A Novice's Process of Object-Oriented Programming

Michael E. Caspersen
Department of Computer Science
University of Aarhus
Aabogade 34, DK-8200 Aarhus N
Denmark
mec@daimi.au.dk

Michael Kölling
Computing Laboratory
University of Kent
Canterbury, Kent CT2 7NF
United Kingdom
mik@kent.ac.uk

Abstract

Exposing students to the process of programming is merely implied but not explicitly addressed in texts on programming which appear to deal with 'program' as a noun rather than as a verb.

We present a set of principles and techniques as well as an informal but systematic process of decomposing a programming problem. Two examples are used to demonstrate the application of process and techniques.

The process is a carefully down-scaled version of a full and rich software engineering process particularly suited for novices learning object-oriented programming. In using it, we hope to achieve two things: to help novice programmers learn faster and better while at the same time laying the foundation for a more thorough treatment of the aspects of software engineering.

Categories and Subject Descriptors D1.5 [**Programming Techniques**]: Object-oriented programming.
D2.3 [**Software Engineering**]: Coding Tools and Techniques – *Object-oriented programming, Structured programming, Top-down programming*.

D2.4 [**Software Engineering**]: Software/Program Verification – *Class invariants, Programming by contract*.

General Terms Algorithms, Design, Documentation, Languages.

Keywords CS1, Systematic Programming, Programming Process, Design by Contract, Representation Invariant, Objects-First, Stepwise Refinement, Top-Down Design, Incremental Development, Testing, Refactoring, Programming Education, UML, Pedagogy.

1. Introduction

I remember when I first learned to program. I had a couple of workbooks covering the fundamentals of programming. I went through them pretty quickly. When I had done that, I wanted to tackle a more challenging problem than the little exercises in the book. I decided I would write a Star Trek game…
My process for writing the programs to solve the workbook exercises had been to stare at the problem for a few minutes, type in the code to solve it, then deal with whatever problem arose. So I sat confidently down to write my game. Nothing came! I needed to do something beyond coding. But I didn't know what else to do.
Kent Beck [3]

Most texts used to teach beginners to program focus on presenting language constructs, programming language concepts, and computer programs (complete or partial). They are concerned mostly with 'program' as a noun rather than as a verb. The process of program development is often merely implied rather than explicitly addressed. A typical structure is the presentation of a problem followed by a presentation of a program to solve that problem and a discussion of the program's elements.

From the viewpoint of a student, the program was developed in a single step, starting from a problem specification and resulting in a working solution. Sometimes, a semi-formal requirements specification is included as an interim step, but this does not fundamentally alter our main point: the process of software development is essentially invisible. The fact that we all start by developing sub-optimal and partial implementations on our way to a solution, which we later refine and improve, often seems to be the best kept secret of the computing profession.

The exercises in texts often compound the problem; they frequently require small, easy-to-understand steps that are quite different in character from the development of a complete software solution. The problems resulting from this approach are potentially two-fold:

- Students may be able to understand every separate construct but do not have the skills to put the constructs together in an organised way. This is succinctly illustrated in Kent Beck's quote above. Or, if they do succeed:
- Students, who labour through various incorrect attempts at solving a problem, slowly improving their solution, running into regular bugs along the way before developing a solution that mostly works, often think they are poor programmers for experiencing so much trouble along the way.

To solve these problems we need to do two things:

- Teach students about the *process* of software development, to enable them to follow organised steps to move toward a solution to a problem, and
- Treat software development explicitly as a process that is carried out in *stages and small steps*, rather than the writing of a single, monolithic solution.

OOPSLA'06 October 22–26, 2006, Portland, Oregon, USA.
ACM 1-59593-491-X/06/0010.

If we do not explicitly teach the programming process, we end up with two groups of students: those who cannot cope with the challenge of development and those who discover their own process.

Developing software is, by its very nature, always a process, whether we are formally aware of it or not.

Some of the first group, those students we lose, might have been saved had we given them better techniques to address this problem.

Students in the second group can also greatly benefit from a systematic process, since the techniques they discover and apply in an ad-hoc manner often (and unsurprisingly) lead to inadequate and badly designed solutions. The most applied development technique among students is probably the “first solution that comes to mind” technique. Many of our students are so happy to find any solution at all that it does not occur to them to investigate alternatives. Thus, a systematic process should not only help those students who have fundamental problems arriving at any solution at all but should improve the quality of solutions of all students.

The problem has been first identified a long time ago [9, 13, 22]. The terms *stepwise refinement* and *top-down design* were introduced in the 1970s, and the general principles appeared in some texts at the time, but few current texts do justice to the topic. Recently, some work in this area has been published. For example, Bennedsen & Caspersen argue for the necessity of teaching a systematic programming process and demonstrate ways to apply programming strategies and techniques [5]. A further discussion of related work is provided in section 5.

