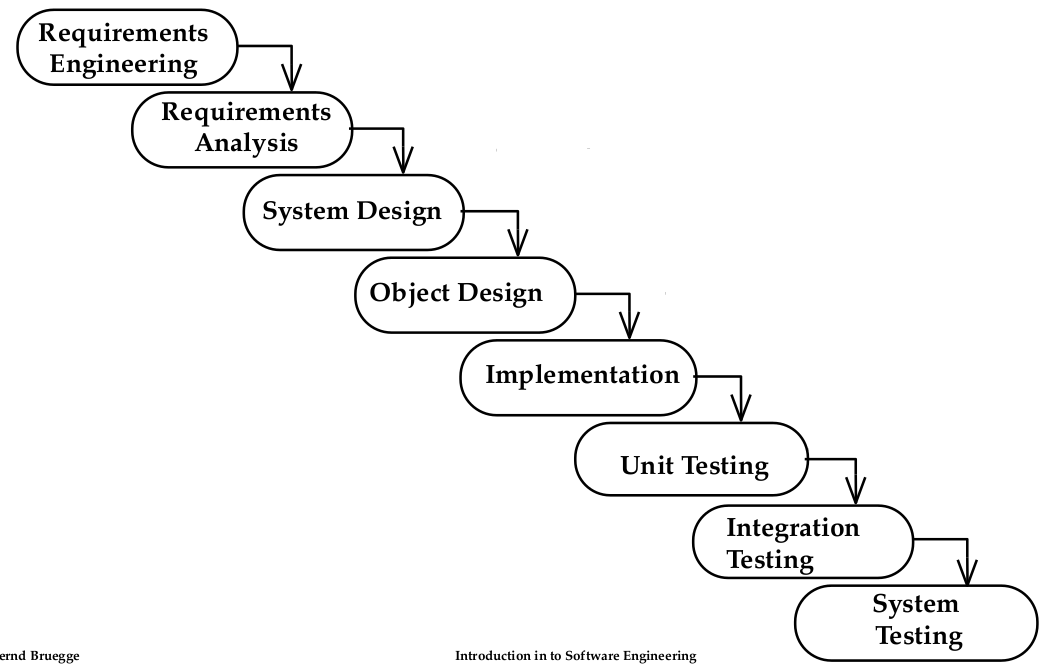
* abstraction representing the development of software

Major Views of Software Life Cycle

* Activity-oriented view: software dev. Consists of a set of development activities (first: problem definition activity and system dev. Activity, then system operation activity)
* Entity-oriented view: software dev. Consists of the creation of a set of deliverables(market survey, system specification documents, executable system, lessons learned document)

Tailoring: -adjusting a life-cycle model to fit a project(naming, cutting, ordering)

Software lifecycle activities



← Waterfall Model (activity diag.)

Properties:

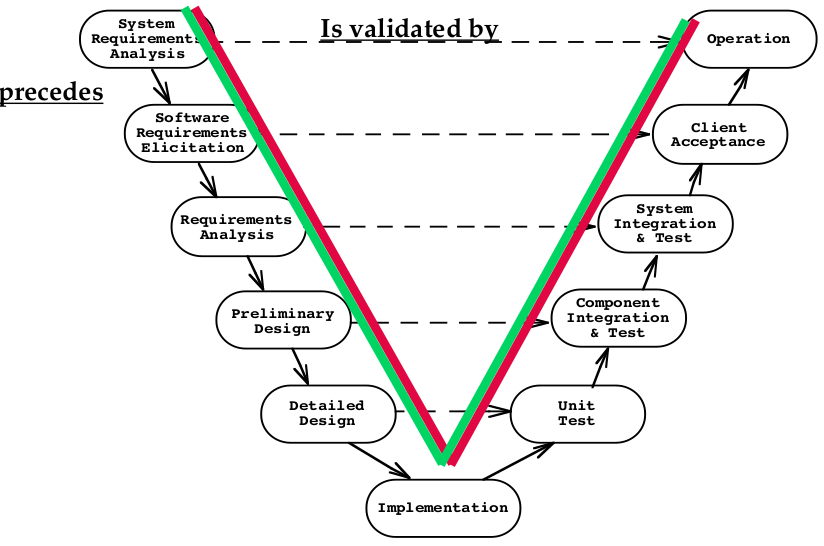
-linear system

-always one activity at a time

-easy to check progress during development

Problems:

