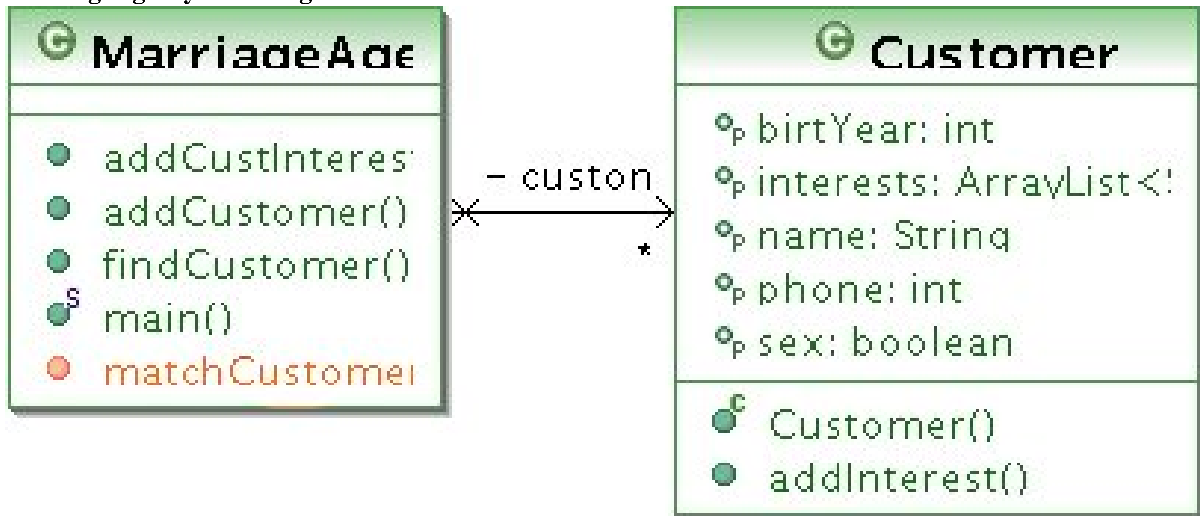
Refactoring Example: Marriage Agency

Assoc. Prof. Hubert Baumeister

Spring 2017

Contents

MarriageAgency class diagram



MarriageAgency code
Method **matchCustomer** in class **MarriageAgency**

```
package dtu.se.marriageagency;

import java.util.ArrayList;

public class MarriageAgency {
   private List<Customer> customers;

   public void setCustomers(List<Customer> customers2) {
      customers = new ArrayList<Customer>(customers2);
   }

   public List<Customer> matchCustomer(Customer customer) {
      List<Customer> res = new ArrayList<Customer>();
      for (Customer potential : customers) {
         // A matching partner needs to have the opposite sex
         if (potential.getSex() != customer.getSex()) {
            // The age difference between matching partners
            // should be less than 10
            int yearDiff = Math.abs(potential.getBirthYear()
                  - customer.getBirthYear());
            if (yearDiff <= 10) {
               for (String interest : potential.getInterests()) {
                  // The customer and the potential candidate should
                  // have one interest in common
                  if (customer.getInterests().contains(interest)) {
                     res.add(potential);
                     break;
                  }
```

```
                }
            }
        }
    }
    return res;
  }
}
```

