Software Product Lines

Exercise 2: Runtime Variability and Design Patterns

ISF (TU Braunschweig) January 2026

1. Configuring Runtime Variability

(a) What is runtime variability?

(b) Provide examples of two real-world software systems that make use of runtime variability. Describe how they use it and how the user configures them.

(c) Imagine you are developing a music player app that offers the following configurable options: visual equalizer (enabled or disabled), streaming quality (high or low), and theme (light or dark).

i. Describe two possible methods to configure these options at runtime.

ii. Using a suitable example, explain what could happen if conflicting options are selected. Describe how you would detect and handle such conflicts at runtime.

(d) What are the pros and cons of realizing variants with runtime variability?

2. Realizing Runtime Variability

You are developing a modern navigation app used by thousands of daily commuters, tourists, and fitness enthusiasts. The app must support three primary modes of operation: driving mode, walking mode, and cycling mode. Each mode provides a different user experience, route calculation, and interface adjustment tailored to the user's current activity. Users may switch between these modes depending on their environment and preferences to receive the most accurate and relevant navigation instructions.

(a) Describe how you would implement this variability using global parameters.

(b) Describe an alternative solution using method parameters instead of global parameters.

(c) Compare the two solutions. Which one is easier to maintain and which one is more flexible for future updates? Justify your answer.

(d) Based on your preferred approach, implement the navigation logic using the scaffold below. Explain which variability approach you used (global configuration, method parameters, or design pattern) and why it is suitable in this case.

Scaffold for implementing navigation variability

```
// Enum for travel modes
enum Mode {
  DRIVING, WALKING, CYCLING
}

class Navigator {
  // Declare configuration or strategy fields if needed

  Navigator() {
    // Initialize configuration if necessary
  }

  void navigate() { // Define the correct parameters for navigate()
    // Insert logic in conditional statements
    if () {
      System.out.println("Calculating driving route...");
    } else if () {
      System.out.println("Calculating walking route...");
    } else if () {
      System.out.println("Calculating cycling route...");
    }
  }
}

class Route {
  // Simple placeholder class for route data
}

public class Main {
  public static void main(String[] args) {
    Route route = new Route();
    Navigator nav = new Navigator();
    // Call nav.navigate() with proper configuration or arguments
    nav.navigate();

  }
}
```

3. **Design Patterns for Runtime Variability**

You are designing customizable software for a coffee machine. The machine can prepare different drinks such as espresso, cappuccino, and latte. The machine can optionally add sugar or milk and can log preparation times to a file if enabled.

(a) Explain *design patterns* and their general purpose! Name and briefly explain one design pattern that can be used to implement variability.

(b) Select one design pattern that you would use to implement the variability in the coffee machine system. Justify why this design pattern is appropriate for this scenario.

(c) Write a small pseudo-code snippet or construct a class diagram demonstrating how the selected design pattern would work in the coffee machine system.

(d) Imagine the user changes the coffee recipe while the machine is already preparing the drink.

i. Explain what could go wrong if configurations are changed at runtime during preparation.

ii. Propose and explain a solution to prevent potential failures.

(e) Could you implement this coffee machine system without using a design pattern? If yes, how? If not, why not?

4. **Parameters Vs. Design Patterns**

The following Java source code fragment implements variability at runtime by using global configuration parameters to manage different audio outputs in a video game.

A method with configurable audio output in Java

```
void playSound(String sound) {
  if (Config.PLAY_ON_SPEAKER) {
    speakerSystem.play(sound);
  } else if (Config.PLAY_ON_HEADPHONES) {
    headphoneSystem.play(sound);
  } else if (Config.PLAY_ON_STREAM) {
    onlineStreamSystem.play(sound);
  }
}
```

The current implementation must be improved to satisfy the following requirements:

- The system should allow dynamically switching audio outputs during gameplay without restarting the game.
- The system should support playing sounds on multiple outputs at the same time.
- The system should allow adding new audio output types, such as VR headsets, without modifying the existing source code.

(a) What are the problems with the current implementation? List and briefly explain the issues.

(b) Which design pattern could solve the problems and why? Explain how the selected design pattern would support runtime switching of audio outputs, multiple simultaneous outputs, and the addition of new audio systems without changing existing code.

(c) Refactor the provided Java code using the proposed design pattern. Provide the refactored Java-style code that satisfies all three requirements.