STREAM: A First Programming Process

MICHAEL E. CASPERSEN
Department of Computer Science, Aarhus University, Denmark
and
MICHAEL KÖLLING
Computing Laboratory, University of Kent, United Kingdom

Programming is recognised as one of seven grand challenges in computing education. Decades of research have shown that the major problems novices experience are composition-based – they may know what the individual programming language constructs are, but they do not know how to put them together. Despite this fact, text-books, educational practice, and programming education research hardly address the issue of teaching the skills needed for systematic development of programs.

We provide a conceptual framework for incremental program development, called Stepwise Improvement, which unifies best practice in modern software development such as test-driven development and refactoring with the prevailing perspective of programming methodology, stepwise refinement. The conceptual framework enables well-defined characterizations of incremental program development.

We utilize the conceptual framework to derive a programming process, STREAM, designed specifically for novices. STREAM is a carefully down-scaled version of a full and rich agile 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 more advanced aspects of software engineering. In this paper, two examples demonstrate the application of STREAM.

The STREAM process has been taught in the introductory programming courses at our universities for the past three years and the results are very encouraging. We report on a small, preliminary study evaluating the learning outcome of teaching STREAM. The study indicates a positive effect on the development of students' process competences.

Categories and Subject Descriptors: D.2.1 [**Software Engineering**]: Requirements/Specifications—*Methodologies*; D.2.3 [**Software Engineering**]: Coding Tools and Techniques—*Object-oriented programming*, *Structured programming*, *Top-down programming*; D.2.10 [**Software Engineering**]: Design—*Methodologies*; K.3.2 [**Computers and Education**]: Computer and Information Science Education—*Computer science education, Curriculum*

General Terms: Design, Experimentation, Human Factors

Additional Key Words and Phrases: Stepwise improvement, stepwise refinement, programming methodology, programming process, agile methods, test-driven development, refactoring, programming education

Authors' addresses: M.E. Caspersen, Department of Computer Science, Aarhus University, IT-parken, Aabogade 34, DK-8200 Aarhus N, Denmark; email: mec@cs.au.dk; M. Kölling, Computing Laboratory, University of Kent, Canterbury, Kent, CT2 7NF, United Kingdom; email: mik@kent.ac.uk.

1. INTRODUCTION

Teaching beginners to program is hard. Many teachers agree that this always has been, and remains, a very difficult task that has no quick solutions. What is especially worrying, though, is that the task of teaching programming did not become easier over the last decades. Initially, one might have claimed that much of the difficulty could be related to the relative immaturity of our discipline: since we do not have the many decades of teaching experience other subjects have gone through, some initial problems could be expected. This might have been viewed as teething problems.

The worrying fact, however, is that this does not seem to be true. Many teachers now agree that the teaching of programming has recently become more difficult, rather than easier, as it should have been had maturity of the discipline been the significant factor. Instructors now look back at the days of teaching Pascal, and mourn the relative simplicity of teaching in those days.

Some attribute the increased level of difficulty to intrinsic characteristics of new paradigms used in teaching, such as object orientation, some attribute it rather to our relatively naïve treatment of object orientation and lack of experience with this paradigm. Others again blame increasingly complex tools and infrastructure. One thing that seems certain is that the number of concepts covered in introductory programming courses has grown. Group work, GUI programming, testing, debugging, concurrency, correctness, patterns, refactoring, and many other topics are now regularly found in first year courses, while they were much less prominent a decade ago.

We can easily accept that most or all of the above contribute to the problems that teachers currently experience. However, in our view, one of the main reasons for the increasing problems is a lack of recognition of *process* as an important topic in introductory programming courses.

Several aspects of programming courses have changed dramatically over the last decade. One significant addition to the traditional discussion of algorithms and data structures at the heart of the course is the coverage of real world ("large") software systems. Issues such as code quality, maintainability, extendibility, testing, modularization, group work, etc., have gained in prominence. With this, the programs under investigation are often larger (not always written by the students alone – often code is provided by the teacher to be corrected, modified, or extended).

With the increase in size and complexity of the artefacts being worked on by students, the concept of a development process has become increasingly important. This is often not clearly reflected in the academic content of introductory courses.

While software development processes are well established among professional programmers, very little is done to address process in introductory programming courses. Most textbooks, and with it most courses, focus on presenting programming language constructs, programming concepts, and computer programs. “Program”, in short, is treated as a noun, not as a verb.

A typical pattern of introducing material is the presentation of a problem, followed by the presentation of a program that solves the problem, followed by a detailed discussion of the language constructs and concepts or algorithms involved.

