Assignment no: 1

Software Construction

Haseeb Irfan

01-131232-029

Question 1: Understanding Metaphors in Software Development

Why are metaphors important?

Metaphors play a crucial role in software development because important developments often arise out of analogies. By comparing a topic you understand poorly to something similar you understand better, you can come up with insights that result in a better understanding of the less-familiar topic. This use of metaphor is called **"modeling."** Metaphors contribute to a greater understanding of software-development issues in the same way that they contribute to a greater understanding of scientific questions.

Classical Examples of Metaphors from Scientific History

1. **Kekulé's Dream of the Snake**: He saw a snake biting its own tail and realized that benzene's molecular structure could be a ring.
2. **Billiard-Ball Model of Gases**: Gas molecules were compared to billiard balls colliding elastically, forming the basis of the kinetic theory of gases.
3. **Wave Theory of Light and Sound**: Light was compared to sound waves because both have amplitude, frequency, and wavelength.
4. **Galileo's Pendulum**: Instead of seeing a swinging stone as "falling with difficulty" (Aristotelian view), Galileo saw it as a pendulum with repeating motion, leading to discoveries about periodic motion.
5. **Heliocentric Model (Copernicus)**: Replacing the earth-centered (Ptolemaic) model with a sun-centered one, which reshaped astronomy and our understanding of the universe.

Difference Between Algorithm and Heuristic

Algorithm

An **algorithm** is a set of well-defined instructions for carrying out a particular task. An algorithm is predictable, deterministic, and not subject to chance. An algorithm tells you how to go from point A to point B with no detours, no side trips to points D, E, and F.

Example: Here is an algorithm for driving to someone's house: Take Highway 167 south to Puyallup. Take the South Hill Mall exit and drive 4.5 miles up the hill. Turn right at the light by the grocery store, and then take the first left. Turn into the driveway of the large tan house on the left, at 714 North Cedar.

Heuristic

A **heuristic** is a technique that helps you look for an answer. Its results are subject to chance because a heuristic tells you only how to look, not what to find. It doesn't tell you how to get directly from point A to point B; it might not even know where point A and point B are. In effect, a heuristic is an algorithm in a clown suit. It's less predictable, it's more fun.

Example: Here's a heuristic for getting to someone's house: Find the last letter we mailed you. Drive to the town in the return address. When you get to town, ask someone where our house is. Everyone knows us—someone will be glad to help you. If you can't find anyone, call us from a public phone, and we'll come get you.

Conclusion

An **algorithm** gives you the instructions directly. A **heuristic** tells you how to discover the instructions for yourself, or at least where to look for them.

Common Software Development Metaphors

1. **Science** – Writing software is a science (David Gries, 1981)
2. **Art** – Software is an art (Donald Knuth, 1998)
3. **Process** – It's a structured process (Watts Humphrey, 1989)
4. **Driving a Car** – Compared to driving, but with different interpretations (P. J. Plauger, 1993; Kent Beck, 2000)
5. **Game** – Like a cooperative and strategic game (Alistair Cockburn, 2002)
6. **Bazaar** – Open, collaborative, and evolving like a marketplace (Eric Raymond, 2000)
7. **Gardening** – Requires nurturing, patience, and constant care (Andy Hunt & Dave Thomas)
8. **Filmmaking** (Snow White and the Seven Dwarfs) – Large, creative, and coordinated production (Paul Heckel, 1994)
9. **Farming / Hunting Werewolves / Drowning in Tar Pits** – Metaphors of struggle, persistence, and complexity (Fred Brooks, 1995)

Analysis of Software Development Metaphors

1. Writing Code Metaphor

Advantages:

- **Simplicity**: Gives an easy-to-understand picture—just sit down and "write" the program from start to finish
- **Individual Work**: Works adequately for small projects or when one person is developing software
- **Readability**: Encourages focus on program readability (like good writing style)
- **Cultural Support**: Influential works (like The Elements of Programming Style) built on this metaphor and helped programmers think of clarity and style

Disadvantages:

- **Too Simplistic**: Implies that programming is casual and unplanned, like writing a letter, instead of requiring structured design
- **One-Person Activity**: Writing is usually solitary, while software development is collaborative with many roles and responsibilities
- **Finality Misconception**: A letter is "done" once mailed, but software keeps evolving—up to 90% of effort happens after release
- **Overemphasis on Originality**: Writing values originality, but software development benefits more from reuse of existing ideas, code, and test cases
- **Trial-and-Error Bias**: Suggests expensive trial-and-error ("throw one away" drafts), which is impractical for large, costly software systems
- **Misleading Perpetuation**: Even popular books like The Mythical Man-Month reinforced this metaphor, encouraging wasteful approaches

2. Software Farming Metaphor

Advantages:

- **Natural Growth Idea**: Like farming, it conveys that software requires patience, time, and care to mature
- **Maintenance Emphasis**: Suggests continuous nurturing, cultivation, and improvement, not just one-time building
- **Predictability**: Farming implies cycles (planting, growing, harvesting), which can parallel planned release cycles in software
- **Collaboration**: Farming usually involves multiple people and roles, much like teamwork in software projects

Disadvantages:

- **Overextension Risk**: Unlike farming, software doesn't "grow by itself." Farming depends on natural processes, while software requires intentional design and coding
- **Overpromises**: Farming implies predictable yields with time, but software projects often face unexpected complexity and uncertainty
- **Limited Adaptability**: Farming cycles are fixed (seasonal), whereas software must adapt continuously and rapidly to change

