CSCI3100 Software Engineering

Complete Line-by-Line Annotated Course Notes

Compiled by OpenAI's GPT-4

May 7, 2025

Contents

1 Chapter 1: Introduction (ch1.pdf)

Hello World Example and Its Implications

Code Example: Using matplotlib to say Hello World

```
# Import the matplotlib library (this is a bad comment)
import matplotlib.pyplot as p
# "Spagetti" code below...
p.rcParams["font.size"] = 20
s = p.figure().add_subplot(xticks=[], yticks=[])
t = s.text(.1, .5, "CSCI3100", color="red")
t = s.annotate(": ", xycoords=t, xy=(1, 0), verticalalignment="bottom", color="gold", wei
t = s.annotate(" hello", xycoords=t, xy=(1, 0), verticalalignment="bottom",color="green",
t = s.annotate(" world!", xycoords=t, xy=(1, 0), verticalalignment="bottom",color="blue",
p.show()
```

Explanation:

- `import matplotlib.pyplot as p`: Imports a plotting library as 'p'. The comment says this is a bad comment: it doesn't add value. This hints at the importance of meaningful documentation.
- The code sets font size, creates a figure, adds a subplot without ticks, and then adds a series of text and annotations to the plot, each in a different color and style.
- `p.show()`: Displays the figure.
- **Lesson:** This is a needlessly complex way of doing a simple "Hello World"; it humorously warns students against overcomplicating code and stresses the value of simplicity and clarity in software engineering.

Course Content and Administration

- Lists staff members, including the lecturer (Dr. Tak-Kei Lam), his office, and email, as well as the tutors and their contact info.
- Explains that lectures and tutorials are in English and will mix theory and practical content.
- **Goals are divided into three parts:**
  - **Let's Learn:** Understand how software teams work, processes, methodologies, and compare their pros and cons; understand tools.
  - **Be a better programmer:** Apply SE practices, propose methodologies, and go beyond just programming to full software engineering.
  - **Be a better guy/gal:** Understand that SE is both technical and human; consider ethics, user focus, and humility in teamwork.
- Meeting schedules are detailed, showing the structure and frequency of lectures and tutorials.

- **Course Materials:** Homepage provided for notes and assignments; textbooks are recommended but not required.
- **Programming Languages:** Python is the main language (for ease), with Go and Rust discussed for comparison.
- **Operating System:** Linux is recommended for development, though Windows and Mac are possible.
- **Text Editor:** Any editor is allowed but IDEs are recommended for productivity.
- **Late Policy:** Each student has three free late days for assignments, with a 20% deduction per day after that.
- **Assessment:** 30% assignments, 30% group project, 40% final (open book/notes).
- **Course Content:** Covers the entire SE process, specification, modeling, OO design, testing, toolchains, and best practices.
- **Acknowledgments:** Credits previous lecturers and sources of material.

2 Chapter 2: Software Development (`ch2.pdf`)

Why Learn About Software Engineering?

- "Software is eating the world": This phrase means software is now a core part of almost all industries and daily life.
- "AI saving the world": SE enables AI solutions for global issues (climate, medicine, space).
- "Help you get a job (unless AI replaces programmers)": A tongue-in-cheek reminder of automation's impact on jobs.

What is Software?

- Software is described as a set of objects: programs, documents, data, user guides.
- Software is engineered (not natural), does not "die" but degrades, and is inherently complex.

Types of Software and Their Roles

- **Product:** Delivers computing power, manages or displays data.
- **Vehicle:** Provides system functionality, controls other programs (like OS), facilitates communications, or helps build more software (like IDEs).
- **Categories:** System, engineering/scientific, web, AI, ubiquitous computing, net sourcing (web as a computing engine), open source, and more.

Difficulties in Writing Software

- Software is intangible—unlike civil engineering, we can't see cracks.
- Hard to know if it's correct, especially at scale; if both code and tests are wrong, errors may go unnoticed.
- Human brain bandwidth is limited ("10 bits/sec" is a humorous exaggeration).
- Software development is a team sport—requires communication; misaligned specs and requirements can cause major failures (example: NASA Mars probe lost due to unit mismatch).

Maintaining Software

- Software is not "write once"; it must evolve with new tech, requirements, interoperability, or bug fixes.
- Bugs can remain hidden for years (Linux bug example).
- If code isn't designed for high cohesion/low coupling, replacements are costly and error-prone.
- Project failures are common (Standish Group: 19% failure rate).

Why Projects Fail?

- Example: Australian Stock Exchange project failed after huge delays and poor code quality (50% rewrite).
- Root causes: poor testing, bad quality management, lack of communication.

Software Development Process

- A systematic set of activities: requirements, design, implementation, testing/debugging, maintenance, and process management.
- No one-size-fits-all model; process must be adapted to context.
- Models are templates; engineering judgment is expected.

Example: Mobile Suit Development Process (Gundam)

- Steps: requirements, high-level design, approvals, detailed design, build, validation, production, maintenance, feedback.
- Parallels to software: staged, iterative, feedback-driven.

Five Main Elements in Software Development

- Requirements capture: most difficult.
- Design: how to meet requirements.
- Implementation: actual coding and documentation.
- Verification: testing and debugging.
- Operations and maintenance: post-release fixes, enhancements, refactoring.

Ordering and Structuring

- Waterfall: sequential, best for well-understood requirements; rarely strictly followed.
- All-connected: more iterative, recognizes feedback loops.

Process Models

- Waterfall: classic, linear.
- Incremental: divides work into manageable increments, can overlap.
- RAD: rapid, parallel, resource-intensive.
- Prototyping: evolutionary, fast feedback, risk of poor implementation choices.
- Spiral: risk-driven, evolutionary.
- Unified Process: iterative, incremental, flexible.

3 Chapter 3: Software Development (Unified Process, Agile) (`ch3.pdf`)

Unified Process (UP)

- **Architecture-centric:** Build executable prototypes for early validation and reuse.
- **Controlled iterative:** Address critical risks early, deliver in increments, gather feedback.
- **Use-case driven:** Requirements and design are driven by how users will interact with the system.
- **Tailorable:** Process can be adapted to fit project needs.

Inside a UP Cycle

- Inception: Feasibility, scope.
- Elaboration: Define architecture, plan construction, identify and prioritize risks.
- Construction: Build increments, prepare for deployment.

- Transition: Integrate, train users, make adjustments.
- Each phase may have multiple iterations; several cycles per project.

Core Practices

- Develop iteratively.
- Manage requirements systematically.
- Use proven architecture.
- Model requirements and user interactions.
- Continuously verify quality.
- Foster collaborative development.

Agile Manifesto and Principles

- **Values:** Individuals/interactions ¿ processes/tools, working software ¿ documentation, customer collaboration ¿ contract negotiation, responding to change ¿ following a plan.
- **Principles:** Customer satisfaction by rapid delivery, welcome change, frequent delivery, working software as progress, sustainable pace, collaboration, face-to-face communication, motivated teams, technical excellence, simplicity, self-organizing teams, regular reflection.

Agile Practices (XP)

- Planning game, small releases, system metaphor, simple design, testing, refactoring, pair programming, collective ownership, CI, 40-hour week, on-site customer, coding standards.

4 Chapter 4: Requirements (`ch4.pdf`)

Why is "requirements capture" so important?

