The Midterm will cover the following:

- REVIEW slides 01 – 11 (inclusive)
- Quizzes 1 – 5 (inclusive)
REVIEW 01: Introduction to Software Engineering
Types of Software

- **Program**
  - Single executable
  - Used by very few people (maybe just the developer)

- **Product**
  - Used by many people
  - Polished
  - Used outside development environment

- **System**
  - Group of programs that work together
  - Complex interfaces/integration between parts

- **System Product**
  - Polish of a **product**
  - Multiple parts of a **system**
Need for Software Engineering

- **Problem with Scale**
  - Larger software $\rightarrow$ higher cost of coding, debugging, etc.
  - Larger teams:
    - Division of labor
    - More communication lines
    - Ensuring everyone on same page

- **Problem with Novelty**
  - Almost always new problem
  - Often “wicked problem”
    - Don’t know all issues until start working on it
Software Engineering Definition

- **Software Engineering**
  - **Design**
    - System/module/class/program/data design
    - Requirements
    - Architecture
  - **Development**
    - Coding
    - Testing
    - Debugging
  - **Management**
    - Work breakdown / team organization
    - Cost and resource estimation
    - Scheduling and workflow
    - Quality control
Software Development Metaphors

- **Penmanship (“writing code”)**
  - Too rigid
  - Not like real SE → suggests one person, little change, and originality
    - Real SE: multiple people, heavy change, reuse
  - Implies “throw one away” → wasteful

- **Farming**
  - Implies incremental approach → good
  - Implies lack of direct control
  - Details don’t quite make sense

- **Oysters**
  - Implies incremental approach
  - Leaves many details unspecified

- **Building**
  - Implies planning, preparation, and execution
  - Works with incremental approach
  - Allows variable scale and quality control
  - Includes concepts of:
    - Buy vs. build
    - Under- and over- planning
Software Construction

- **Software Construction** focuses on the following:
  - Detailed design
  - Coding
  - Debugging
  - Integration
  - Developer testing (unit testing and integration testing)
Why Software Construction?

- Large part of software development
- Central activity
- Can improve individual programmer’s productivity
- Source code → most accurate description of software
- Only activity guaranteed to be done
REVIEW 02: Early History of Software Engineering
Pre-SE

- Early on, no “software engineering”
  - Batch processing $\rightarrow$ slow development
  - Constantly changing hardware
  - Hardware-specific assembly language
- Increase in software development (60’s)
  - Higher-level languages $\rightarrow$ portable
  - Interactive, time-sharing machines
  - More powerful computers
The “Software Crisis”

- “Software Crisis” → identified in the mid/late 60’s
  - Projects over time and budget (or never completed at all)
    - Poor estimation of how long things would take
    - Poor monitoring of progress
  - Projects poorly managed (or not at all)
    - Assumption that people and months are interchangeable
    - Adding manpower when behind schedule
  - Software did not meet requirements and/or was of poor quality
    - ...sometimes with fatal results... (e.g., Therac-25)
Software Engineering is Born

- 1968 – first NATO Software Engineering Conference (Germany)
  - Term “software engineering” coined
  - Trying to address “software crisis”
  - Recognized need to some formalized process of software development
    - Also called “software process”

Fred Brooks

**Dr. Frederick P. Brooks, Jr.**
- Project manager of IBM System/360 (hardware)
  - Coined term “computer architecture”
- Manager of OS/360 software project
  - Experience resulted in book “Mythical Man-Month”
- Won Turing Award in 1999
The Problems with Software Development

- Brooks → why software projects fail:
  - Poor estimation
    - Unwarranted optimism
    - Need previous project data
    - Not enough time for testing/debugging
  - Assuming people and months are interchangeable
    - More people → more lines of communication
  - Adding manpower to a late project
    - **Brook’s Law:** Adding manpower to a *late* software project makes it later.
      - More lines of communication
      - More training time
      - Possible poor partitioning of remaining work
  - “Gutless estimating”
    - If project is late:
      - Rush it → bad idea
      - Tell customer it will be late
      - Offer alternative with fewer features
  - Poorly monitored progress
REVIEW 03: Discipline vs. Agility
“Discipline” vs. “Agility”

- **Discipline (Plan-Driven) → “The Elephant”**
  - Implies training, repeatability, and control
    - Ingrains and strengthens through repeated action
  - Provides strength and comfort
  - Creates well-organized memories, history, and experience

- **Agility → “The Monkey”**
  - Implies creativity, innovation, and responsiveness to change
  - Releases and invents
  - Applies memory and history to:
    - Adjust to new environments
    - React and adapt
    - Take advantage of unexpected opportunities
    - Update experience base for the future
“You Need Both to Climb the Tree...”

To be successful, you need both discipline and agility:

- **Discipline without agility** → bureaucracy and stagnation
  - Resistant to change
  - Excessive focus on process/procedure/Starfleet regulations

- **Agility without discipline** → unencumbered enthusiasm with little direction
  - Chaos rules the day
Process vs. Plan

- **Process**
  - Defined, *generic* set of steps for doing a job
  - Template

- **Plan**
  - Set of steps for a *specific* job, plus other things such as effort, costs, and dates
Process Improvement

- **Process Improvement** = program of activities designed to improve the performance and maturity of the organization’s processes, and the results of such a program
  - I.e., Figuring out how to do what you’re doing better
Process Improvement Cycle

1. Define Process
2. Control Process
3. Perform Process
4. Measure Process
5. Improve Process

The cycle starts with Define Process, moves to Control Process, then to Perform Process, next to Measure Process, and finally back to Improve Process.
Process Capability

- **Process capability** = inherent ability of a process to produce planned results
  - I.e., Is it capable of doing what you said it would do?