In this paper, we identify and describe systematic programming techniques particularly suited for novices learning object-oriented programming. More specifically, we present part of an informal, but systematic, process of decomposing a programming problem. The process is designed to be applied by beginners. This paper does not completely describe the whole process, but the largest part of it. Some additional work remains to be done.

The aim of this process is to be applied in an introductory learning and teaching situation. Thus, some of the design goals are that the process has very little bureaucratic overhead, is easy to understand, and is simple to follow.

Our hope is that the result is not only to enable more students to develop programs but also to achieve an improvement in code quality (such as readability, correctness, testing, and extendability) of student solutions.

Section 2 presents the techniques in an abstract form, followed by two examples in sections 3 and 4 that illustrate and discuss the techniques. Section 5 discusses related and future work and section 6 presents our conclusions.

2. A SYSTEMATIC PROCESS FOR NOVICES

In this section, we describe, in a general way, some simple steps that can be followed to implement classes whose intended behaviour is essentially understood.

This section is kept brief and is intended as an initial overview – we will discuss the techniques in more detail using an example in the following section.

Our techniques do not address the analysis phase or the finding of the classes from the problem domain. This may be achieved by using the noun/verb method or other simple methodologies. More likely, in very early student exercises, the teacher or the textbook will provide the class structure.

2.1 Step 1: Create the class (with method stubs)

We assume that the classes and their observable (public) functionality is understood and given, for example in the form of a Java interface or carefully written *javadoc* comments.

The first step towards implementation is to create an implementation class that implements this interface (or, if the interface is not formally given, provides methods with the intended signatures). The method implementations at this stage are stubs (i.e. minimal method bodies).

For methods that do not return values, the method body is empty. For methods with return values, the method body consists of a single return statement. The value returned is a default value (zero for numbers, null for object types, etc.).

Repeat this for every class in the project.

2.2 Step 2: Create tests

Once method stubs have been defined, test cases can be written for every method. This is commonly done using JUnit [16]. Several educational tools support JUnit testing (e.g. BlueJ and Dr. Java [18, 11]), and in environments that support recording of interactive testing, such as BlueJ [17], the existence of stubs enables the test interaction to be recorded.

Initially, most tests will fail. Details about how these tests should be developed are beyond the scope of this paper and have been discussed elsewhere [4, 15].

2.3 Step 3: Alternative representations

The next step aims at deciding on an implementation representation for the objects to be defined. The representation is defined by the instance fields of the class.

For every class, alternative representations must be considered. These can be as many as a student can think of, but must be *at least two*.

We label each of our candidate representations R_1 to R_n.

Next, we create a *Representation Evaluation Matrix* (REM). A REM is a table with one column for each candidate representation, and one row for each method in our class to be implemented (Table 1). Above the table is a short description of each alternative.

R_1: a short description of the first representation alternative here

R_2: a short description of the first representation alternative here

IMPL. EFFORT	R_1	R_2
***method1*()**	Challenging	Trivial
***method2*()**	Trivial	Hard
***method3*()**	Easy	Hard

Table 1: *Implementation effort estimation matrix*

We use this matrix to compare each method that must be implemented for each possible object representation. The comparison criteria may vary – leading to different tables – but is initially always “implementation effort”.

Table 1 shows an example of an *Effort REM*. In this table, we compare the estimated effort it takes to implement each method using a particular object representation. As values, we use a small ordered set of effort qualifiers. They are *Trivial*, *Easy*, *Average*, *Challenging*, and *Hard* (the "*TEACH* scale").

In later exercises, different REMs may be used for other criteria that are explicitly mentioned in the task specification. For example, if runtime performance is an explicitly stated goal, a *Performance REM* may be used.

It is crucial not to judge representations on imaginary requirements. Especially, performance consideration should *not* play a role in early exercises, and it should be made clear that performance is entirely irrelevant for judgement of the Effort REM. We recommend focusing on Effort REMs in early exercises.

Initially the instructor can supply the REM, but gradually the students should be responsible for filling in the REM.

Once the Effort REM is complete, we choose the representation that is judged to have the simplest overall implementation.

2.4 Step 4: Instance fields

When we have settled on one particular representation, we can refine our implementation class.

We now define the fields needed to represent the object. (The field definitions need not be complete; further fields may be added later to support method implementations. However, many important fields are derived from the implementation representation.) The field definitions may include their role (in the form of a comment) and possible constraints on their values (also in comment form).

At this stage, we also provide appropriate initialisations for the fields, either in the form of default values or by using client-supplied values. This includes at least partial implementation of the class's constructor.