- **Most problems come from inadequate requirements analysis:**
  - Stakeholders have different technical backgrounds, which can cause misunderstandings.
  - Improper assumptions may be made about the system or users.
  - Unknown characteristics of the inputs (e.g., data types, ranges) can lead to errors.
  - Engineering concerns (how to build) often conflict with economic concerns (cost, time).
- **Historical lesson:** Spending time early to clarify problems and solutions is cheaper than fixing mistakes later.

What is a requirement?

- Is it a process or a product? **Answer:** Both.
- **As a process:** All stakeholders define what the problems are and what the solution needs to include.
  - Output: a document describing high-level concerns, what the software will and will not do.
  - No concrete design at this stage.
- **As a product:** The documentation resulting from the process, e.g., informal memo, formal Software Requirement Specification (SRS).
  - IEEE templates (1984, 2011), ReadySET templates are mentioned as standards.

Requirements vs. Specification

- **Who is the audience for SRS documents?** Both users and developers.
- **Requirements:** What users want, written in user language.
- **Specification:** How the software will meet those requirements, written for developers to answer:
  - Which modules must be constructed?
  - Should we use object-oriented design?
  - What algorithms should be used?

Example: Requirements and Specification

- **Requirements (User-focused):**
  - "An easy to use source control system that can store multiple versions of code, can rollback, able to split branches."

- **Specification (Developer-focused):**
  - Must have fast algorithms for computing differences between versions.
  - Write operations must be atomic and journaled.
  - Must be distributed.
  - Must be secure.

Functional and Non-functional Requirements

- **Functional:** What functions the software must achieve.
- **Non-functional:** Constraints or system properties (e.g., security, usability, performance, memory limits, process requirements like using Jira for bug tracking).
- **Classification:**
  - Product (protocols, encodings, encryption).
  - Organizational (coding style, processes).
  - External (constraints outside your control, e.g., "must use FAX").

Systematic Requirements Collection

- Always consider all types: functional, product, organizational, and external non-functional.

WRSPM Reference Model

- **Purpose:** Map requirements (problem domain) to specifications (solution domain).
- **Components:**
  - **Interface:** e.g., GUI, TUI.
  - **Environment:**
    * W - world assumptions
    * R - requirements
    * S - specification
  - **System:**
    * P - program (software)
    * M - machine (hardware)
  - **Four phenomena:**
    * e_h: hidden elements (e.g., user skills)
    * e_v: elements visible to the system (e.g., user input)
    * s_v: shared system elements, visible to users (e.g., buttons)
    * s_h: internal representations (e.g., data structures)
- **Goals:**
  - $W + S \Rightarrow R$
  - $P + M \Rightarrow S$
  - $W + P + M \Rightarrow R$

Requirements Elicitation Techniques

- Interviews, scenarios, prototypes, observation.

Requirements Validation

- **Validity:** Do requirements reflect real needs?
- **Consistency:** Are they consistent among stakeholders and scenarios?
- **Completeness:** Do they solve the original problem?
- **Realism:** Are they achievable within budgets?
- **Verifiability:** How easily can we test that they are met?

5 Chapter 5: Source control system (`ch5.pdf`)

Why Source Control System is Crucial?

- **Many versions of code exist:**
  - Changing requirements, bug fixes, new environments.
  - New versions accumulate even after shipping.
- **Why use a tool?**
  - Maintain multiple versions (traceability, rollback).
  - Support parallel development (different projects, same project by many devs).
  - Resource estimation, release tracking.

General Terminology

- **Repository (repo):**
  - Database (can be local or remote) that stores file content and history (metadata).
- **Commit:**
  - A snapshot of files at a point in time; a repo is a series of commits.
- **Working copy:**
  - A checked-out copy of the repo at a particular commit.

Source Control Evolution: RCS, CVS, Git

Take 1: RCS (first released 1982)

- **Requirements:**

- Keep copies of all edits, change logs, and diffs between versions.

- **Specification:**
  - Keep changes and logs, support rollback/diff/merge, resolve conflicts, possibly use client-server.
  - Handle text and binary files.
- **Design:**
  - Each file has its own repository; no project-level repo.
  - To release multi-file software, you must manually check out each file version and combine them.
  - Uses "lock-modify-unlock": only one editor per file at a time.
  - Teamwork is hard (no parallel edits).

Take 2: CVS (first released 1990)

- **Requirements:**
  - All from Take 1, plus allow multiple devs to work on same files.
- **Specification:**
  - Each commit can contain multiple files.
  - Support for multiple users, networked access (SSH, email), login/security.
- **Design:**
  - Server-client based, centralized repository.
  - "Copy-modify-merge-before-commit": No locks, but you must merge with others' changes before committing.
  - Commits are not atomic: if two users commit at once, data corruption can result.
  - File versions are still tracked individually; a commit may contain fileA v1.4 and fileB v1.3.
  - Branching and renaming are difficult and error-prone.

Take 3: Git (first released 2005)

- **Requirements:**
  - All from Take 2, plus: assign same version number to all files in a snapshot.
- **Specification:**
  - Distributed repository: every user has a full copy.
  - Commits are atomic (all-or-nothing).
  - Branching is fast and easy.
  - Better support for renaming, compression, and diffs.
  - Same version applies to all files in a commit.

- **Design:**
  - "Copy-modify-commit-before-merge": You can commit locally without merging others' changes, then merge later.
  - The concept of "staging" allows preparing and splitting commits as needed.
  - Central repo is still used for integration, CI/CD, and backup.
  - More complex concepts (local vs remote repo, staging area, etc.).

Branching Strategies

- **Trunk-only:**
  - Only one main branch. Simple, but causes merge conflicts and makes feature exclusion difficult.
- **Feature branching:**
  - One branch per feature or hotfix.
  - Frequent merges from main to feature branches.
  - "Pull requests" are used for code reviews before merging.
  - Short-lived branches are best; long-lived branches lead to hard-to-resolve conflicts.
  - Naming conventions (e.g., feature/<ticket_id>/description) are recommended.
- **More complex branching:**
  - Multiple long-lived branches (e.g., main, dev) are possible, but require robust CI/CD.

Do's and Don'ts

- **Do:**
  - Delete branches after merging.
  - Communicate regularly.
  - Keep branch scope limited.
  - Use clear, consistent naming.
- **Don't:**
  - Keep branches forever.
  - Work in isolation (causes merge conflicts).
  - Push code that cannot be compiled.

Git Commands and Concepts

- Common commands: clone, pull, fetch, merge, rebase, stash, subtree, submodule, remote, add, rm, commit, push, branch, checkout, diff, difftool.
- **merge vs rebase:**
  - **merge:** Combines branches, preserving all history and showing where they diverged.
  - **rebase:** Moves a branch to begin on the tip of another branch, making history linear.

Other Tools and Practices

- Communication and tooling (IDE, CI/CD, automated tests) are crucial.
- **Feature flags:** Boolean variables in code for toggling features at runtime, helpful for testing and release management.

6 Chapter 6: Software architecture (`ch6.pdf`)

What is software architecture?

- **Definition (Carnegie-Mellon SEI):**

  > "The architecture of a software-intensive system is the structure or structures of the system, which comprise software elements, the externally-visible properties of those elements, and the relationships among them."