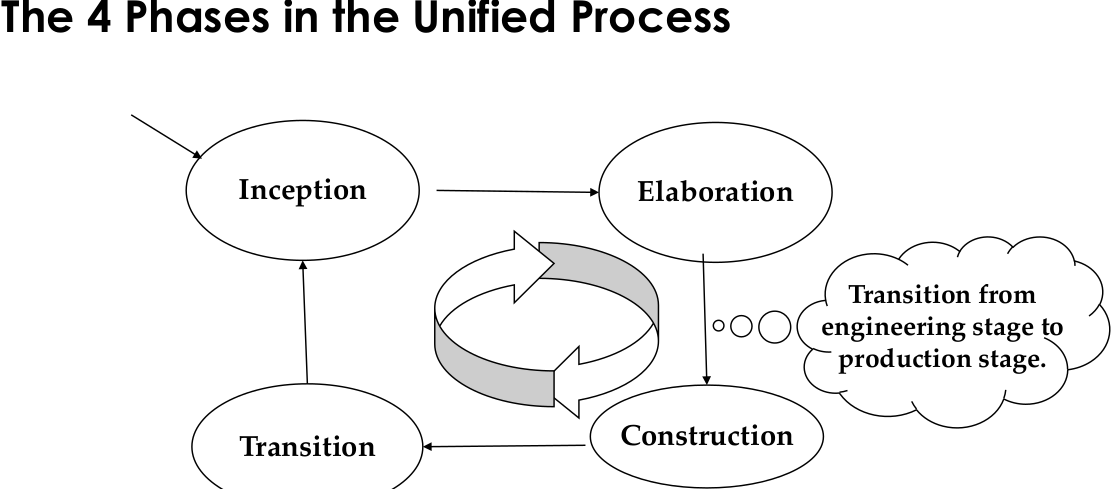
-software dev. Is not linear

Spiral Model

V-model→

* iterative model with 4 activities(are applied iteratively in each of 9 rounds)
* **rounds**:(1. Concept of Operations, 2. Software req. , 3. Software Product Design, 4. Detailed design, 5. Code, 6. Unit Test, 7. Integration and Test, 8. Acceptance Test, 9. Implementation)
* each round goes through this activities: 1. Define objects, alternatives, constraints ; 2. Evaluate alternatives, identify and resolve risks; 3. Develop and verify a prototype, 4. Plan the next round

Unified Process

* each cycle consist of 2 **stages** and 4 **phases**
* stages: -Engineering stage(smaller teams,focusing in design activities) with 2 phases(Inception, Elaboration), -Production stage(lager teams, focusing on construction, test and deployment activities) with 2 phases(construction and transition)

Linear and Spiral Model Limitation

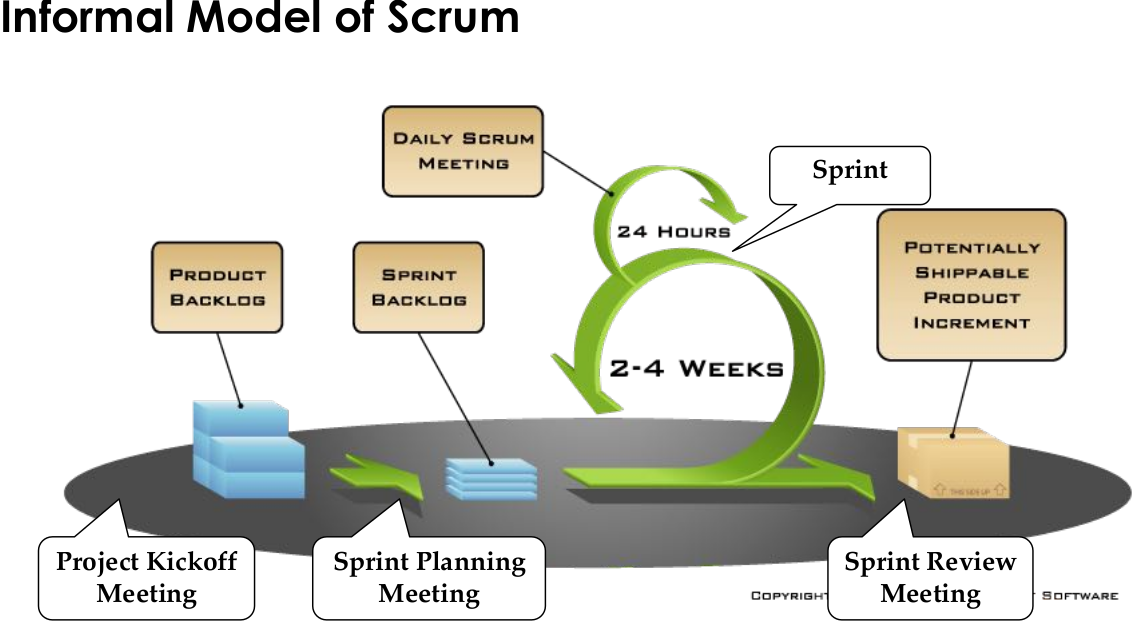
* both don’t deal with frequent change

Frequency of Change

* no change during project → linear model(waterfall, v)
* infrequent changes during project → iterative models (spiral, unified process)
* changes are frequent → agile model (scrum)

Incremental vs Iterative vs Adaptive

|  |  |  |
| --- | --- | --- |
| Incremental(=add onto something)  -improves your process | Iterative(=redo something)  -improves your product | Adaptive(=react to changing requirements)  -improves the reaction to changing customer needs |

Scrum

* Scrum is a technique to manage and control software and product development when the requirements and the technology may be rapidly changing during the project.

