1. Software Engineering

SE vs programming

1) Time & Changes → sustainable, adaptable code
2) Scale & Growth → team effort
3) Trade-offs & Costs → no single solution, context

def| the application of systematic, disciplined, quantifiable approach to the development, operation and maintenance of software

Timeline

1880-1936 Software before hardware

1843 - Ada Lovelace, first programming language, Analytical Engine, Bernoulli numbers

1936 - Alan Turing, Turing machine, basis to Computer Science and Theory of Computability

1937-1960 Hardware Era

Colossus - cryptanalysis of German communication (WW2); binary arithmetic (switches plugs)

ENIAC - solve numeric problems, decimal operators (loops, conditions, subroutines)

1960-1968 Software Crisis

computational power of machines → grow several orders of magnitude

1968-1969 First Software Engineering Conference

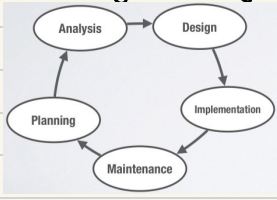
1970s Waterfall Model

1975 "The Mythical Man-Month", Fred Brooks

Brooks' Law: adding manpower to a late software project makes it later

"there is no single development, in either technology or management technique, which by itself promises even one order of magnitude improvement within a decade in productivity, in reliability, in simplicity"

Software Development Life Cycle - process for planning, designing, developing, testing and deploying software projects



Analysis → requirement engineering

requirements → description of systems services and constraints

2 Requirement Engineering

stakeholder - person or organization that influences a system's requirements or is impacted by that system

- active → client, user
- passive → environment constraint (legislation)

elicitation techniques

- → survey
- → creativity
- → document-centric
- → observation
- → support

requirement classification:

Functional - what system should do (functionality, input, output, exception)

non-functional

- → product (behavior: performance, memory, reliability)
- → organizational (policies: standards, language, OS)
- → external (legislative)

MoSCoW method

- → Must
- → Should
- → Could
- → Won't

3 Domain Driven Design

software design - process of defining the architecture components, interfaces and other characteristics of a system or component

def: approach to software development that centers the design around processes and rules of a domain

ubiquitous language - common, rigorous language between developers and domain experts

storytelling - technique of "recording" specific examples of what happens in the system

bounded contexts - sub-concepts within the whole domain

types of domains:
- -> core
- -> generic
- -> supporting

context map - illustrates how contexts (boundaries) are defined, their integration points and the flow of data between bounded contexts

relationship: upstream (influence), downstream (influenced)

organizational patterns:
- partnership (cooperation between U and D)
- conformism (U doesn't support D, D conforms to U)
- anti-corruption layer ("translation" layer to equate semantics)
- shared kernel
- customer-supplier (U take into account needs of D)
- open-host service (protocol for inter-context access)
- published language
- separate ways

monolithic architecture - all bounded contexts in one module

Scalability Cube:
- X-axis -> load balances requests across multiple, identical instances (cloning)
- Z-axis -> routes requests based on attribute of request (splitting similar things)
- Y-axis -> decomposition of application into services (splitting different things)

service - mini application that implements narrowly focused functionality

microservice-based architecture:
- -> adhere to principles such as fine-grained interfaces
- -> small changes require rebuilding and redeploying one or a few services

Advantages:
- → Modularity - microservices are easy to maintain, test, understand
- → Scalability - independent monitor and scale of microservices
- → Integration - can be integrated with legacy systems
- → Distributed development (independent)

Disadvantages:
- → Network overhead - higher network latency and message processing time
- → Deployment - more complicated
- → Security - larger attack surface with larger data
- → Moving responsibilities

Ports and Adapters (Hexagonal) Architecture

Advantages:
- → Testability - independently tested core logic
- → Maintainability - replace one service with another

Disadvantages:
- → Security & Reliability
- → Higher complexity

domain layer → behaviour and constraints
application layer → orchestrates what happens in domain
framework layer → code used by application, but is not application
outside layer → connections to others

Inversion of Control
interface in lower layer (Port), implementation in higher level (Adapter)

Domain Layer:
- → Use cases - objects that specify intent (DTO - Data Transfer Object)
- → Domain events - interface and objects that allow other parts of the application to act on events without tight coupling
- → Repositories - interface for querying a data resource