- **Purpose:** Partition large systems into smaller subsystems/modules to achieve:
  - Reusability
  - Testability
  - Manageability
  - Integratability
  - Scalability
  - Security
  - Usability
  - Performance
  - Code elegance
- **References:** IEEE 42010, software architecture documentation frameworks.
- **Key lesson:** Bad architecture can't be fixed by good coding later (e.g. a cryptocurrency network with centralized nodes undermines the whole purpose).

Decomposition

- `decompose(software system) -> components`:
  - Each component should be self-contained, independently constructible, and have its own business value.
  - Enables better organization, parallel development, third-party integration, and subsystem cooperation.
- Decomposition is recursive: you can decompose classes into methods, packages into classes, subsystems into packages, and systems into subsystems.

What must a component have?

- A clearly defined set of responsibilities (e.g., UI logic separate from business logic).
- A clearly defined boundary (no shared internal data or responsibilities with other components).
- A set of well-defined interfaces (APIs) for interconnection.

Architectural structuring

- Define a **static hierarchy** of interacting components.
- Also define the **dynamic behavior** (runtime interaction) between components.

High-level system structures

- **Client-server**
- **Layered**
- **Blackboard**
- **Pipe-and-filter**
- **Event-driven**
- Often, real systems are a mix of these.

Client-server

- Distributed components provide services to clients.
- Example code:

```
def collect_news_from_news_server(sym: str) -> str:
    return f"No news about {sym} is good news"

def collect_price_history_from_archive_server(sym: str) -> list[float]:
    return [1.1, 1.2, 1.3, 1.4, 1.5, 1.6, 0.7]

def magic_calc_price_by_news_price_history(
    sym: str, news: str, price_history: list[float]
) -> float:
    if news.find("good"):
        return statistics.mean(price_history) + statistics.stdev(price_history)
    else:
        return statistics.mean(price_history) - statistics.stdev(price_history)

def calc_stock_price(sym: str) -> float:
    news: str = collect_news_from_news_server(sym)
    price_history: list[float] = collect_price_history_from_archive_server(sym)
    return magic_calc_price_by_news_price_history(sym, news, price_history)
```

- **Explanation:** The code is illustrative; ‘collect$_{n}ews_{f}rom_{n}ews_{s}erver‘and‘collect_{p}rice_{h}istory_{f}rom_{a}rchive$

Layered

- System is organized in layers, each supporting and depending on the one above/below.
- Example: Library checkout system.
- **Business logic layer** (`Library`, `CheckoutRule`, `Notification` classes): Implements the actual behaviors.
- **Facade layer** (`LibraryFacade`): Provides a simple interface for application use.
- **Application layer**: Instantiates the facade and performs actions.
- **Explanation:** Each layer abstracts and hides details from the layer above. This allows for replacing or reusing layers independently.

Blackboard

- Used for complex, multi-domain problems (e.g., IDEs, speech recognition).
- Components:
  - **Blackboard**: Shared data, problem state.
  - **Knowledge Sources**: Sub-problem solvers.
  - **Control Shell**: Orchestrates the process.
- Example: Counting from 0 to 10 using blackboard approach (see code in chapter).
- **Explanation:** Each knowledge source reads from and writes to the blackboard; the control shell coordinates which sources act.

Pipe-and-filter

- Data flows through a series of filters (transformers); each filter processes data and passes it on.
- Example: Linux pipeline `fortune | cowsay | tee saved_cowsay.txt`.
- Useful for sequential or parallel transformations, and for reusability.

Event-driven

- Producers push data into a message broker (e.g., RabbitMQ, Kafka).
- Consumers listen for events and act.
- Example: Web app produces events; services listen and process them.
- Useful for highly decoupled and scalable systems.

7 Chapter 7: Software architecture 2 (`ch7.pdf`)

Recap: What must a component have?

- Clearly defined responsibilities.
- Clearly defined boundary.
- Well-defined interfaces.
- **Ultimate goal:** Loosely coupled system → high reusability, testability, manageability, scalability, etc.

Client-server: When and why?

- **When to use:**
  - Centralized resource management (e.g., user DB, access control).
  - Many clients accessing shared resources.
  - Heterogeneous clients (laptops, mobile, bots).
- **Potential issues:**
  - Server failure disables all clients.
  - Security: requires authentication and authorization for remote access.
- **Misconceptions:**
  - Server is not always remote.
  - Communication isn't always network-based (example: Docker, where `docker` client and `dockerd` server run on same machine via IPC socket).

Layered: When and why?

- **When to use:**
  - Functions can be abstracted recursively into levels.
  - Standardization needed (e.g., OSI 7-layer model).
  - Flexibility needed (replace a layer's implementation).
  - Reusability needed.
- **Potential issues:**
  - Abstraction may reduce efficiency (e.g., many function calls, added headers in network stacks).
  - Local changes may propagate to higher layers, indicating a bad layering scheme.
- **Example:**
  - Two Python versions of an OR gate: one composed of many small layered functions (slower), the other more direct (faster).

Blackboard: When and why?

- **When to use:**
  - Problem can be divided into sub-problems.
  - No deterministic overall solution, but sub-problems are tractable.
  - Partial knowledge or large search space.
  - Want to try different algorithms with reusable components.
- **Potential issues:**
  - Testing is difficult (results not reproducible).
  - High development effort (complex control strategy).

Pipe-and-filter: When and why?

- **When to use:**
  - Standardized input/output formats.
  - High reusability and flexibility (components reusable and reorderable).
  - Rapid prototyping needed.
- **Potential issues:**
  - Passing large data along the pipeline can be expensive.
  - For maximum flexibility, all filters must use the same format, which increases conversion overhead.

Event-driven (Broker): When and why?

- **When to use:**
  - System is highly decoupled or distributed.
  - Super scalability is required.
  - Components just need to know message format, not server locations or APIs.
  - Cross-platform communication is needed.
- **Potential issues:**
  - Less efficient due to indirections.
  - Easy to overuse because it sounds so flexible.
- **Examples:**
  - Intra-process: Qt's "signals and slots".
  - Inter-process/network: Apache Kafka, RabbitMQ, Faststream for Python.

8 Chapter 8: Introduction to Risk Analysis: Fault Tree Analysis (`ch8.pdf`)

Risk analysis: Why?

- **Mistakes are inevitable:** Planning for mistakes is essential in software engineering.
- **Testing is limited:** Can detect bugs, but cannot prove absence of all bugs.
- **All components are unreliable:** Some more so than others. This mindset is important for robust system design.
- **Risk/Safety analysis:** Required in certain standards (e.g., ARP4761 for aerospace).

Use cases for risk analysis

- Identify risks.
- Predict failure probability and system reliability.
- Prioritize tasks based on risk.
- Provide documented analysis for stakeholders.
- Checklist for architecture design.
- Define system metrics and rubrics.
- Debugging support.

Fault Tree Analysis (FTA): What and how?

- **Definition:** A graphical and mathematical method for predicting the causes and likelihood of system failures.
- **Origins:** Developed in the 1960s for launch control systems.
- **Output:**
  - **Qualitative:** Minimal combinations of failure causes.
  - **Quantitative:** Probabilities of failures.
- **Method:** Model a system fault as a tree of events, where each event is caused by other events, linked by logic gates (AND, OR).