Core Components

* 3 Artifacts(Product and Sprint Backlog, PSPI)
* 4 Meeting Activities(Kickoff Meeting, Sprint Planing Meeting: list of prioritized features, Daily Scrum, Sprint Review Meeting)
* 3 Roles(Scrum Master: responsible for process, Product Owner: responsible for product, Developer: responsible for realization of the PSPI)

Scrum Roles

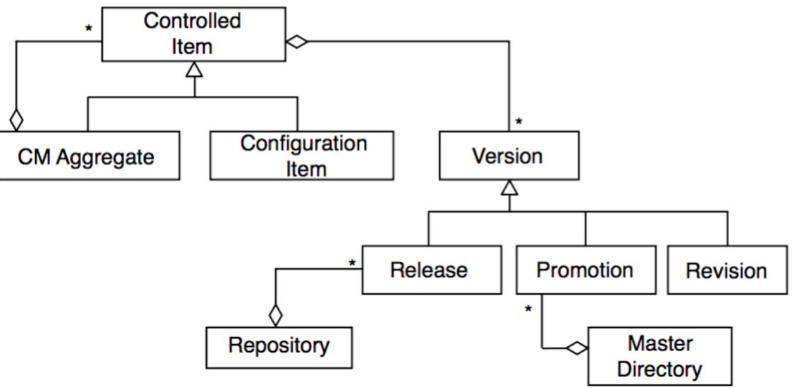
* Scrum Master: -should moderate and coach the team, -main job is to remove impediments[[5]](#footnote-6)
* Product Owner: knows what need to be build and in what order, responsible for product, value and prioritzation
* Scrum Team: 5-6 people, cross-functional (analyst, programmer,designer, tester), team is self-organizing

# Software Configuration Management

* Set of management disciplines within a software engineering process to develop a baseline

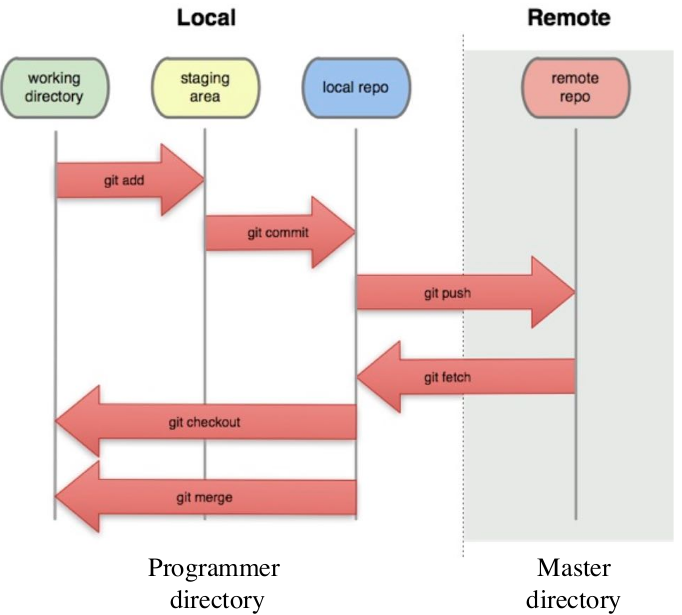
Configuration Management Activities

* Configuration item identification: modeling the system as a set of evolving components

  
Illustration 3: Object Model for Configuration Management

* Change management: management of change requests (e.g repository architecture)
* Promotion management: creation of versions for other developers
* Branch management: management of concurrent development
* Release management: creation of versions for clients and end users
* Variant management: management of coexisting versions

Git

Merge Conflict: happen if two persons work on the same lies in the same file at the same time

**Continuous Integration**: technique where members of a team integrate their work frequently

**Continuous delivery**: team keeps producing valuable software in short cycles and ensure that software ca reliable released

**Continuous deployment**: every change that passes automated test is deploey automatically

# Testing

1. Requirements Elicitation and Analysis [↑](#footnote-ref-2)
2. Ganz bestimmte relation, die nur einmal geerbt wird, zb “no charge” kann nur von “ColectMoney” extenden [↑](#footnote-ref-3)
3. Nicht spezifische relation, kann von mehreren erben “purchase card” muss nicht nur von “collect money” erben [↑](#footnote-ref-4)
4. Deutsch: Lastenheft [↑](#footnote-ref-5)
5. Hindernisse [↑](#footnote-ref-6)