- Improve process capability $\rightarrow$ process becomes **predictable** and **measurable** $\rightarrow$ improves **quality** and **productivity**
**Process Group**

- **Process group** = a collection of specialists that facilitate the definition, maintenance, and improvement of the process(es) used by an organization
  - I.e., people in your organization who figure out (in general) what to do, make sure it's being done right, and how we can do things better

- SEPG = software engineering processes only
- EPG = engineering processes
Organizational Maturity

- Organizational Maturity
  - As processes are improved → organization “matures”
  - Not just individual projects, but *common application of standard procedures* across organization
    - Standard process → personnel training → measure effectiveness → improve based on measures
Risk Management

- **Risk management** = organized, analytic process to:
  - Identify *risks* (uncertainties that might cause harm or loss)
  - Assess and quantify risks
  - Develop and apply plans to prevent/handle risks
Software System Architecture

- **Software System Architecture** = defines:
  - (1) A collection of software and system components, connectors, and constraints
  - (2) A collection of system stakeholders’ need statements
  - Rationale which demonstrates that (1) will satisfy (2)
Verification and Validation

- **Verification** = confirms work products (e.g., specifications, designs, models) properly reflect the requirements specified for them
  - I.e., building the product right
  - “Working VERY well”

- **Validation** = confirms the fitness or worth of a work product for its operational mission
  - I.e., building the right product
  - “Is this a VALID solution?”
Waterfall Model

- Waterfall model
  - Early paradigm
  - “Plan to throw one away; you will, anyhow”
    - Originally recommended by Brooks, although later he realized it was a bad idea
  - Problems:
    - Puts testing at END of process \(\rightarrow\) only see problems after the whole thing is built
    - Assumes whole thing is built in one shot \(\rightarrow\) doesn’t have to be, and probably SHOULDN’T be
  - Incremental approach is better in general
- Plan-driven approaches can use either waterfall or incremental
Origins

- Plan-driven
  - Motivations:
    - “Software Crisis”
  - Derived from methods in hardware engineering
    - DoD, companies like IBM, Hitachi, and Siemens
  - Fit well with academic → math-driven software verification

- Agile
  - Motivations:
    - “Programming is a craft” → disdain for mechanical, dehumanizing application of plan-driven software development
    - Response to rapidly changing nature of Internet-based economy
  - Outgrowth of rapid prototyping/development experiences
  - 2001 – 17 people met in a ski resort in Snowbird, Utah → developed Agile Manifesto
  - Targets “chaordic” work
    - Chaordic = chaos + order
Plan-Driven: Pros and Cons

- **Plan-driven**
  - **Pros:**
    - **Quantitative** measurement and improvement of process
    - **Predictability**
    - Common training across organization → loss of employees not devastating to project
  - **Cons:**
    - Mechanical, checklist mentality → customer becomes secondary
    - Less innovation
    - Cost of documentation
    - Cost of standardization
    - Certification over actual improvement
  - **Needs:**
    - Management support
    - Organizational infrastructure
    - Supportive environment/culture
The Agile Manifesto

- **Agile Manifesto**
  - Individuals and interactions **over** processes and tools
  - Working software **over** comprehensive documentation
  - Customer collaboration **over** contract negotiation
  - Responding to change **over** following a plan

- Weighing of alternatives → NOT binary choice!
1. Our highest priority is to **satisfy the customer** through early and continuous delivery of valuable software.

2. **Welcome changing requirements, even late in development.** Agile processes harness change for the customer's competitive advantage.

3. **Deliver working software frequently**, from a couple of weeks to a couple of months, with a preference to the shorter timescale.

4. **Business people and developers must work together** daily throughout the project.

5. Build projects around **motivated individuals**. Give them the environment and support they need, and trust them to get the job done.

6. The most efficient and effective method of conveying information to and within a development team is **face-to-face conversation**.
7. **Working software** is the primary measure of progress.

8. **Agile processes promote sustainable development.** The sponsors, developers, and users should be able to maintain a constant pace indefinitely.

9. Continuous attention to **technical excellence and good design** enhances agility.

10. **Simplicity**—the art of maximizing the amount of work not done—is essential.

11. The best architectures, requirements, and designs emerge from **self-organizing teams**.

12. At regular intervals, the **team reflects** on how to become more effective, then tunes and **adjusts its behavior accordingly**.
Other Agile Values

- **Pair programming**
  - Style of programming in which two programmers work side by side at one computer
  - Often used (but not in all Agile methods)

- **Retrospective**
  - Post-iteration review of effectiveness of work performed, methods used, and estimates
  - Sometimes called “reflection”

- **Test-driven development**
  - Module or method tests are incrementally written by developers and customers before and during coding
  - Supports and encourages very short iteration cycles
Agile: Pros and Cons

- **Agile**
  - **Pros:**
    - Works very well with small teams (5-10) and/or small projects
    - Less time on documentation → more time for development
  - **Cons:**
    - Potential issues with:
      - Large team size
      - Very complex software development
      - Safety-critical / mission-critical software
    - Analyzing improvement a challenge → nothing written down
  - **Needs:**
    - Close relationship between customer/users and developers
    - Informed, involved, and authorized on-site customer representative
    - Critical mass of highly motivated, knowledgeable team members
    - Cultural acceptance (e.g., OK with pair programming)
# Cockburn’s Levels of Skill (Modified)

<table>
<thead>
<tr>
<th>Level</th>
<th>Characteristics</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Able to revise a method (break its rules) to fit an <em>unprecedented</em> new situation.</td>
</tr>
<tr>
<td>2</td>
<td>Able to tailor a method to fit a <em>precedented</em> new situation.</td>
</tr>
</tbody>
</table>
| 1A    | With training, able to perform discretionary method steps  
|       | With experience, can become Level 2.  
|       | Need guidance from Level 2 on both agile and plan-driven teams. |
| 1B    | Average-and below, less-experienced, hard-working developers.  
|       | Work well in plan-driven team  
|       | Likely to slow down agile team  
|       | With training, able to perform procedural method steps  
|       | With experience, can master some Level 1A skills. |
| -1    | May have technical skills, but unable or unwilling to collaborate or follow shared methods.  
|       | Should be rapidly identified and reassigned to work other than performing on either agile or plan-driven teams. |
Each approach is most comfortable in its “home ground”

- **Plan-driven**
  - Large, complex systems
  - Often have safety-critical or other high-reliability attributes
  - Fairly stable requirements
  - Predictable environment

