**Refactoring Exercises**

**PART 2: Typical Smells + Basic Refactorings**

**Prefer automatic IDE based refactorings**

**Exercise 1 20 min**

Have a look at the following classes.

1. Identify possible problems and smells
2. Address and solve them by using base refactorings like rename, inline, and extract

**public** **class** Person

{

String name;

**int** age;

**public** **static** **void** main(String[] args) {

Person mike = **new** Person();

mike.name = "Mike";

mike.age = 51;

}

}

// Which Smell?

**public** **class** PersonPrinter {

**public** **static** **void** main(String[] args) {

Person mike = **new** Person();

mike.name = "Mike";

mike.age = 51;

System.***out***.println("Person " + mike.name + " is " + mike.age + " years old.");

}

}

**Exercise 2: Refactoring Long Parameter List 20 min**

Have a look at the following class.

1. Identify possible problems and smells
2. Address and solve them by using base refactorings like rename, inline, and extract

**class** PollutantEntry {

String country;

String state;

String city;

String place;

LocalDateTime localDateTime;

Float average;

Float max;

Float min;

String pollutant;

**public** PollutantEntry(String country, String state, String city, String place, LocalDateTime localDateTime, Float average, Float max, Float min, String pollutant) {

**this**.country = country;

**this**.state = state;

**this**.city = city;

**this**.place = place;

**this**.localDateTime = localDateTime;

**this**.average = average;

**this**.max = max;

**this**.min = min;

**this**.pollutant = pollutant;

}

}

Have a look at:

<https://konfhub.medium.com/refactoring-long-parameter-list-in-constructors-java-e9c9f2cb1fb>

**Exercise 3: Feature Envy 10 min**

Have a look at:

<https://www.thecodebuzz.com/awesome-code-feature-envy-code-smell-resolution/>

**Exercise 4: Replace Type Code with Class / Enum 30 min**

Have a look at the following class.

1. Identify possible problems
2. Address and solve them by using base refactorings, especially move

**public** **class** BusinessClass

{

**public** **static** **final** **int** ***JOBSTATUS\_UNDEFINED*** = -1;

**public** **static** **final** **int** ***JOBSTATUS\_ACTIVE*** = 1;

**public** **static** **final** **int** ***JOBSTATUS\_FINISHED*** = 2;

**private** **static** **final** String ***JOBSTATUS\_UNDEFINED\_NAME*** = "UNDEFINED";

**private** **static** **final** String ***JOBSTATUS\_ACTIVE\_NAME*** = "Active";

**private** **static** **final** String ***JOBSTATUS\_FINISHED\_NAME*** = "Finished";

**public** **static** **final** **int** ***ERRORCODE\_OK*** = 0;

**public** **static** **final** **int** ***ERRORCODE\_WARN*** = 1;

**public** **static** **final** **int** ***ERRORCODE\_ERROR*** = 2;

**private** **final** BusinessJob job;

**public** BusinessClass(**final** BusinessJob job) {

**this**.job = job;

}

**public** **int** getJobState() {

**if** (job.isActive())

**return** ***JOBSTATUS\_ACTIVE***;

**if** (job.isFinished())

**return** ***JOBSTATUS\_FINISHED***;

**return** ***JOBSTATUS\_UNDEFINED***;

}

}

**Exercise 5: Clean up design 30 min**

1. look at all classes roughly

2. what are the problems in address and person?

3. how to proceed step by step? Which refactorings are useful?

4. run AddressPrinterApp class

**PART 4: Testing + JUnit Refresher**

**Exercise 1: Calculator 15 min**

**Exercise 1a**:

Create or inspect a simple class Ex01\_Calculator and a method add() to add and divide()to divide two numbers.

**Exercise 1b**:

Create corresponding test methods:

a) for any two positive numbers

b) the same value positive and negative: how to make the naming beautiful?

