Software Engineering

WS 2022/23, Sheet 04

Prof. Dr. Sven Apel
Yannick Lehmen, Maurice Vincon

Handout: 21.11.2022

Task 1

a) Define the terms *Code Scattering* and *Code Tangling.*

b) Calculate the scattering degree SD for the features *Color* and *Encryption* from the code on the bottom of the page. Then calculate the tangling degree TD of the modules containing those features. Note that every line of code that is only executed if `Conf.ENCRYPTION = true` belongs to the feature *Encryption.* Every line of code that is only executed if `Conf.COLORED = true` belongs to the feature *Color.* The variable `Conf.COLOR` describes the used color if the feature *Color* is activated.

c) Discuss some problems of crosscutting features.

```java
import java.util.ArrayList;
import java.util.List;

public class ChatApplication {
  Client client;
  List<Message> messages =
      new ArrayList<>();

  void newMessage(String content) {
    Message message = new Message(content);
    if (Conf.ENCRYPTION) {
      message.encryptMessage();
    }
    messages.add(message);

  }

  void createClient() {
    if (Conf.COLORED) {
      client = new Client(Conf.COLOR);
    } else {
      client = new Client();
    }
  }
}
```

```java
public class Client {
  Color color;

  public Client(Color color) {
    this.color = color;
  }

  public Client() {
  }

  void displayMessage(Message message) {
    String messageToDisplay =
        message.getContent();
    if (Conf.ENCRYPTION) {
      messageToDisplay =
          decryptMessage(messageToDisplay);
    }
    System.out.println(messageToDisplay);
  }

  String decryptMessage(String message) {
    return new StringBuilder(message)
        .reverse().toString();
  }
}
```

```java
public class Message {
  String content;

  public Message(String content) {
    this.content = content;
  }

  void encryptMessage() {
    content = new StringBuilder(content)
        .reverse().toString();
  }

  public String getContent() {
    return content;
  }

  @Override
  public String toString() {
    return "Message{" +
        "content='" + content + '\'' +
        '}';
  }
}
```

```java
public class Conf {
  public static boolean ENCRYPTION = true;
  public static boolean COLORED = true;
  public static Color COLOR = Color.BLACK;
}
```

Solution

a) *Code Scattering* defines how distributed code belonging to one feature is across multiple modules. *Code Tangling* occurs when code belonging to different features is present in one module.

b) SD(*Color*) = 3, SD(*Encryption*) = 4
TD(ChatApplication) = 2, TD(Client) = 2, TD(Message) = 1, TD(Conf) = 2

c) Problems:

- Features *vanish* in the code base (What code belongs to a feature?)
- Dividing work becomes more difficult because developers working on different features modify the same code.
- It becomes more difficult to evolve and extend the program since adding new features requires dealing with code of many other features.
- These factors make the development process more difficult and less productive.

Task 2

a) Why is inheritance not a good way to implement variability?

b) What is the problem with using multiple inheritance as a solution?

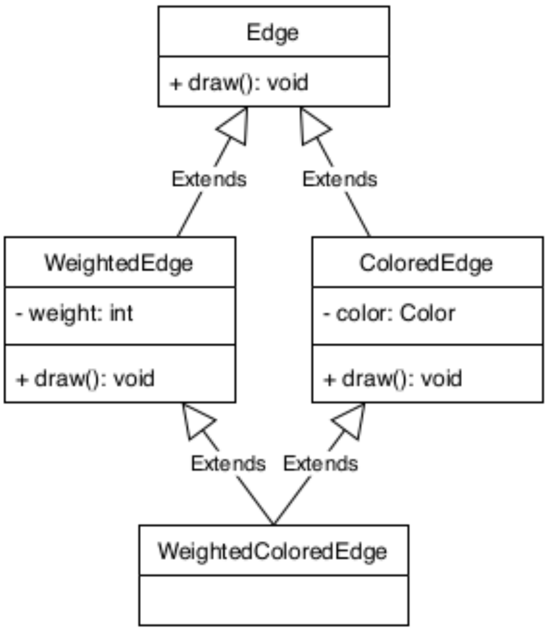
c) Is delegation better suited for implementing variability than inheritance?

d) **Discussion:** Prepare arguments for and against implementing variability with design patterns. You can present your arguments during the discussion in the tutorial sessions.

