CS101: Software Engineering Principles

Week 3: Breaking Problems Down

Milestone 1: Conceptual Scoping & Problem Engineering

Before writing a single line of production code, a software engineer must establish absolute clarity regarding why a feature exists and what boundary constraints it must satisfy to prevent system scope creep.

Lesson 1: Writing a Clear Feature Brief

A professional Feature Brief is a tight, 4–6 sentence paragraph explaining a feature's scope without leaking technical implementation details. Every high-quality brief uses the 1-2-3 Drafting Method to guarantee three core dimensions are accounted for:

■ Click to Expand: The 1-2-3 Drafting Breakdown

- **Goal (Line 1):** The direct business value or user problem the code solves.
- **Target Users (Line 2):** Exactly who will be interacting with this system space.
- **Context (Line 3):** The concrete situation, time of day, or environment where execution occurs.

```
# Case Study Application: Daily Bedtime Routine Feature Brief
# "This feature gives parents a simple way to remind and track pre-bedtime tasks their children
# should complete each night. It runs on the parent's phone and helps establish a consistent
# routine for young children. Parents can create their own list of tasks or use a built-in
# checklist with items like brushing teeth, washing their face, or saying prayers."
```

Lesson 2: Separating Purpose from Technical Details

A good purpose sentence is implementation-neutral—it tells why the code exists in a way that remains true even if the internal mechanics are completely rewritten.

■ Click to Expand: Spotting Technical Leakage & Red-Flags

If a high-level architectural comment mentions specific fields, control flows, loops, or variable scopes, it has drifted into an implementation detail.

```
# ■ INCORRECT (Technical Leakage):
# Updates the items list and sets Active = False when the user clicks the cancel button.

# CORRECT (Implementation-Neutral Purpose):
# Allows the user to cancel their subscription if they no longer want service so they are not
billed.
```

Lesson 3: Writing a Structured Problem Statement

Once a brief is set, we expand it into a formal 4-part Problem Statement. This serves as a testable checklist split into clear, observable pass/fail rules.

■ Click to Expand: Structural Blueprint Requirements

1. Header: A clear, concise title marking the problem domain scope.
2. Purpose: A 1-2 sentence summary of human intent.
3. Constraints: Explicit boundaries specifying what the system must not do.
4. Success Criteria: Numbered, observable pass/fail rules.

```
problem_statement = """
Header: Daily Bedtime Routine Tracker
Purpose: This program helps parents establish and track structured nightly routines for children.
Constraints:
    - The program must run as a phone application layout canvas.
    - If the split input count is <= 0, the program must trigger an input error prompt.
"""
```

Milestone 2: Data Routing Contracts & Invariant Testing

Lesson 4: Defining Example Inputs and Outputs

Every subproblem requires an input/output map. When constructing these tables, you must explicitly account for standard paths and edge boundaries.

■ Click to Expand: Boundary Analysis Theory

• Normal Cases: Typical valid test criteria that pass easily under basic conditions.

• Edge Cases: Boundary values sitting on the absolute minimum or maximum limits of allowed inputs.

```
# Subproblem Contract: calculate_tip: bill, tip_pct, people -> tip, total, split
# ================================================================================
# INPUTS | ASSUMPTION PROFILE | EXPECTED OUTPUTS
# ================================================================================
# (100, 10, 5) | Normal Case: Standard Split| (10.0, 110.0, 22.0)
# (100, 0, 5) | Low Edge: Zero Percent Tip | (0.0, 100.0, 20.0)
```

Lesson 12: Writing a Full Problem Statement with Assertions

Instead of using visual screen print statements to manually inspect behavior, professional environments use assertions to enforce hard constraints.

■ Click to Expand: Why Assertions Beat Manual Print Checks

Console log print statements require human eyes to manually verify calculations on every single rerun. An assert statement evaluates instantly and completely silently.

```
def calculate_tip(bill, tip_percent, num_people):
    tip_amount = round(bill * (tip_percent / 100), 2)
    grand_total = round(bill + tip_amount, 2)
    per_person = round(grand_total / num_people, 2)
    return tip_amount, grand_total, per_person

tip, total, split = calculate_tip(50, 15, 4)
assert tip == 7.5, f"Error: Got {tip}"
```

Milestone 3: Functional Decomposition & Structural Patterns

Lessons 5, 6, & 7: Functional Decomposition, Contracts, and Data Flow Sequences

When faced with a complex feature request, trying to write it all at once invites logic errors. Instead, engineers map requirements step-by-step through structural decomposition.

■ Click to Expand: The 3-Step Decomposition Pipeline

1. Lesson 5: Single-Responsibility Cut: Break a requirement down into individual tasks that do exactly one kind of work starting with a strong action verb.
2. Lesson 6: Subproblem Contracts: Define a clear input/output contract signature for every step before coding.
3. Lesson 7: Logical Sequencing: Order tasks into a strict execution chain where data outputs pass smoothly.

Lesson 8: Matching Subproblems to Patterns

To lower your cognitive overhead when turning a plan into logic, map the active verb of your subproblem directly onto a structural Python layout pattern:

Subproblem Core Verb	Target Pattern Name	Production Code Hint Style
look up / find / translate	Lookup Table	`value = dictionary_map.get(key, default)`
calculate total / count / collect	Accumulator Loop	`total += score OR list.append(item)`
skip bad data / remove optional	Filter Loop	`for item in items: if invalid: continue`
exit on bad input / protect boundary	Guard Clause	`if not parameter_list: return "Error Signal"`
convert each item / map one-to-one	Transform Loop	`for x in raw_items: transformed.append(modify(x))`

Lesson 9: Prioritizing Tasks Systematically

When managing multiple subproblems, development priority scores are calculated by analyzing three core vectors: Impact (business value), Effort (complexity), and Risk (blast radius).

■ Click to Expand: The Task Priority Scoring Formula

Priority Score = (Impact * 2) - Effort + (5 - Risk). A higher score reveals mission-critical tasks.

```
for t in tasks:
    t["priority"] = (t["impact"] * 2) - t["effort"] + (5 - t["risk"])
tasks.sort(key=lambda t: t["priority"], reverse=True)
```

Lesson 10 & 11: Case Study Extraction & Pipeline Synthesis

The final design stage is taking an unstructured user request, separating pure computational logic away from UI presentation components, and orchestrating them into a unified data channel.

```
def calc_totals(monthly_transactions):
    category_totals = {}
    for item, category, amount in monthly_transactions:
        category_totals[category] = category_totals.get(category, 0) + amount
    return category_totals
```