**Exercise 1c**:

Check/test exceptions for

a) a division by 0.

b) inspect the text of the exception

**Exercise 1d**:

Extend the calculator with the following method and write a test that has a meaningful name, like **0.1 + 0.1 + 0.1 = 0.3**

**public** **double** sumUp\_0\_1(**final** **int** factor)

{

**double** value = 0.1;

**double** result = 0;

**for** (**int** i = 0; i < factor; i++)

{

result += value;

}

**return** result;

}

**Exercise 2: Leap Year: Conversion to parameterized test with hint 15 min**

Given a calculation of leap years in a class Ex03\_LeapYear. Improve the implementation of the tests in the class Ex03\_LeapYearTest.

There are a few special cases as well as the “rule of 4”:

* If a year is divisible by 4, it is usually a leap year.
* Years that are divisible by 100 are called secular years and are not leap years.
* However, secular years that are also divisible by 400 are leap years again.

**Exercise 2a**: These rules can be tested individually as follows, but since JUnit 5, a Parametrized Test is the obvious choice. Convert the following into such a test:

@Test

**void** testIsLeap\_4\_Years\_Rule()

{

**final** **boolean** result = Ex01\_LeapYear.*isLeap*(2020);

*assertTrue*(result);

}

@Test

**void** testIsLeap\_100\_Year\_Rule()

{

**final** **boolean** result = Ex01\_LeapYear.*isLeap*(1900);

*assertFalse*(result);

}

@Test

**void** testIsLeap\_400\_Years\_Rule()

{

**final** **boolean** result = Ex01\_LeapYear.*isLeap*(2000);

*assertTrue*(result);

}

**Exercise 2b**: In addition to a simple test, there should also be an indication of why such a decision was made. To do this, write a test method that produces output such as the following:

**Ein Bild, das Text enthält.

Automatisch generierte Beschreibung**

**Bonus:** Improve readability and use these characters:

**Ein Bild, das Text enthält.

Automatisch generierte Beschreibung**

**Exercise 3: Roman Numbers Adder 15 min**

Given a class RomanNumbers for converting to and from Roman numbers with the two methods:

* int fromRomanNumber(String)
* String toRomanNumber(int)

Conveniently, both are already well tested with unit tests.

But now comes business and wants to provide an addition functionality. Develop this in a class Ex03\_RomanNumberAdder and rely on unit tests. The business is willing to contribute a CSV file roman-addition.csv, which looks shortened as follows:

Roman Roman Summe Berechnung = Wert

---------------------------------------------

I, I, II, 1 + 1 = 2

I, II, III, 1 + 2 = 3

I, III, IV, 1 + 3 = 4

I, IV, V, 1 + 4 = 5

V, II, VII, 5 + 2 = 7

V, IV, IX, 5 + 4 = 9

X, VII, XVII, 10 + 7 = 17

X, XX, XXX, 10 + 20 = 30

…

**PART 5: Advanced Refactorings**

**Exercise 1: Refactor to get rid of external dependencies 45 min**

Use (a lot of) basic refactoring steps to get rid of external dependencies and reach the desired design shown below. Create an enum SupportedFrequencies instead of using ComplexFrequency. Prefer standard Java Date and Time API classes like LocalDateTime instead of ExtTimePeriod.

