Automated Change Propagation from Source Code to Sequence Diagrams

Test Cases Description

1 Test Cases

We evaluated the proposed architecture via sixteen test cases, that was organized in three test sets based on level of their complexity:

- Evaluation of basic functionalities
  - TC01: Adding a synchronous message
  - TC02: Adding a synchronous message and a lifeline
  - TC03: Removing a synchronous message
  - TC04: Removing a synchronous message and a lifeline
  - TC05: Adding a combined fragment opt
  - TC06: Removing a combined fragment opt
- Evaluation of synchronization rules
  - TC07: Filtration of system calls
  - TC08: Restriction of lifelines count
  - TC09: Filtration of get/set calls
  - TC10: Filtration of external calls
  - TC11: Filtration of combined fragments
- Evaluation of propagation of complex changes
  - TC12: Replacing two messages with one new message, which contains internally six new calls
  - TC13: Condition change and movement of existing calls to new operation
  - TC14: Part of the functionality has been moved to new operation
  - TC15: Removing a sequence diagram implementation from the source code
  - TC16: Adding a loop over a condition and adding a new synchronous call into the condition

2 Evaluation of Basic Functionalities

Fig. 1 shows sequence diagram used in the first set of test cases. The source code implementing the sequence diagram is listed by List. 1.

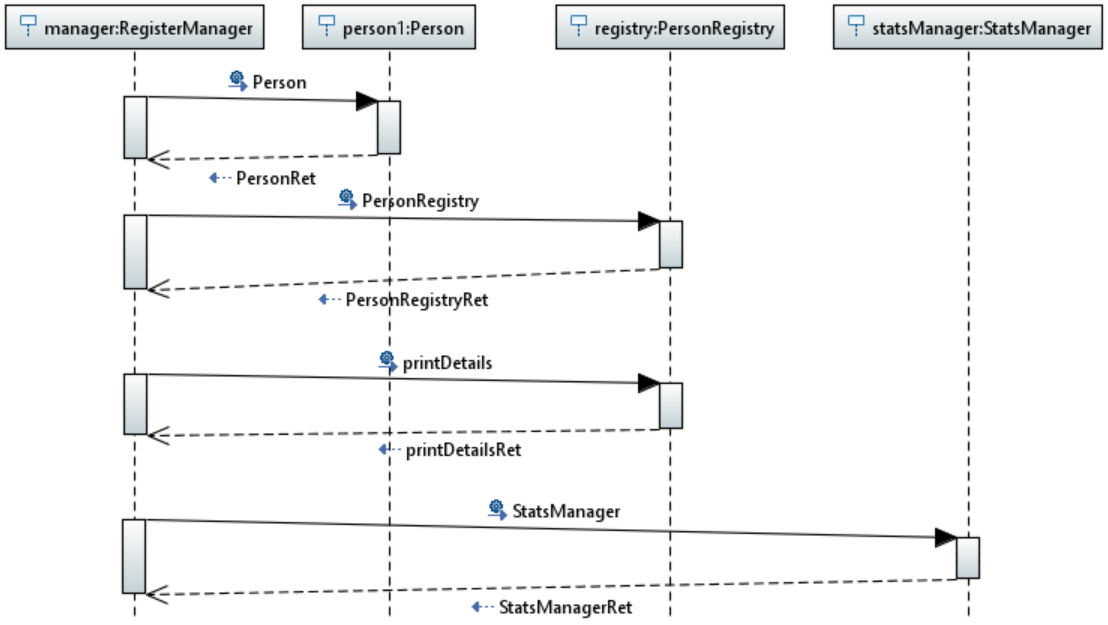

Fig. 1. Sequence diagram used in test cases from the test set Evaluation of basic functionalities

```
public class RegisterManager {
    public void createRegistry() {
        Person person1 = new Person("Andrej","Mlyncar", null);
        PersonRegistry registry = new PersonRegistry();
        registry.printDetails();
        StatsManager statsManager = new StatsManager(registry);
    }
}

public class PersonRegistry {
    private final List<Person> persons = new ArrayList<>();
    public int getPersonSize() {
        return persons.size();
    }
    public void addPerson(Person person) {
        persons.add(person);
    }
    public void printDetails() {
        System.out.println(persons.toString());
    }
}

public class StatsManager {
    private final PersonRegistry registry;
    private Integer stats;
    public StatsManager(PersonRegistry registry) {
        this.registry = registry;
    }
    public void computeStats() {
        if (stats == null) {
            extractsStats();
        }
    }
    private void extractsStats() {
        this.stats = registry.getPersonSize();
    }
}
```

List. 1. Source code for the sequence diagram in Fig. 1

2.1 TC01: Adding a synchronous message

Description: Call *addPerson* is added to the method *createRegistry*.

```
    public void createRegistry() {
        Person person1 = new Person("Andrej","Mlyncar", null);
        PersonRegistry registry = new PersonRegistry();
        registry.addPerson(person1);
        registry.printDetails();
        StatsManager statsManager = new StatsManager(registry);
    }
```

List. 2. Edited source code of TC01

Result: Message *addPerson* has been added into the sequence diagram (see Fig. 1). Synchronization log is shown below (note: message *add* has been filtered out by system calls filter).

```
Sun Apr 23 20:23:04 CEST 2017: message_add = After:PersonRegistry;
addPerson
Sun Apr 23 20:23:04 CEST 2017: message_add = After:add; addPersonRet
```