2.5 Step 5: Method implementation

Step 5 is actually more than a single step: it has the form of a nested loop. The definition is:

```
while there is an unfinished method:
      Pick an unfinished method;
      Implement the method
```

The "Implement the method" step itself contains a loop:

```
while not done:
      improve the method;
      test
```

The order in which a student chooses the methods is essentially arbitrary. Our recommendation for students who are not entirely confident is to choose the method that, according to the Effort REM, is easiest to implement first.

It is easy to see that this completes the implementation. If a student successfully completes this step, the class is finished.

All the magic now lies in the "Implement method" step. This is still a large task, and needs further advice to break it down into smaller steps.

2.6 Method implementation rules

Implementing a method is potentially a large and non-trivial task. We aim to provide a process that breaks this task into smaller steps as well. This time, we cannot give a single recipe, since details of the method may vary widely. Instead, we give a set of rules that can be applied in certain cases.

Some methods, of course, consist of only a few lines of code and may be easy to write. Our rules aim at breaking all methods down into smaller chunks, until they approach the complexity of those easy-to-write methods. This is essentially a small variation of stepwise refinement [22].

At the heart of this technique is the *Mañana Principle*. The Mañana Principle says

> *When – during implementation of a method – you wish you had a certain support method, write your code as if you had it. Implement it later.*

Thus, the Mañana Principle encourages separation of concerns and the use of many small methods. We discuss an example below.

To get beginners used to the Mañana Principle, there are some more specific forms of this rule, each of which state a more concrete situation in which this principle should be used. They are:

> ***Special Case rule***: If you write code to treat a special case in your algorithm, treat the special case in a separate method.
>
> ***Nested Loop rule***: If you have a nested loop, move the inner loop into a separate method.
>
> ***Code Duplication rule***: If you write the same code segment twice, move the segment into a separate method.
>
> ***Hard Problem rule***: If you need the answer to a problem that you cannot immediately solve, make it a separate method.
>
> ***Heavy Functionality rule***: If a sequence of statements or an expression becomes long or complicated, move some of it into a separate method.

The special methods created as part of these rules are usually private methods, unless they are created in different classes – we discuss this further below.

It is important to remind students that these separate methods do not need to be implemented straight away. The calling method can be written as if the method existed. Following this, a stub for the Mañana method should be created. (If the programming environment had specific tool support for the Mañana principle, this could be automated by the IDE.)

The specific rules are initially easier to apply, because they provide concrete hints to times when they should be applied. They are, however, just instances of the Mañana Principle, and, if applied regularly, develop a coding habit that encourages the understanding and application of the principle in general.

This principle – and the specific rules – may sound abstract or complicated when presented in this theoretical form, but they are quite easy to understand when presented in the context of an example. In the next section, we discuss the development of a class defining objects for dates (time, month and year) to illustrate these techniques in practice.

3. A FIRST EXAMPLE: DATE

We demonstrate the techniques discussed above in the context of a simple programming problem: the implementation of a class representing a date.

3.1 Specification of Date

Here, we give the specification of the problem as a Java interface. It could easily be presented more informally; the introduction of interfaces is not a requirement for this process.

```
interface Date {
  /**
   * Advance the date to the next day
   */
  void setToNextDate();

  /**
   * Return a string representation of this date
   * in the format yyyy-mm-dd
   */
  String toString();
}
```

Figure 1: *Specification of Date*

3.2 Creating method stubs

The first step is to create a class for the implementation that contains method stubs. The resulting class is presented in Figure 2. (Note that we do not formally implement the interface given above to demonstrate that the use of Java interfaces is not a requirement.)

If the specification was provided in the form of a Java interface, this process is essentially mechanical and could be automated by a development environment. For students in early stages of learning, however, it might help to write this class skeleton by hand. The important thing is: simple rules can be given to guide the creation of this class.

```
/** An instance contains a date */
class Date1 {

  /**
   * Advance the date to the next day
   */
  public void setToNextDate() {
  }

  /**
   * Return a string representation of this date
   * in the format yyyy-mm-dd
   */
  public String toString() {
    return null;
  }
}
```

Figure 2: *Date class with method stubs*

3.3 Test cases

The next step is to ensure that appropriate test cases exist.

Our techniques do not necessarily prescribe a strict test-first approach, in which students create tests for all methods themselves. A viable alternative for early programming tasks is to use teacher-provided tests. The teacher may provide a test suite for the expected methods as part of the specification of the task.

The important step here is to ensure that tests exist, can be compiled, and can be executed (but do not need to pass).