- **Misleading Simplicity**: Farming suggests a steady, organic process, but software development often requires rework, restructuring, or radical redesigns

3. Software Building / Construction Metaphor

Advantages:

1. **Structured Stages**: Emphasizes planning, preparation, and execution, just like real construction (problem definition → design → construction → optimization)
2. **Scalability Awareness**: Shows that small projects (like a doghouse) need little planning, while larger projects (like a skyscraper) demand detailed design and coordination
3. **Parallels with Roles and Activities**:
   - Architects → software architects
   - Inspectors → software reviews
   - Materials → libraries, prebuilt components
   - Interior design → software optimization
4. **Highlights Cost of Mistakes**: Just as moving a wall in a house is costly, refactoring or redesigning late in a software project wastes time and effort
5. **Encourages Reuse**: Like buying prefabricated cabinets instead of building your own, developers should use libraries and frameworks instead of reinventing everything
6. **Supports Customization**: When high quality is needed, both builders and developers customize components (custom furniture ↔ custom algorithms/UI)
7. **Promotes Appropriate Planning**: Shows that some details can be decided later, but structural planning is essential to avoid collapse
8. **Explains Different Development Approaches**: Different projects (toolshed vs. medical center ↔ small app vs. safety-critical system) require different processes (lightweight vs. heavyweight)
9. **Addresses Large-Scale Projects**: Emphasizes over-engineering, safety margins, documentation, and careful scheduling—critical for massive software systems
10. **Rich Terminology**: Terms like software architecture, scaffolding, foundation classes all come naturally from this metaphor

Disadvantages:

1. **Overemphasis on Rigidity**: Buildings are fixed once constructed, but software is more flexible and can be changed after release (though at a cost)
2. **Potential for Over-Planning**: May encourage excessive documentation and bureaucracy, slowing down projects that could benefit from agility
3. **Material Cost vs. Labor Cost**: In construction, materials are costly, but in software, the main cost is people's time. The metaphor can mislead managers into undervaluing flexibility
4. **Illusion of Predictability**: Implies that projects can always be planned in detail up front, but software development often deals with changing requirements and uncertainty

5. **Not Self-Growing**: Buildings don't evolve once completed, whereas software is continuously updated, refactored, and extended
6. **Limits Creativity**: Suggests software is like assembling bricks, which may understate the creative and exploratory aspects of programming

---

Question 2: Summary of Heuristics

What are heuristics?

A heuristic is a problem-solving approach or technique that helps you find an answer, but doesn't guarantee the best or exact solution. It is different from an algorithm (which gives precise, step-by-step instructions). Instead, a heuristic guides you on where or how to look for a solution.

Understanding of heuristics

Think of heuristics as "rules of thumb" or "educated guesses." They rely on experience, intuition, or practical strategies rather than rigid logic. They're useful when exact algorithms are impossible, impractical, or too expensive.

Advantages of heuristics

- **Speed**: Faster than exact algorithms
- **Flexibility**: Useful for complex, vague, or unique problems
- **Creativity**: Encourages creativity and flexibility in problem-solving
- **Practicality**: Often "good enough" even if not perfect

Disadvantages of heuristics

- **Uncertainty**: Not always accurate (results vary)
- **Suboptimal**: May miss the optimal or best solution
- **Assumption-based**: Relies on assumptions that can be wrong
- **Bias-prone**: Can lead to biases or errors in thinking

Example Code: Heuristic Number Guessing

```
import java.util.Random;
import java.util.Scanner;

public class HeuristicGuess {

    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Random random = new Random();

        int secret = random.nextInt(100) + 1; // secret number between 1 and 100
        int guess = 50;                   // heuristic: start from the middle
        int step = 25;                    // heuristic step size

        System.out.println("Heuristic Guessing Game (1 to 100)");
        System.out.println("Secret number generated (hidden): " + secret);
        System.out.println("Computer will use binary search heuristic to find it...\n");

        int attempts = 0;
        while (true) {
            attempts++;
            System.out.println("Attempt " + attempts + " - Computer guesses: " + guess);

            if (guess == secret) {
                System.out.println("🎉 Computer found the number: " + secret + " in " + attempts + " attempts!");
                break;
            } else if (guess < secret) {
                System.out.println("   Too low! Guessing higher...");
                guess += step; // guess higher
            } else {
                System.out.println("   Too high! Guessing lower...");
                guess -= step; // guess lower
            }

            // reduce step size for better accuracy (heuristic refinement)
            step = Math.max(1, step / 2);

            // prevent infinite loops by ensuring guess stays in range
            if (guess < 1) guess = 1;
            if (guess > 100) guess = 100;

            System.out.println("   Next step size: " + step + "\n");
        }

        scanner.close();
    }
}
```

Code Explanation: This program demonstrates a heuristic approach to number guessing:

- **Starting Point**: Begins with 50 (middle of range 1-100)
- **Step Size**: Uses binary search logic, halving the step size each iteration
- **Adaptive**: Adjusts direction based on feedback (too high/too low)
- **Bounds Checking**: Ensures guesses stay within valid range
- **Not Guaranteed Optimal**: While efficient, it's a heuristic that finds a solution rather than guaranteeing the absolute best approach