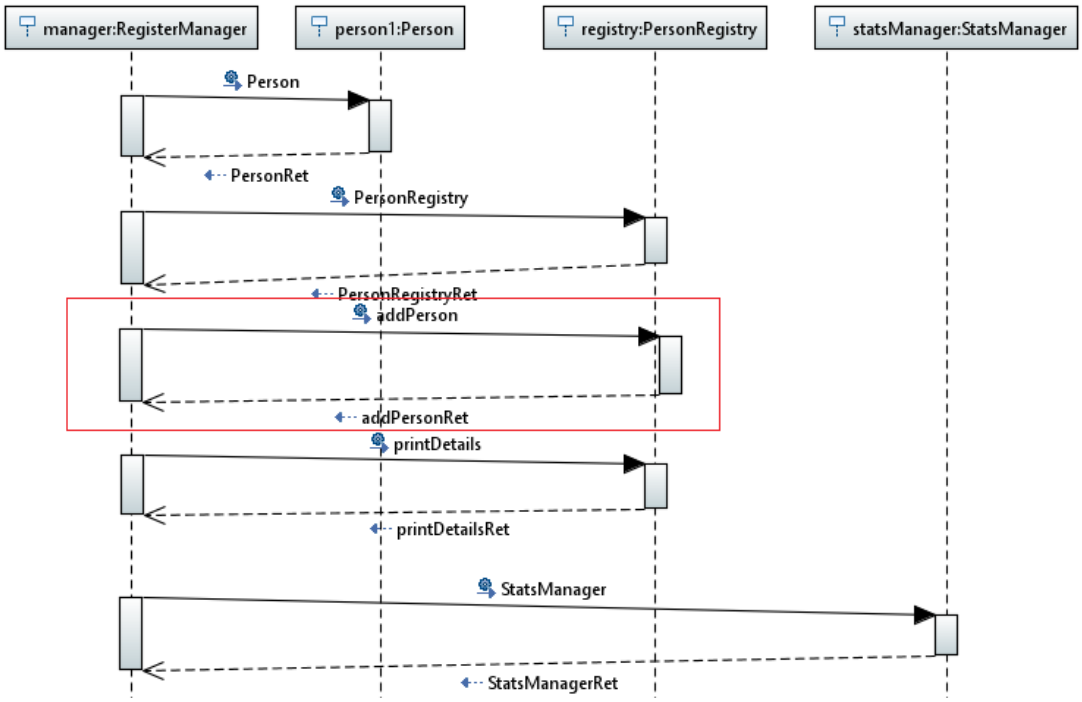

Fig. 2. Modified sequence diagram, after execution of TC01

2.2 TC02: Adding a synchronous message and a lifeline

Description: Call constructing object *Address* is added to the method *createRegistry*.

```
public void createRegistry() {
    Person person1 = new Person("Andrej","Mlyncar", null);
    PersonRegistry registry = new PersonRegistry();
    Address address1 = new Address("Bratislava","Kresankova");
    registry.printDetails();
    StatsManager statsManager = new StatsManager(registry);
}
```

List. 3. Edited source code of TC02

Result: Create message and new the *Address* object lifeline have been added to the sequence diagram (see Fig. 3). Synchronization log is shown below.

```
Sun Apr 23 20:35:27 CEST 2017: lifeline_add = address1:Address
Sun Apr 23 20:35:27 CEST 2017: message_add = After:PersonRegistry; Ad-
dress
Sun Apr 23 20:35:27 CEST 2017: message_add = AddressRet
```

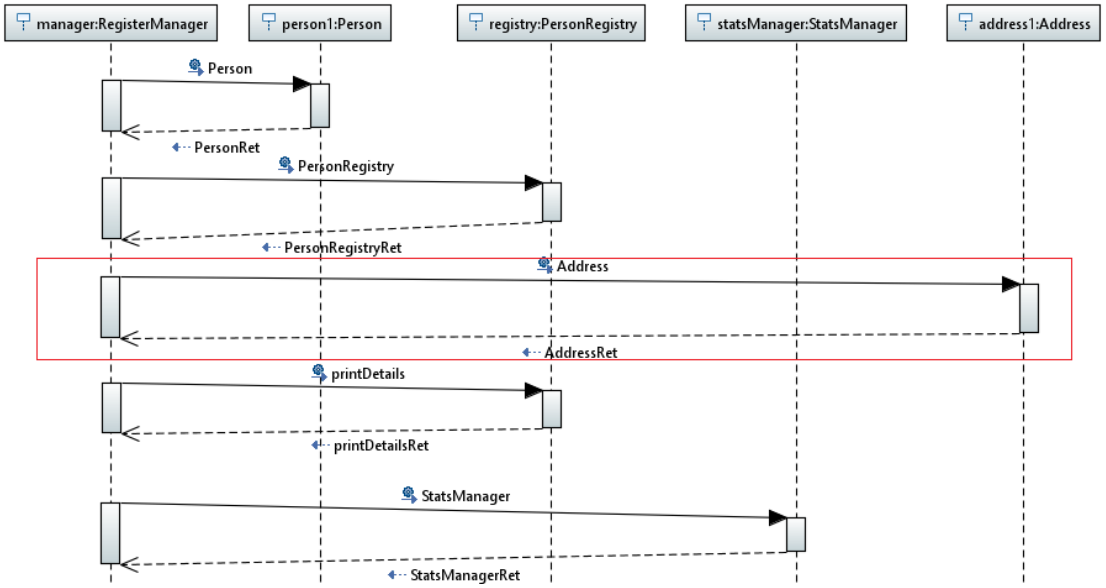

Fig. 3. Modified sequence diagram, after execution of TC02

2.3 TC03: Removing a synchronous message

Description: Call *printDetails* is removed from the method *callRegistry*.

```
public void createRegistry() {
    Person person1 = new Person("Andrej","Mlyncar", null);
    PersonRegistry registry = new PersonRegistry();
    registry.printDetails();
    StatsManager statsManager = new StatsManager(registry);
}
```

List. 4. Edited source code of TC03

Result: The message *printDetails* has been removed from the sequence diagram (see Fig. 4). Synchronization log is shown below.

```
Sun Apr 23 20:37:47 CEST 2017: message_remove = printDetailsRet
Sun Apr 23 20:37:47 CEST 2017: message_remove = After:PersonRegistry;
printDetails
```

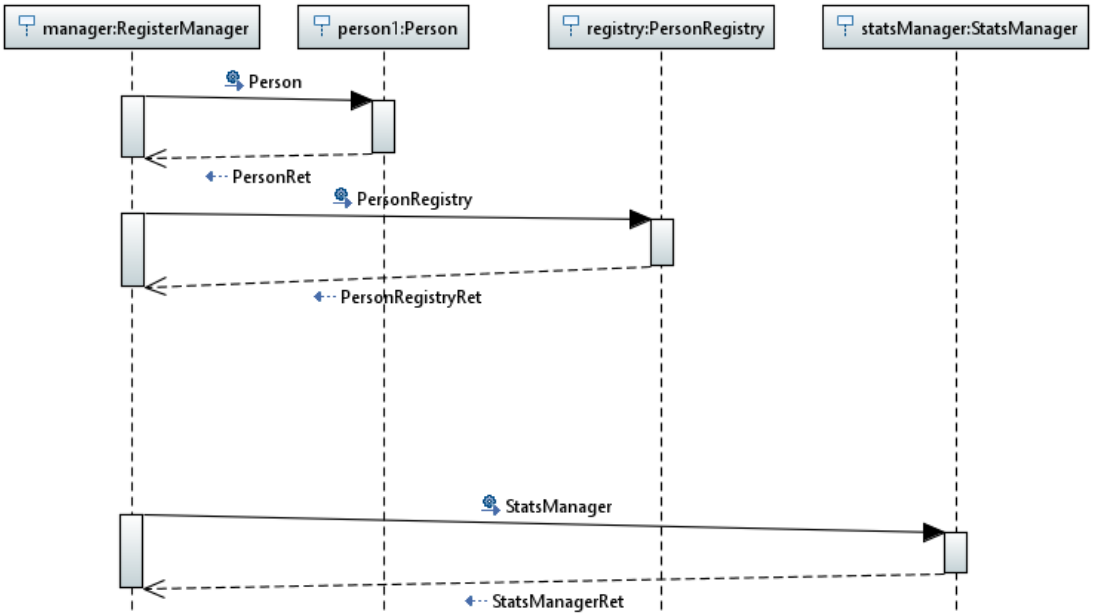

Fig. 4. Modified sequence diagram, after execution of TC03

2.4 TC04: Removing a synchronous message and a lifeline

Description: Call constructing object *StatsManager* has been added to the method *createRegistry*.

```
public void createRegistry() {
    Person person1 = new Person("Andrej","Mlyncar", null);
    PersonRegistry registry = new PersonRegistry();
    registry.printDetails();
    StatsManager statsManager = new StatsManager(registry);
}
```