This pattern of introducing material creates – unintentionally – the illusion that these programs were developed (by an expert programmer) in a single step from the problem formulation. The fact that we all start with incomplete and incorrect programs when we start addressing a problem, which we then slowly modify to improve and extend our implementation until we arrive at an acceptable solution, seems to be swept under the carpet as if it was an embarrassing secret that must not be mentioned.

While the ultimate solution to the problem is explained in detail, the *process* – how we go about developing the solution – is often entirely neglected in beginners’ courses.

Developing software is, by its very nature, always a process, whether we are formally aware of it or not. 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 can and who discover their own implicit process.

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.

Another problem with hiding the development process is the misleading impressions students develop about the nature of software development, which can lead to a discouraging experience. When going through a rather normal activity of struggling with their implementation, chasing bugs, getting stuck, and tearing their hair out before finishing an assignment, they often think that they are inadequate programmers because they find software development difficult. It would help them to know that most developers as a matter of course go through exactly the same experience.

The problem with teaching a formal process from the start in an introductory course is the overhead in time and complexity this imposes on the course. Most software processes described in the literature have been developed for professional software engineers working on large systems in teams. Lately, agile processes have become more popular, which are described as "light weight" – they remove some of the overhead associated with more traditional processes. However, they are still not easy to follow for new students. An even simpler process is needed.

Stepwise Improvement is a model of program development that unifies elements of Stepwise Refinement and agile methods. It provides a conceptual framework which can be used to derive a simple, semi-formal programming process. This process, named *STREAM*, is simple enough to be taught to beginners. It provides clear guidance through development steps while its overhead is kept low enough to be integrated into many introductory courses.

Stepwise Improvement itself is an instance of an approach to programming which we call *Growing Islands of Functionality*.

In this paper, we will first briefly introduce the ideas of Growing Islands of Functionality and Stepwise Improvement. We will then present the main elements of the STREAM process. We present this process at a sufficient level of detail for it to be used directly in teaching situations by interested readers.

We do not mean to suggest that the introduction of a semi-formal process for beginners will somehow remove all problems that students have in developing software. Providing a process, even one that is light weight and reasonably precise, will still leave plenty of grey areas that students have to struggle with. Some decisions to be made will still be difficult. Some students will still have problems, but we believe that teaching a process can help with *some* problems that students face and result in some meaningful improvement.

2. GROWING ISLANDS OF FUNCTIONALITY

In the 1970s and 1980s, *Structured Programming* (a.k.a. *Stepwise Refinement* and *Top-Down Development*) was the dominant development paradigm. In this model, the development of a computer program starts at the highest level of abstraction, which is repeatedly refined until its level of detail reaches that of an available machine for execution (Figure 1).

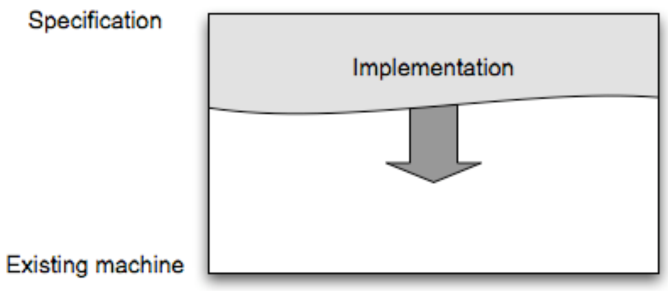

Figure 1. Top-down programming

This idea, while compelling at a theoretical level, does not usually work very well in practice for non-trivial systems [Caspersen 2007, p.90-91]. Experience over time has shown that it presents a useful way to reason about the structure of a program, but is problematic as a development process in practice, since it requires the developer to have the full (abstract) solution in mind before commencing the implementation.

The widespread acceptance of object-oriented programming in the early 1990s saw increased prominence of the idea of *Bottom-Up Development* (Figure 2).

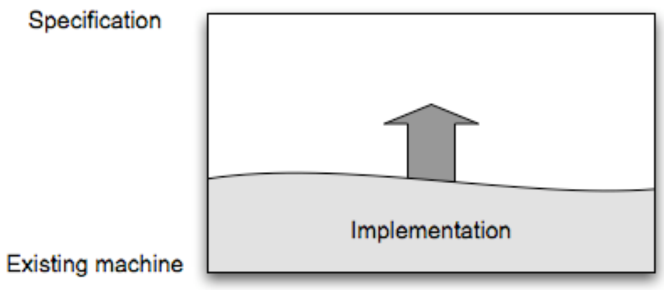

Figure 2. Bottom-up programming

In this model, lower level components are developed first, with higher level functionality slowly being built on top of the low-level modules. The concrete machine is slowly built up towards the required functionality.