Example of a Fault Tree

- **Scenario:** "Safety system fails to respond" is the top event.
- **Branches:**
  - Failure to detect heat (linked to heat detector or security guard fails to wake up).
  - Failure to detect smoke (linked to smoke detector or fire detection system).
- **Explanation:** Each branch represents logical dependencies; AND/OR gates model how failures combine.

Fault tree construction: Rules

- **Top event:** System failure mode.
- **Basic events:** Leaf-level component failures.
- **Gates:** AND/OR logic connects events.
- **Procedure:** Recursively break down system failures until only basic events remain.
- **Rules:**
  - Assume component failures propagate and become observable (no cancellations).
  - Define all gate inputs before expanding further (breadth-first).
  - Gates should not connect directly to other gates.

Second Fault Tree Example: Ineffective security

- **Top event:** Ineffective security.
- **Branches:** Weak password, broken encryption.
- **Sub-branches:** Password length, specific weak password, weak encryption algorithm, short encryption key.
- **Symbols:** Top event = logic gate; failure event = internal node; basic failure event = leaf.

Minimal cut set

- **Definition:** A (minimal) set of basic events whose occurrence guarantees the top event (system failure).
- **Example:** For the security tree, minimal cut sets are:
  - Both short encryption key **and** weak encryption algorithm.
  - Password length too short.
  - Password is "12345678".
- **Boolean representation:** Each basic event is a Boolean variable (a, b, c, d). The top event formula might be $ab + c + d$.

System failure probability

- For each minimal cut set C_i (composed of independent events X_{ij}): $P(C_i) = \prod_j P(X_{ij})$
- The top event's probability is bounded above by the sum of cut set probabilities: $P(\text{top event}) \leq \sum_i P(C_i)$.
- Alternatively, $P(\text{top event}) \leq 1 - \prod_i (1 - P(C_i))$.
- **Note:** Only the upper bound is usually practical to compute, since overlapping of cut sets is complex.

Importance measures

- **Purpose:** Quantify how much each component contributes to system failures.
- **Critical system state:** For each component, a "critical state" is when all other components are in a specific state such that failure of this component alone causes system failure.
- **Example:** Table shows for each combination of other components, whether a specific component is critical.

Structural importance measure

- **Definition:** Number of critical states for a component divided by total number of states.
- **Calculation:** For the example, a and b each have 1/8; c and d have 3/8.
- **Interpretation:** Focus on improving the most critical components (highest measure) to boost system reliability.
- Probabilistic measures can refine this further by considering failure probabilities.

Further references

- Handbook of Software Reliability Engineering, Chapter 15.

9 Chapter 9: Software design: Introduction (`ch9.pdf`)

What is software design?

- **Definition:**
  - **Process:** Steps taken to move from requirements to implementation.
  - **Product:** The output artifacts (design documents, diagrams, pseudocode, etc.).
- **Stages of design:**
  - Understanding the problem.
  - Identifying solutions.
  - Describing solution abstractions (technology-independent, e.g., UML).
- **Diagram:** Shows flow from requirements → specification → architecture → design → implementation.
- **Iteration:** Each stage may reveal new problems or abstractions, requiring revisiting earlier stages.

Architecture vs. Design

- **Architecture:**
  - High-level structure of the system.
  - What are the main components, their responsibilities, and APIs.
- **Design:**
  - Internal structure of components (classes, data structures, algorithms).
  - No code yet, just detailed blueprints.

Design process example: Fibonacci

- **Problem:** Implement $f(n) = f(n-1) + f(n-2)$, with $f(0) = 1$, $f(1) = 1$.
- **Solution options:**
  - **Iterative pseudocode:**
    * Initialize an array for previous/current values.
    * Shift and update values in a loop.
  - **Recursive pseudocode:**
    * If $n = 0$ return 0; if $n = 1$ return 1; else return $f(n-1) + f(n-2)$.
- **Analysis:**
  - Iterative: efficient, no repeated work.
  - Recursive: simple, but wastes memory and time on repeated calls.
- **Design output:** Pseudocode, technology-independent, is enough for direct implementation.
- **Implementation:** Actual code (Python) can differ from pseudocode for efficiency.

Essentials of a good design

- **Divide and conquer:** Break system into subsystems (which have business value) and modules (smaller, internal units).
- **Single responsibility:** Each component should only have one job.
- **Loose coupling:** Components can be easily swapped; changes don't ripple through the system.
- **High cohesion:** Functions and data in a component work tightly together towards a single goal.
- **Information hiding:** Details are hidden behind clear interfaces.
- **Data encapsulation:** Only expose what's necessary.

Coupling

- **Definition:** How strongly components depend on each other.
- **Desirable:** Loose coupling (easy substitution, minimal change ripple).
- **Types:**
  - **Tight:** Content coupling (directly manipulating another's data), Common coupling (global data), External coupling (depends on uncontrollable components, e.g., DB vendor).
  - **Medium:** Control coupling (passing control flags), Data structure/stamp coupling (sharing data structures but only using part).
  - **Loose:** Data coupling (only data passed via parameters), Message coupling (interaction only via messages, as in event-driven systems).
- **Examples:**
  - Code examples for each type are given (e.g., user/card classes for content coupling, global counter for common coupling, etc.).

Cohesion

- **Definition:** How well the elements of a component belong together.
- **High cohesion:** All internal data/functions are essential; no unused elements; ideally, each function uses all attributes.
- **Guideline:** "Put only closely related code together as a unit."
- **Goal:** Each unit should have a single clear responsibility and no code duplication.

10 Chapter 10: High level design strategies, and considerations in the design process (`ch10.pdf`)

Design at different levels

- **After requirements:** Multiple levels of design must be performed:
  - **Architectural Design:** How to divide the entire system into subsystems/modules?
  - **Interface Design:** What are the inputs/outputs for each subsystem or module? How do they interact? What is the user interface?
  - **Component Design:** How can a subsystem/module be decomposed further? How do submodules and data structures interact?
  - **Data Design:** What are the formats of data to be consumed/produced?
  - **Algorithm Design:** How are data structures and algorithms used to implement functions?
- **Note:** In practice, these are often considered simultaneously.

High-level design strategies

- **Top-down vs. bottom-up**
  - **Top-down:** Start from the root abstraction, decompose into finer levels. Organized, but real design rarely purely top-down.
  - **Bottom-up:** Identify essential standalone components, assemble into a complete system. Allows distributed development, but needs big picture planning.
- **Centralised vs. decentralised** (in this context: how you conceptualize the architecture, not deployment)

Top-down design example: Song composition

- Steps: Select a key; think about song structure; develop melody/chords for each section.

Bottom-up design example: Song composition

- Steps: Develop melodic/chord riffs; adjust riffs for consistency; join riffs using chord progressions.

Centralised design

- **Start:** Model the system from one component's perspective (e.g., the library system itself).
- **Example:**

```
class LibrarySystem:
    def __init__(self):
        self.borrow_records = BorrowRecords()
    def checkout(self, user: User, books : [Book]):
        self.borrow_records.add(user, books)
```

- **Explanation:** The system evolves as you realize new required components (BorrowRecords, User, Book).

Distributed design

- **Start:** Identify all "nouns" (major objects/actors, e.g., User, Book).
- **Example:**

```
class User:
    def __init__(self):
        self._borrowed_books = []
    def borrow_book(self, book: "Book"):
        self._borrowed_books.append(book)
        book.lend(self)
```

```
class Book:
    def __init__(self):
        self._borrower = None
    def lend(self, user: User):
        self._borrower = user
```

- **Explanation:** Each object manages its own state; interactions are through method calls.