List. 5. Edited source code of TC04

Result: Create message and the *StatsManager* object lifeline have been removed from the sequence diagram (see Fig. 5). Synchronization log is shown below.

```
Sun Apr 23 20:39:25 CEST 2017: message_remove = StatsManagerRet
Sun Apr 23 20:39:25 CEST 2017: message_remove = After:printDetails;
StatsManager
Sun Apr 23 20:39:25 CEST 2017: lifeline_remove = statsManag-
er:StatsManager
```

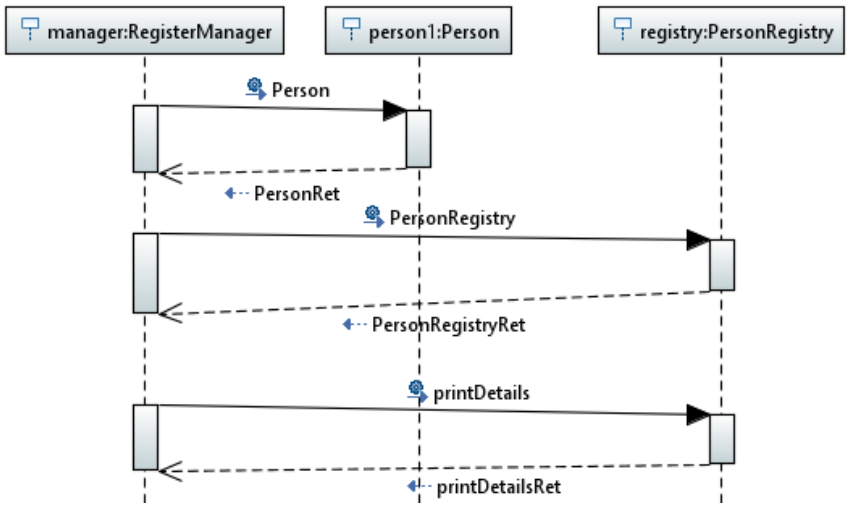

Fig. 5. Modified sequence diagram, after execution of TC04

2.5 TC05: Adding a combined fragment opt

Description: If statement is added above call *printDetails* to the method *createRegistry*

```
public void createRegistry() {
    Person person1 = new Person("Andrej","Mlyncar", null);
    PersonRegistry registry = new PersonRegistry();
    if(registry!=null)
       registry.printDetails();
    StatsManager statsManager = new StatsManager(registry);
}
```

List. 6. Edited source code of TC05

Result: Combined fragment opt has been added around the message *printDetails* to the method *createRegistry* (see Fig. 6). Synchronization log is shown below.

```
Sun Apr 23 20:42:16 CEST 2017: fragment_add = opt:registry!=null; message: printDetails
```

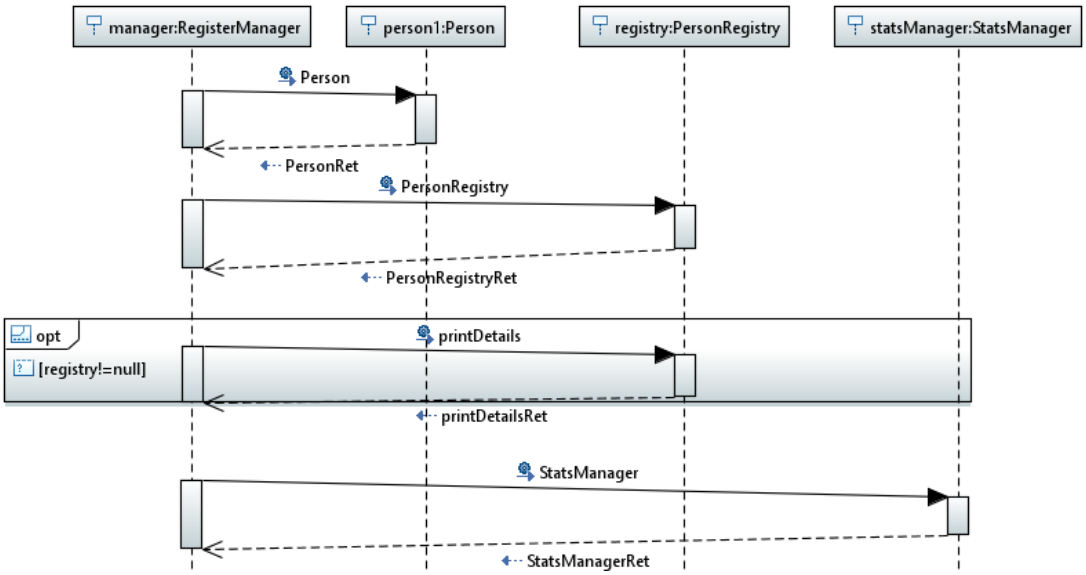

Fig. 6. Modified sequence diagram, after execution of TC05

2.6 TC06: Removing a combined fragment opt

Description: If statement is removed from the method *createRegistry* (note: source code of TC05 is used)
Result: The sequence diagram of the TC05 has been corrected to the original state. Synchronization log is shown below.

```
Sun Apr 23 20:45:27 CEST 2017: fragment_remove = opt:registry!=null;
message:printDetails
```

3 Evaluation of synchronization rules

Fig. 7 shows sequence diagram used in the second set of test cases. The source code implementing the sequence diagram is listed by List. 7.