Solution

a) Inheritance is inflexible in the sense that different features cannot be combinable and optional at the same time. Trying to do so leads to a combinatorial explosion since *all possible feature combinations* must be implemented separately. This makes inheritance a non-optimal way of implementing variability in most cases.

b) The problem with multiple inheritance is that there can be ambiguity about which parent class a particular variable or method is inherited from. This is illustrated by the diamond problem: The class `WeightedColoredEdge` inherits from both `WeightedEdge` and `ColoredEdge` which in turn have the same superclass `Edge`. The superclass `Edge` has a function `draw` which is overridden by both `WeightedEdge` and `ColoredEdge`. The question is now: Which implementation of `draw` should the class `WeightedColoredEdge` use?



c) Delegation has the advantage that it solves the flexibility issue of inheritance. With delegation, features can be composed dynamically. However, delegation requires features to be completely independent from each other meaning that there cannot be special functionality for a special combination of features. Furthermore, delegation does not work with certain language mechanisms, e.g., it is not possible to add new methods and it is also not possible to utilize late binding the same way as with inheritance. Delegation also leads to *object schizophrenia* which means that one conceptual object is composed of many technical objects. This can negatively affect maintainability. Another effect of this is that delegation introduces many indirections to the program's control flow which may negatively influence performance.

Task 3

a) **Discussion:** Prepare arguments for and against implementing variability with frameworks. You can present your arguments during the discussion in the tutorial sessions.

b) Describe the difference between *blackbox* and *whitebox* frameworks.

c) In this context, explain the preplanning problem.

Solution

b) Whitebox frameworks use subclassing for customization, which means that the plugin (extension) needs to know the code of the superclass to extend and overwrite the functionality. Moreover, the plugin and the base code are compiled together. Blackbox frameworks, on the other hand, can only see and extend functionality exposed in the framework's public interface. This makes it more modular and it allows using more than one extension at a time. The extension code is also deployed separately from the framework.

c) The preplanning problem describes the difficulty of anticipating where somebody would like to extend a program in the future. This is a hard task which requires good analysis of possible extensions and plugins. However, good preplanning is important since adding extension points for plugins later on is more difficult than including them from the start.

Task 4

a) Describe the expression problem in your own words.

b) Discuss pros and cons of using data-centric or operation-centric modularity.

c) What is the dilemma that arises from the expression problem. In that context, also explain the *tyranny of the dominant decomposition.*

Solution

a) The expression problem describes the problem of how to extend recursive data structure with new operations and new data types while considering some additional side constraints. These constraints state that we do not want to change the existing code and also avoid trivial code replication. Furthermore, we want to extend the data structure multiple times in arbitrary order and the extensions should be independent from each other (they can be compiled in isolation).

b)
- data-centric:
  + it is easy to add new data types to the existing code
  - we have to change all existing classes to add new operations
- operation-centric:
  + it is easy to add new operations without changing anything
  - we have to change a lot of existing code to add new data types

To summarize: For data-centric code, it is easy to add new data types but hard to add new operations. For operation-centric code, it is easy to add new operations but hard to add new data types. However, because of our side constraints, we need a way that allows to easily add new data types *and* new operations. Both, the data-centric and the operation-centric approaches fail to do so.

c) It is not always possible to write code such that all features are modular. The reason is that some features are in a sense *orthogonal* to each other. This means that, regardless of which feature you pick to be your dominant dimension, there will always be some kind of dependencies in the other dimensions. Hence, these features will exhibit some degree of scattering and tangling because runtime variability and object-oriented mechanisms are insufficient to modularize these orthogonal features. This problem is called the tyranny of the dominant decomposition.