**public** **class** TimeStampUtils

{

**public** **static** String createTimeStampString(**final** ExtTimePeriod currentPeriod,

**final** ComplexFrequency frequency)

{

**final** LocalDateTime start = currentPeriod.getDateTime();

**final** **int** divisor = frequency == ComplexFrequency.***P1M*** ? 1 : 3;

**final** String addition = frequency == ComplexFrequency.***P1M*** ? "" : "Q";

**final** **int** value = ((start.getMonthValue() - 1) / divisor + 1);

**return** start.getYear() + "-" + addition + value;

}

}

**public** **class** ExtTimePeriod

{

**public** LocalDateTime getDateTime()

{

**return** LocalDateTime.*now*();

}

}

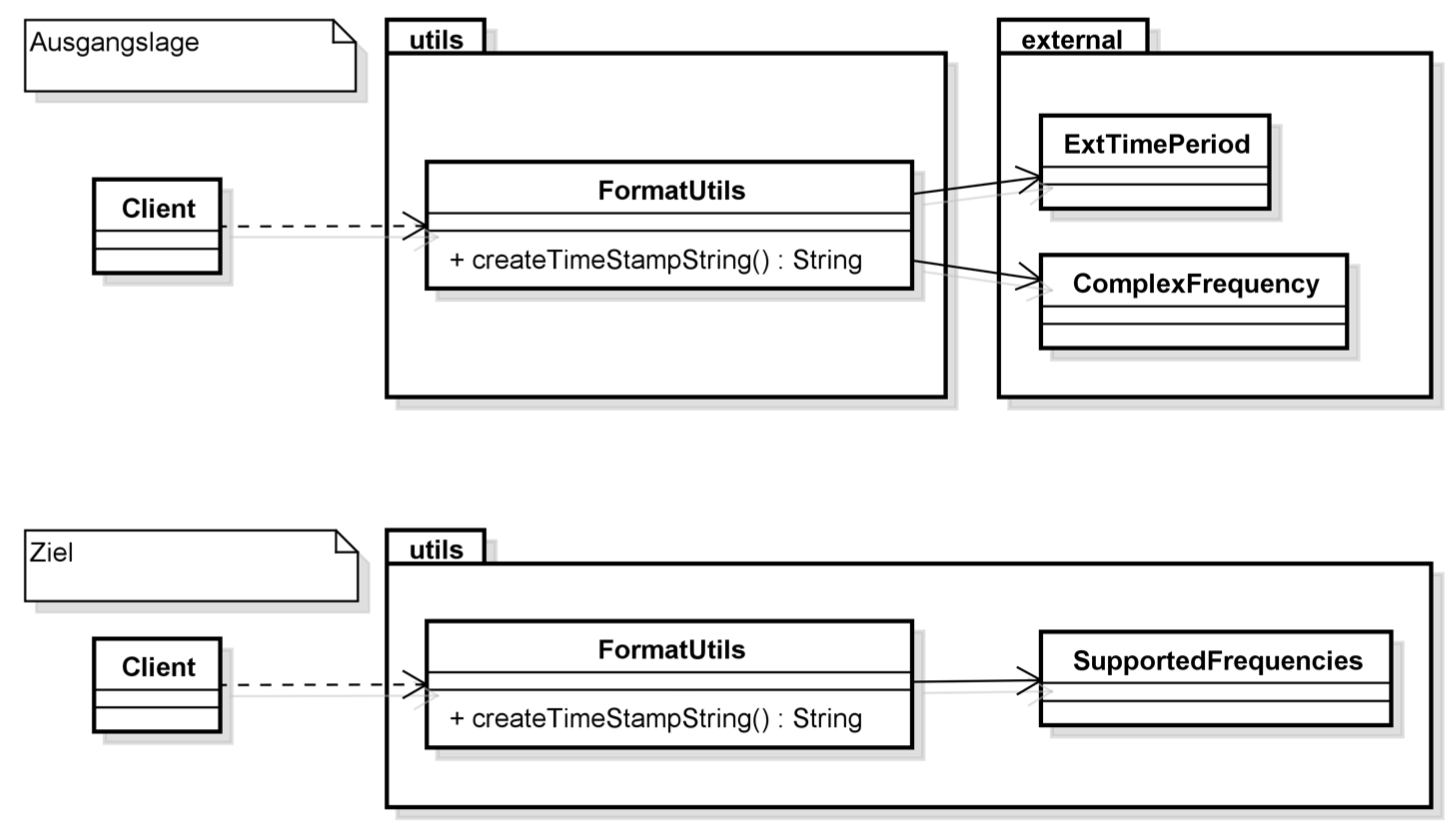
**public** **enum** ComplexFrequency

{

***P1D***, ***P1W***, ***P1M***, ***P3M***, ***P6M***, ***P1Y***;

}

which results in the following dependency overview and the below goal where no dependencies to external components exists any longer.