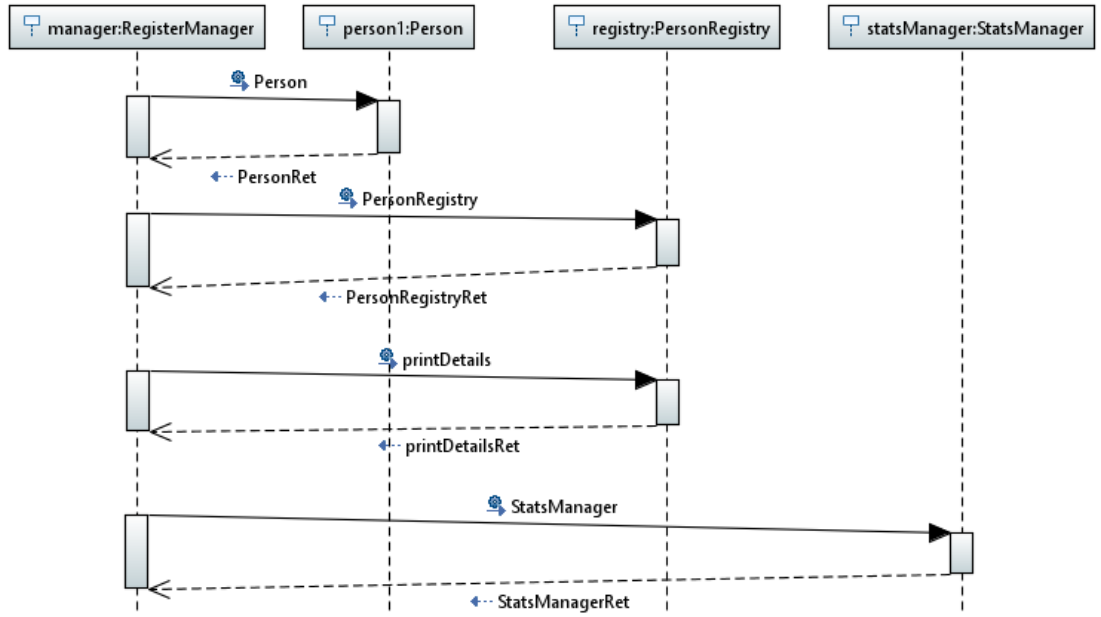

Fig. 7. Sequence diagram used in test cases from the test set Evaluation of synchronization rules

```
public class RegisterManager {
    public void createRegistry() {
        Person person1 = new Person("Andrej","Mlyncar", null);
        PersonRegistry registry = new PersonRegistry();
        registry.printDetails();
        StatsManager statsManager = new StatsManager(registry);
    }
}

public class PersonRegistry {
    private final List<Person> persons = new ArrayList<>();
    public int getPersonSize() {
        return persons.size();
    }
    public void addPerson(Person person) {
        persons.add(person);
    }
    public void printDetails() {
        System.out.println(persons.toString());
    }
}

public class StatsManager {
    private final PersonRegistry registry;
    private Integer stats;
    public StatsManager(PersonRegistry registry) {
        this.registry = registry;
    }
    public void computeStats() {
        if (stats == null) {
            extractsStats();
```

```
        }
    }
    private void extractsStats() {
        this.stats = registry.getPersonSize();
    }
}
```

List. 7. Source code for the sequence diagram in Fig. 7

3.1 TC07: Filtration of system calls

Description: Call *addPerson* is added to the method *createRegistry*. Test case has been executed with and without the rule for filtration of system calls.

```
    public void createRegistry() {
        Person person1 = new Person("Andrej","Mlyncar", null);
        PersonRegistry registry = new PersonRegistry();
        registry.addPerson(person1);
        registry.printDetails();
        StatsManager statsManager = new StatsManager(registry);
    }
```

List. 8. Edited source code of TC07

Result: In case when the rule was enabled, only message *addPerson* has been added to the sequence diagram (see Fig. 8). When the rule was disabled, messages of Java libraries have been added to the sequence diagram (see Fig. 9).

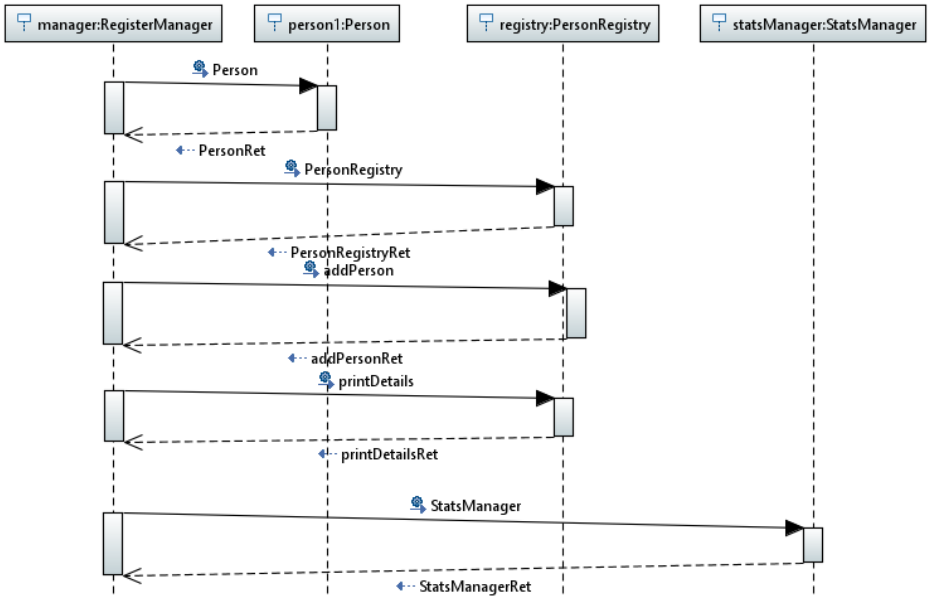

Fig. 8. Modified sequence diagram, after execution of TC07 with the rule for filtration of system calls

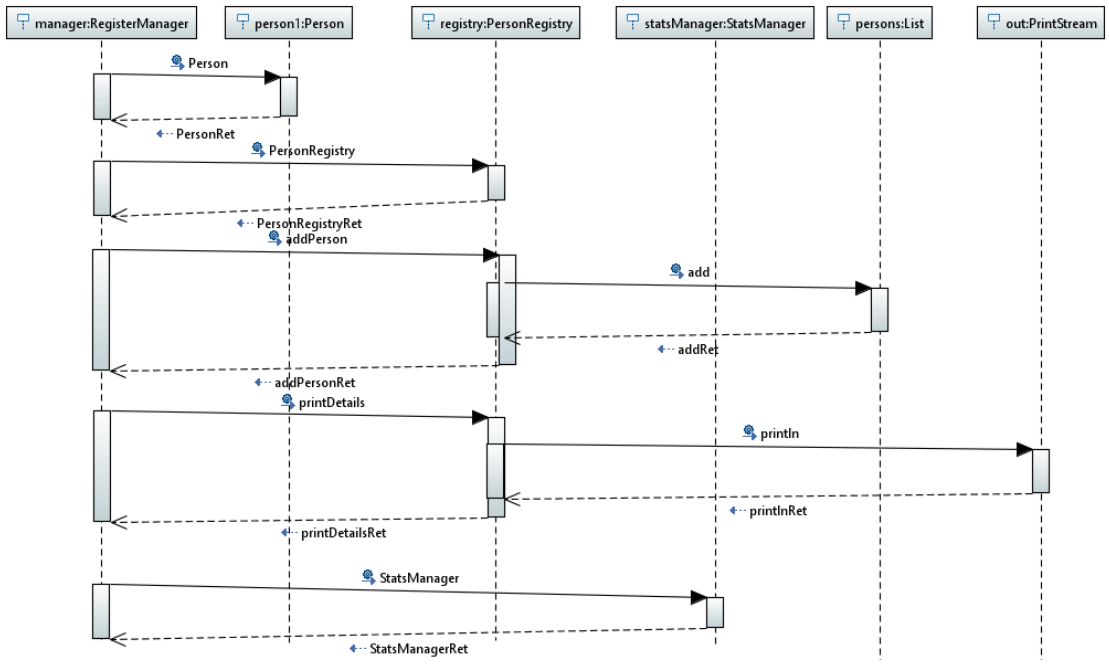

Fig. 9. Modified sequence diagram, after execution of TC07 without the rule for filtration of system calls