While the top-down development process results in partial programs during the development that do not compile, the bottom-up process produces programs that compile, but do not (yet) provide required functionality.

The model of *Growing Islands of Functionality* is based on an approach that initially implements small subsets of functionality. This functionality is implemented completely (from the user interface down to the available machine), but it can be very "thin" (Figure 3). The overall available functionality is then gradually increased by growing the available islands of implementation.

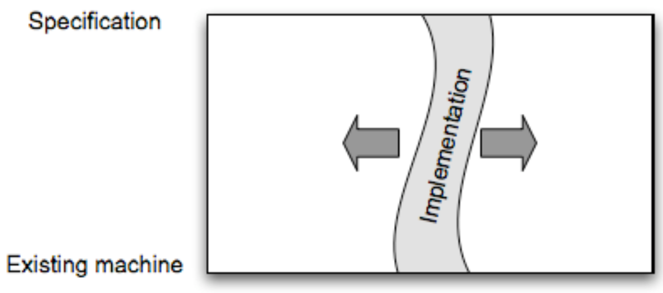

Figure 3. Islands of functionality

Systems developed according to this model reach compilable and functional stages early and often. Most agile methodologies are compatible with this model.

The model of Growing Islands of Functionality and the notion of Stepwise Improvement are inspired by a qualitative study of the programming practice of experts [Caspersen 2007, pp. 81-85] as well as recent developments in best-practice in modern software development.

Computing is a vocational discipline, which means that a large group of professionals are developing and expanding the practices of the discipline, in parallel with academia. Examples of recent major contributions to the programming practices primarily offered by people outside academia are design patterns and frameworks, extreme programming, refactoring, agile development, and test-driven development [Beck 2000, Beck 2003, Cockburn 2002, Fowler 1999, Gamma 1995, Martin 2003]. Further references to this are presented in section 8 on related and future work.

3. STEPWISE IMPROVEMENT

In traditional stepwise refinement [Dijkstra 1969, Wirth 1971, Back 1978, Morgan 1990, Back 1998], programming is regarded as the one-dimensional activity of refining abstract programs (i.e. programs containing non-executable specifications) to concrete programs (i.e. executable code) through a series of behaviour preserving program transformations. The fundamental assumption of traditional stepwise refinement is that the complete specification, the requirements, is known and addressed from the outset. Algorithmically, stepwise refinement can be characterised as follows (*req* is the requirements, *impl* the implementation, and *abstract* means ‘not executable’):

```
impl:= abstract solution (that meets req);
do impl is abstract -> refine impl od
```

Figure 4. Stepwise refinement: programming as a one-dimensional activity

Typically, stepwise refinement is described as a strict top-down process of programming.

Programming by Stepwise Improvement [Caspersen 2007], on the other hand, is characterised as an explorative activity of discovery and invention that takes place in the three-dimensional space of *extension*, *refinement*, and *restructuring*. Extension is the activity of extending the specification to cover more (use-) cases; refinement is the activity of refining abstract code to executable code to meet the current specification; and restructure is the activity of improving non-functional aspects of a solution without altering its observable behaviour, such as design improvements through refactoring, efficiency optimisation, or portability improvements. Algorithmically, stepwise improvement can be characterised as follows (*spec* is the current specification that the implementation is supposed to meet):

```
spec:= the empty specification;
impl:= empty (do nothing);
do spec does not imply req -> extend spec
[] impl does not meet spec -> refine impl
[] impl needs restructuring -> restructure impl
od
```

Figure 5. Stepwise improvement: programming as a three-dimensional activity

Stepwise Improvement captures the Islands of Functionality model. It can be illustrated as a movement graph in a three-dimensional space. Figure 6 illustrates a development scenario consisting of five activities that in order are *refine*, *extend*, *refine*, *restructure*, and *refine*.

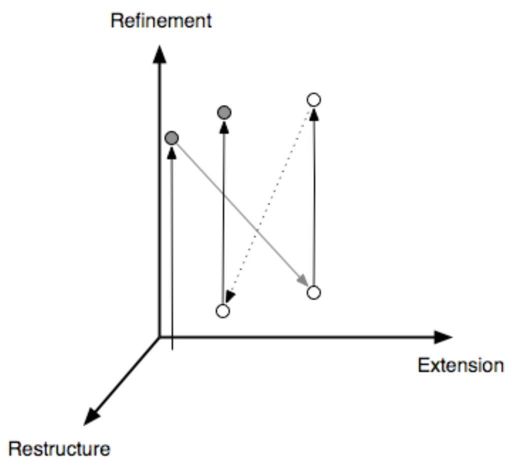

Figure 6. Stepwise improvement: moving through three-dimensional space

Different software development methodologies put different emphasis on the order of the activities described in Stepwise Improvement. Waterfall methods are characterized by a strict separation of the activities (extension first, refinement and restructure later) whereas agile methods allow a much more fine-grained interleaving of the activities.