**Some Tips on how to proceed, e.g., for resolving the dependencies**

Step 1: Introduce auxiliary variable (EXTRACT LOCAL VARIABLE)

Step 2: Remove dependency on frequency (EXTRACT METHOD)

Step 3: Inlining method call(s) (INLINE)

Step 4: Remove dependency on ExtTimePeriod class

Step 5: Ensure consistent parameter order

Step 6: Inlining method call(s)

Step 7: Add tests

…

**Exercise 2: Project Group: Refactor To Composite 30 min**

Given the two classes ProjectTask and ProjectGroup to define workflows on projects your challenge is

* to get rid of some programming problems and
* to allow hierarchical projects.

Keep in mind that comments should be available not only for tasks.

**public** **class** ProjectGroup {

String name;

List<ProjectTask> tasks = **new** ArrayList<>();

**public** ProjectGroup(String name) {

**this**.name = name;

}

**public** ProjectGroup(String name, List<ProjectTask> tasks) {

**this**.name = name;

**this**.tasks = tasks;

}

**public** **int** calcDuration()

{

**int** durationInDays = 0;

**for** (**var** task : tasks)

{

durationInDays += task.durationInDays;

}

**return** durationInDays;

}

}

**public** **class** ProjectTask {

String name;

**int** durationInDays;

String comment;

**public** ProjectTask(String name, **int** durationInDays) {

**this**.name = name;

**this**.durationInDays = durationInDays;

}

}

So please refactor these classes towards composite pattern, like the following:



# Solutions

**PART 2: Typical Smells + Basic Refactorings**

**Exercise 5: Clean up design**

class Address

* Remove unused attribute street
* ***Encapsulate Field*** for city and country
  + Attribute private
  + Introduce Setter / Getter
* Attribute could be made final, since only assigned in constructor
* ***Generate toString()***

class Person

* Zuweisungsfehler in personInit() bereinigen
* Extract method printAddress(Address) out of printAddresses()
* Move method printAddress(Address: ***Move***
* Rename methode setFirstAddress(Address: ***Rename***
* ***Encapsulate Field*** for name and age
  + Attribute private
  + Introduce Setter / Getter einführen
  + 2 x **Inline** in personInit()
* private final for attribute addresses
* ***Generate constructor using fields***
* Call constructor with personInit()
* execute ***Inline***
* final for Attribut name

**Optional**

* Design problem: How to get older? Modeling as birthday

**PART 5: Advanced Refactorings**

**Exercise 1: Refactor to get rid of external dependencies 45 min**

**Step 1: Introduce helper variable (EXTRACT LOCAL VARIABLE)**

**public** **static** String createTimeStampString(ExtTimePeriod currentPeriod,

ComplexFrequency frequency)

{

**final** LocalDateTime start = currentPeriod.getDateTime();

**final** **boolean** isMonthly = frequency == ComplexFrequency.***P1M***;

**final** **int** divisor = isMonthly ? 1 : 3;

**final** String addition = isMonthly ? "" : "Q";

**final** **int** value = ((start.getMonthValue() - 1) / divisor + 1);

**return** start.getYear() + "-" + addition + value;

}

**Step 2: Remove dependence on frequency (EXTRACT METHOD)**

@Deprecated

**public** **static** String ~~createTimeStampString~~(**final** ExtTimePeriod currentPeriod,

**final** ComplexFrequency frequency)

{

**final** **boolean** isMonthly = frequency == ComplexFrequency.***P1M***;

**return** *createTimeStampString*(currentPeriod, isMonthly);

}

**public** **static** String createTimeStampString(**final** ExtTimePeriod currentPeriod, **final** **boolean** isMonthly) {

**final** **int** divisor = isMonthly ? 1 : 3;

**final** String addition = isMonthly ? "" : "Q";

**final** LocalDateTime start = currentPeriod.getDateTime();

**final** **int** value = ((start.getMonthValue() - 1) / divisor + 1);

**return** start.getYear() + "-" + addition + value;

}

**Step 3: Inlining of the method call for usages (INLINE)**

**public** **static** **void** main(String[] args) {

**final** ExtTimePeriod currentPeriod = **new** ExtTimePeriod();

**final** ComplexFrequency frequency = ComplexFrequency.***P6M***;

**final** String timeStamp = TimeStampUtils.~~createTimeStampString~~(currentPeriod, frequency);

System.***out***.println(timeStamp);