3.2 TC08: Restriction of lifelines count

Description: Call constructing object *Address* is been added to the method *createRegistry* and the restriction of lifelines count is set to four. The source code modification will lead to addition of the fifth lifeline for the object *Address* to the sequence diagram.

```
public void createRegistry() {
    Person person1 = new Person("Andrej","Mlyncar", null);
    PersonRegistry registry = new PersonRegistry();
    Address address1 = new Address("Bratislava","Kresankova");
    registry.printDetails();
    StatsManager statsManager = new StatsManager(registry);
}
```

List. 9. Edited source code of TC08

Result: The sequence diagram has not been changes. Synchronization log is shown below.

```
Sequence diagram contains maximum number of lifelines 4. Addition of
lifeline address1:Address is ignored.
```

3.3 TC09: Filtration of get/set calls

Description: Call *getPersonSize* is added to the method *createRegistry*. Because the sequence diagram does not contains any get/set message, the call should be ignored.

```
public void createRegistry() {
    Person person1 = new Person("Andrej","Mlyncar", null);
    PersonRegistry registry = new PersonRegistry();
    registry.getPersonSize();
    registry.printDetails();
```

```
    StatsManager statsManager = new StatsManager(registry);
}
```

List. 10. Edited source code of TC09

Result: The sequence diagram has not been changes. Synchronization log is shown below.

```
Message addition getPersonSize is ignored because sequence diagram
does not contain any get messages.
```

3.4 TC10: Filtration of external calls

Description: Call to an external object is added to the method *createRegistry*. The external object is in different package group as original application.

```
public void createRegistry() {
    Person person1 = new Person("Andrej","Mlyncar", null);
    PersonRegistry registry = new PersonRegistry();
    registry.printDetails();
    StatsManager statsManager = new StatsManager(registry);
    ExternalObject.executeExternalFunction();
}
```

List. 11. Edited source code of TC10

Result: The sequence diagram has not been changes. Synchronization log is shown below.

```
Ignoring addition of lifeline: ExternalObject. Object is located in
different package group. eu.external
```

3.5 TC11: Filtration of combined fragments

Description: If statement is added above call *printDetails* to the method *createRegistry*. The sequence diagram does not contain any combined fragment, therefore the if statement should be ignored.

```
public void createRegistry() {
    Person person1 = new Person("Andrej","Mlyncar", null);
    PersonRegistry registry = new PersonRegistry();
    if(registry!=null)
       registry.printDetails();
    StatsManager statsManager = new StatsManager(registry);
}
```

List. 12. Edited source code of TC11

Result: The sequence diagram has not been changes. Synchronization log is shown below.

```
Fragment addition registry!=null is ignored. Diagram does not contain
any fragments.
```

4 Evaluation of propagation of complex changes

Source codes for test cases from the third test set are available at: https://github.com/rastocny/SOFSEM_SeqDiag_ChangeProp/tree/master/ReplicationPackage/TestProjects/dptest2.

4.1 TC12: Replacing two messages with one new message, which contains internally six new calls

Description: Sequence diagram in Fig. 10 describes a fingerprint log-on process. This functionality is modified to password-based log-on in source code.

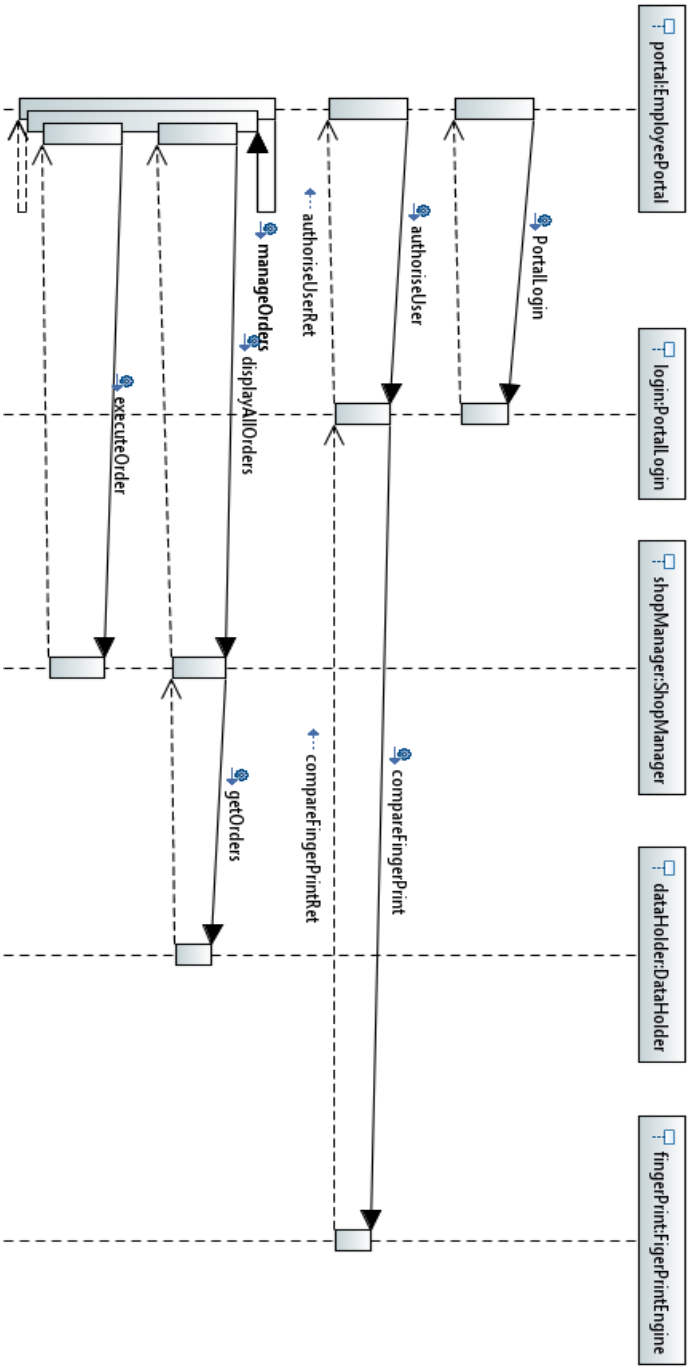

Fig. 10. Original sequence diagram of TC12

Result: The sequence diagram in Fig. 10 has been modified to the sequence diagram in Fig. 11. Messages *authoriseUser* and *compareFingerPrint*, and *Fingerprint* object lifeline have been removed. Messages and lifelines for password-based login has been added. There have been also added messages implementing the method *executeOrder*. Conditions from the modified source code have not been added to the synchronized sequence diagram, because the original sequence diagram does not contain any combined fragment.