Application Layer:
- → Command Bus – orchestration component
- → Event Dispatcher – dispatches events to listeners
- → Service Interfaces – events fired at important points in the business logic
- → Hibernate ORM – database abstraction layer

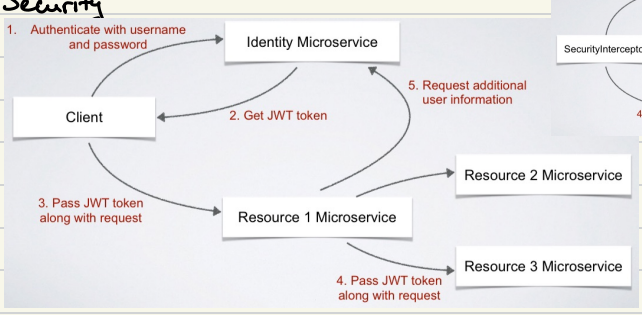
Framework Layer:
- → HTTP/CLI/API – interfaces for communicating with users and other services
- → Service Implementation – libraries for connecting with third party services
- → Database – physical connection to database engine

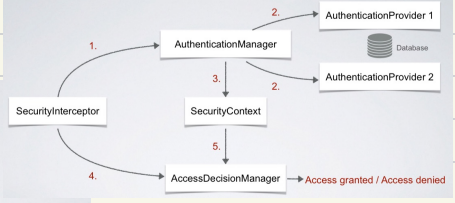
Value Objects – immutable objects whose attribute values are important
Entities – equality/interchangability determined by unique id (attributes can differ)
- → publish Domain Events to communicate state of business logic state

Aggregates – clusters of associated objects acting as a unit for data changes
Services – stateless; manipulate business logic elements of objects

Spring
@Entity + @Id; no-argument constructor
@Embeddable – complex value objects; no final fields; set methods in constr.
Events – basic immutable DTOs; timestamp + unique identifier
@DomainEvents (collecting), @AfterDomainEventPublication (clearing)
Handlers – automatic when Data base class called (@EventListener)

Security





4. Software Architecture

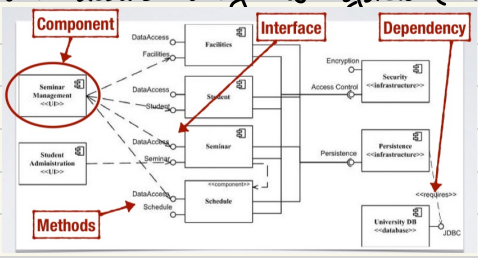
def.] the fundamental concepts or properties of a system in its environment embodied in its elements, relationships and in the principles of its design and evolution

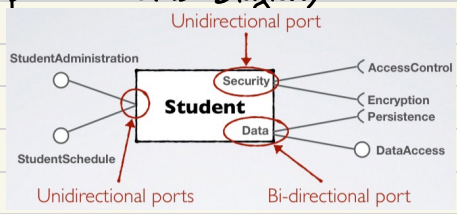
1969 - NATO Conference (discussion on structure of software systems)
1980s - "system" architecture (physical computers)
1991 - software architecture as discipline
2000 - first IEEE standard in software architecture

Software Architecture - how components of a software system are organized and assembled, how they communicate with each other, and the constraints that rule the whole system
→ Architectural Patterns - how components are organized
→ Messages + APIs - how components interact with one another
→ Quality Attributes - constraints + non-functional requirements

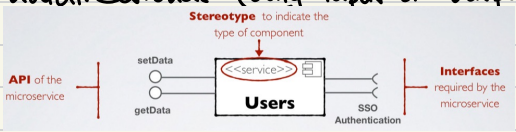
Component Diagram

→ UML - graphical modeling language; elicit requirement (Use Case) and describe design of system (Component + Class Diagram)





sockets (—C) - interfaces component requires (input)
lollipops (—O) - interfaces given component provides (output)
unidirectional (only input or output) or bi-directional (both)



Architectural Patterns

- general, reusable solution to commonly occuring problem

→ Client & Server

server - hosts, delivers and manages most of resources and services to be consumed by client
- invoked to provide set of services

client - uses provided services to initiate communication (remote messages or invocations)