Tests for MarriageAgency

Refactoring is best applied with a strong test suite. Here is the test suite for the MarriageAgency:

```
package dtu.se.marriageagency;

import static org.junit.Assert.*;

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

import org.junit.Before;
import org.junit.Test;


public class TestMarriageAgency {
   MarriageAgency agency;
   Customer customer1, customer2;

   @Before
   public void setUp() {
      agency = new MarriageAgency();
      List<Customer> customers = new ArrayList<Customer>();
      agency.setCustomers(customers);
      customer1 = new Customer(Sex.MALE,"Harold", 1970);
      customer1.addInterest("skiing");
      customer1.addInterest("check");
      customers.add(customer1);
      customer2 = new Customer(Sex.MALE,"William", 1950);
      customer2.addInterest("biking");
      customer2.addInterest("photograpy");
      customers.add(customer2);
      agency.setCustomers(customers);
   }

   @Test
   public void testMatch() {
      Customer customer = new Customer(Sex.FEMALE,"Annie", 1975);
      customer.addInterest("skiing");
      customer.addInterest("biking");
      List<Customer> matches = agency.matchCustomer(customer);
       assertEquals(1,matches.size());
       assertSame(customer1,matches.get(0));
   }

   @Test
   public void testMatchNoCommonInterests() {
      Customer customer = new Customer(Sex.FEMALE,"Dessie", 1975);
      customer.addInterest("reading");
      customer.addInterest("cycling");
      List<Customer> matches = agency.matchCustomer(customer);
       assertEquals(0,matches.size());
   }

   @Test
   public void testMatchAgeDifferenceTooBig() {
      Customer customer = new Customer(Sex.FEMALE,"Leona", 1981);
      customer.addInterest("skiing");
      customer.addInterest("biking");
      List<Customer> matches = agency.matchCustomer(customer);
       assertEquals(0,matches.size());
   }

   @Test
   public void testMatchWrongSex() {
      Customer customer = new Customer(Sex.MALE,"Christian", 1975);
```

```
        customer.addInterest("skiing");
        customer.addInterest("biking");
        List<Customer> matches = agency.matchCustomer(customer);
         assertEquals(0,matches.size());
    }
}
```

Bad Smell: Long methods

- Methods in object-oriented programs should be short
- They should do a lot of delegation
- Long methods are an indication that an object is doing too much work by themselves
  - → We already discussed this when we discussed about centralised- and decentralised control
- Solution
  - Use **ExtractMethod** to introduce delegation

Bad Smell: Comments

- Comments are good if they explain things that are not visible in the code, e.g. design decisions (*why* you wrote the code in this way)
- Sometimes comments try to hide badly written code; they act as a **deodorant** to cover the bad smells
- Solution
  - Remove the underlying bad design and see if the comments are still necessary in the refactored code
  - Use ExtractMethod or RenameMethod to let the code speak for itself

Refactoring: ExtractMethod

- Motivation:
  - For code fragments that can be grouped together introduce a new method
- Mechanics
  1) Create a new method with a name revealing the *intention* of the code
  2) Copy the code fragment into the new method

  → Complications: references to variables defined outside the scope of the code fragment have to passed as parameter

  → Sometimes other refactorings have to be done before **ExtractMethod** becomes possible (e.g. in our example to **InlineTemp**
- Note that Eclipse has nice support for this refactoring

→ Together with the support from Eclipse, this becomes a very powerful tool to improve ones code

Eclipse: ExtractMethod: hasOppositeSex

```
private boolean hasOppositeSex(Customer customer, Customer potential) {
   return potential.getSex() != customer.getSex();
}
```

Note that we don't need the comment about the opposite sex anymore. The name of the method hasOppositeSex takes over this role.

```
public List<Customer> matchCustomer(Customer customer) {
   List<Customer> res = new ArrayList<Customer>();
   for (Customer potential : customers) {
      if (hasOppositeSex(customer, potential)) {
         // The age difference between matching partners
         // should be less than 10
         int yearDiff = Math.abs(potential.getBirthYear()
               - customer.getBirthYear());
         if (yearDiff <= 10) {
            for (String interest : potential.getInterests()) {
               // The customer and the potential candidate should
               // have one interest in common
               if (customer.getInterests().contains(interest)) {
                  res.add(potential);
                  break;
               }
            }
         }
      }
   }
   return res;
}
```

Eclipse: ExtractMethod: hasAppropriateAge

To apply extract method to introduce a new method hasAppropriateAge, it is a good idea to inline the computation of yearDiff, so that the arguments to hasAppropriateAge are both of class customer.:

1. Step: inline temp yearDiff

```
public List<Customer> matchCustomer(Customer customer) {
   List<Customer> res = new ArrayList<Customer>();
   for (Customer potential : customers) {
      if (hasOppositeSex(customer, potential)) {
         // The age difference between matching partners
         // should be less than 10
         if (Math.abs(potential.getBirthYear()
               - customer.getBirthYear()) <= 10) {
            for (String interest : potential.getInterests()) {
               // The customer and the potential candidate should
               // have one interest in common
               if (customer.getInterests().contains(interest)) {
                  res.add(potential);
                  break;
               }
            }
         }
      }
   }
   return res;
}
```

2. Step: extract method hasAppropriate Age:

```
private boolean hasAppropriateAge(Customer customer, Customer potential)    {
   return Math.abs(potential.getBirthYear()
         - customer.getBirthYear()) <= 10;
}

public List<Customer> matchCustomer(Customer customer) {
   List<Customer> res = new ArrayList<Customer>();
   for (Customer potential : customers) {
      if (hasOppositeSex(customer, potential)) {
```

```
            if (hasAppropriateAge(customer, potential)) {
               for (String interest : potential.getInterests()) {
                  // The customer and the potential candidate should
                  // have one interest in common
                  if (customer.getInterests().contains(interest)) {
                     res.add(potential);
                     break;
                  }
               }
            }
         }
      }
      return res;
   }
```

Refactoring: InlineTemp

- Motivation:
  - A temp is assigned to once and comes in the way of further refactorings
- Mechanics
  1) Declare the temp as final and compile; Why is this step helpful?
  2) Find all references of temp and replace them with the right hand side of the expression; compile and test
  3) Remove the declaration of temp and the assignment to temp; compile and test
- Note: There exists also an opposite refactoring that introduces temps for expressions

→ A lot of refactorings have versions that do exactly the opposite

→ Refactorings should be used to avoid code smells

Extract method: hasOneInterestInCommon

We also extract the computation of a common interest in one method called hasOneInterestInCommon

```
private void hasOneInterestInCommon(Customer customer, List<Customer> res,
        Customer potential) {
    for (String interest : potential.getInterests()) {
         if (customer.getInterests().contains(interest)) {
            res.add(potential);
            break;
        }
    }
}

public List<Customer> matchCustomer(Customer customer) {
    List<Customer> res = new ArrayList<Customer>();
    for (Customer potential : customers) {
        // A matching partner needs to have the opposite sex
        if (hasOppositeSex(customer, potential)) {
            // The age difference between matching partners
            // should be less than 10
            if (hasAppropriateAge(customer, potential)) {
                hasOneInterestInCommon(customer, res, potential);
            }
        }
    }
    return res;
}
```

Code smell: A method is doing two things

- A method should usually do one thing and only one thing and that good.
- If a method tries do to too many things, try to split the method in several methods
- In our example hasOneInterestInCommon determines what it means to have one interest in common *and* also adds a potential customer to the list of matches

Split hasOneInterestInCommon and extractMethod

The method hasOneInterestInCommon the result list as an input parameter and modifies the result from inside the method. This in itself is a code smell. The solution is to seperate the two responsibilities. This is done, by first computing if the two customers have an interest in common and storing the result in a variable, and only then add the customer to the result list, if the variable is true.

```
private void hasOneInterestInCommon(Customer customer, List<Customer> res,
      Customer potential) {
   boolean hasOneInterestInCommon = false;
   for (String interest : potential.getInterests()) {
      if (customer.getInterests().contains(interest)) {
         hasOneInterestInCommon = true;
         break;
      }
   }
   if (hasOneInterestInCommon) {
      res.add(potential);
   }
}
```