In this paper, we do not present the specific tests, since the actual test development is not the main focus of this paper. The example (including the test), however, is available from a web site. The URL is given at the end of this paper.

3.4 Alternative representations for Date

The next step in our technique is to consider alternative representations (at least two).

An obvious representation for this problem is to use three integer variables *day*, *month* and *year*; we will denote this alternative R_1. An alternative representation is to count the number of days from a certain start date, say 0001-01-01; we denote this alternative R_2.

R_1 simplifies the implementation of *toString* whereas the implementation of *setToNextDate* will be more challenging, since it must deal with the special case of the last day of a month.

R_2 leads to a simple implementation of *setToNextDate* (a simple increment), whereas implementing *toString* will be hard.

The result of this analysis is the Effort REM for Date (Table 2).

***R*₁:** *Use three integers for date: day: int; month: int; year: int*

***R*₂:** *Use one integer: number of days since 1 Jan 0001*

IMPL. EFFORT	R_1	R_2
***setToNextDate*()**	Challenging	Trivial
***toString*()**	Trivial	Hard

Table 2: *Estimate of required effort to implement Date*

We choose to use R_1 for our class, since it seems to be the representation that allows for the quickest implementation of *Date*.

3.5 Instance fields of Date

Choosing R_1 as the basis for our implementation determines the instance fields. The definition of class *Date1* after adding the fields is presented in Figure 3. The method stubs are unchanged. Comments from previous code segments are left out for brevity; only comments for new methods are included from here on.

```
class Date1 {

  private int day;    // 1 ≤ day ≤ daysInMonth
  private int month;  // 1 ≤ month ≤ 12
  private int year;

  /**
   * Create a date instance with an arbitrary
   * (fixed) value.
   */
  public Date1() {
    day = 23; month = 10; year = 2006;
  }

  public void setToNextDate() {
  }

  public String toString() {
    return null;
  }
}
```

Figure 3: *Adding instance fields to Date*

3.6 Implementing the methods

The next step is to implement and test the methods. Some methods may be easy to implement in one step; *toString* in our example falls into this category. Other methods may require more work. In this case, partial solutions may be used for initial versions. Figure 4 shows our class after implementing function *toString* and a first, naïve version of *setToNextDate*.

```
class Date1 {
  private int day;    // 1 ≤ day ≤ daysInMonth
  private int month;  // 1 ≤ month ≤ 12
  private int year;

  public Date1() {
    day = 23; month = 10; year = 2006;
  }

  public void setToNextDate() {
    day = day + 1;
  }
  public String toString() {
    return year + "-" + month + "-" + day;
  }
}
```

Figure 4: *Naïve implementation of Date*

This partial solution is indeed a very naïve implementation. Nevertheless, we might claim that the *setToNextDate* method is 97% correct since it works correctly in 353 out of 365 cases! In some sense, we are very close to a full solution, and if the class is part of a larger system, it can now be used (as a test stub) by other parts of the system.

Incrementing the field *day* might violate the representation invariant, and in this special case the above implementation of *setToNextDate* fails to work properly. We have to check for this special case and handle it appropriately. For simplicity, we temporarily assume 30 days in every month.

In the special case where *day* after being incremented exceeds the number of days in the month, we must set *day* to 1 and increment field *month*. Following our *Special Case* rule from section 2, we deal with this special case by introducing a new private method, *checkDayOverflow*. Figure 5 shows the resulting code.

```
class Date1 {
  private int day;    // 1 ≤ day ≤ daysInMonth
  private int month;  // 1 ≤ month ≤ 12
  private int year;

  public Date1() {
    day = 23; month = 10; year = 2006;
  }

  public void setToNextDate() {
    day = day + 1;
    checkDayOverflow();
  }

  /**
   * Check for special case where day > daysInMonth;
   * in that case, set day to 1 and add 1 to the month
   */
  private void checkDayOverflow() {
    if ( day > 30 ) {
      day = 1;
      month = month + 1;
    }
  }

  public String toString() {
    return year + "-" + month + "-" + day;
  }
}
```

Figure 5: *Partial implementation of Date*

Now, incrementing the variable *month* might also violate the representation invariant; this special case is handled similarly by introducing a new private method *checkMonthOverflow*, which is called after incrementing *month*. Except for the assumption of 30 days in every month, the method is now finished.

To finish our implementation, we have to replace the literal 30 with the correct number of days in every month. Here, the *Mañana Principle* comes in again, this time in the form of the *Hard Problem* rule: If we need some information that we do not have, we pretend we have a method that gives us the answer. Thus, we just assume a method *daysInMonth* that does exactly what we need. We do not worry about the implementation of this method now; it is postponed until later.