- **Agile**
  - Systems and development teams are smaller
  - Customers/users readily available
  - Requirements and environment are volatile
<table>
<thead>
<tr>
<th>Plan-Driven</th>
<th>Agile</th>
</tr>
</thead>
<tbody>
<tr>
<td>Predictive mindset</td>
<td>Adaptive mindset</td>
</tr>
<tr>
<td>Tame change</td>
<td>Embrace change</td>
</tr>
<tr>
<td>Standardize (and improve) processes</td>
<td>Use processes as needed</td>
</tr>
<tr>
<td>Focus on software quality</td>
<td>Focus on customer satisfaction</td>
</tr>
<tr>
<td>Heavier processes</td>
<td>Lighter processes (“barely sufficient” mentality)</td>
</tr>
<tr>
<td>Tends towards waterfall/sequential development</td>
<td>Short cycle times and iterative, evolutionary development</td>
</tr>
<tr>
<td>Contract view of requirements</td>
<td>Actively involve customer (on-site) in requirements throughout development</td>
</tr>
<tr>
<td>Documentation for shared knowledge</td>
<td>Tacit knowledge within the team → no “shelfware”</td>
</tr>
<tr>
<td>Defined processes and procedures</td>
<td>Self-organizing teams</td>
</tr>
<tr>
<td>“Big Design Up Front” (BDUP)</td>
<td>Simple design (“YAGNI” = “You Aren’t Going to Need It”)</td>
</tr>
<tr>
<td>Predictive design → plan ahead</td>
<td>Just-in-time redesign → refactoring</td>
</tr>
<tr>
<td>Easier for less-capable developers to contribute</td>
<td>Usually needs highly-skilled developers</td>
</tr>
</tbody>
</table>
REVIEW 04: The Importance of Prerequisites
Introduction

- **Prerequisites:**
  -Called “upstream work”
  -Include:
    - Requirements
    - Architecture/Design
The Case for Prerequisites

- **Appeal to Logic**
  - Need to know cost
  - Know what you’re building!
  - Risk reduction

- **Appeal to Analogy**
  - Building a house
  - “Food chain” analogy $\Rightarrow$ avoiding “pollutants” in system

- **Appeal to Data**
  - Early a defect is in the “food chain” $\Rightarrow$ more expensive to repair
    - Error in requirements $\Rightarrow$ MOST expensive to repair later
Different Approaches

- Two development approaches:
  - Sequential ("Waterfall model")
  - Iterative
- Usually project is some combination of the two

- **Plan-driven** → traditionally more **sequential**
- **Agile** → traditionally more **iterative**
Do We Need Prereqs with Iterative Projects?

- Projects without prerequisites $\rightarrow$ iterative cost $<$ sequential cost
  - Errors detected closer to time of insertion
  - HOWEVER:
    - Discovering errors at end of iteration $\rightarrow$ still need to redesign, recode, retest $\rightarrow$ unnecessary cost
    - Costs absorbed piecemeal $\rightarrow$ average cost will be about the same
- (Iterative + prerequisites) cost $<$ (Iterative only) cost
Which to Choose...

- **Sequential, up-front**
  - Requirements → fairly stable
  - Design → straightforward and fairly well understood
  - Dev team → familiar with applications area
  - Project → contains little risk
  - Long-term predictability → important
  - Cost of changing requirements/design/code downstream → likely to be high

- **Iterative**
  - Requirements → not well understood or unstable
  - Design → complex and/or challenging
  - Dev team → unfamiliar with applications area
  - Project → contains lot of risk
  - Long-term predictability → NOT important
  - Cost of changing requirements/design/code downstream → likely to be low
REVIEW 05:
Writing Software Requirements
Software Requirements

- Software Requirements
  - Attributes and functionality of the software *from the user’s perspective → what the software does*
  - Also should include *hardware and software constraints*
  - In general, should NOT contain implementation details
Plan-Driven vs. Agile Requirements

- **Plan-driven → Software Requirements Specification (SRS)**
  - **Contract** between developers and customer
  - Specific, **formal** document
  - Defined **before work starts**
  - Best when requirements:
    - Largely determinable in advance
    - Mission-critical
    - Have low change rate
    - Affect many features at once
    - Quality or nonfunctional

- **Agile → often User Stories**
  - **Written by customer** → should not be too technical
  - **Informal** → often on **index cards** or in **spreadsheet**
    - *Stricter format:* “As a (role) I want (something) so that (benefit).”
    - *Can also include:* priority, time estimate, identifier
  - **Product Backlog** = priority sorted features/functions/tech enhancements
  - Stories may be **modified, added, or removed** as development proceeds
  - Best when requirements may **change considerably**
Why Have Official Requirements at All?

• Explicit requirements important because:
  ○ User (not programmer) drives system’s functionality
  ○ Avoids arguments and conflict
  ○ Minimizes changes to system after development begins
Dealing with Changing Requirements

- Review (changed) requirements and ensure they are good enough before continuing
- Make sure everyone understands cost of requirement changes
- Set up change-control procedure
- Use development approaches that accommodate changes
- Unworkable/unresolvable requirements → dump the project
- Keep your eye on the business case for the project
SRS should address:
- Functionality
- External interfaces
- Performance
- Attributes
  - E.g., portability, correctness, maintainability, security, etc.
- Design constraints imposed on an implementation

SRS on design:
- Limits range of valid designs
- Does not specify PARTICULAR design
IEEE Std 830-1998: Characteristics of Good SRS

- An SRS should be:
  - Correct
  - Unambiguous
  - Complete
  - Consistent
  - Ranked for importance and/or stability
    - Clears out hidden assumptions
    - Properly prioritize effort and time
  - Verifiable
  - Modifiable
    - Good structure
    - Coherent cross-referencing
    - Avoid redundancy
    - Don’t intermix requirements
    - Less detail in high-level sections
  - Traceable
    - Backward traceability to previous documents ➔ why we have a requirement
    - Forward traceability ➔ which requirement does this satisfy?
      - Each requirement should have unique name/reference number
IEEE Std 830-1998: SRS Outline

Table of Contents

1. Introduction → background for product
   - Purpose of SRS
   - Scope → identify products, explain functionality, talk about benefits/objectives/goals
   - Definitions, acronyms, and abbreviations
   - References
   - Overview of SRS organization

2. Overall description → background for requirements
   - Product perspective → interfaces, operations, site adaptation
   - Product functions
   - User characteristics
   - Constraints
   - Assumptions and dependencies
   - Apportioning of requirements → requirements that may be delayed until future versions

3. Specific Requirements
   - Specify (input) (operation on input) (output)
   - Externally perceivable
   - Uniquely identified
   - Includes: External Interfaces, Functions, Performance requirements, Logical database requirements, Design constraints, Software system attributes

Appendixes
Index
IEEE Std 830-1998: Organizing Requirements

- **System mode**
  - E.g., In-game, in-game-menu, main-menu, level selection
- **User class**
  - E.g., elevator control system → passengers, maintenance workers, fire fighters
- **Objects**
  - Real-world entities modeled by system
- **Feature**
  - E.g., telephone system → local call, call forwarding, conference call
- **Stimulus**
  - E.g., aircraft landing system → loss of power, wind shear, etc.
- **Response**
  - E.g., all functions associated with generating list of current employees
- **Functional hierarchy**
  - Organized by common inputs, common outputs, or common internal data