**final** String timeStamp2 = TimeStampUtils.~~createTimeStampString~~(currentPeriod, ComplexFrequency.***P1M***);

System.***out***.println(timeStamp2);

}

**=>**

**public** **static** **void** main(String[] args) {

**final** ExtTimePeriod currentPeriod = **new** ExtTimePeriod();

**final** ComplexFrequency frequency = ComplexFrequency.***P6M***;

**final** **boolean** isMonthly = frequency == ComplexFrequency.***P1M***;

**final** String timeStamp = TimeStampUtils.*createTimeStampString*(currentPeriod, isMonthly);

System.***out***.println(timeStamp);

**final** String timeStamp2 = TimeStampUtils.*createTimeStampString*(currentPeriod, isMonthly);

System.***out***.println(timeStamp2);

}

* **Remove deprecated method**

**--- REJOIN POINT 1**

**Step 4: Remove dependency on ExtTimePeriod class**

@Deprecated

**public** **static** String ~~createTimeStampString~~(**final** ExtTimePeriod currentPeriod, **final** **boolean** isMonthly) {

**final** LocalDateTime start = currentPeriod.getDateTime();

**return** *createTimeStampString*(isMonthly, start);

}

**public** **static** String createTimeStampString(**final** **boolean** isMonthly, **final** LocalDateTime start) {

**final** **int** divisor = isMonthly ? 1 : 3;

**final** String addition = isMonthly ? "" : "Q";

**final** **int** value = ((start.getMonthValue() - 1) / divisor + 1);

**return** start.getYear() + "-" + addition + value;

}

**Step 5: Ensure consistent parameter order**

**public** **static** String createTimeStampString(**final** LocalDateTime start, **final** **boolean** isMonthly) {

**final** **int** divisor = isMonthly ? 1 : 3;

**final** String addition = isMonthly ? "" : "Q";

**final** **int** value = ((start.getMonthValue() - 1) / divisor + 1);

**return** start.getYear() + "-" + addition + value;

}

**Step 6: Inlining of the method call for usages (INLINE)**

**Conclusion**

The created method no longer has any dependencies on external classes or at least only on classes that come from standard libraries of the JDK. This makes it much easier to check the whole thing with unit tests. So far, however, we have not executed any unit tests, especially because there simply were none, which we now want to change, and because we have only applied safe basic refactorings.

**--- REJOIN POINT 2**

**Step 7: Get rid of boolean parameter, use enum**

**Step 8: Simplification of the statements**

1. The ternary operator (? operator) can be converted into an if-else statement. To do this, we use the keyboard shortcut CTRL+1 for QUICK FIX and select REPLACE CONDITIONAL WITH 'IF-ELSE' there.
2. Now we want to group the respective lines for the two conditions together. A prerequisite for this, however, is that we convert the if-else statements into blocks. To do this, we select the if and again use the keyboard shortcut CTRL+1, which now offers us the option CHANGE 'IF-ELSE' STATEMENTS TO BLOCKS, which we select. Then we rearrange the lines by grouping the lines from the respective conditions.

**public** **static** String createTimeStampString(**final** LocalDateTime start, **final** **boolean** isMonthly) {

**final** **int** divisor;

**final** String addition;

**if** (isMonthly) {

divisor = 1;

addition = "";

} **else** {

addition = "Q";

divisor = 3;

}

**final** **int** value = ((start.getMonthValue() - 1) / divisor + 1);

**return** start.getYear() + "-" + addition + value;

}

**Step 9: Simplification of the statements**

Before we can simplify, it gets a bit more confusing, and we also need some manual work to perform the inverse variant of the basic refactoring CONSOLIDATE DUPLICATE CONDITIONAL FRAGMENT. Normally, you want to use this to merge duplicated elements in if branches into one statement at the end. Here we do the opposite and duplicate the return statement to provide it in each if-else branch:

**public** **static** String createTimeStampString(**final** LocalDateTime start, **final** **boolean** isMonthly) {

**final** **int** divisor;

**final** String addition;

**if** (isMonthly) {

divisor = 1;

addition = "";

**final** **int** value = ((start.getMonthValue() - 1) / divisor + 1);

**return** start.getYear() + "-" + addition + value;

} **else** {

addition = "Q";

divisor = 3;

**final** **int** value = ((start.getMonthValue() - 1) / divisor + 1);

**return** start.getYear() + "-" + addition + value;

}

}

The variables divisor and addition are actually unnecessary, since they each contain only simple constants. We can now replace the values for both variables directly in the source code.