Which design is better?

- **Desirable properties at all levels:**
  - Loose coupling
  - High cohesion
  - Information hiding
  - Data encapsulation
  - Fulfils all requirements
- **Trade-offs:** Sometimes decoupling adds complexity and overhead; balance is needed.

Software design metrics

- **Metrics exist:** (e.g., coupling metric)
- **Tools:** Some IDEs (Visual Studio, etc.) provide quality measurement.
- **Instinct:** Developing good judgment is key, not just metrics.

Learning from examples

- Reference: "The Architecture of Open Source Applications" (book).

Warning: Overengineering

- **Don't overengineer:** Avoid "Builder's Trap" (spending too much time on architecture at the expense of shipping working software).
- **Meme:** Overengineering is a killer; a simple, working product is better than a complex, unfinished one.

11 Chapter 11: Object-oriented programming (OOP) in Python (`ch11.pdf`)

Why classes and OOP?

- Model real-world entities as objects (classes define types, attributes, methods).
- Promotes reusability, data abstraction, encapsulation.

What is OOP?

- Groups related fields (attributes) and procedures (methods) as objects.
- Models intuitive real-world relationships (e.g., social media accounts that follow/message each other).

Four pillars of OOP

- **Encapsulation:** Expose only essential info, restrict direct data access.
- **Abstraction/Information hiding:** Hide details, modularize code.
- **Inheritance:** Share common code/behavior, enable reuse.
- **Polymorphism:** Change behavior while keeping common interfaces.

Encapsulation

- Data and methods are bundled into objects; data is accessed intentionally via methods.
- Example: Ticket website tracks tickets sold/available internally.
- Python uses conventions (`_attr`) for "private" fields.

Python class example: Social Media

```
class Post:
    def __init__(self, post_id: str):
        self._post_id = post_id
        self._content = ""
    def update_content(self, content: str):
        self._content = content

class SocialMediaAccount:
    def __init__(self, account_id: str, username: str):
        self._account_id = account_id
        self.username = username
        self._following: set[SocialMediaAccount] = set([])
        self._posts: list[Post] = []
    def add_to_following(self, another_account: "SocialMediaAccount"):
        self._following.add(another_account)
    def add_post(self, post: Post):
        self._posts.append(post)
```

- **Explanation:** Shows encapsulation (private attributes), public methods, and object relationships.

UML class diagram notation

- **Visibility:** Private (-), Public (+), Protected (#), Package ( ).

Abstraction/Information hiding

- Hide internal workings; provide simple, clear interfaces.
- Achieved via system modularization and easy-to-use function interfaces.

Class vs. object/instance

- **Class:** Blueprint for objects; defines data and behavior.
- **Object/Instance:** Concrete realization of a class.

Instance creation and usage

```
class Cat:
    def __init__(self):
        self.name = 'Kitty'
        self.breed = 'domestic short hair'
        self.age = 1
    def print_info(self):
        print(self.name, 'is a ', self.age, 'yr old', self.breed)
pet_1 = Cat()
pet_2 = Cat()
```

- **Explanation:** Demonstrates instantiation, attribute assignment, and method invocation.

Instance vs. class attributes

```
class CoffeeOrder:
    loc = "Cafe Coffee"
    cls_id = 1
    def __init__(self):
        self.order_id = CoffeeOrder.cls_id
        self.cup_size = 16
        CoffeeOrder.cls_id += 1
    def print_order(self):
        print(CoffeeOrder.loc, "Order", self.order_id, ":", self.cup_size, "oz")
order_1 = CoffeeOrder()
order_2 = CoffeeOrder()
order_3 = CoffeeOrder()
order_3.print_order()
```

- **Explanation:** Class attributes (`loc`, `cls_id`) are shared; instance attributes are unique to each object.

Object instantiation and initialization

- **Steps:**

  - Creation: __new__ allocates memory.
  - Initialization: __init__ sets attributes.
- **Destructor:** __del__ cleans up resources.
- **Singleton example:** __new__ can be used to implement a singleton pattern.

Class methods and static methods

```
class Person:
    @classmethod
    def new_with_name_age(cls, name, age):
        return cls(name, age)
    @staticmethod
    def create_with_name_age(name, age):
        return Person(name, age)
```

- **Explanation:** Class methods operate on the class; static methods do not access class or instance state.

Instance methods

```
class ProductionCar:
    def update_max(self, speed):
        self.max_mph = speed
```

- **Explanation:** Instance methods operate on object state.

Inheritance

- **Is-a relationship:** Subclass inherits from superclass (e.g., `class Daisy(Plant):`).
- **Has-a relationship:** Composition (e.g., Employee has a Laptop).
- **Terminology:** Superclass/parent/base, subclass/child/derived.
- **Access:** Subclasses access public/protected, but not private, members.

OOP: Inheritance example

```
class SuperClass:
    def __init__(self):
        self.feat_1 = 1
        self.feat_2 = ""
    def bc_display(self):
        print(f"Superclass: {self.feat_2}")

class SubClass(SuperClass):
    def dc_display(self):
        print(f"Subclass: {self.feat_2}")
```

12 Chapter 12: Software design: Design Patterns (`ch12.pdf`)

What is a pattern?

- The idea comes from architect Christopher Alexander, who defined a pattern as a three-part rule: context, problem, and solution.
- In software, a pattern is "a solution to a problem in a context" and can be applied to many disciplines.

Why patterns?

- Designing reusable object-oriented software is hard (Erich Gamma).
- Experienced designers reuse successful solutions (patterns).
- Recognizing and naming recurring structures (patterns) increases productivity and flexibility.

Software patterns history

- 1987: Cunningham and Beck apply Alexander's ideas in Smalltalk.
- 1990-1995: "Gang of Four" (Gamma, Helm, Johnson, Vlissides) publish the seminal book *Design Patterns*.
- 1991+: Patterns community forms (Hillside Group, PLoP conference, etc).

Types and levels of software patterns

- **Types:** Analysis, design, organizational, process, project planning, configuration management.
- **Levels:** Abstract (entire systems), mid-level (GoF design patterns), concrete (e.g., linked list).

GoF Design Patterns

- **Purpose:**
  - Creational: how objects are created.
  - Structural: composition of classes/objects.
  - Behavioral: interactions among classes/objects.
- **Scope:**
  - Class patterns: inheritance (class relationships).
  - Object patterns: composition (object relationships).

GoF pattern template

- Pattern Name and Classification
- Intent (purpose)
- Also Known As (aliases)
- Motivation (scenario)
- Applicability (where to use)
- Structure (diagram)
- Participants (classes/objects involved)
- Collaborations (how they interact)
- Consequences (pros/cons)
- Implementation (how to code it)
- Sample Code
- Known Uses
- Related Patterns

Benefits of design patterns

- Capture and communicate expertise.
- Provide a shared vocabulary.
- Encourage reuse and avoid poor alternatives.
- Facilitate change and documentation.

Factory Patterns (Creational)

- **Factory patterns** abstract object creation, hiding details and allowing system independence from concrete classes.
- **Factory Method:** Lets subclasses decide which class to instantiate by defining a method for object creation, which subclasses override.