→ Layered Architecture

system - decomposed into groups of layers, each of which is at particular level of abstraction

layer - provide services to next layer, consume from lower one

module (class or package) - allocated to only one layer

Three-tier Architecture

→ Presentation Layer (HTML page, GUI, user interface)

→ Business Layer - real point logic of system

→ Storage Layer - communication with databases, messaging systems, transaction managers, other packages

Advantages:

→ Testability - test layer in isolation (mocks and stubs)

→ Maintability - independent to change layers

→ Flexibility - interchangable layers if same interface is implemented

→ Reusability

Disadvantages:

→ Higher complexity - additional abstraction

→ Lower performance - request may pass through all layers

→ Model-View-Controller (MVC)

- application divided into three main layers

→ View - user input and output (HTML)

→ Controller - forward requests to model and results to view

→ Model - verify request and output results (DataBase)

Layer - sapare system into logical modules with design elements

Tier - physical layer where layers are deployed

→ Pipe-and-Filter Pattern
- system that produces a data stream
→ Pipe - order-preserving connector that transports data between filters
→ Filter - component that reads data and returns transformed one
- Sorce - origin of data in the system (user input)
- Sink - component receiving final data (system output)

Advantages:
→ Extendability - easy to add filters
→ Testability - independent testing of filters
→ Concurrency - consistancy of stream
→ Reusability

Disadvantages:
→ Performance - large data transformation overhead
→ Deadlocks
→ Low Responsiveness

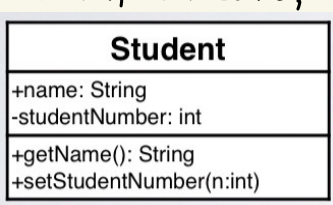
→ Publish-Subscribe Pattern
- producers and consumers exist independently and unaware
→ Publisher - component that publishes (documented) event
→ Subscriber - component that subscribes to event
→ Event Bus - registers subscriptions and delivers events
message filtering - by topic or content

Advantages:
→ Reusability
→ Testability
→ Scalability
→ Separation
→ Loose Coupling

Disadvantages:
→ Bottleneck (event bus)
→ Security
→ Availability

→ Event-Driven Architecture
- producers (agents), consumers (sinks) and channels
events - inputs that trigger state change
producers - detect, gather and transfer data
consumers - apply reaction to event
→ distributed asynchronous architecture
Mediator Topology:
→ Queue - transport event to mediator
→ Mediator - send asynchronous events to channels
→ Processor - listen to channels and execute business logic
Broker Topology:
- no central mediator
→ Broker - contains all channels
→ Processor
Advantages:
→ Real-time
→ Loose Coupling
Disadvantages:
→ Increased Complexity
→ Security
→ Reliability

5. Design Patterns
Class Diagrams (UML)
→ classes, attributes, methods (operations), relationships

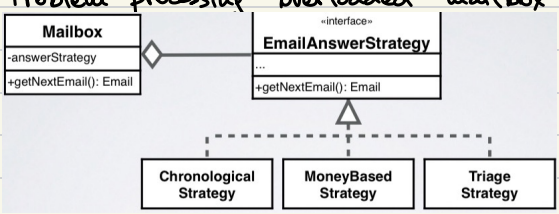


visibility: + public, - private, # protected, ~ package
static → underline
abstract → italics
A extends B: A ——▷ B
A implements B: A - - ▷ B
associations, A to B is 1 to many: A 1 ——* B

aggregation (A ◇—— B): A contains B, B can live without A
composition (A ◆—— B): A manages lifecycle of B, B cannot exist without A

Cohesion vs. Coupling
cohesion - degree to which methods inside module belong together
coupling - degree to which one class relies on others
high cohesion - independent classes, facilitate reuse
low coupling - higher maintainability, simplified modification

Behavioral Patterns - realization of common communication patterns
→ Strategy
Problem: processing overloaded mailbox