**public** **static** String createTimeStampString(**final** LocalDateTime start, **final** **boolean** isMonthly) {

**if** (isMonthly) {

**return** start.getYear() + "-" + "" + ((start.getMonthValue() - 1) / 1 + 1);

} **else** {

**return** start.getYear() + "-" + "Q" + ((start.getMonthValue() - 1) / 3 + 1);

}

}

**Simplification of calculation**

(start.getMonthValue() - 1) / 1 + 1 => (start.getMonthValue() - 1) + 1 => (start.getMonthValue() - 1)

**=>**

**public** **static** String createTimeStampString(**final** LocalDateTime start,

**final** **boolean** isMonthly) {

**if** (isMonthly) {

**return** start.getYear() + "-" + "" + start.getMonthValue();

} **else** {

**return** start.getYear() + "-" + "Q" + ((start.getMonthValue() - 1) / 3 + 1);

}

}

**Step 10: Move functionality**

If we take a closer look at the latest version of the createTimeStampString(LocalDateTime, SupportedFrequencies) method, we can see that it contains functionality in the if-else that is strongly related to the SupportedFrequencies enumeration in each case. We defined these when creating the unit tests to improve the slightly unwieldy API. It now makes sense to move even more functionality there. To do this, we take advantage of the fact that an enum enumeration can have at- tributes and methods, and the latter can even be overridden:

1. **remove static**
2. **move**
3. **add static**

**enum** SupportedFrequencies

{

***MONTHLY***, ***QUARTERLY***;

**public** String createTimeStampString(**final** LocalDateTime start) {

**boolean** isMonthly = **this** == SupportedFrequencies.***MONTHLY***;

**if** (isMonthly) {

**return** start.getYear() + "-" + "" + start.getMonthValue();

} **else** {

**return** start.getYear() + "-" + "Q" + ((start.getMonthValue() - 1) / 3 + 1);

}

}

}

**Step 11: Enhance enum**

1. **BY HAND: assign to enum values**
2. **Inline in usage**
3. **Remove class TimeStampUtils**

**enum** SupportedFrequencies

{

***MONTHLY*** {

@Override

**public** String createTimeStampString(LocalDateTime start) {

**return** start.getYear() + "-" + "" + start.getMonthValue();

}

}, ***QUARTERLY*** {

@Override

**public** String createTimeStampString(LocalDateTime start) {

**return** start.getYear() + "-" + "Q" + ((start.getMonthValue() - 1) / 3 + 1);

}

};