Factory Method Example

- **Motivation:** Framework with an Application class and Document classes; subclasses instantiate appropriate Document via a factory method.
- **Structure:**

```
Creator
  +create_product() -> Product
Product
  +open()
  +close()
  ...
ConcreteProduct, ConcreteCreator implement interface
```

- **Usage:** Code can work with Creator and Product interfaces, and concrete subclasses provide actual classes.
- **Refactored:** Application has a create$_{d}ocument()factorymethod; MyApplicationoverridesittocreateMy$

Factory Method: Participants/Collaborations

- **Product:** Interface for objects created by the factory.
- **ConcreteProduct:** Implements Product.
- **Creator:** Declares the factory method returning Product.
- **ConcreteCreator:** Overrides factory method to return ConcreteProduct.

Factory Method: Consequences

- **Benefits:** Decouples code from specific classes, makes code reusable/flexible.
- **Liabilities:** May require subclassing for each new Product.
- **Implementation:** Factory method can be abstract or concrete; may take parameters to decide what to instantiate.

Abstract Factory Pattern

- Provides an interface for creating families of related objects without specifying concrete classes.
- Difference from Factory Method: delegates instantiation to a factory object (composition), not just a subclass (inheritance).

Abstract Factory Example

- **GUI toolkit:** WidgetFactory creates ScrollBar and Window; MotifWidgetFactory creates MotifScrollBar/MotifWindow, PMWidgetFactory creates PMScrollBar/PMWindow.
- **MazeFactory:** MazeGame uses a MazeFactory to create rooms, walls, doors; can swap in EnchantedMazeFactory, etc.
- **Structure:**

```
AbstractFactory
  +create_product_a()
  +create_product_b()
ConcreteFactory1
  +create_product_a()
  +create_product_b()
...
Client uses AbstractFactory/AbstractProduct interfaces
```

Abstract Factory: Consequences

- Isolates clients from concrete classes.
- Makes swapping product families easy.
- Enforces use of only one family at a time.
- **Liabilities:** Can become complex; adding new products requires interface changes.
- **Implementation:** Typically a singleton; factories use factory methods to create products.

Singleton Pattern

- Ensures a class has only one instance, provides a global point of access.
- Motivation: e.g., window manager, resource manager, factory.
- Structure: static instance variable, static factory method.
- Implemented in Python using __new__ or metaclasses; care required for thread safety and to prevent direct instantiation.

Adapter Pattern

- Converts the interface of a class into another expected by clients.
- **Class Adapter:** Uses multiple inheritance; Adapter inherits from Target and Adaptee.
- **Object Adapter:** Uses composition; Adapter holds an Adaptee and implements Target.
- **Applicability:** Use when you need to work with an existing class but its interface is incompatible.
- **Implementation:** May adapt a single method or a full interface; can provide two-way adaptation if needed.

Adapter Examples

- **Round peg / square peg:** Adapts different interfaces for insertion.
- **Two-way adapter:** Implements both interfaces (via abstract base class).
- **Selective copying:** Adapts objects to a Copyable interface for a utility.

13 Chapter 13: Software design: Design Patterns 2 (`ch13.pdf`)

Composite Pattern

- **Intent:** Compose objects into tree structures to represent part-whole hierarchies; treat individual objects and compositions uniformly (recursive composition).
- **Structure:**

```
Component
  +operation()
  +add(), +remove(), +get_child()
Leaf
  +operation()
Composite
  +children: [Component]
  +operation(), +add(), +remove(), +get_child()
```

- **Example:** Graphics (Line, Rectangle, Picture, etc.); GUI widgets (Windows, Buttons, Containers).
- **Benefits:** Easy to add new components; clients can treat all as Components.
- **Liabilities:** Over-generalization if classes differ too much; may waste space if every leaf has a children list.

Composite Pattern: Implementation Issues

- Should components know their parent? Depends (e.g., for traversal).
- Where to put add/remove/get_child? In Component (uniform interface) or only in Composite (more type safety).
- Is child ordering important? What data structure to use for children?

Composite Pattern: GUI Example

- **Bad design:** Each widget type has a different interface; Window must know how to update each.
- **Better:** All widgets implement a common interface. Window can call draw() on all.
- **Best:** Composite pattern; both simple widgets and containers implement the Widget interface; containers can contain children, and Window can treat everything uniformly.

Decorator Pattern

- **Intent:** Add responsibilities to objects dynamically; a flexible alternative to subclassing.
- **Structure:**

```
Component
  +operation()
ConcreteComponent
  +operation()
Decorator
  +component: Component
  +operation()
ConcreteDecoratorA/B
  +added_state/behaviour
  +operation()
```

- **Example:** FileReader can be decorated with BufferedFileReader for buffering.
- **Benefits:** Add features at runtime, combine features flexibly.
- **Liabilities:** Many small classes; can look like Adapter.

Observer Pattern

- **Intent:** One-to-many dependency; when subject changes, observers are notified.
- **Also Known As:** Publish-Subscribe, Dependents, Model-View.
- **Structure:**

```
Subject
  +attach(), +detach(), +notify()
Observer
  +update()
ConcreteSubject, ConcreteObserver
```

- **Benefits:** Decouples subject from observer; supports event broadcasting; dynamic addition/removal.
- **Liabilities:** Cascading notifications; observers may need to deduce what changed.

Observer Pattern: Implementation Issues

- How does the subject track observers? (Array, set, etc.)
- Observers may subscribe to specific events (publish-subscribe).
- Observers can be subjects themselves (chained notifications).
- Subject can use push (send data) or pull (just notify, observer pulls info).

Other Patterns and Pitfalls

- Iterator, Strategy, Visitor, MVC/MVVM.
- **Pattern-abuse:** Don't use patterns everywhere just because they're cool; over-abstraction leads to complexity, cognitive overload, and performance costs.
- **Anti-patterns/code smells:** Deep class hierarchies, unnecessary abstractions, 1:1 class-to-noun mapping, etc.

Over-engineered Design Example

- Example: Pokemon system using Singleton, Composite, Decorator, Factory, etc. All patterns at once make the code hard to understand, maintain, and extend.
- **Simpler alternative:** Use basic classes with only the needed features.
- **Lesson:** Only use patterns as needed; defer adding abstractions unless justified.

Summary

- Deferred implementation (via polymorphism) achieves low coupling.
- Composition enables safe multiple inheritance.
- Apply patterns judiciously, not for their own sake.

14 Chapter 14: Software Testing Terminologies and Strategies (`ch14.pdf`)

Attitudes toward testing

- Humorous myths: "Real programmers need no testing!" "Testing is for the weak." "Most functions rely on built-in types, so no need to test."
- **Message:** These are dangerous attitudes; thorough testing is an essential part of quality software.

Three hard problems in Computer Science

- **Phil Karlton's quote:** "There are only two hard things in Computer Science: cache invalidation and naming things." Off-by-one errors are added as a third by the slides.

Building Quality Software

- **External qualities:** Correctness, reliability, efficiency, integrity.
- **Internal qualities:** Portability, maintainability, flexibility.
- **Quality Assurance:** The process (including testing) to uncover and fix problems.

Testing: Limitations and Value

- **Testing can find bugs, but cannot guarantee bug-free software** (Halting Problem analogy).
- **Full testing is unachievable** for complex systems; some bugs will always slip through.
- Testing increases confidence and quality, but is inherently incomplete.