- May use multiple levels of organization (system mode, then objects)
- May use diagrams, state charts, etc. where appropriate
REVIEW 06: Overview of Software Engineering Approaches
## Methods Covered Here

<table>
<thead>
<tr>
<th>Agility Rank</th>
<th>Method</th>
<th>Very Low</th>
<th>Low</th>
<th>Med.</th>
<th>High</th>
<th>Very High</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Scrum</td>
<td>●</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Adaptive Software Development (ASD)</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Lean Development (LD)</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Crystal</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>eXtreme Programming (XP)</td>
<td>●</td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>Dynamic Systems Development Method (DSDM)</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>Rational Unified Process (RUP)</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>Team Software Process (TSP)</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>Feature-Driven Development (FDD)</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>Capability Maturity Model Integration (CMMI)</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>11</td>
<td>Capability Maturity Model for Software (SW-CMM)</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td>●</td>
</tr>
<tr>
<td>12</td>
<td>Personal Software Process (PSP)</td>
<td></td>
<td></td>
<td></td>
<td>●</td>
<td></td>
</tr>
<tr>
<td>13</td>
<td>Cleanroom</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>●</td>
</tr>
</tbody>
</table>
Scrum

- Scrum
  - Project management technique
    - **Scrum Master** = Project/team management
    - **Daily Scrum meetings** = 30 min. team meetings to monitor status and communicate problems
    - **Product Backlog** = All project features/functions/tech enhancements
    - **Sprint** = 30-day development cycle
      - Before Sprint → pick features and plan work
      - **Sprint Goal** = minimum success criterion for Sprint
  - Can introduce Agile to Plan-driven environment
  - Can be scaled up
  - Need Cockburn Level 3 coaches/personnel
Scrum Development Cycle

http://www.methodsandtools.com/archive/scrum1.gif
Adaptive Software Development (ASD)

- Acknowledges **uncertainty**
- Repeating cycles of:
  - **Speculate** = explore and experiment
  - **Collaborate** = work together, share knowledge, make decisions
  - **Learn** = project retrospectives, customer focus groups
- Characteristics:
  - **Mission-focused**
  - **Feature based**
  - **Iterative**
  - **Timeboxed**
  - **Risk driven**
  - **Change tolerant**
- **Non-prescriptive method** → difficult to apply to large, critical, relatively stable development efforts
- Need Cockburn Level 3
Lean Development

- Proprietary approach by Bob Charette (ITABHI)
- Business strategy and project management
  - Not really development process
- Focus on software reuse \(\rightarrow\) product lines
- Involves entire enterprise
- Most explicit focus on assessing and achieving business value
  - Deliver value sooner rather than later:
  - Buy rather than build if you can

• **Crystal**
  - Not single method → framework/family of methods
  - Vary approach based on environment and project (“crystal metaphor”)
    - “Color” = # of people involved
      - Clear, Yellow, Orange, Red
    - “Hardness” = criticality in terms of type of loss
      - Comfort (C), Discretionary money (D), Essential money (E), Life (L)
  - Characteristics:
    - People and communications centric
    - “Ultra-light” and “Stretch-to-fit”
      - Use harder version ONLY if significant consequences/risks involved
  - Needs Cockburn Level 2

eXtreme Programming (XP)

- **eXtreme Programming**
  - Most widely recognized Agile method
    - Often used in tandem with Scrum
  - Four values:
    - **Communication** → force in positive fashion
    - **Simplicity** → simplest product that meets customer’s needs
    - **Feedback** → customers and developers
    - **Courage** → make hard decisions
XP: Prerequisites

- **Planning game** = customers and developers negotiate requirements
  - Requirements → “user stories”
  - Design
    - Emphasis on *simple design* (YAGNI)
    - “*Metaphor*” = capture system concept
    - If you need to redesign → *refactoring*
    - *Spike* = very simple program/prototype to explore solutions
      - Also helps team get better estimates
      - *Architectural spike* = helps develop project metaphor

- Plan **multiple iterations**
XP: Development

- **Small** teams with *customer continuously present* on-site

- **Short cycle time** → no more than 3 weeks

- **Project velocity** = how much work you are getting done per iteration

- **Collective ownership** of product
  - Anyone can work on anything
  - **Pair programming** often employed
    - Two programmers sit next to each other and work on same problem/code

- Coding standards established by team

- Emphasis on 40-hour work week
XP: Quality Assurance

- Quality assured through:
  - Test-first approach
    - Customers defines **acceptance tests**
    - Pass test $\Rightarrow$ requirement met
  - Continuous integration
    - Make simpler but complete versions as you go along
    - Build on previous iterations
XP: Development Lifecycle

http://www.extremeprogramming.org/map/project.html
Advantages:
- Works very well with small applications

Disadvantages:
- Scaling an issue → max. 20-person team
- Simple design/YAGNI → inappropriate for stable systems with predictable evolution
- Issues when teams not collocated
Dynamic Systems Development Method (DSDM) Atern

- Framework for developing software rather than particular method
- Phases:
  - Feasibility + Foundations
    - Scope, requirements, schedule
  - Exploration
    - Develop partial solution
  - Engineering
    - Make partial solution robust for use
  - Deployment
    - Solution put into use
    - Either:
      - Scope larger $\rightarrow$ go back to Foundations
      - More requirements to implement $\rightarrow$ go back to Exploration
      - Technical issues $\rightarrow$ go back to Engineering
      - All requirements done $\rightarrow$ go to Post-Project
- Timeboxing and MoSCoW priorities (Must have, Should have, Could have, Want)
- Designed for small teams $\rightarrow$ can be scaled up
- Clear project roles
- Traditional feel but uses risk management (Agile-ish)
RUP

Rational Unified Process (RUP)

- “Serial in the large; Iterative in the small”
  - 4 Phases (some with multiple iterations):
    - Inception
    - Elaboration
    - Construction
    - Transition
  - Activities overlap, but more emphasis in certain phases
  - Clear phase exit criteria and milestones
- Heavier process with Agile attributes
Team Software Process (TSP)

Five objectives of TSP:
1) Self-directed teams
2) Managers that coach/motivate and sustain peak team performance
3) Make SW-CMM Level 5 (Optimizing) normal and expected
4) Provide improvement guidance to high-maturity organizations
5) Facilitate university teaching of industrial-grade team skills