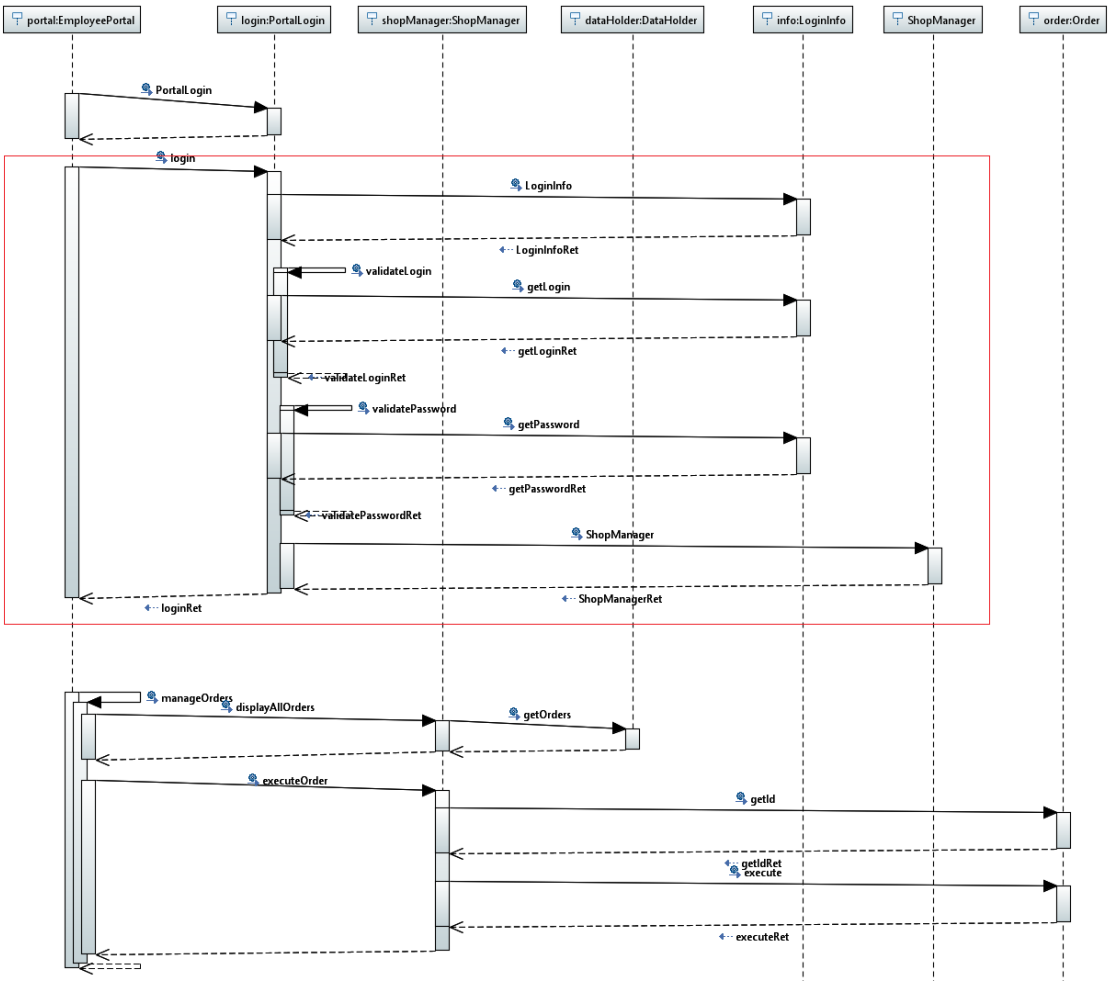

Fig. 11. Modified sequence diagram after execution of TC12

4.2 TC13: Condition change and movement of existing calls to new operation

Description: Sequence diagram in Fig. 12 describes ordering process with a constraint to registered customers. The constraint is modified to check, if order information is specified. Set methods are also moved to new encapsulating method *fillData*.

Result: Constraint in the combined fragment has been changed. Set messages have been moved to the new message *fillData*. Synchronized sequence diagram is shown in Fig. 13.