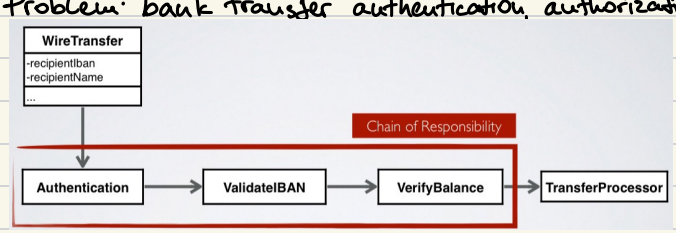


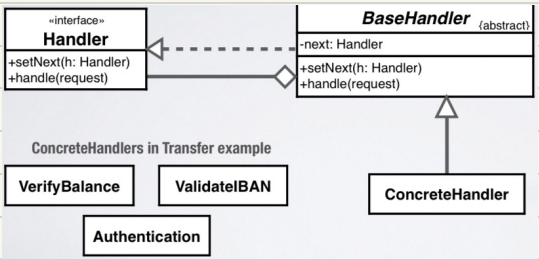
a family of algorithms, encapsulated, thus making them interchangeable; many related classes only differing in behaviour

Advantages:
- Run-time changing of behaviour
- Interfaces instead of sub-classing
- Eliminates conditional statements

Disadvantages:
- Context must be aware of strategy
- Communication overhead between Strategy and Context
- Increased number of objects

→ Chain of Responsibility
Problem: bank transfer authentication, authorization, validation, ...





avoid coupling of sender and request to receiver, more than one object can handle automatically/dynamically determine next handler

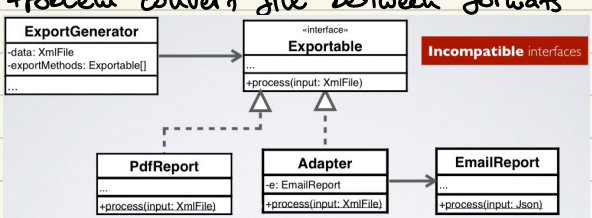
Advantages:
- Reduced coupling
- Receiver/sender don't know each other
- Run-time changing of chain

Disadvantages:
- No explicit receiver, unhandled requests

Structural Patterns - organization of classes/objects to form structures

→ Adapter

Problem: convert file between formats



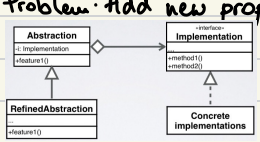
convert from one interface to another (compatible)
use existing incompatible class, create reusable class

Advantages:
- Client unaware of implementation details
- Reduce code duplication

Disadvantages:
- Doesn't work with subclasses
- One additional class

→ Bridge

Problem: Add new property to similar context classes



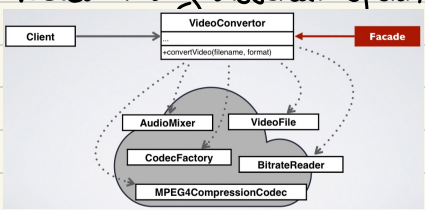
decouple abstraction from implementation, two can vary independently
divide and organize monolithic class with variants

Advantages:
- Open/Close principle: independent abstraction and implementation
- One responsibility per abstraction/implementation

Disadvantages:
- More complex code (if cohesive classes)

→ Facade

Problem: Many different operations, however talk to simple interface



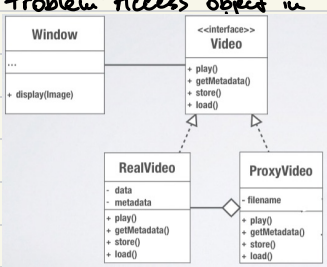
unified interface to a set of interfaces in subsystem
simple interface to complex subsystem

Advantages:
- Isolate clients from subsystems
- Open-closed principle

Disadvantages:
- Facades coupled to all classes of application

→ Proxy

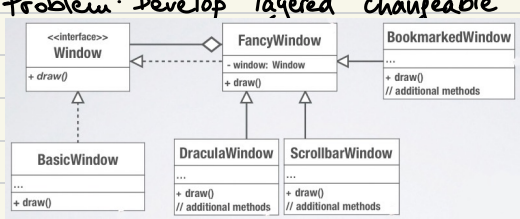
Problem: Access object in memory-saving manner



surrogate/placeholder for another object in order to control access to it, access the original objects
Remote → distributed object communication
Virtual → handling complex or heavy objects
Protected → control access to resource
Caching → cache result requests

