Software Product Lines

Exercise 6: Techniques for Modular Variability

ISF (TU Braunschweig) January 2026

1. Modular Design Principles

(a) Explain the terms *modularity*, *cohesion*, *coupling*, and *encapsulation*. How are they related?

(b) Consider the following implementation.

Change Implementation Example

```
public class Change {
  public Object change(Object obj, Operation op) {
    if (op.objectIsChangeable(obj)) {
      saveUndo(obj);
      try (FileWriter fw = new FileWriter("log.txt", true)) {
        fw.write("Changing: " + obj);
      } catch (IOException e) {
        e.printStackTrace();
      }
      return op.apply(obj);
    } else {
      try (FileWriter fw = new FileWriter("log.txt", true)) {
        fw.write("Not changeable: " + obj);
      } catch (IOException e) {
        e.printStackTrace();
      }
    }
    return obj;
  }

  private void saveUndo(Object obj) {
    try (FileWriter fw = new FileWriter("log.txt", true)) {
      try (ObjectOutputStream oos = new ObjectOutputStream(new
          FileOutputStream("backup"))) {
        oos.writeObject(obj);
      }
      fw.write("Saved backup");
    } catch (IOException e) {
      e.printStackTrace();
    }
  }
}
```

(i) Describe the code's cohesion and coupling characteristics.

(ii) Propose a modular redesign that addresses the problems identified above. In your solution, also explain why modularization is beneficial in this context. How could your design support product line variants such as `UndoSupport`?

2. Component Granularity and Library Scaling

You are developing a sensor-based embedded system that supports multiple hardware devices. Some sensors provide only temperature (in Celsius or Kelvin), others only air pressure (in Bar or Pascal), and a few support both. However, not all sensors support the same measurements or units. The current implementation combines everything into a single monolithic component:

Monolithic Sensor Component

```
public class SensorComponent {
  public double readTemperature(String unit) {
    if (unit.equals("C")) {
      return Math.random() * 100;
    } else if (unit.equals("K")) {
      return (Math.random() * 100) + 273.15;
    }
    throw new IllegalArgumentException("Unsupported temperature unit");
  }

  public double readPressure(String unit) {
    if (unit.equals("Bar")) {
      return 1.013; // default bar value
    } else if (unit.equals("Pascal")) {
      return 101300;
    }
    throw new IllegalArgumentException("Unsupported pressure unit");
  }

  public boolean supports(String capability) {
    // TEMP or PRESSURE
    return capability.equals("TEMP") || capability.equals("PRESSURE");
  }
}
```

(a) Based on this example, describe how would you design a modular sensor component suitable for a software product line. How can the design be adapted to handle different sensor capabilities and unit formats? Does *glue code* play a role?

(b) Explain the **library scaling problem**.

3. **Service Composition**

You are developing a product line for a vehicle tracking system with services like `TrackerService`, `AlertService`, and `ReportService`.

Microservices Example

```
public class TrackingCoordinator {
  private TrackerService tracker = new TrackerService();
  private AlertService alert = new AlertService();
  private ReportService reporter = new ReportService();

  public void process() {
    String loc = tracker.trackVehicle("X123");
    if (loc.equals("restricted")) {
      alert.sendAlert("X123 entered restricted zone");
    }
    reporter.log(loc);
  }
}
```

(a) Which architectural approach is used in this design? What are the trade-offs of using this approach in a software product line?

(b) What is an alternative approach for service composition used in above code? How might it affect handling feature variability in a software product line?

4. **Frameworks and Plugin Conflicts**

You are given a minimalistic plugin-based framework in Java.

Minimalistic Dummy-Framework

```
public interface Plugin {
  void start(Framework f);
  void terminate(Framework f);
}

public class Framework {
  // lots of fields here

  public void start(final List<Plugin> plugins) {
    plugins.forEach(p -> p.start(this));
    log("The framework has been set up and is now running!");

    // Doing some complex UI or network stuff
    // here and communicate with plugins.

    log("Framework is shutting down.");
    plugins.forEach(p -> p.terminate(this));
  }

  public void log(String message) { ... }
  public void setTitle(String message) { ... }
  public JButton getMainButton() { ... }
}
```

(a) Explain the term *framework* by considering additionally the terms *plugin/add-on*, *interface*, and *inversion of control*.

(b) Implement the following plugins:

- A plugin that measures how long the application ran (approximately). The plugin should log the time so that it is visible to the user.
- A plugin that changes the color of the main button to red.
- A plugin that sets the application title to `"Hello World"`.
- A plugin that changes the color of the main button to blue.

(c) Implement a dedicated `main` function that launches the framework with all your plugins. Do you observe any problems or unexpected behaviors? If so, how could they be addressed? Where does inversion of control occur in this design?