The new version of the *checkDayOverflow* method is shown in Figure 6.

```
private void checkDayOverflow() {
  if ( day > daysInMonth() ) {
    day = 1;
    month = month + 1;
    checkMonthOverflow();
  }
}
```

Figure 6: *Final version of checkDayOverflow()*

This method will not compile until we provide a method stub for *daysInMonth*. The stub, in this case, should not return a zero, but should return 30 – the approximation we have used previously.

The most important thing at this stage is that we have explicitly separated two independent problems: the correct use of this method and the implementation of the method. Separating these problems makes each half easier to solve.

Since our *checkDayOverflow* method is now complete, we might now proceed to implement *checkMonthOverflow*. In the general case, implementing one method may generate several other methods via the Mañana Principle, which can then be gradually implemented.

For our example, implementing the *daysInMonth* method is the last thing that is missing. To calculate the number of days in the current month, we declare a local array variable in this method to hold the number of days per month (with 28 days for February), and the method returns the number of days in the current month by looking up the number in the array. This brings us almost to the finishing line: the implementation now works, except for the special case where the current year is a leap year ("99.93% correctness").

As previously, we treat a special case by introducing a new private method to deal with it. In this case, we introduce a boolean method *isLeapYear* that returns true if the current year is a leap year. The implementation of this method is a straightforward implementation of the leap year rule: a year is a leap year if the year is divisible by 4 but not by 100 or if it is divisible by 400.

The hardest part of this calculation is the check whether a number can be divided by another so, again, following the Mañana Principle, we use a method *divides* that gives us the result, and then we implement that method later.

The complete implementation of our Date class including these methods is shown in Figure 7.

```
class Date1 {
  private int day;   // 1 ≤ day ≤ daysInMonth
  private int month; // 1 ≤ month ≤ 12
  private int year;

  public Date1() {
    day = 23; month = 10; year = 2006;
  }

  public void setToNextDate() {
    day = day + 1;
    checkDayOverflow();
  }
```

```
    private void checkDayOverflow() {
      if ( day > daysInMonth() ) {
        day= 1;
        month= month + 1;
        checkMonthOverflow();
      }
    }

    /**
     * Check for special case where month > 12;
     * in that case, set month to 1 and add 1 to the year
     */
    private checkMonthOverflow() {
      if ( month > 12 ) {
        month= 1;
        year= year + 1;
      }
    }

    /**
     * Return the number of days in the current month
     */
    private int daysInMonth() {
      // month:            1  2  3 ... 12
      int[] daysInMonth = {31,28,31,...,31};

      int result = daysInMonth[month-1];
      // special case: February in a leap year
      if ( month == 2 && isLeapYear() ) {
          result= result + 1;
      }
      return result;
    }

    /**
     * Return true iff the current year is a leap year
     */
    private boolean isLeapYear() {
      return (divides(4, year) && !(divides(100, year))
             || divides(400, year);
    }

    /**
     * Return true iff a divides b
     */
    private boolean divides(int a, int b) {
      return b % a == 0;
    }

    public String toString() {
      return year + "-" + month + "-" + day;
    }
  }
```

Figure 7: *Complete implementation of Date*

3.7 Discussion of Date implementation

The above development of a class implementing *Date* demonstrates the application of the techniques set out in section 2. The most relevant observation is that every step is broken into small, manageable chunks.

Some of the steps in our technique are fairly easy to learn (creating method stubs, defining the instance fields after deciding on a representation); others require much practice (creating tests, implementing methods).

The detailed discussion of the method implementation has shown that – at least in this case – the harder tasks can also be broken down into small parts. This technique can be applied to any implementation of a method.

4. A SECOND EXAMPLE: CALENDAR

Our second example is one that consists of two classes: a class *Appointment* to record personal appointments, and a class *Calendar* class to hold the appointments. We discuss this example to illustrate some additional points (while mostly skipping those parts that we have already covered above).

4.1 Specification of Calendar

Again, we give the specifications of *Calendar* and *Appointment* in the form of Java interfaces (Figure 8). Alternatively, they may be provided as a UML diagram or informally as a list of required methods.

To abstract from the details of actual time and date information, start time and duration of appointments are represented as integer values. For example, we may have an appointment that starts at 6 and has duration 5 and another that starts at 9 and has duration 1.

```
/** A calendar that can hold appointments */
interface Calendar {

  /** Add appointment a to this calendar */
  void add (Appointment a);

  /** Remove appointment a from this calendar */
  void remove(Appointment a);

  /** Return the first free slot of
      duration d at or after time s */
  Appointment getFirstAvailable(int s, int d);
}

/** An appointment */
interface Appointment {
  int getStartTime();
  int getDuration();
  String getDescription();

  /** collidesWith is true iff this and a overlap */
  boolean collidesWith(Appointment a);
}
```

Figure 8: *Specification of Calendar and Appointment*

4.2 Creating method stubs and test cases

For this example, we skip the discussion of method stub creation and test case definitions, since the process is essentially the same as in the first example. Instead, we jump straight ahead to the discussion of representation alternatives.

