

# Software Engineering

WS 2021/22, Sheet 07



Prof. Dr. Sven Apel  
Yannick Lehmen, Maurice Vincon

**Handout:** 12.12.2022

## Task 1

- a) What is the *Feature Traceability Problem*?
- b) What consequences are there if a software project suffers from the *Feature Traceability Problem* problem?

## Solution

- a) The *Feature Traceability Problem* describes the difficulty of locating all implementation artifacts of a feature in the source code.
- b) Consequences of suffering from the *Feature Traceability Problem*:
  - Features vanish in the code base → difficult maintenance
  - Difficult division of work
  - Less productivity
  - Difficult evolution

## Task 2

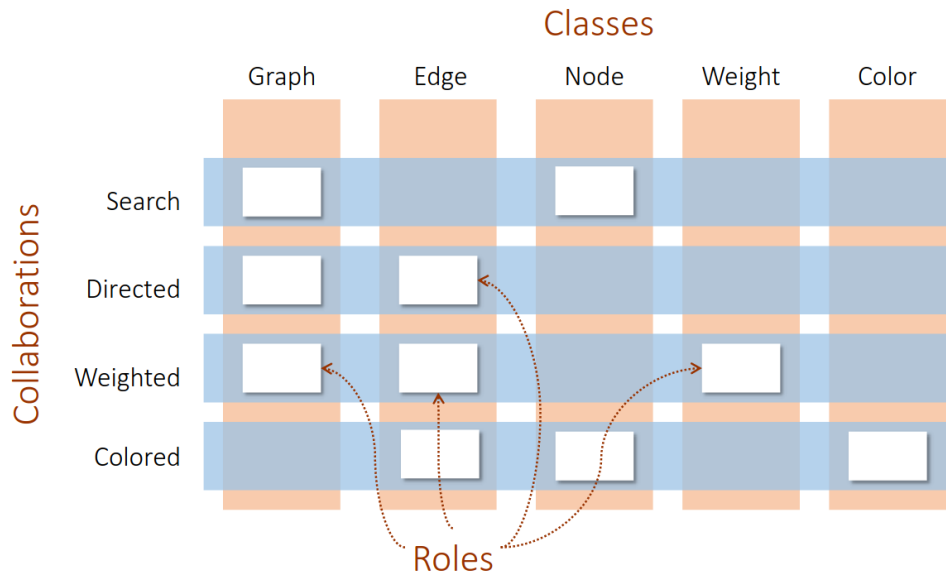
- Define the terms *Class*, *Collaboration*, and *Role*. Visualize the relationship between these concepts in a diagram.
- Do *Collaborations* and *Roles* solve the *Feature Traceability Problem*? Explain why or why not.
- Do *Collaborations* and *Roles* solve the *Preplanning Problem*? Explain why or why not.
- Can you implement crosscutting concerns using *Collaborations* and *Roles*? Explain why or why not.

### Solution

- Class** An abstract model for a group of similar objects.

**Collaboration** Classes or their parts that cooperate to implement the functionality of a feature.

**Role** A description of the responsibilities of a class in a collaboration (implemented, e.g., by a refinement).



- Collaborations* and *Roles* solve the *Feature Traceability Problem*, because each feature is implemented by one collaboration.
- Collaborations* and *Roles* solve the *Preplanning Problem*, because new collaborations can be easily added to an existing system.\*
- You can implement crosscutting concerns with *Collaborations* and *Roles*, because you can have multiple roles in different classes.\*

\* FOP implementation techniques (like superimposition or merging) have limitations. So there can be features that cannot be implemented easily this way.

## Task 3

- Combine the following two roles via *Superimposition*. In this example, we use *FeatureHouse* syntax, meaning that `original()` refers to the original function.
- Implement the same program in SCALA using *traits and mixin composition*.

Combine the following two roles via *Superimposition*. In this example, we use *FeatureHouse* syntax, meaning that `original()` refers to the original function.

```
1 // Base class
2 class ChatClient {
3     ClientConnection connection;
4
5     void sendMessage(String message) {
6         connection.sendMessage(message);
7     }
8
9     void receiveMessage() {
10        String message =
11        ↪ connection.receiveMessage();
12        System.out.println(message);
13    }
```

```
1 // Class refinement
2 class ChatClient {
3     EncryptionModule encryptor;
4
5     void sendMessage(String message) {
6         String encryptedMessage =
7         ↪ encryptor.encrypt(message);
8         original(encryptedMessage);
9     }
10
11    void receiveMessage() {
12        String message =
13        ↪ connection.receiveMessage();
14        String decryptedMessage =
15        ↪ encryptor.decrypt(message);
16        System.out.println(decryptedMessage);
17    }
18 }
```