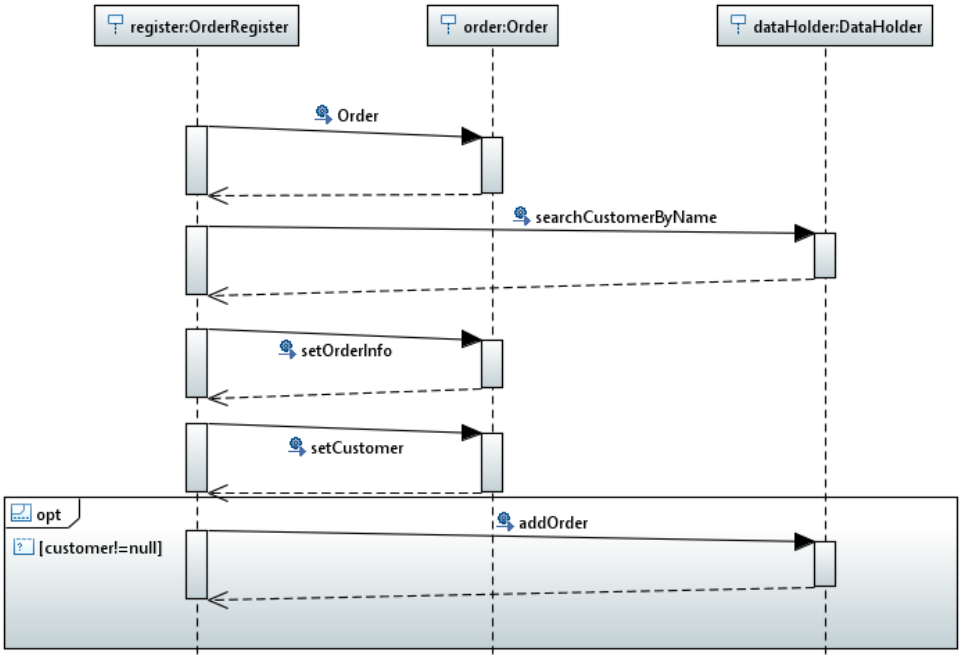

Fig. 12. Original sequence diagram of TC13

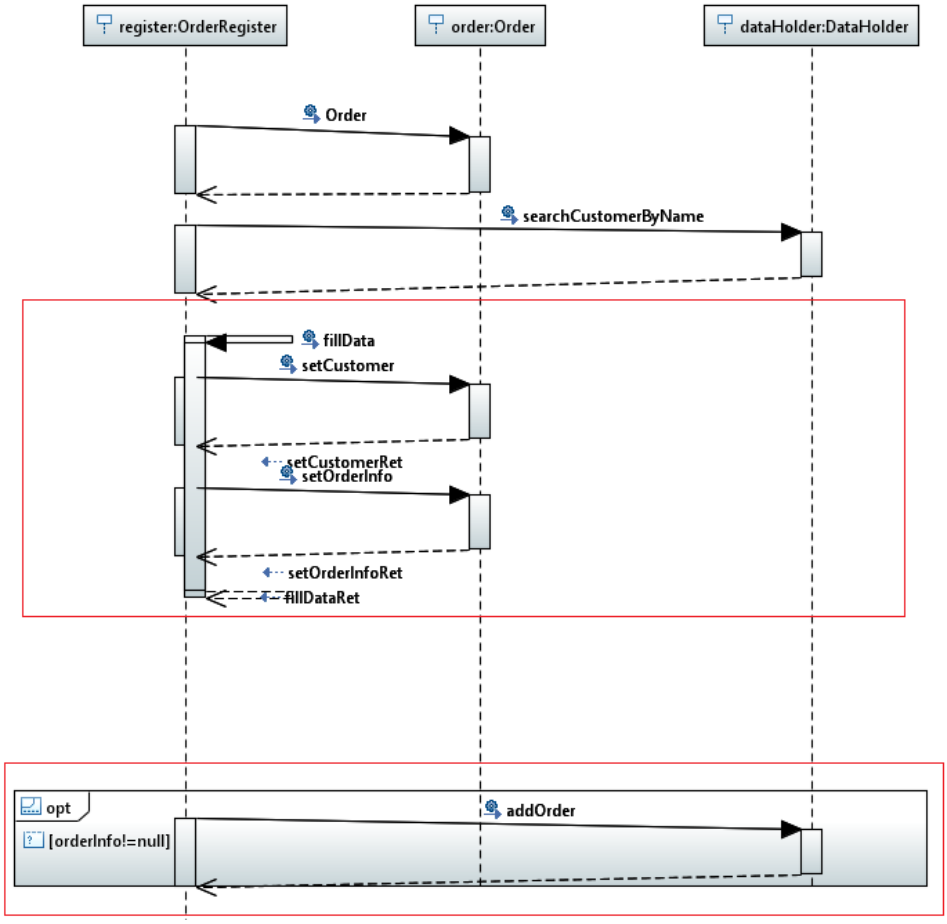

Fig. 13. Modified sequence diagram after execution of TC13

4.3 TC14: Part of the functionality has been moved to new operation

Description: Sequence diagram in Fig. 14 describes order management process. In the source code, the call *displayAllOrders* is moved into new method *accessOrders*. The call login is replaced with a call of *ShopMannager* contructor.

Result: The method *displayAllOrders* has been moved into the merhod *accessOrders*. Missing internal implementations of methods have been added with respect of restriction to lifeline count (defiantly set to five). The message *login* has been replaced with the create message *ShopManager*. The original sequence diagram does not contain any get/set messages, so get/set calls have not been synchronized. Synchronized sequence diagram is shown in Fig. 15.