The next step is again use extract method to get the function hasOneInterestInCommon that returns true or false instead of changing the result list directly:

```
private boolean hasOneInterestInCommon(Customer customer, Customer potential) {
   boolean hasOneInterestInCommon = false;
   for (String interest : potential.getInterests()) {
      if (customer.getInterests().contains(interest)) {
         hasOneInterestInCommon = true;
         break;
      }
   }
   return hasOneInterestInCommon;
}

private void hasOneInterestInCommon(Customer customer, List<Customer> res,
      Customer potential) {
   boolean hasOneInterestInCommon = hasOneInterestInCommon(customer,
         potential);
   if (hasOneInterestInCommon) {
      res.add(potential);
   }
}
```

What we can do now is inline the void method hasOneInterestInCommon. After having done this, we get each a boolean function for the sex, the age, and the interests.

```
public List<Customer> matchCustomer(Customer customer) {
   List<Customer> res = new ArrayList<Customer>();
   for (Customer potential : customers) {
      if (hasOppositeSex(customer, potential)) {
         if (hasAppropriateAge(customer, potential)) {
            if (hasOneInterestInCommon(customer,
                  potential)) {
               res.add(potential);
            }
         }
      }
   }
   return res;
}
```

ExtractMethod match

We can still do a further refactoring by separating the concern of going through the list of potential candidates and computing the match itself.

```
private void match(Customer customer, List<Customer> res, Customer potential) {
   if (hasOppositeSex(customer, potential)) {
      if (hasAppropriateAge(customer, potential)) {
         if (hasOneInterestInCommon(customer,
               potential)) {
            res.add(potential);
         }
      }
   }
}
```

```
public List<Customer> matchCustomer(Customer customer) {
   List<Customer> res = new ArrayList<Customer>();
   for (Customer potential : customers) {
      match(customer, res, potential);
   }
   return res;
}
```

The match method has now a similar problem as the void method hasOneInterestInCommon. It takes the result list as an argument and modifies the result list. The same trick as with hasOneInterestInCommon can be used to actually get.

```
private boolean match(Customer customer, Customer potential) {
   boolean match = false;
   if (hasOppositeSex(customer, potential)) {
      if (hasAppropriateAge(customer, potential)) {
         if (hasOneInterestInCommon(customer,
               potential)) {
            match = true;
         }
      }
   }
   return match;
}

public List<Customer> matchCustomer(Customer customer) {
   List<Customer> res = new ArrayList<Customer>();
   for (Customer potential : customers) {
      if(match(customer, potential)) {
         res.add(potential);
      }
   }
   return res;
}
```

Code smell: Inappropriate Intimacy

- If a class uses too many features of another class this can be a sign that code and fields may need to move to other classes

→ In our example, class MariageAgency itself computes the matching conditions instead of the delegating this to the customer class

Refactoring: Move Method

- Motivation
  - A method is using or used by more features of another class than the class on which it is defined
  - Move that method to the other class
  - E.g. the methods hasOppositeSex, hasAppropriateAge, hasOneInterestInCommon use instance variables of the customer object; thus these should be methods of class customer
- Mechanics
  1) Declare the method in the target class
  2) Copy the body of the old method to the new method
  3) If necessary, pass on the original object
  4) Replace the body of the old method by the call to the new method; compile and test
  5) Replace all references to the old method with calls to the new method; compile and test
  6) Remove the old method; compile and test

