Software Product Lines

Exercise 7: Language-Based Techniques

ISF (TU Braunschweig) January 2026

1. **Feature-Oriented Programming Fundamentals**

   (a) Explain the basic ideas of *feature-oriented programming* (FOP). What do the terms *feature*, *collaboration*, *role*, *class*, and *refinement* mean in the context of FOP? Illustrate your explanations with a short example.

   (b) Discuss how FOP solves the *feature-traceability problem*.

   (c) What does the *principle of uniformity* mean in FOP? Why is it important when managing multiple types of artifacts in a software-product line? Give an example where a feature spans more than one artifact type.

2. **Composition in FeatureHouse**

   Let `Game`, `Multiplayer`, and `Achievements` be three classes created using FeatureHouse.

`Game`

```
class Game {
  int level = 1;

  void start() {
    prt("Starting Game");
  }
}
```

`Multiplayer`

```
class Game {
  List<Player> players
   = new ArrayList<>();

  void addPlayer(Player p) {
    players.add(p);
  }

  void start() {
    original();
    for (Player p
     : players) {
     prt(p
         +
         " has joined.");
    }
  }
}
```

`Achievements`

```
class Game {
  int achievementPoints = 0;

  void addPlayer(Player p) {
    achievementPoints += 10;
  }

  void start() {
    original();
    achievementPoints = 0;
  }
}
```

   (a) What is the purpose of the keyword `original` in FeatureHouse? How does this differ from Java's `super` keyword?

   (b) Suppose you compose the features in the order

   [`Game`, `Multiplayer`, `Achievements`]

   using FeatureHouse, how is the composition performed? Show the final composed source code.

   (c) Does the order of feature composition matter in this example?

   (d) The code example above contains a variability bug. Can you identify and fix it?

3. **Aspect-Oriented Programming Fundamentals**

   (a) What is the core idea of *aspect-oriented programming* (AOP)? Provide a concrete example in (pseudo-)code that shows how AOP handles cross-cutting concerns differently from traditional object-oriented programming (OOP).

   (b) Define the following terms and explain how they relate to AOP: *aspect*, *pointcut*, *join point*, *advice*, *inter-type declaration*, and *quantification*.

   (c) What is the *fragile-pointcut problem* in AOP? When does it occur? Can it be avoided? How does it relate to *obliviousness*? Use an example to illustrate your explanation.

4. **AspectJ: Pointcut Matching and Advice Application**

   Consider the following class:

Handling payment operations with logging and error handling

```java
public class PaymentProcessor {
  public void processPayment(String user, double amount) {
    try {
      double newBalance = getBalance(user) - amount;
      if (newBalance < 0) {
        throw new RuntimeException("Insufficient funds");
      }
      updateBalance(user, newBalance);
      logTransaction(user, amount);
    } catch (Exception e) {
      handleError(e);
    }
  }

  public void logTransaction(String user, double amount) {
    System.out.println("Logged: $" + amount + " for " + user);
  }

  private double getBalance(String user) {
    // Simulated account balance
    return 100.0;
  }

  private void updateBalance(String user, double newBalance) {
    System.out.println("Updated balance for " + user + ": $" + newBalance);
  }

  private void handleError(Exception e) {
    System.err.println("Payment error: " + e.getMessage());
  }
}
```

   (a) Write pointcuts[1] that match the following join points:

      i. Any call to `processPayment(..)` from outside the `PaymentProcessor` class.

      ii. Any execution of the method `logTransaction()`.

      iii. The constructor call `new PaymentProcessor(..)` but only when it is triggered from a class in the `admin.*` package.

   (b) Explain the difference between `call` and `execution`. In which situations would you use one over the other?

[1]https://eclipse.dev/aspectj/doc/released/progguide/semantics-pointcuts.html

(c) Refactor the code using AspectJ so that the following behavior is moved to advice:

- Logging when payment processing ends.
- Exception handling during payment processing.

Provide both the modified class and the aspect definition.

5. **Comparison of Variability Implementation Techniques**

   Compared to previous implementation techniques, to what extent are FOP and AOP a solution for cross-cutting concerns, feature traceability, and the preplanning problem? When would you (not) resort to FOP or AOP but to a different technique?

6. **A Chart of Implementation Techniques** ABGABE <u>VERPFLICHTEND</u> FÜR ALLE

   Develop an infographic, diagram, or poster that summarizes and explains the most important relationships regarding the implementation techniques covered in this lecture. Which relationships and concepts you consider important and which presentation form you choose is up to you, as long as you can explain your graphic. You can limit yourself to the relationships discussed in the lecture and exercise, but you can also incorporate your own thoughts and ideas. For example, what conclusion you personally draw from this section of the lecture. Below you will find some suggestions that you can consider when designing your graphic (but don't have to).

   (a) possible presentation forms:

      i. Concept/Mind Map, ontology/semantic network, organizational chart, flowchart, history/timeline, pictograms/illustrations, mosaic, ...

   (b) possible relationships:

      i. *Dimensions of variability* such as binding time, tool-/language-based, annotative/compositional, static/dynamic

      ii. *Quality criteria* such as feature traceability, separation of concerns, information hiding/cohesion/encapsulation, granularity, uniformity, obliviousness

      iii. *Problems* such as code tangling/scattering, code replication/cloning, preplanning, crosscutting concerns, fragile evolution

      iv. *Scenarios* such as safety- or performance-critical systems, fine-grained extensions, adoption strategy, embedded systems, training effort, scalability

      v. and possibly more

   When in doubt, follow the Unix philosophy *do one thing and do it well*. That is, don't try to explain *all* relationships (which would probably lead to an overloaded and complex graphic), but rather some selected ones (which you then explain meaningfully and in sufficient detail and put into context). Tables are not admitted as a solution.