→ Decorator

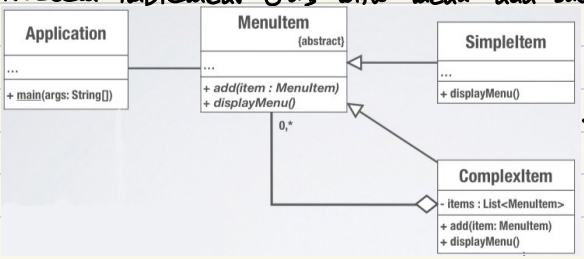
Problem: Develop layered changeable Editor GUI



dynamically attach additional responsibility to objects (instead of subclassing)
transparently add responsibilities that can be withdrawn, when subclassing is impractical

→ Composite

Problem: implement GUI with menu and sub-menus

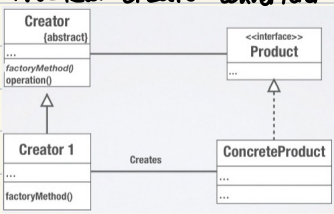


objects into trees structure
→ part-whole hierarchies
treat composition and individual uniformly

Creational Patterns – create objects in controlled way, based on requirements

→ Factory

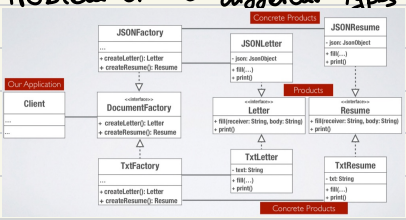
Problem: Create converterer whose formats can be extended



interface for creating objects, subclasses decide

→ Abstract Factory

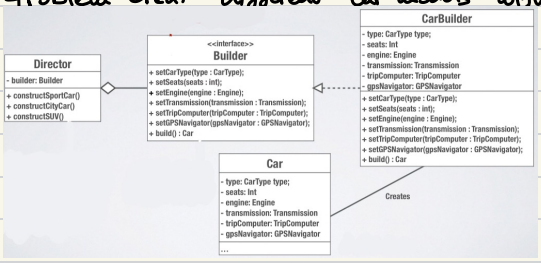
Problem: Create different types of documents in different formats



interface for creating families of related or dependent objects
independent product creation, composition and representation, system configured with one of multiple families

→ Builder

Problem: Create different car models with basic features and optional additional



step-by-step construction of object separated from its representation
independent creation of complex object allowing for different representations

-> Singleton
Problem: Only one instance of class at any time
Mostly used for handling threads, processes, logging

6. Software Analytics
Technical Debt
intrinsic complexity - essential complexity to solve problem
additional complexity / cruft - added complexity for quick implementation
types: Code-related, Testing-related, Organizational

Code Smells - surface indication of design weaknesses
-> Blob code - methods/classes too large
-> Object-oriented abusers - incomplete/incorrect applications of OO principles
-> Change preventers - changes in some part require many changes elsewhere
-> Dispensable - pointless/unnecessary methods/classes
-> Couplers - methods/classes with very high coupling

Software Metrics
measure - indication of extent, amount, dimension, capacity or size of attribute
metric - measure of the degree to which an attribute is present
->Process - analyze software development process (productivity)
->Project - analyze quality of project (variance)
->Product - analyze quality of software product (CK Metrics)

CK Metrics - A Metric Suite for Object Oriented Design
->WMC - Weighted Methods (Complexity) for Class
->CBO - Coupling between objects
->RFC - Response for Class
->LCOM - Lack of Cohesion of Methods
->DIT - Depth of Inheritance Tree
->NOC - Number of Children

Blob Code

Code Smell: Long method

- Metrics of Code Size:
  - -> LOC (Lines of Code)
  - -> KLOC (1000 Lines of Code)
  - -> eLOC (Effective Lines of Code)

  Pros: easy to compute and use

  Cons: Language and programmer dependent
- Metrics of Code Complexity:
  - -> CC (Cyclomatic Complexity)

  Pros: determine number of white-box tests

  Cons: not easy to compute

  Computation:
  - -> Control Flow Graph
  - -> Basic Block (Node) - straight-line code sequence with no branches
  - -> Dependency edge - jump in control flow

  $V(G) = E - N + 2 = D + 1$ (E-edge, N-node, D- decision point)
- Refactoring
  - -> Extract repetitive code in new method