MoveMethod hasOppositeSex, hasAppropriateAge, and hasOneInterestInCommon

We now move the methods hasOppositeSex, hasAppropriateAge, and hasOneInterestInCommon from the class MariageAgency to their first arguments (a customer). This results in the following methods in class customer. Note that the arguments of the methods have changed. The first argument is now this, the customer on which the methods are called. We also do the same with method match, as this looks like a responsibility of the customer and not the MarriageAgency.

```
public enum Sex { MALE, FEMALE; }

public class Customer {
   private Sex sex;
   private String name;
   private int birthYear;
   private List<String> interests = new ArrayList<String>();

   ...

   boolean hasOneInterestInCommon(Customer potential) {
      boolean hasOneInterestInCommon = false;
      for (String interest : potential.getInterests()) {
         if (getInterests().contains(interest)) {
            hasOneInterestInCommon = true;
            break;
         }
      }
      return hasOneInterestInCommon;
   }

   boolean hasAppropriateAge(Customer potential) {
      return Math.abs(potential.getBirthYear() - getBirthYear()) <= 10;
   }

   boolean hasOppositeSex(Customer potential) {
      return potential.getSex() != getSex();
   }

   boolean match(Customer potential) {
      boolean match = false;
      if (hasOppositeSex(potential)) {
         if (hasAppropriateAge(potential)) {
            if (hasOneInterestInCommon(potential)) {
               match = true;
            }
         }
      }
      return match;
   }
}
```

We can still do two simplifactions. The first is to convert the if statements that return a boolean value into a boolean expression. This shortens the program code and enhances the clarity of the code. The second change is that we return from the loop directly when we have found an element instead of breaking out of the loop. The following is the final class customer.

```
public class Customer {
   private Sex sex;
   private String name;
   private int birthYear;
   private List<String> interests = new ArrayList<String>();

   public Customer(Sex sex, String name, int birthYear) {
      this.sex = sex;
      this.name = name;
      this.birthYear = birthYear;
   }

   public String getSex() {
      return sex;
   }

   public int getBirthYear() {
      return birthYear;
   }

   public List<String> getInterests() {
```

```
      return interests;
   }

   public void addInterest(String interest) {
      interests.add(interest);

   }

   boolean hasOneInterestInCommon(Customer potential) {
      for (String interest : potential.getInterests()) {
         if (getInterests().contains(interest)) {
            return true;
         }
      }
      return false;
   }

   boolean hasAppropriateAge(Customer potential) {
      return Math.abs(potential.getBirthYear() - getBirthYear()) <= 10;
   }

   boolean hasOppositeSex(Customer potential) {
      return potential.getSex() != getSex();
   }

   boolean match(Customer potential) {
      return hasOppositeSex(potential) & hasAppropriateAge(potential)
            & hasOneInterestInCommon(potential);
   }
}
```

The following remains in the class MarriageAgency

```
public class MarriageAgency {
   private List<Customer> customers;

   public void setCustomers(List<Customer> customers2) {
      customers = new ArrayList<Customer>(customers2);
   }

   public List<Customer> matchCustomer(Customer customer) {
      List<Customer> res = new ArrayList<Customer>();
      for (Customer potential : customers) {
         if(customer.match(potential)) {
            res.add(potential);
         }
      }
      return res;
   }
}
```

Advantages of the new design

The new design is much easier to read than the original design, because each method is now responsible for one thing only and has a name that shows what it is responsible for. In addition, we have moved the responsibilities into the right class. For example, the code related to the actual matching has been moved to the customer class. The remaining responsibility of the MarriageAgency is going through the list of its customer but does not have anymore the responsibility to determine when one customer matches another. Because we now have small methods that are only responsible.for one task, the algorithm can be simply modified. We could have a special type of customer (i.e. a subclass of customer) that defines hasAppropriateAge in a different way. Also, we could add a gay customer to the system. He would not look for the opposite sex, but for a same sex partner. The implementation will override hasOppositeSex to return true if the sex is the same. However, then the name hasOppositeSex does not fit anymore, and the method should be renamed hasAppropriateSex.

Summary

- Refactoring improves the quality of the code by removing **code smells**
- Refactoring is applied to prepare the code for adding new functionality and improving the code of existing functionality (after, e.g., new functionality has been added)

→ Refactoring plays an important role in good design

- Refactoring depends on sufficient test cases
- Refactoring itself are described through their mechanics
  - Small steps leading from correct (compilable and tests pass) to correct programs

→ Today, IDE's support the most important refactorings