4.3 Alternative representations for Calendar

As always, before embarking on implementing a specification, alternative representations must be considered. This must be done for each class. In this discussion, we consider only the implementation of class *Calendar* and ignore class *Appointment*.

One representation of a calendar is an unordered set of appointments; we will denote this representation R_1. An alternative representation is a sorted set of appointments; we will denote this representation R_2.

For both R_1 and R_2, implementation of *add* and *remove* is trivial (a delegation to the similar *Set* method).

R_1 simplifies the programming task of *getFirstAvailable* (at the expense of runtime efficiency). The method can be implemented as a simple linear search where each repetition requires another repetition over the set of appointments (i.e. *getFirstAvailable* will be $O(n^2)$ where n denotes the number of appointments in the calendar), but the required programming effort is manageable.

We know that R_2 allows for a more efficient implementation (*getFirstAvailable* will have time complexity $O(log(n) + m)$ where m denotes the number of collisions until a free slot is found), but clearly this is at the expense of a considerable increase in the complexity of the programming task. R_2 requires the definition of a total ordering (natural order) of appointments as well as fluency with the *SortedSet* interface, which is an order of magnitude more complex than the more straightforward *Set* interface.

The result of the analysis is summarized in the Effort REM for Calendar (Table 3).

***R_1:** Use unordered set to store appointments*

***R_2:** Use a sorted set to store appointments*

IMPL. EFFORT	R_1	R_2
***add*()**	Trivial	Trivial
***remove*()**	Trivial	Trivial
***getFirstAvailable*()**	Average	Challenging

Table 3: *Estimate of required effort to implement Calendar*

We choose R_1 because it clearly allows for the simplest implementation of *Calendar*.

4.4 Implementation of Calendar

Having decided upon a representation of a calendar (i.e. having defined the representation invariant), we have decoupled the three subtasks of implementing the methods of the *Calendar* interface. This is an instance of the principle *separation of concerns* – Dijkstra's mantra and primary instrument of thought [10, pp. 209-217].

Having decided upon a set representation, where we are free to choose any concrete class that implements the *Set* interface, we can make a partial implementation of *Calendar* (Figure 9).

```
/** A calendar with appointments */
class CalendarUnsorted {
  private Set<Appointment> appointments;

  /** Create an empty calendar */
  public CalendarUnsorted() {
    appointments = new HashSet<Appointment>();
  }

  public void add(Appointment a) {
    // FixMe
  }

  public void remove(Appointment a) {
    // FixMe
  }

  public Appointment getFirstAvailable(int s, int d) {
    return null;  // FixMe
  }
}
```

Figure 9: *Partial implementation of Calendar*

This is indeed a very small step toward a complete implementation of *Calendar*, but it compiles and maybe even makes a few test cases run. For novices (and indeed for others), making small successful steps toward the goal is a rewarding and satisfying way of developing software.

Using a set as the representation of a calendar allows for a straightforward implementation of each of the three methods independently of each other.

Methods *add* and *remove* can be implemented simply by delegating the method call to the similar *Set* methods. Adding this to the initial implementation gives the next two methods of our solution to the problem (Figure 10).

```
/** Add appointment a to this calendar */
public void add(Appointment a) {
  appointments.add(a);
}

/** Remove appointment a from this calendar */
public void remove(Appointment a) {
  appointments.remove(a);
}
```

Figure 10: *Implementation of methods add and remove*

Method *getFirstAvailable* is somewhat more complicated. It can be implemented as a linear search by successively checking for availability of appointment slots (s, d), (s+1, d), (s+2, d), ... until an available appointment slot is found ((s, d) denotes the appointment with start time s and duration d). A first attempt at implementing *getFirstAvailable* is shown in Figure 11.

```
/** Return the first free slot of
   duration d at or after time s */
public Appointment getFirstAvailable(int s, int d) {
  Appointment result;
  boolean available = false;
  do {
    result = new Appointment(s++, d);
    // set 'available' such that available holds iff
    // result does not collide with any appointment
    // already in the calendar
  } while ( !available );
  return result;
}
```