Pros:
- Can be scaled up using “team of teams”
- Very thoroughly defined process

Cons:
- Undisciplined teams
**Feature-Driven Development (FDD)**

- **Five phases:**
  1. Architecture
  2. Establish list of features based on business needs
  3. Create development plan based on list of features
  4. Develop design packages and work packages for features assigned to current iteration
  5. Build features in software
     - Implement, inspect, test at unit level
     - When complete → integrate into current build of overall system
  - Repeat phases (4) and (5) each iteration

- "Object-oriented" mindset
  - Specifically defines process and roles in OO fashion
  - Individual code ownership

- Get architecture right the first time

- **Problems:**
  - Issues with unpredictable, “architecture-breaker” changes
  - Relies on “good people” (Cockburn Levels 2 and 3) to recognize when to backtrack to previous phases
• **CMM** = Capability Maturity Model
  - Organized set of practices to improve *process capability* in one or more discipline areas
    - *Example areas*: software engineering, systems engineering
SW-CMM

- Capability Maturity Model for Software (SW-CMM)
  - Mostly management model
  - Developed as framework for U.S. Government to assess software contractors
    - SEI (Watts Humphrey → “Father of Software Quality”) and MITRE corporation
  - Five maturity levels:
    1) Ad hoc, chaotic
    2) Repeatable
    3) Defined
    4) Managed
      - I.e., measuring processes
    5) Optimizing
  - Key Process Areas (KPAs)
    - Describe minimum activities and common features of a process for a particular area
    - I.e., the stuff you should be doing at a given maturity level
CMMI

**Capability Maturity Model Integration (CMMI)**
- Provide standard for CMMs
  - Includes systems, not just software
  - Replaces SW-CMM
- NOT: development life cycle
- Includes:
  - **Process Areas (PAs)**
    - Sets of activities you should be doing
  - **Generic Practices**
    - General activities in process areas
  - **Capability scale:**
    - 1) Not performed
    - 2) Performed
    - 3) Managed
    - 4) Defined
    - 5) Quantitatively managed process
    - 6) Optimizing
- Can be viewed as:
  - **Staged**
    - Process Areas organized by maturity levels, like SW-CMM
  - **Continuous**
    - Process Areas evaluated and improved separately, similar to ISO
Personal Software Process (PSP)

- Focuses on improving individual programming skills
- Four levels:
  - PSP0 → establish personal baseline
  - PSP1 → adds planning and disciplined approach to testing
  - PSP2.0 → focus on code/design quality
  - PSP3 → start of improvement cycle
- Complementary with TSP:
  - TSP → team processes
  - PSP → individual processes
Cleanroom

- **Cleanroom**
  - Focus on **defect-free** code
    - Program = complex mathematical function → can “certify” it is reliable
  - VERY heavy processes
    - Initial **black box** approach to system
    - **Stringent design/code reviews**
    - Incremental approach with **strict construction rules**
    - **Statistical testing** (probabilistic input/test cases)
  - **Advantages:**
    - Excellent for developing highly reliable software
  - **Disadvantages:**
    - Need highly skilled practitioners
    - Difficult to scale up
    - Problematic in highly dynamic environments
REVIEW 07: Introduction to Software Architecture
What is Architecture?

**Software System Architecture** = defines:
- (1) A collection of software and system components, connectors, and constraints
- (2) A collection of system stakeholders’ need statements
- Rationale which demonstrates that (1) will satisfy (2)

**System stakeholders**
- An individual, team, or organization (or classes thereof) with interests in, or concerns relative to, a system
  - **Concerns** = interest pertaining to system’s development, operation, or any other critical aspect (performance, reliability, security, etc.)
- Includes customers/users, architects, developers, and evaluators
What is Architecture?

- In other words, Software Architecture is...
  - High-level part of software design
  - Frame that holds the more detailed parts of the design
  - Important because:
    - Conceptual integrity
    - Partitioning of work

- “Architecture specification” / “Top-level Design”
  - Document that describes architecture
Architecture vs. High-Level Design

Some make a distinction between:

- Architecture $\rightarrow$ system-wide constraints
- High-level design $\rightarrow$ subsystem constraints
What is a System?

- **System**
  - A collection of components organized to accomplish a specific function or set of functions
  - Can loosely refer to individual applications, product lines/families, enterprises, etc.

- **Every system has an architecture**
  - May not be documented
  - May be good, mediocre, or terrible, but still has one
General Architectural Quality

- **Good architecture:**
  - Discusses:
    - Classes (and info hidden in classes)
    - Rationale/justification for major decisions
  - Attributes:
    - Internally consistent
    - Complete (w.r.t. requirements)
    - Matches problem
    - Machine- and language- independent
    - Balances over- and under-designing
    - Identifies risky areas
  - Has multiple **views**
    - **View** = a representation of whole system from perspective of a related set of concerns
  - Something YOU understand and are comfortable with
Structure, Components, and Connections

- Program Organization
  - Major building blocks
  - How block communicate

- Major Classes
  - Includes:
    - Responsibilities, interactions, hierarchies, state transitions, persistence
    - **80/20 rule**
      - Specify 20% of classes that make up 80% of system’s behavior

- Data Design
  - Files, tables, etc.

- User Interface Design
  - Must specify now (if not at requirements time)
  - Should be modularized (separate from other subsystems)
Constraints

- Business Rules
  - Rules and impact on design
- Resource Management
  - How resources managed
  - Normal and maximum cases
Quality

- Quality Attributes
  - Security
    - Design and code level
    - Build threat model
  - Performance
    - Goals and priorities
    - Budgets and estimates
    - Feasibility and risks
    - Algorithms/data types needs
  - Scalability
    - How system will deal with growth
      - If NOT to be expanded → state explicitly
  - Interoperability
    - Sharing data/resources with other software/hardware
  - Internationalization/Localization
    - Internationalization ("I18n") = support multiple locales
    - Localization ("L10n") = support specific local language
    - Important → buckets of strings in any given interactive program
Error Handling and Robustness

- **Error Handling and Robustness**
  - **Input/Output**
    - Describe level where I/O errors detected
  - **Error Processing**
    - Estimated that 90% of code is for error checking/processing
    - Should have error processing strategy in architecture
  - **Fault Tolerance**
    - Techniques to increase system reliability in the face of errors
      - Detection, recovery, and containment of bad effects
  - **Overengineering**
    - Determine system-wide level of robustness desired
    - Don’t want mix of hyper-robust and barely-adequate classes
Management, Planning, and Business Decisions

- Architectural Feasibility
  - Is it possible to build this at all?
- Buy-vs.-Build Decisions
- Reuse Decisions
- Change Strategy
  - Anticipate possible changes
REVIEW 08: Writing Software Architecture Descriptions
Architectural Description

- **Architectural description (AD) contains:**
  - One or more **views** of the system
  - Other information outside of the views → system overview, system context, concerns, etc.