Validation vs. Verification

- **Validation ("Are we building the right product?"):** Does the software meet user needs?
- **Verification ("Are we building the product right?"):** Is each function correct and defect-free?
- **Acceptance tests:** Focus on validation; most other testing is verification.

Phases of Testing

- **Unit Testing:** Does each function/module do what it's supposed to?
- **Integration Testing:** Do modules work together correctly?
- **Validation Testing:** Does the software meet requirements?
- **System Testing:** Does the software work in the overall system?
- **Complexity:** Lower for unit, higher for system testing.
- **Participants:** Developers do unit/integration; customers are involved in validation/system testing.

What's so hard about testing?

- **Example: Boolean circuit with a stuck-at-0 fault.** Testing all inputs is impractical; need to find minimal "test vectors" that activate and propagate the bug.
- **Exhaustive testing is impractical.** Must select a small but effective test suite using heuristics.

Test Planning

- Specify strategy (which tests, coverage, outcome percentage), schedule, and resource estimates.

Input Space Partitioning

- **Ideal:** Partition inputs into sets with same behavior; test one value from each.
- **Reality:** Use heuristics to approximate these sets.
- **Execution equivalence:** Inputs leading to same code path are equivalent (e.g., all $x < 0$ for `abs(x)`).
- **Heuristics:** For a range, test at/around boundaries; for a set, test valid/invalid; for a value, test $x - 1, x, x + 1$.
- **Revealing subdomains:** Partition inputs into sets that will reveal an error if present.

Test Case: Four parts

- What to test, under what conditions, with what input, and what is the expected result.
- Steps: select input/config, specify expected, execute and document, compare to expected.

Types of test case designs

- **Black Box:** Functionality from outside; no code knowledge.
- **White Box:** Internals, control/data flow; code is required.
- **Regression:** Re-test to ensure no new defects after changes.

Black Box Testing

- **Heuristic:** Explore alternative paths through specification.
- **Boundary cases:** Test overflow, duplicates, nulls, aliasing, etc.
- **Advantages:** Not influenced by code, allows independent testers, can be written before code, robust to implementation change.
- **Limitations:** Cannot reveal certain bugs (e.g., logic bug at $x = -2$ in a broken `abs(x)`).

White Box Testing

- **Goal:** Execute all code components; measure coverage.
- **Coverage:** Statement, branch, path.
- **Heuristics:** Test all paths, all logical conditions, loops at boundaries, data flows.
- **Examples:** Condition testing (regex for email), loop testing (sum over list), data flow testing (average calculation, bank account).
- **Tools:** Automated coverage tools are vital.

Regression Testing

- **Purpose:** Prevent recurrence of fixed bugs.
- **Process:** Reproduce bug, document inputs/outputs, add to test suite, confirm fix.
- **Remember:** If a bug happened once, it can happen again.

Test case types in different phases

- Unit/Integration: black box, white box.
- Validation/System: black box.

15 Chapter 15: Software Testing, and with Python (`ch15.pdf`)

Testing in Python

- **Definition:** Evaluate system/components to verify requirements are met.
- **Tools:** Python has several, e.g., `pytest`.

pytest and its plugins

- **pytest:** User-friendly, powerful testing framework.
- **Installation:** `pip install pytest`
- **Plugins:** `pytest-cov` (coverage), `pytest-xdist` (parallel), `pytest-mock`, `pytest-benchmark`.

pytest: Example

```
class SimpleNeuralNetwork:
    def __init__(self):
        self.weights = [0.5, -0.51]
    def predict(self, inputs):
        return sum(w * i for w, i in zip(self.weights, inputs))
    def train(self, inputs, target):
        prediction = self.predict(inputs)
        error = target - prediction
        self.weights = [w + 0.1 * error * i for w, i in zip(self.weights, inputs)]
```

- **Test file:** Test functions start with `test_` and assert expected behavior.
- **Assertions:** Used to check conditions; a failed assertion fails the test.
- **Running tests:** `pytest` in directory, with options (`-v`, `--maxfail=1`, specific files, etc.).
- **Test discovery:** Files and functions starting with `test_` are discovered automatically.

Fixtures

- **Definition:** Fixed state/environment for tests.
- **Purpose:** Consistency, isolation, reusability, efficiency.
- **Types:** Setup/teardown, function/class/module/session scope.
- **Usage:** Define with `@pytest.fixture`; inject in test by argument.
- **Best practices:** Keep simple, scope properly, document, avoid interdependencies.

Parameterization

- **Definition:** Run the same test with different input data.
- **Benefits:** More coverage, less duplication, maintainability, efficiency.
- **Usage:** `@pytest.mark.parametrize` decorator.
- **Example:** Test a function for multiple input/output pairs.

Mocks and Stubs

- **Mocking:** Simulate object behavior to isolate tests.
- **Benefits:** Isolation, control, speed, reliability, state collection.
- **pytest-mock:** Plugin for mocking; use `mocker.Mock()`.
- **Stubs:** Simplified module used to simulate lower-level modules in top-down testing.

Testing Strategies with Mocks and Stubs

- **Top-Down:** Test high-level first, use stubs for lower modules.
- **Bottom-Up:** Test low-level modules first, use drivers for higher levels.
- **Sandwich (Hybrid):** Combine both; practical for real projects.
- **Each has advantages/disadvantages** (e.g., stub/driver complexity, timing of bug detection).

Performance Testing and Benchmarking

- **Benchmarking:** Measure performance under specific conditions; micro/macro/synthetic/real-world.
- **pytest-benchmark:** Plugin to benchmark code; use `benchmark()` in test.
- **Best practices:** Isolate benchmarks, use realistic workloads, repeat for accuracy, document results.

Profiling

- **Definition:** Analyze program to find performance bottlenecks.
- **Key:** More detailed than benchmarking; focus on functions, calls, and resource usage.
- **Tools:** `cProfile`, `profile`, `line_profiler`.
- **Difference from benchmarking:** Benchmarking measures overall performance; profiling breaks down by code unit.
- **Visualization:** Use KCachegrind (with `pyprof2calltree`) to analyze call graphs.

Code Coverage

- **Definition:** Fraction of code executed by tests.
- **pytest-cov:** Plugin for coverage; generates terminal and HTML reports.
- **Best practices:** Aim for high coverage, but focus on meaningful tests, not just numbers.

Suggested Testing Practices

- **Directory structure:** `src/` for source, `tests/` for tests.
- **Naming:** Consistent `test_*.py` naming.
- **Use fixtures for common setup.**
- **Keep tests independent.**
- **Document tests with docstrings and README.**
- **Code duplication:** Occasionally okay for clarity or legacy, but prefer to avoid by using fixtures/helpers.

Reminder

- Testing is essential—tools may change, but the principles apply across languages and environments.
- Regular testing and optimization ensure reliable, maintainable, high-quality software.

16 Chapter 16: Profiling, and code optimisation (`ch16.pdf`)

More on Profiling

- **Profiling:** Analyzing and measuring program performance to see where time is spent.
- **Why:** Useful for long-running or complex code; helps pinpoint bottlenecks.
- **Benefits:** Quick process; can reveal peace of mind or identify slow areas. Identifying bottlenecks allows precise, effective optimization—sometimes yielding 100x–1000x speedups!
- **Energy:** Profiling also helps ensure efficient use of system resources (important for High Performance Computing).