Figure 11: *Partial implementation of method getFirstAvailable*

It is obvious that the calculation of *available* involves an iteration over the appointments in the calendar, and consequently a nested loop. One of our rules for method implementation is the *Nested Loop* rule: *use a new private method to unfold nested loops*. Instead of proceeding with development of the inner loop, we define a new private method for the calculation of the boolean expression *available* as defined above. We name the method *isAvailable* (Figure 12).

```
/** Return true iff Appointment a does not collide
   with any appointments in this calendar */
private boolean isAvailable(Appointment a) {
  return true; // FixMe
}
```

Figure 12: *Specification of method 'isAvailable'*

With method *isAvailable* to serve us, we can now finish the loop body of method *getFirstAvailable* (Figure 13).

```
public Appointment getFirstAvailable(int s, int d) {
  Appointment result;
  boolean available = false;
  do {
    result = new Appointment(s++, d);
    available = isAvailable(result);
  } while ( !available );
  return result;
}
```

Figure 13: *Implementation of method getFirstAvailable*

Removing the unnecessary variable *available* gives the final version of *getFirstAvailable* (Figure 14).

```
public Appointment getFirstAvailable(int s, int d) {
  Appointment result;
  do {
    result = new Appointment(s++, d);
  } while ( !isAvailable(result) );
  return result;
}
```

Figure 14: *Improvement of method getFirstAvailable*

(Side note: we assume here an unbounded calendar, i.e. there will always be an available slot, and the loop will always terminate.

For a bounded calendar, we would have to add a test for reaching the end of the calendar in the loop condition. This would, of course, again involve the Mañana Principle, and we would use a method a*tCalendarEnd*.)

Now we only need to implement the new private method *isAvailable*. As mentioned earlier, this can be done by a repetition checking for collision between *a* and each appointment *i* in the set (Figure 15).

```
private boolean isAvailable(Appointment a) {
  for ( Appointment i : appointments ) {
    if ( a.collidesWith(i) ) return false;
  }
  return true;
}
```

Figure 15: *Implementation of method 'isAvailable'*

This completes the development of an implementation of *Calendar* based on R_1. The development of an implementation of *Appointment* is left to the reader.

4.5 Discussion of development of Calendar

The discussion of the calendar example has shown the application of the *Nested Loop* rule. When consistently applying this rule, the code remains considerably simpler (and easier to understand for beginners) than an alternative using a nested loop.

In this example, all the methods introduced through our rules were private methods in class *Calendar*. In the general case, this does not always have to be the case. If, for instance, class *Appointment* did not have a method *collidesWith*, this method may have been introduced by applying the *Hard Problem* rule while implementing the calendar's *isAvailable* method.

In early exercises, we usually start with problems where the methods that naturally develop are in the same class. This can then – a bit later – be extended and linked to a discussion of responsibility-driven design, and the question which class should provide a new, required method.

5. RELATED AND FUTURE WORK

Numerous software engineering topics relate to our efforts of identifying a systematic programming process for novices. We will discuss these topics in turn.

Stepwise refinement. More than 35 years ago Dijkstra and Wirth identified the need for a constructive and systematic approach to programming – not only for novices but for the community as a whole [8, 9, 22, 23]. Our work builds on the work of Wirth and Dijkstra but concentrates on a specialized process for novices learning object-oriented programming.

Programming methodology. In the early seventies Dijkstra formalized his ideas about structured programming and developed a methodology for systematic construction of programs using functional specifications (pre and post conditions) and loop invariants to drive the development process [10]. In continuation of Dijkstra's seminal work, Back developed a refinement calculus [1, 2] while Gries and others produced text books based on the methodology (e.g. [6, 14, 20]). Our approach differs from this work by being a formally-based but informally-practiced approach to systematic program development.

Responsibility-driven design. The Mañana Principle is related to responsibility-driven design [21]. In this paper, we apply the Mañana Principle only for functional decomposition, but even here it reveals its relationship to responsibility-driven design (the nested loop rule factors a part of the program to a separate method with the responsibility of implementing the nested loop functionality).

Refactoring. During a programming session, it is inevitable that decisions made earlier in the session need to be altered at a later stage. Realizing and learning that this is the rule rather than the exception helps novice programmers come to terms with the fact that programming is not a linear process. This is refactoring-in-the-small [12]. An interesting aspect here is programming environment support: in a similar manner in which refactoring is now commonly supported in development environments, the Mañana Principle could easily be supported by automating the creation of method stubs whenever a new private method is introduced.

XP and agile software development. Extreme programming and agile software development covers many aspects of software engineering [3, 19]; two of the basic principles are: "*Take small steps*" and "*Always do the simplest thing that will work*". We use these principles as guidelines for choosing among several possible implementations of an abstraction (a method specification or an interface) and for the process of implementing it. They are wise guidelines for novices as well as experts.