**public** **abstract** String createTimeStampString(**final** LocalDateTime start);

}

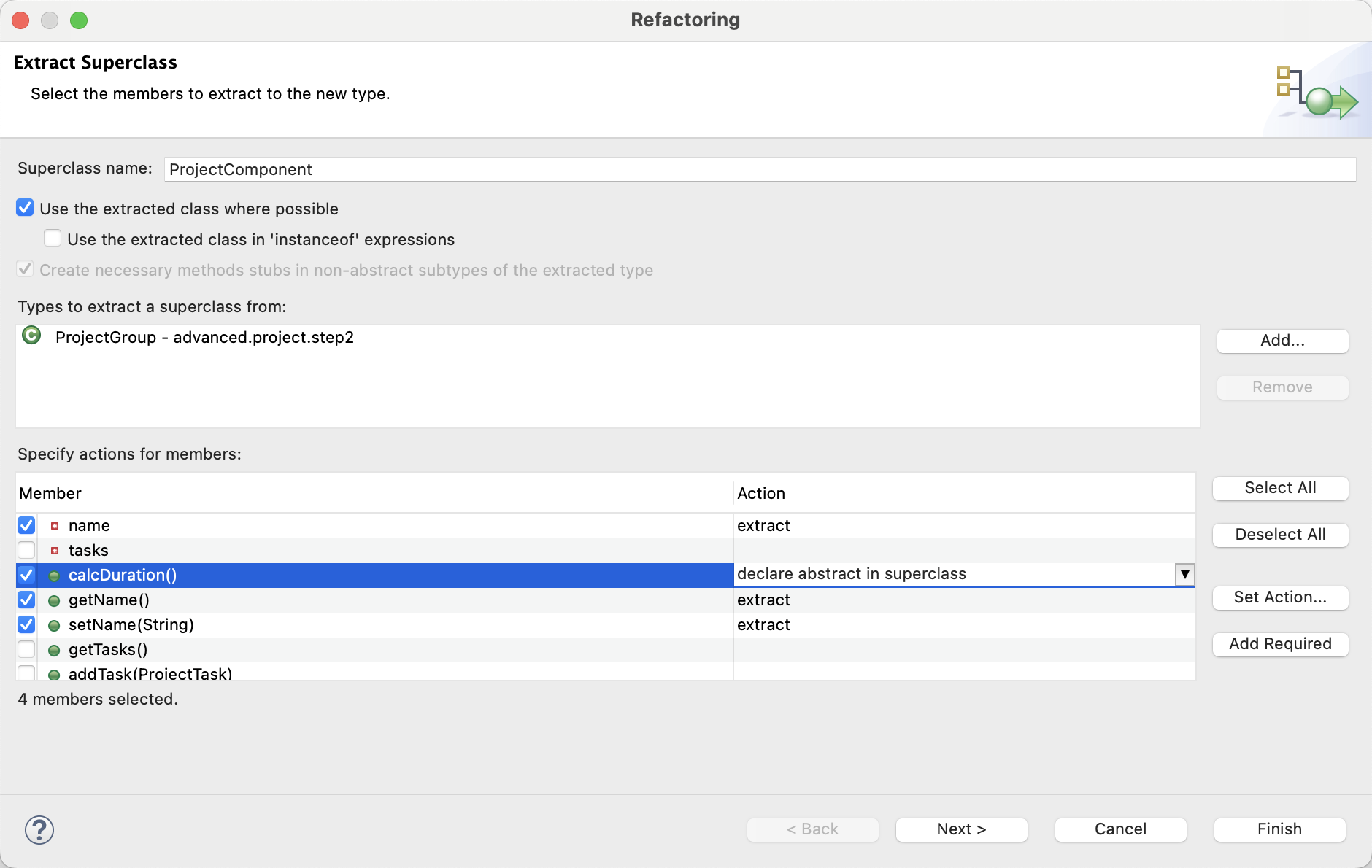
**Exercise 2: Project Group: Refactor To Composite 30 min**

**Step 1: Encapsulate**

1. **2 x Encapsulate Field for ProjectTask**
2. **2 x Encapsulate Field for ProjectGroup**
3. **Apply Encapsulate Collections for Tasks**
   1. **Return Unmodifiable**
   2. **setTasks() => rename addTask()**
   3. **Change Method Signature**
   4. **Modify impl**
   5. **Copy and impl removeTask()**

**Step 2: Introduce BaseClass**

1. **Extract SuperClass on ProjectGroup**

****

1. **Create ctor using field in ProjectComponent**
2. **BY HAND: adjust ctor calls to super(name)**
3. **Remove default ctor in ProjectComponent**
4. **Extract SuperClass on ProjectTask “ProjectComponent2”**
5. **Delete “ProjectComponent2”**
6. **BY HAND: Redirect ProjectTask from “ProjectComponent2”**
7. **BY HAND: adjust ctor calls to super(name)**
8. **Add unimplemented method and return durationInDays**
9. **getDurationInDays() => call new method**
10. **Inline getDurationInDays()**

**Step 3: Use ProjectComponent instead of Tasks and apply composite**

1. **BY HAND: Replace ProjectTask with ProjectComponent**
2. **Rename tasks => projectComponents**
3. **Pull up comment**

**Ein Bild, das Text enthält.

Automatisch generierte Beschreibung**