## Solution

FeatureHouse output:

```
1 // After superimposition
2 class ChatClient {
3     ClientConnection connection;
4     // add member from refinement
5     EncryptionModule encryptor;
6
7     // rename and keep original method
8     void sendMessage_original(String message) {
9         connection.sendMessage(message);
10    }
11
12    void sendMessage(String message) {
13        String encryptedMessage = encryptor.encrypt(message);
14        // call renamed original function
15        sendMessage_original(encryptedMessage);
16    }
17
18    // original method overridden
19    void receiveMessage() {
20        String message = connection.receiveMessage();
21        String decryptedMessage = encryptor.decrypt(message);
22        System.out.println(decryptedMessage);
23    }
24 }
```

Scala implementation using traits:

```
1 trait ChatClient {
2   val connection: ClientConnection
3
4   def sendMessage(message: String): Unit
5   def receiveMessage(): Unit
6 }
7
8 class ChatClientImpl extends ChatClient {
9   val connection = new ClientConnection();
10
11   def sendMessage(message: String): Unit = {
12     connection.sendMessage(message)
13   }
14
15   def receiveMessage(): Unit = {
16     println(connection.receiveMessage());
17   }
18 }
19
20 trait Encryption extends ChatClient {
21   private val encryptor = new EncryptionModule()
22
23   abstract override def sendMessage(message: String): Unit = {
24     super.sendMessage(encryptor.encrypt(message))
25   }
26
27   abstract override def receiveMessage(): Unit = {
28     println(encryptor.decrypt(super.connection.receiveMessage()))
29   }
30 }
```

## Task 4

- a) Define the terms *Advice*, *Aspect*, *Join Point*, and *Pointcut*.
- b) Do aspects solve the *Feature Traceability Problem*? Explain why or why not.
- c) Do aspects solve the *Preplanning Problem*? Explain why or why not.
- d) Can you implement crosscutting concerns using aspects? Explain why or why not.
- e) What problem arises from the use of pointcuts?

## Solution

- a) **Aspect** Encapsulates the implementation of a crosscutting concern or feature.  
**Join Point** An event during the execution of a program; Aspects can interfere here.  
**Pointcut** A declarative specification of a set of *Join Points* (predicate).  
**Advice** A piece of code that is executed when a *Join Point* is selected by a *Pointcut*.
- b) Aspects solve the *Feature Traceability Problem*, because each feature is implemented by one aspect.
- c) Aspects solve the *Preplanning Problem*, because new advice can be inserted very flexibly using pointcuts.
- d) You can implement crosscutting concerns with aspects, because pointcuts can select join points in different locations across the whole program.
- e) Pointcuts suffer from the *Fragile Pointcut Problem*: Changes to the base code can inadvertently add or remove join points from a pointcut.

## Task 5

- a) Mark all join points in the first code example.
- b) List all join points from the first code example that are selected by the following pointcuts:
- `call(void Locker.lock(int))`
  - `call(* Locker.*( *))`
  - `get(* Foo.locker)`
  - `execution(void Foo.log(String))`
  - `execution(void Foo.*(String))`
  - `execution(void Foo.*(..))`
- c) Write an aspect for the code in Example 2, so that it has the same functionality as Example 1.

```
1 // Example 1
2 public class Foo {
3     private Locker locker = new Locker();
4
5     void main(String parameter) {
6         int lockId = locker.lock();
7         try {
8             int i = parameter.length();
9             firstOperation(i);
10            log("First Operation executed with parameter " + i);
11            secondOperation();
12        } finally {
13            locker.unlock(lockId);
14        }
15    }
16 }
```

```
1 // Example 2
2 public class Foo {
3     void main(String parameter) {
4         int i = parameter.length();
5         firstOperation(i);
6         secondOperation();
7     }
8 }
```

## Solution

- a)
- line 3: constructor call
  - line 3: field access (set locker)
  - line 5: method execution
  - line 6: field access (get locker)
  - line 6: method call
  - line 8: method call
  - line 9: method call
  - line 10: method call
  - line 11: method call
  - line 13: field access (get locker)

- line 13: method call

See <https://www.eclipse.org/aspectj/doc/next/progguide/semantics-joinPoints.html> for reference.

b)

- –
- line 13: method call
- line 6: field access (get locker), line 13: field access (get locker)
- –
- line 5: method execution
- line 5: method execution

c)

```

1 public aspect Advice {
2     private Locker locker = new Locker();
3
4     void around(String parameter): execution(void Foo.main(String)) && args(parameter) {
5         int lockId = locker.lock();
6         try {
7             proceed(parameter);
8         } finally {
9             locker.unlock(lockId);
10        }
11    }
12
13    after(Foo foo, int i): call(void Foo.firstOperation(int)) && args(i) && this(foo) {
14        foo.log("firstOperation executed with parameter " + i);
15    }
16 }

```