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

a) Define the terms *Class*, *Collaboration*, and *Role*. Visualize the relationship between these concepts in a diagram.

b) Do *Collaborations* and *Roles* solve the *Feature Traceability Problem*? Explain why or why not.

c) Do *Collaborations* and *Roles* solve the *Preplanning Problem*? Explain why or why not.

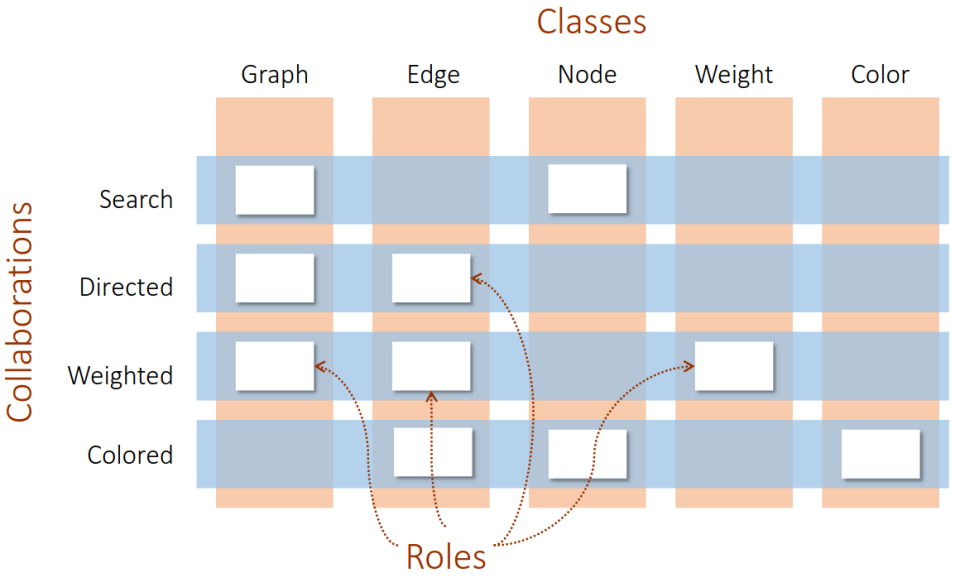
d) Can you implement crosscutting concerns using *Collaborations* and *Roles*? Explain why or why not.

Solution

a) **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).



b) *Collaborations* and *Roles* solve the *Feature Traceability Problem*, because each feature is implemented by one collaboration.

c) *Collaborations* and *Roles* solve the *Preplanning Problem*, because new collaborations can be easily added to an existing system.*

d) 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

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

b) 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.

```
// Base class
class ChatClient {
  ClientConnection connection;

  void sendMessage(String message) {
    connection.sendMessage(message);
  }

  void receiveMessage() {
    String message =
  ↪ connection.receiveMessage();
    System.out.println(message);
  }
}
```

```
// Class refinement
class ChatClient {
  EncryptionModule encryptor;

  void sendMessage(String message) {
    String encryptedMessage =
  ↪ encryptor.encrypt(message);
    original(encryptedMessage);
  }

  void receiveMessage() {
    String message =
  ↪ connection.receiveMessage();
    String decryptedMessage =
  ↪ encryptor.decrypt(message);
    System.out.println(decryptedMessage);
  }
}
```

Solution

FeatureHouse output:

```
// After superimposition
class ChatClient {
  ClientConnection connection;
  // add member from refinement
  EncryptionModule encryptor;

  // rename and keep original method
  void sendMessage_original(String message) {
    connection.sendMessage(message);
  }

  void sendMessage(String message) {
    String encryptedMessage = encryptor.encrypt(message);
    // call renamed original function
    sendMessage_original(encryptedMessage);
  }

  // original method overridden
  void receiveMessage() {
    String message = connection.receiveMessage();
    String decryptedMessage = encryptor.decrypt(message);
    System.out.println(decryptedMessage);
  }
}
```

Scala implementation using traits:

```
trait ChatClient {
  val connection: ClientConnection

  def sendMessage(message: String): Unit
  def receiveMessage(): Unit
}

class ChatClientImpl extends ChatClient {
  val connection = new ClientConnection();

  def sendMessage(message: String): Unit = {
    connection.sendMessage(message)
  }

  def receiveMessage(): Unit = {
    println(connection.receiveMessage());
  }
}

trait Encryption extends ChatClient {
  private val encryptor = new EncryptionModule()

  abstract override def sendMessage(message: String): Unit = {
    super.sendMessage(encryptor.encrypt(message))
  }

  abstract override def receiveMessage(): Unit = {
    println(encryptor.decrypt(super.connection.receiveMessage()))
  }
}
```

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)

```
public aspect Advice {
  private Locker locker = new Locker();

  void around(String parameter): execution(void Foo.main(String)) && args(parameter) {
    int lockId = locker.lock();
    try {
      proceed(parameter);
    } finally {
      locker.unlock(lockId);
    }
  }

  after(Foo foo, int i): call(void Foo.firstOperation(int)) && args(i) && this(foo) {
    foo.log("firstOperation executed with parameter " + i);
  }
}
```