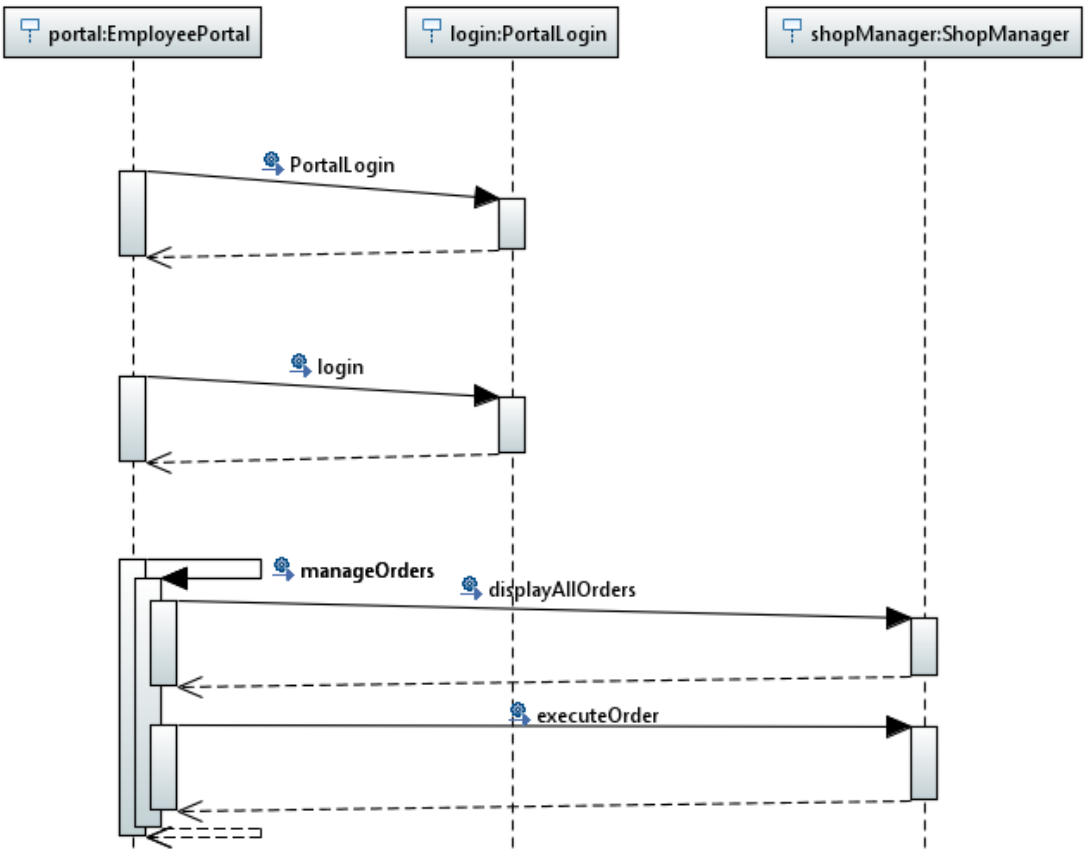

Fig. 14. Original sequence diagram of TC14

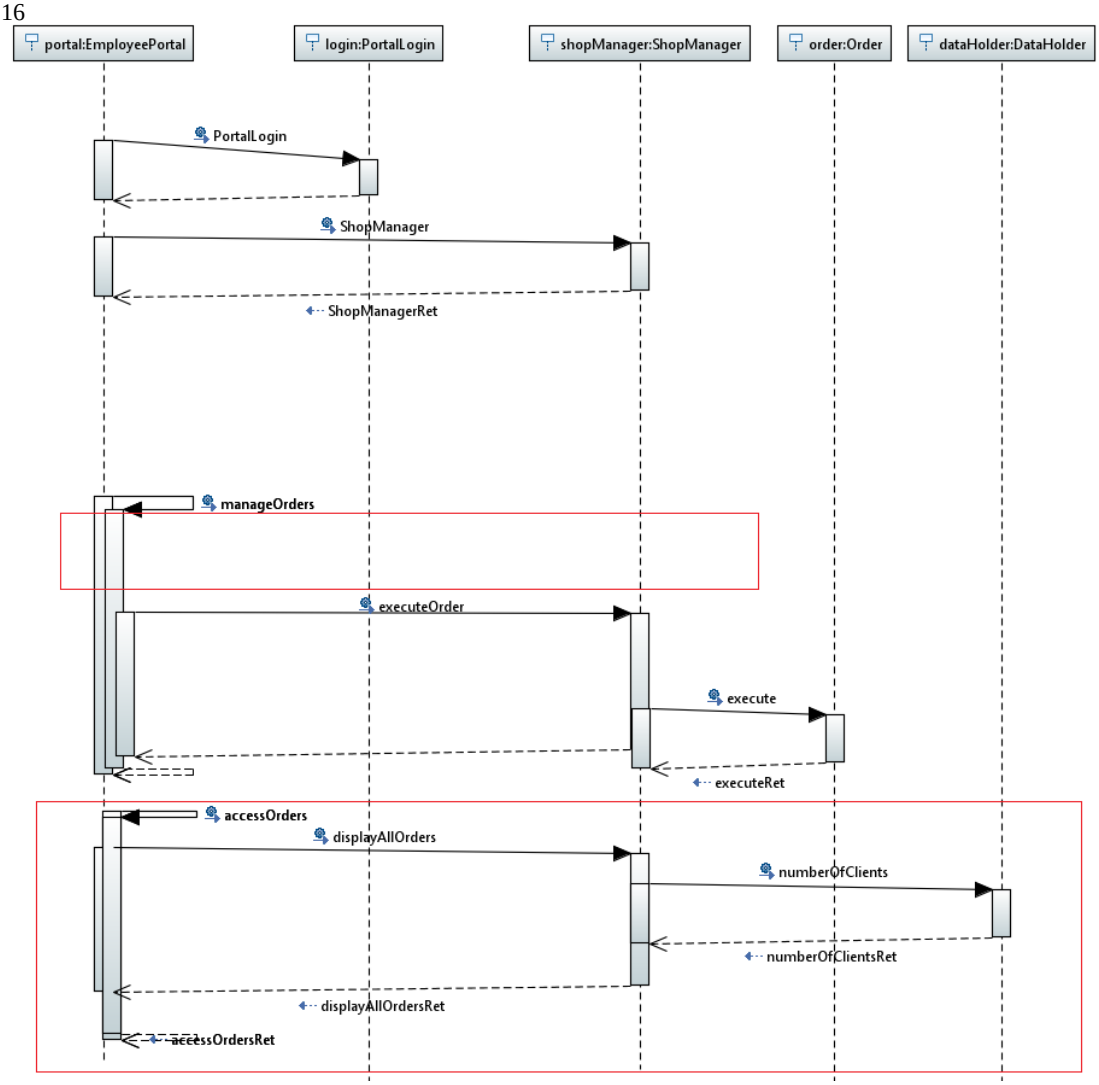

Fig. 15. Modified sequence diagram after execution of TC14

4.4 TC15: Removing a sequence diagram implementation from the source code

Description: The source code implementing the sequence diagram in Fig. 14 is makrely modified by:

- Removing initialization of the object *PortalLogin*
- Log-in process is modified as in TC12
- Call *accessOrders* is added similarly as in TC14
- Call *invariantMethod* is added

Result: The sequence diagram has not been synchronized, because proportion of matched and modified messages is less as defined threshold 0.35. Synchronization log is shown below.

```
Child comparison finished, number of equal nodes: 1
Comparison index: 0.3333333333333333
Unable to synchronize source code and diagrams because of the excep-
tion thrown in previous module.
Comparison Service failed: Error while finding sequence diagram root
in reference structure.
```

4.5 TC16: Adding a loop over an existing condition and adding a new synchronous call into the condition

Description: The source code implementing the sequence diagram in Fig. 12 is modified by adding loop around the if statement, by inserting the call *execute* into the if statement, and by adding call *numberOfClients*.

Result: The combined fragment loop has been added around the combined fragment opt. In addition of the combined fragment loop we observed problems with its visualization – its position is not calculated properly and it covers *setCustomerRet* message, the loop constraint is hidden behind the combined fragment opt, and the background of combined fragments is broken. Other modifications are synchronized correctly. Synchronized sequence diagram is shown in Fig. 16.

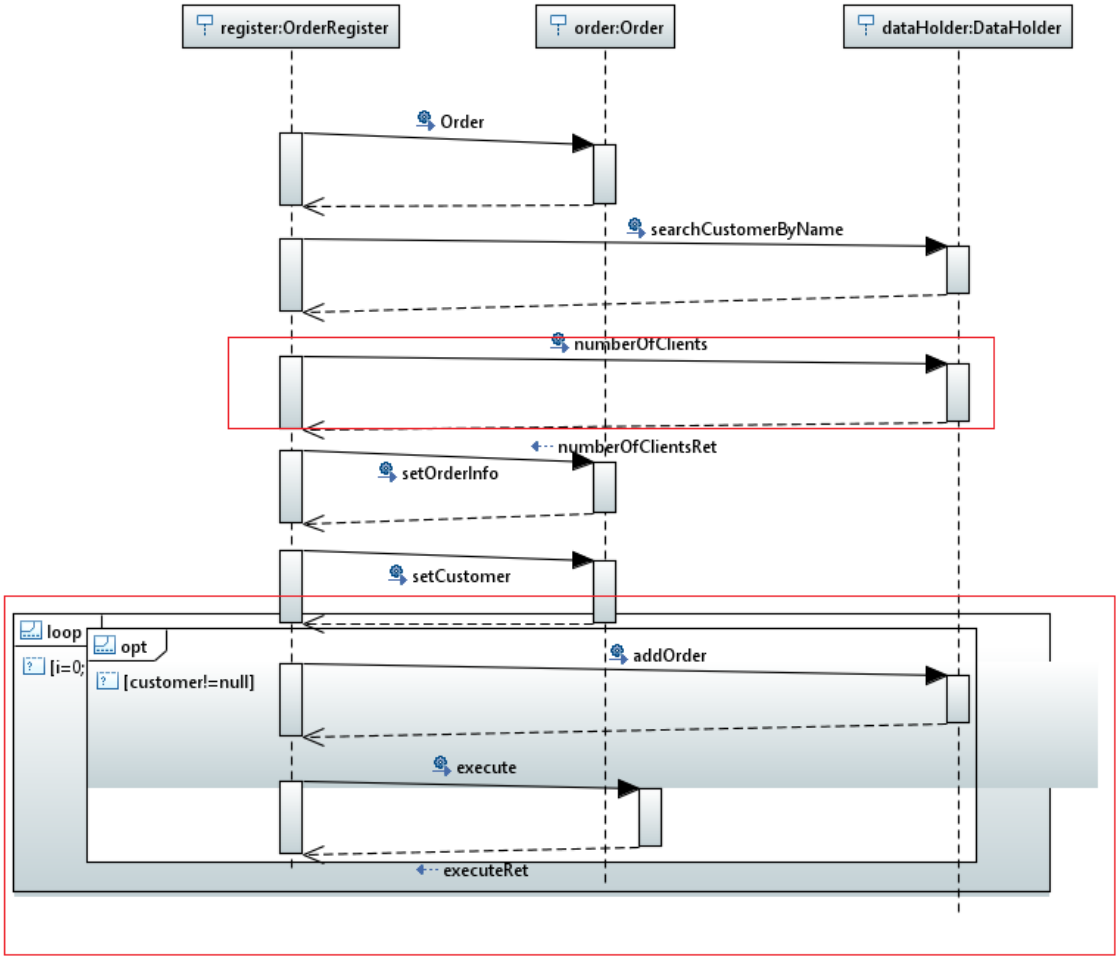

Fig. 16. Modified sequence diagram after execution of TC16