(d) How would you extend the framework to support plugin priorities, so that potential conflicts between plugins can be resolved in a controlled way? How do such conflicts reflect the **preplanning problem**?

5. **Comparison of Variability Implementation Techniques**

Compare the advantages, disadvantages, and use cases of implementation techniques discussed in the lecture so far. Use examples to support your arguments.

6. **Implementierung einer Software-Produktlinie** Abgabe verpflichtend für M.Sc.

Für Bachelorstudierende ist diese Aufgabe fakultativ (Bonus von +5 Votierungspunkten).

Das Ziel dieser Aufgabe ist es, Erfahrungen mit einer (oder mehreren) Implementierungstechnik für Software-Produktlinien zu sammeln. Deine Lösung sollte den folgenden Kriterien entsprechen. Bei Fragen gilt: Bitte melde dich so früh wie möglich.

- **Implementierungstechnik** Du kannst frei aus folgenden Implementierungstechniken wählen: Laufzeitvariabilität (z.B. Parameter oder Design Patterns), Präprozessoren, Buildsysteme, Komponenten, Services, Frameworks, Feature-orientierte Programmierung und aspektorientierte Programmierung. Wo du es für sinnvoll und technisch umsetzbar hältst, sind auch Kombinationen erlaubt. Mit Absprache sind auch andere Ansätze erlaubt. Clone-and-Own ist bei dieser Aufgabe unerwünscht – das macht ihr nach dem Studium noch genug ☺
- **Werkzeuge** Dir steht frei, welche konkreten Werkzeuge du zur Umsetzung wählst. Auch die Programmiersprache kannst du frei wählen. Wichtig ist, dass der Code am Ende ausführbar ist und "auf Knopfdruck" verschiedene Produkte generiert werden können. Gängige Werkzeugkombinationen könnten zum Beispiel sein: FeatureIDE + Java + ..., C + KConfig + KBuild oder AspectJ + Java.
- **Domäne** Du kannst die Domäne frei wählen, es soll sich aber explizit *nicht* um ein Beispiel aus der Vorlesung oder einer der Übungsaufgaben handeln (also z.B. keine Graphbibliotheken, Taschenrechner oder FeatureIDE-Beispiele). Hier ein paar Anregungen in Form von Domänen, die in früheren Jahren von Studierenden verwendet wurden: Shopping, Spiele (z.B. Text-Adventures, Tic Tac Toe, ...), Wetter, das römische Reich, Kalender, Datum/Uhrzeit, Arduino-Sensordaten, ...
- **Features** Deine Software-Produktlinie sollte ≥ 10 Features enthalten, die jeweils sowohl an- als auch abwählbar (also frei von Anomalien) sind. Nicht alle diese Features sollten vollständig optional und unabhängig sein, d.h. es sollte auch Features geben, die von anderen Features abhängen oder anderweitige Constraints erfordern. Diese Constraints sollten im Code berücksichtigt werden (z.B. durch ein Feature-Modell, falls du ein solches verwendest), so dass keine ungültigen Produkte generiert werden können. Es sollte aber auch nicht so viele Constraints geben, dass nur sehr wenige Produkte generiert werden können.
- **Code** Deine Lösung sollte ≥ 1000 selbst geschriebene *source lines of code* (SLOC) enthalten. Ob der Code explizit für diese Lehrveranstaltung geschrieben wurde oder ob du auf bestehenden Code zurückgreifst, ist unerheblich. Wichtig ist, dass es *dein* Code ist (u.a. kein KI-generierter Code, kein fremder Code aus dem Internet oder von Kommiliton*innen – unter Vorbehalt einer Nachprüfung). Am einfachsten ist es, ein komplett neues Projekt zu beginnen. Es bietet sich aber auch an, Code aus früheren Lehrveranstaltungen zu "Produktlinien-fizieren", also nach den in der Lehrveranstaltung dargestellten Prinzipien konfigurierbar zu machen.
- **Abgabe und Vorstellung** Der Code kann als Anhang per Mail oder als Weblink abgeschickt werden. In jedem Fall ist Git zur Versionsverwaltung zu verwenden und der `.git`-Ordner mit abzugeben. Bitte vermeide, nur einen Commit zu erzeugen, damit die Autorenschaft besser nachvollziehbar ist. Bereite dich außerdem darauf vor, deine Lösung kurz in der Übung vorzustellen (d.h., den Code ausführen, Produkte konfigurieren, und deine Vorgehensweise erläutern). Sowohl Abgabe als auch Vorstellung sind zur erfolgreichen Bearbeitung der Aufgabe notwendig.