When to Profile

- **Relevant when:** Code is working and approaching deployment; anything running for more than a few minutes.
- **Profiling is cheap:** If no major bottlenecks, you quickly know code is OK; if there's a bottleneck, targeted optimization can yield massive improvements.

Types of Profiler

- **Manual:** Use `print()` and timing (e.g., `time.monotonic()`) to measure code sections.
- **Function-level:** Counts calls and execution time per function, including/excluding child functions (e.g., `cProfile`).
- **Line-level:** Measures time for individual code lines (good for complex or long functions).

Selecting Appropriate Test Case

- Profilers add overhead (code runs slower).
- High-resolution profiling uses more CPU/memory/storage.
- Choose a test case representative of workload, but small enough to run quickly (ideally a few minutes).
- Example: Simulate one day of a year-long simulation to profile, not the whole year.

Function-level Profiling - Example

```
def a_1():
    for i in range(3): b_1()
    time.sleep(1)
    b_2()
def b_1(): c_1(); c_2()
def b_2(): time.sleep(1)
def c_1(): time.sleep(0.5)
def c_2(): time.sleep(0.3); d_1()
def d_1(): time.sleep(0.1)
a_1()
```

- Run with `python -m cProfile -o out.prof ...`; view output with `snakeviz` or `kcachegrind`.

Optimisation

- **Profiling:** Understands program behavior.
- **Optimization:** Changes code or config based on profiling.
- **Major methods:**

- **Algorithm:** Analyze/replace with better ones (hash tables vs lists, parallelism, heuristics, randomized algorithms).
- **Bottleneck identification:** Use profiling tools to find "hotspots."
- **Code and memory:** Inlining, loop unrolling, cache awareness, recursion best practices, GC tuning, efficient memory allocation.
- **I/O:** Buffering, async I/O.
- **Concurrency:** Thread pools, lock-free data structures.
- **Caching:** Data/result caching.

- **Rules:**
  1. Don't optimize when unnecessary.
  2. Don't optimize before the code is correct.
  3. Don't optimize without regression tests.
  4. Know when to stop.

Performance vs. Maintainability

- **Knuth's quote:** "Premature optimization is the root of all evil."
- Maintainability is as important as performance; optimize only the critical 3% of code.
- Don't micro-optimize everything; focus on algorithms/data structures and profile-driven changes.

Understanding Computer Program Execution and Optimisation

- **How programs run:** Load from storage, fetch/decode/execute/store instructions.
- **Function call and stack:** Each call creates a stack frame with parameters, locals, return address; stack grows/shrinks per call.
- **Example in C:** Shows stack frames for main(), bar(), foo().

Memory System

- **Hierarchy:** L1/L2/L3 cache (fast, small), RAM, disk (slow, large).
- **Cache:** Data moves from main memory to caches based on temporal/spatial locality, cache misses, replacement policy, prefetching.
- **Example:**
  - `sequential_access()` faster than `random_access()` due to better cache locality.

Code Optimisation Techniques

- **Inlining:** Replace function calls with body to avoid call overhead (may hurt readability/code size).
- **Loop unrolling:** Replicate loop body to reduce control overhead and allow instruction-level parallelism.
- **Struct field alignment:** Arrange struct fields to minimize padding and cache misses (e.g., C structs).
- **Tail recursion:** Special case of recursion where last operation is recursive call; allows stack frame reuse (Python does not optimize this by default).
- **GC tuning:** Adjust thresholds to balance memory use and pause times; understand reference counting and cycles.
- **Memory pools:** Allocate/deallocate memory in bulk to reduce fragmentation and allocation overhead (hard in Python, easier in C).
- **Object pools:** Reuse objects rather than create/destroy repeatedly (can be implemented in Python via queues).
- **Language-specific:** Use built-in functions (e.g., Python's `sum`, `in`), list comprehensions, etc., for speed and readability.

Benchmarking/Profiling and Results

- Use benchmarking and profiling to prove optimizations are effective.
- Don't optimize before software is proven correct.
- Abstractions hide both complexity and performance implications.

17 Toolchains (Python) And Consistent Development Environment (`ch_toolchains.pdf`)

Toolchain: Overview

- **Toolchain:** Series of development tools from code writing to deployment.
- **Key question:** For each new project/language/framework: Is there a tool for each important task?

Source control

- **Git:** The de facto standard; platforms like GitHub/GitLab/Bitbucket.
- **Purpose:** Collaboration, history, rollback.

Development Environment - Interpreter

- **Python interpreter:** CPython, PyPy (faster), Jython (Java integration).
- **Choose version:** E.g., Python 3.11/3.12 for compatibility.

Development Environment - Virtual Environments

- **Problem:** "Python version hell"—project dependencies can conflict.
- **virtualenv/venv:** Create isolated environments per project (`python -m venv myenv`).

Development Environment - Package Manager

- **pip:** Installs/updates Python libraries from PyPI.
- **Poetry:** Modern, manages dependencies and builds.
- **requirements.txt:** Lists package/version; ensures reproducibility.

Build and Dependency Management

- **setuptools:** For packaging/distribution (setup.py).
- **Poetry:** Also handles builds (`pyproject.toml`).
- **Wheel:** Binary distribution format.

Consistent and Easy-to-setup Environments

- **Docker:** Bundles app with dependencies (solves "Python version hell").
- **VMs:** Emulates OS/hardware (heavyweight).
- **Nix/devenv:** Reproducible environments; all versions, all packages, fully specified.

IDE/Text Editor

- **PyCharm:** Full-featured (not free).
- **VSCode:** Lightweight, extensible.
- **Jupyter:** For interactive data science.
- **Vim, Emacs, Helix, etc.:** Powerful with language server support.

Linters and Formatters

- **Flake8:** Style and syntax errors.
- **Black:** Automatic code formatting.
- **Pylint:** Code analysis, style, errors.

Static Code Analysis

- **Bandit:** Checks security issues.
- **SonarQube:** Advanced metrics (free community edition available).
- **Pylint:** Also does static analysis.

Testing Frameworks

- **unittest:** Built-in.
- **pytest:** Popular, feature-rich.
- **pytest-cov:** Coverage plugin.
- **pytest-benchmark:** Performance benchmarking.
- **doctest:** Tests in docstrings.
- **Fuzzers:** E.g., Google Atheris for random input testing.

CI/CD (Continuous Integration/Deployment)

- **GitHub Actions, GitLab Pipeline, Jenkins:** Automate testing, building, deployment.
- **Tox:** Tests across multiple Python versions.

Debugging

- `print()`: Simple but manual and error-prone.
- `pdb`: Built-in debugger.
- IDE debuggers: Breakpoints, variable inspection.

Profiling

- `cProfile`: CPU/time profiling.
- `profile`: Pure Python, slower.

Documentation

- `docstring`: In-code docs.
- `Sphinx`: Generates professional docs from docstrings.

Deployment

- `Docker, VM`: Bundle and deploy with dependencies.
- `Nuitka`: Compile Python to binary.

Using Nix and devenv: Example

- **Install Nix & devenv:** See provided commands.
- **devenv.nix:** Specifies Python packages, linters, tasks, git hooks.
- **poetry:** Used for package/dependency management within the environment.
- **Reproducibility:** Any developer can use the same files to get an identical environment.