- **View** = a representation of whole system from perspective of a related set of concerns

- **Viewpoint** = establishes conventions on how the view is created, depicted, and analyzed → notations used, model used, etc.
Views

• Using the 4+1 view model:
  ○ Logical View
    - Structure
    - Dynamic Behavior
  ○ Process View
  ○ Physical View
  ○ Development View
  ○ Scenarios (that overlap all 4 views)

AD Template

- Title Page
- Introduction
- System Purpose
- Structure
- Dynamic Behavior
- (Other Views)
- Conceptual Framework
- Conclusion
1. Introduction

- Should include:
  - Name of the architecture
  - Architecture design team/authors (with contact information)
  - Creation and modification history
  - Audience and purpose of document
  - Selected viewpoints
  - Related documents
2. System Purpose

- **Includes:**
  - Context (with respect to environment)
  - System Interface(s)
  - Non-functional requirements
    - *Qualities, Constraints, Principles*

- **Focus:** how the system fits into its environment
3. Structure

- STATIC structure of the architecture
  - INSIDE the system

- Includes:
  - Overview
    - Diagrams of classes and interactions
    - Commentary
      - Rationale
      - Architectural Constraints
      - Alternatives
  - Components
    - Classes or groups of classes
  - Interfaces
    - Abstraction of some kind of conversation between components
Levels of Design: AD

1. **Software System**

2. **Division into subsystems/packages**

3. **Division into classes within packages**

4. *Division into data and routines within classes*

5. **Internal routine design**
Common Subsystems

- **Business Rules**
  - Laws, regulations, policies, and procedures of system

- **User Interface**
  - Classes for GUI, menu, windows, help systems, etc.

- **Database Access / Data Storage**

- **Graphics**

- **System Dependencies**
  - Wrap OS-specific functionality

- **Application Level Classes**
  - App-specific classes that do not include any of the above
4. Dynamic Behavior

- **Includes:**
  - Scenarios
    - System responses to EXTERNAL stimuli
  - Mechanisms
    - Internal operations
5. Other Views

- Other views are possible:
  - **Process View**
    - Shows processes and threads from static or dynamic point of view
      - E.g., what classes run in separate threads, etc.
  - **Development View**
    - Describes how code, files, libraries, packages, etc. all fit together
  - **Physical View**
    - Shows deployment of system and how it connects in a broad sense with external hardware, servers, etc.

- Each of these views would be granted a separate section.
6. Conceptual Framework

- Refers to a network of concepts and the relationships between them
- Includes:
  - Domain Lexicon
    - Specific text definitions of certain concepts
  - Lexicon diagrams
    - Show relationships between concepts
    - Can use UML Class Diagram
7. Conclusion

- Includes an assessment section:
  - Advantages and limitations of chosen solution
  - How well non-functional requirements met
  - Any known inconsistencies
  - Optionally discuss future directions
  - Optionally documents any consistency analysis done across views
Introduction

- **UML**
  - Unified Modeling Language
  - Very well recognized specification for modeling architectures, use cases, etc.
Two basic kinds of UML Diagrams:

- **Behavior diagrams** → dynamic behavior
  - Use case
  - Activity
  - State machine
  - *(Unofficial)* Information flow
  - *Interaction* → emphasize object interactions
    - Sequence

- **Structure diagrams** → static structure
  - Class
  - Object
Use case diagram
- Describe...
  - a set of actions (**use cases**)...
  - that some system or systems (**subject**) should or can perform...
  - in collaboration with one or more external users of the system (**actors**).
**Use cases**
- Action or sequence of actions

**Actors**
- Person, organization, or external system that plays a role in one or more interactions with your system

**Associations**
- Actor – Use Case
  - Invocation (NOT data flow)
- Actor – Actor
  - One actor is a specific kind of actor
UML Use Case Diagram: Notation

- **Extending Use Cases**
  - Adds optional behavior

- **Including Use Cases**
  - Means a given use case is part of another use case
UML Use Case Diagram Example

http://www.agilemodeling.com/artifacts/useCaseDiagram.htm
Activity Diagram

- High-level business processes or complex logic
  - Does NOT identify who does what

UML Activity Diagram: Notation

- **Initial node**

- **Activity final node**

- **Activity**
  - Single step of activity

- **Flow/edge**
  - Shows flow of control from one action to the next

- **Flow final**
  - Flow stops ONLY on this path (does not affect other flows)

http://www.sparxsystems.com/resources/uml2_tutorial/uml2_activitydiagram.html
UML Activity Diagram: Notation

- **Condition**
  - Must be true to traverse node
  - Text on a flow/edge

- **Decision**
  - Diamond with one flow in and several out

- **Merge**
  - Merges ALTERNATE flows (NOT parallel flows)
  - Diamond with several flows entering and one leaving

- Can actually combine decision and merge into one diamond
UML Activity Diagram: Notation

- **Fork**
  - Beginning of PARALLEL flows

- **Join**
  - All flows MUST COMPLETE before processing may continue out of join

- Can combine fork and join
You can also have flows that trigger on or send events:

- **Send signal action**
  - Trigger some event

- **Accept signal action**
  - Asynchronous event occurred; flow starts there

- **Wait time action**
  - Event occurs at certain time or on certain schedule
UML Activity Diagram: Example

Process Order

Requested Order → Receive Order

[order accepted] → Fill Order → Send Invoice

[order rejected] → Ship Order

Accept Payment

Close Order
• **State machine diagram**
  ○ Describes the states of object/interaction and transitions
    ▪ Generally only for very complex classes
**UML State Machine Diagram: Notation**

- **Simple state**
  - No substates or regions
  - Optionally has *internal activities*
    - Actions performed while in this state
    - *Special keywords:*
      - **entry** → behavior performed upon entry to the state
      - **do** → ongoing behavior; performed as long as the element is in the state
      - **exit** → behavior performed upon exit from the state
    - Can also add your own conditions:
      - *condition / operation*
**Composite state**
- Has nested states