Code Smell: Large class

- Software Metrics:
  - -> # methods
  - -> # attributes
  - -> # instance variables
- Metrics of Cohesion
  - -> LCOM (Lack of Cohesion of Methods)
  - -> CM (Connectivity Metric)

  Pros: Identify different cohesion aspects

  Cons: Not easy to compute

  Computation:

  $P = \{(m_i, m_j) \mid A_i \cap A_j = \emptyset\}$ - methods do not share any attribute

  $Q = \{(m_i, m_j) \mid A_i \cap A_j \neq \emptyset\}$ - methods share at least one attribute

$$LCOM = \begin{cases} |P| - |Q| & \text{if } |P| > |Q| \\ 0 & \text{otherwise} \end{cases}$$

idea: methods of a cohesive class should share common attributes

$$CM = \frac{\#\text{method pairs sharing attributes or calling}}{\#\text{method pairs in class}}$$

↑ CM = ↑ Cohesion
↓ CM = ↓ Cohesion

- Refactoring
  - → Split class into multiple classes

Code Smell: Long Parameter List
- Metric
  - → # parameters in method (>4)
- Refactoring
  - → parameter object

Code Smell: Couplers

Feature envy - method highly coupled with other classes
- Metrics of Coupling
  - → CBO (Coupling Between Objects)
  - → MPC (Message Passing Coupling)

Pros: Evaluate impact of maintenance on other classes

Cons: Sometimes expensive to compute

Computation:

CBO = # dependencies with other classes (parameters, attributes, method calls)

MPC = # messages/method calls between two classes

CAM (Cohesion Among Methods of Class) = A / K.L

A = # distinct parameter types across all methods in class

L = # distinct parameter types

K = # methods

- Refactoring
  - → Move methods coupled to other classes in said classes

Code Smell: Coupled Classes (inappropriate intimacy)
- Software Metrics:
  - → CBO (Coupling Between Objects)
  - → DIT (Depth of Inheritance Tree)
  - → NOC (Number Of Children)

- Refactoring
  → Replace Delegation with Inheritance

Object-Oriented Abusers
switch statements (nested if-else statements) → violate Open-Closed
↳ replace with Polymorphism

7. Mutation Testing
adequacy criteria - measure how thoroughly the program is exercised
White-box testing:
- Statement coverage
- Branch coverage
- Condition coverage
- MC/DC (Modified Condition Coverage)
- Path coverage

Black-Box Testing:
- Functional Testing
- Model-based Testing
- Input space partitioning (Equivalence, Category)
- Specification-based coverage

Mutation Testing - Fault-based Testing
idea: artificial defects (mutants) simulate (realistic) mistakes
technique: create mutants to assess fault detection capability of tests
mutant: introduce simple syntactic change in code

Mutation Operators
→ Arithmetic Operator Replacement (AOR)
(+, -, *, /, %) or drop operator (a + a → a)
→ Relational Operator Replacement (ROR)
(<, >, <=, >=, ==, !=)
→ Conditional Operator Replacement (COR)
(&&, ||, &, |, !, ^)

→ Assignment Operator Replacement (AOR)
(+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=

Mutation Score = $\frac{\#\text{ killed mutants}}{\#\text{ non-equivalent mutants}} = \frac{K}{M-E}$ M = # mutants, E = # equivalent mutants

8. Regression and Performance Testing
existing test cases in previous version of program during regression testing:
→ Obsolete - no longer correct or related to deleted functionality
→ Reusable - execute unchanged parts of the system
→ Re-testable - execute changed parts; updated and re-executed

Strategies:
Strategy I: Retest all
- expensive execution of test suit in large projects
Strategy II: Test Case Selection
- $T' \subseteq T$ to test P'; limited resources
Strategy III: Test Case Prioritization
- execute tests in decreasing likelihood of fault order
→ bugs in previous versions (past-fault coverage)
→ largest coverage (white-box criteria)
→ most critical requirements
→ interaction with unchanged code (impact and dependency analysis)
→ input/output diversity (different data input partitions)

Heuristics:
White-box
→ use (past) code coverage metric, select minimal number of tests
Black-box
→ Input/Output Diversity, Model-based Distance
Impact Analysis
→ Impact level based on changed class