Test-driven development. The strategy of test-driven development [4, 15] relates closely to step 2 in our process: Create tests. Test-driven development is gaining increased recognition, and it is beneficial to apply this strategy with novices for several reasons (e.g. force a consumer view as well as producer view of program components). But it is not necessary to adopt test-driven development in order to apply our process; instead test cases can be provided as part of the specification of a programming task.

In this paper, we have concentrated on a part of the process where decomposition generates support methods. This part is not exclusively object-oriented and is equally applicable to functional and procedural languages, even though we have presented it in the context of an object-oriented language. Future work includes extending the set of rules that unfolds the Mañana Principle to cover cases of decomposition that generate not only new methods but also new classes (or interfaces).

A second direction of future work will focus on investigating and designing tool support for the process in general and in particular for the Mañana Principle.

6. CONCLUSIONS

We have argued that we need to teach novices about the process of software development in order to enable them to follow organised steps to move toward a solution to a problem, and that we must treat software development explicitly as a process that is carried out in stages and small steps, rather than the writing of a single, monolithic solution.

Furthermore we have identified and described principles and systematic programming techniques particularly suited for novices learning object-oriented programming. To complement the principles and techniques, we have presented an informal but systematic process designed to be applied by beginners. Through two examples we have demonstrated the application of the process.

The process we propose is a carefully down-scaled version of a full and rich software engineering process. By using it we hope to achieve two things: To help novice programmers learn faster and better while at the same time laying the foundation for a more thorough treatment of the various aspects of a software engineering process.

The complete programs discussed in this paper are available at www.daimi.au.dk/~mec/oopsla2006/.

7. Acknowledgement

It is a pleasure to thank David Gries for numerous careful comments and improvements to an earlier version of the paper.

References

[1] Back, R.-J., *On the Correctness of Refinement Steps in Program Development*, PhD thesis, Department of Computer Science, University of Helsinki, 1978.

[2] Back, R.-J., *Refinement Calculus: A Systematic Introduction*, Springer-Verlag, 1998.

[3] Beck, K. *Extreme Programming Explained: Embrace Change*, Addison-Wesley, 2000.

[4] Beck, K., *Test-Driven Development by Example*, Addison-Wesley, 2003.

[5] Bennedsen, J. and Caspersen, M.E., ”Revealing the Programming Process”, *Proceedings of the thirty-sixth SIGCSE Technical Symposium on Computer Science Education*, St. Louis, Missouri, USA, 2005, pp. 186-190.

[6] Cohen, E., *Programming in the 1990's*, Springer-Verlag, 1990.

[7] Dahl, O.-J., Dijkstra, E.W. and Hoare, C.A:R., *Structured Programming*, Academic Press, 1972.

[8] Dijkstra, E.W., “A Constructive Approach to the Problem of Program Correctness”, BIT 8, 1968.

[9] Dijkstra, E.W., “Notes on Structured Programming”, EWD 249, 1969. In [7].

[10] Dijkstra, E.W., *A Discipline of Programming*, Prentice-Hall, 1976.

[11] Dr. Java, http://drjava.org, Accessed 12 July 2006.

[12] Fowler, M., *Refactoring: Improving the Design of Existing Code*, Addison-Wesley, 1999.

[13] Gries, D., “What Should We Teach in an Introductory Programming Course”, *Proceedings of the fourth SIGCSE Technical Symposium on Computer Science Education*, 1974, pp. 81-89.

[14] Gries, D., *The Science of Programming*, Springer-Verlag, 1981.

[15] Hunt, A. and Thomas, D., *Pragmatic Unit Testing in Java with JUnit*, The Pragmatic Programmers, 2003.

[16] JUnit. www.junit.org.

[17] Kölling, M., *Unit Testing in BlueJ*. www.bluej.org/tutorial/testing-tutorial.pdf. Accessed 12 July 2006.

[18] Kölling, M., Quig, B., Patterson, A. and Rosenberg, J., “The BlueJ System and its Pedagogy”, Computer Science Education, Vol. 13, No. 4, 2003, pp. 249-268.

[19] Martin, R.C., *Agile Software Development: Principles, Patterns, and Practices*, Prentice-Hall, 2003.

[20] Morgan, C., *Programming from Specifications*, Prentice-Hall, 1990. http://users.comlab.ox.ac.uk/carroll.morgan/PfS/ Accessed 12 July 2006.

[21] Wirfs-Brock, R. and McKean, A., *Object Design: Roles, Responsibilities, and Collaborations*, Addison-Wesley, 2003.

[22] Wirth, N., “Program Development by Stepwise Refinement”, *Communications of the ACM*, Vol. 14, No. 4, April 1971, pp. 221-227.

[23] Wirth, N., *Systematic Programming*, Prentice-Hall, 1973.