- Can also use shorthand notation (and define sub-states elsewhere)
You can also have initial nodes, end nodes, conditions, decisions, forks, and joins like in activity diagrams.
UML State Machine Diagram: Water Example
State vs. Activity Diagram

- **State diagram**
  - Nodes are *states*
  - Transitions are events

- **Activity diagram**
  - Nodes are *activities*
  - Transitions are when activities complete (and conditions are met)
Example: Cheese Matcher In-Game State Diagram

- **Cleanup Game**: go to In-Menu
- **Ask if Restart**
  - Yes
  - **Initialize Game**: game ready
    - new game started
- **Won Game**: [game won]
- **Game Lost**: [game lost]
- **Check Game State**: [no more matching cheeses]
- **Board Update**: [matching cheeses]
- **Waiting for User Action**: cheeses swapped
- **[game still going]**
Example: Cheese Matcher Cheese Swap Activity Diagram

in "Waiting for User Action" state

[selected cheeses NOT 4-neighbors]

[selected cheeses are 4-neighbors]

Swap Cheese

Cheeses Swapped

[another cheese selected]

Select Current Cheese

[another cheese not selected]

[current cheese not selected]

Deselect Current Cheese

[player taps cheese]

[current cheese already selected]
(Unofficial) UML Information Flow

- **Information Flow diagram**
  - Shows exchange of information between actors and system entities at very high level
  - Not an official part of UML
• **Sequence diagram**
  - Shows *message* exchanges between **lifelines**
    - **Lifeline** = individual participant in the interaction
      - Represents a SINGLE instance
      - Primarily objects and actors
    - **Message** = one specific kind of communication
      - Specifies sender and receiver
      - Either start of execution or sending/receiving a signal
**UML Sequence Diagram: Message Notation**

- **Synchronous call**
  - Typically operation/function call
  - Wait until response
  - Filled arrow head

- **Asynchronous call/signal**
  - Send message $\rightarrow$ proceed immediately without waiting for return value
  - Open arrow head
UML Sequence Diagram: Message Notation

- **Create**
  - Creates a new lifeline (often class instance)

- **Delete**
  - Destroys a lifeline
  - Usually an X on the bottom to indicate end

- **Reply**
  - Reply to an operation call
  - Dashed line with arrow
UML Sequence Diagram: Notation

- **Execution**
  - Period in participant's lifetime when it is:
    - Executing behavior/action
    - Sending signal
    - Waiting for reply
  - Two execution occurrences → start and finish
  - Thin grey or white rectangle
**Interaction fragment**
- Most general interaction unit
  - E.g., Execution

**Combined fragment**
- Interaction fragment that defines a combination (expression) of interaction fragments
- Many different kinds:
  - `alt` - alternatives
  - `opt` - option
  - `loop` – iteration
  - `break` - break
  - `par` - parallel
  - `ref` - reference
UML Sequence Diagram: Combined Fragments

- **alt** – alternatives
  - At most one of these will be executed, subject to guard conditions

- **opt** – option
  - Either guard condition true and it executes, or nothing happens

- **loop** – iteration
  - Loops through execution
    - No guards/limits → loop forever
    - Can set max iterations
    - Can set boolean guard
**UML Sequence Diagram: Combined Fragments**

- **break**
  - If guard is true, break out of loop

- **par**
  - Execute in any order, possibly in parallel
  - *Shorthand notation when order on ONE lifeline is insignificant:*
ref → interaction use

- Allows you to refer to another interaction
- Combined fragment with operator ref
UML Sequence Diagram: Facebook Auth. Example
UML Class Diagram

- **Class diagram**
  - **Static** model → classes, types, contents, relationships
    - Inheritance, aggregation, and association
    - Operations and attributes
UML Class Diagram: Notation

- **Class**
  - Represented by box
  - Name, attributes, and methods

- **Association**
  - Regular arrow
  - Can have numbers on either end:
    - $1 \rightarrow 1$ student
    - ...is enrolled in...
    - $1..* \rightarrow 1$ or more courses
**UML Class Diagram: Notation**

- **Abstract class**
  -Italicize the name

- **Interface**
  -Used the phrase <<interface>> > above name
UML Class Diagram: Notation

- **Inheritance**

![Inheritance Diagram]

- **Implements**

![Implements Diagram]
- *Uses*

```
«interface»
Search

«use»
Librarian
```
UML Class Diagram: Notation

- **Aggregation**
  - “Weak aggregation”
    - A has a B, but:
      - B could also be in other aggregations
      - B still may exist if A deleted

- **Composition**
  - “Strong aggregation”
    - whole/part relationship
    - binary association
    - part could be included in at most one composite (whole) at a time
    - if a composite (whole) is deleted, all of its composite parts are “normally” deleted with it.
UML Class Diagram: Domain Model Example
UML Object Diagram

- **Object diagram**
  - Depicts objects and their relationships at a point in time
Specific instance of class

- **Name:** Classname

- **Marvin:** Student
Class vs. Object Diagram

- **Class diagram**
  - Shows **class type** hierarchies, contents, and relationships

- **Object diagram**
  - Shows **instances of classes** and their relationships
REVIEW 10: Introduction to Design
Challenges of Design

- Design is:
  - A wicked problem
  - A sloppy process (even if it produces a tidy result)
  - About tradeoffs and priorities
  - Involves restrictions
  - Nondeterministic
  - A heuristic process

- In other words, design is “emergent”
  - Don’t spring fully formed from designer’s brain
  - Designs evolve and improve
Challenges: Wicked Problem

- Wicked problem
  - Can only be clearly defined by solving all or part of it
  - Implies:
    - Define problem → “solve” it → define it better → “solve” better → etc.
Software’s Primary Technical Imperative

- Software’s Primary Technical Imperative → managing complexity!
How to Attack Complexity

Two-prong approach:

1) Minimize amount of essential complexity that anyone’s brain has to deal with at any one time
2) Keep accidental complexity from needlessly proliferating

ALL OTHER TECHNICAL GOALS SECONDARY to MANAGING COMPLEXITY.
Desirable Characteristics of a Design

- **Minimal complexity**
  - Avoid “clever” designs → make “simple” and “easy-to-understand” designs
  - Can safely ignore other parts while working on specific part

- **Ease of maintenance**
  - Make design **self-explanatory**

- **Loose coupling**
  - As few interconnections as possible
  - Minimizes work during integration, testing, and maintenance

- **Extensibility**
  - Can enhance a system without causing violence to underlying structure

- **Reusability**
  - Design for reuse of parts in other systems (and same system)
Desirable Characteristics of a Design

- **High fan-in**
  - Refers to having a high number of classes that use a given class \( \rightarrow \) e.g., good utility classes

- **Low-to-medium fan-out**
  - Having a given class use a low-to-medium number of other classes

- **Portability**
  - Can move easily to another environment

- **Leanness**
  - Designing system so that it has no extra parts

- **Stratification**
  - Can view system at same level and get consistent view

- **Standard techniques**
  - Use Design Patterns where reasonable and useful
Levels of Design

1. Software System
2. Division into subsystems/packages
3. Division into classes within packages
4. Division into data and routines within classes
5. Internal routine design
Subsystems and Communication

- Communication between subsystems
  - Need-to-know basis
    - Can always relax later on
  - Avoid cycles
Common Subsystems

- **Business Rules**
  - Laws, regulations, policies, and procedures of system

- **User Interface**
  - Classes for GUI, menu, windows, help systems, etc.

- **Database Access / Data Storage**
  - Change storage method → doesn’t break everything

- **Graphics**
  - Wrap graphics-specific code

- **System Dependencies**
  - Wrap OS-specific functionality

- **Application Level Classes**
  - App-specific classes that do not include any of the above
### Design Practices

- **Iterate**
- **Divide and Conquer**
  - Break the problem/project into manageable, self-contained chunks
  - Design each chunk
- **Top-down vs. Bottom-up**
  - **Top-down**
    - Start at highest-level → work down
    - Easier, defers construction details
    - *Problem*: low-level complexity can rise to top
  - **Bottom-up**
    - Start with what system needs to do (low-level) → generalize/work up
    - More tangible, encourages reuse, design more compact
    - *Problems*: more difficult, pieces may not work as system
  - Both
    - Complimentary
- **Experimental Prototyping**
  - Writing the **ABSOLUTE MINIMUM** amount of **throwaway** code needed to answer a *very specific question*!
REVIEW 11: Design Heuristics
Major Design Heuristics

- **Find real-world objects**
  - Data and functions

- **Form consistent abstractions**
  - **Abstraction** = “You can look at an object at a high level of detail.”
  - Base classes, class interfaces, function interfaces

- **Encapsulate implementation details**
  - **Encapsulation** = “FURTHERMORE, you aren’t allowed to look at an object any other level of detail”
  - Hiding lower levels
Major Design Heuristics

- Inherit when simplifies design
  - However, avoid:
    - Base classes with only one subclass
    - Deep inheritance trees

- Hide secrets (information hiding)
  - “Black box” design
  - Things to hide:
    - Inner workings (as much as possible)
    - Data formats
    - Areas likely to change
    - Area with errors that needs to be “walled off”

- Identify areas likely to change
  - Design interface so change will not affect interface
Major Design Heuristics

- Keep coupling loose
  - **coupling** = how tightly class/routine is related to other classes/routines
    - **Size** = # of connections
    - **Visibility** = how obvious connection is
      - *Simple-data-parameter* → normal
      - *Simple-object* (one object instantiates another) → fine
      - *Object-parameter* → tighter
      - *Semantic* → most insidious; uses knowledge of classes/routines inner workings
    - **Flexibility** = how easily you can change connections
Major Design Heuristics

• Look for common design patterns
  ○ **Design patterns** = time-tested patterns that can be applied to specific situations
    - *Advantages:*
      - Easier to discuss and communicate design
      - Reduce errors → standardized way to solve certain problems
      - Gives you suggestions for design alternatives
    - *Warnings:*
      - Don’t force-fit a pattern
      - Make sure pattern fits problem
Barriers of Information Hiding

- Things that break Information Hiding
  - Excessive distribution of information
    - E.g., hardcoded 100 instead of MAX_EMPLOYEES
  - Circular dependencies
  - Global data (and to a lesser extent class data)
Popular Design Patterns

- Design patterns fall into one of three categories:
  - Creational
    - Way to create objects while hiding creation logic
  - Structural
    - Class and object composition
  - Behavior
    - Communication between objects
Popular Design Patterns

- **Creational**
  - Factory Method
    - Makes classes derived from a base class
  - Abstract Factory
    - Effectively a factory of factories
  - Singleton
    - Only one instance; accessed without explicit instantiation
Popular Design Patterns

- **Structural**
  - **Adapter**
    - Converts class interface → different class interface
  - **Bridge**
    - Interface and implementation → can vary each without varying the other
  - **Composite**
    - Group of objects treated like single object
    - Creates tree structure
  - **Decorator**
    - Add new functionality to an existing class without altering its structure
      - Wrapper around existing classes
      - Keeps class method signatures intact
  - **Facade**
    - Provides consistent interface to code that wouldn’t otherwise offer a consistent interface
      - Single class with simplified methods that delegates calls to existing classes
Popular Design Patterns

- **Behavior**
  - **Iterator**
    - Access elements sequentially without knowing underlying representation
  - **Observer**
    - Keeps multiple objects in sync with one another
    - Object (observer) notifies set of objects of changes
  - **Strategy**
    - Defines a set of algorithms/behaviors that are dynamically interchangeable with each other
  - **Template**
    - Defines structure of an algorithm but leaves some of the detailed implementation to subclasses
Other Useful Design Heuristics

- **Aim for strong cohesion**
  - Classes/routines should have a well-defined central purpose

- **Build hierarchies**
  - Class hierarchies, routine hierarchies, etc.

- **Formalize class contracts**
  - Pre-conditions and post-conditions
  - Asserts

- **Assign responsibilities**
  - What is each object responsible for?
  - Similar but broader question than, “What does each object hide?”
Other Useful Design Heuristics

- **Design for test**
  - In particular, design interfaces thinking about how you would test classes/subsystems/routines

- **Avoid failure**
  - Think about how things might fail (and how to prevent it)

- **Choose binding time consciously**
  - Binding time = time a specific value is bound to a variable
  - Earlier → simpler, but less flexible
Other Useful Design Heuristics

- **Make central points of control**
  - “The Principle of One Right Place”
  - Even a named constant (e.g., MAX_EMPLOYEES) centralizes control

- **Consider using brute force**
  - A brute force solution that works better than elegant solution that doesn’t

- **Draw a diagram**
  - Doesn’t have to formal → helps you focus your ideas and thoughts

- **Keep your design modular**
  - Try to design things with the “black box” idea in mind