The traditional approach to programming education is to "invite" the students for a random walk in the 3D space. Students are shown a few finished programs and told to solve programming problems on their own. Our approach to programming education offers an alternative to random walks. Instead, we suggest guided tours. By providing guidance and scaffolding[1] with respect to all dimensions involved, we can ensure that students exercise the important aspects of programming while keeping the cognitive load within the bounds where learning outcome is optimized. Our primary means of providing guidance with respect to extension (incremental development) is through the structure of the teaching material (textbook, exercises and assignments, and videos) and an apprentice-based teaching approach. Guidance with respect to refinement is provided through a carefully designed novice's process of object-oriented programming. The process, which we call STREAM, is described in the next section.

4. STREAM: 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.

The name STREAM is an acronym for the six steps that make up this process: Stubs, Tests, Representation, Evaluation, Attributes, Methods.

4.1 Step 1: Stubs (Create a Skeleton Class with Method Stubs)

We assume that the classes and their observable (public) functionality are understood and given, for example in the form of a Java interface and 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. The resulting project, which compiles but does noting when executed, is called *a walking skeleton*.

4.2 Step 2: Tests (Ensure that Tests are Available)

Once method stubs have been defined, test cases can be written for every method. This is commonly done using JUnit [JUnit 2009]. Several educational tools support JUnit testing (e.g. BlueJ and Dr. Java [Kölling 2003, Dr. Java]), and in environments that support recording of interactive testing, such as BlueJ [Kölling 2009], 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 [Beck 2003, Hunt 2003]. In early teaching examples, the tests may be provided by the teacher.

---

[1] Scaffolding is a term from cognitive apprenticeship describing support provided by the master to apprentices in order to carry out some given task: "this can range from doing almost the entire task for them to giving them occasional hints on what to do next" [Collins 1991, p. 24].

4.3 Step 3: Representations (Consider Alternative Representations)

The next steps aim 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*. The alternative representations should be briefly described in writing.

We label our candidate representations R_1 to R_n.

4.4 Step 4: Evaluation (Evaluate the Alternative Representations)

Next, we evaluate each representation with respect to difficulty of implementation. To do this, 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 the short description of each alternative.

R_1: a short description of the first representation alternative here

R_2: a short description of the second representation alternative here

IMPL. EFFORT	***R1***	***R2***
***method*$_1$()**	Challenging	Trivial
***method*$_2$()**	Trivial	Hard
***method*$_3$()**	Easy	Hard

Table 1: Representation Evaluation Matrix (Effort)

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.

It is also worth noting that the emphasis on minimising implementation effort does not at all mean that we intend to encourage "quick-and-dirty" implementations, and thus are prepared to compromise implementation quality. On the contrary! Usually, a simpler implementation (one that requires less effort to understand and implement) will include fewer bugs, and therefore be of higher quality. This focus merely intends to avoid premature optimisation efforts, and represents a straightforward application of Dijkstra's principle of "Separation of Concerns".

Initially the instructor can supply the representation alternatives and the REM, but gradually the students should be responsible for finding representation alternatives and filling in the REM. (This is a good group exercise.)

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

4.5 Step 5: Attributes (Define 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.

4.6 Step 6: Methods (Implement the Methods)

Step 6 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
```

In the latter loop, 'improve' means one of three things: Extend (the specification), refine (the implementation), or restructure (the implementation).

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 the method" steps. This is still a large task, and needs further advice to break it down into smaller steps.

4.7 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 [Dijkstra 1969, Wirth 1971].

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 (day, month and year) to illustrate these techniques in practice.

5. 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.

5.1 Specification of Date

Here, we give the specification of the problem as a Java interface (Figure 7). 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 7. Specification of Date

5.2 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 8. (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 represents 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 8. Date class with method stubs

5.3 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. There is, however, nothing special about these tests, and any standard test-first strategy can be applied.

5.4 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 store the number of days from a certain start date, say 0001-01-01; we denote this alternative R_2. (In in-class discussions, students typically come up with more creative alternatives, e.g. representing the month or the complete date as a string.)

5.5 Evaluation of Alternative Representations for Date

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	***R1***	***R2***
***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*.

5.6 Attributes

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 9. 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 9. Adding instance fields to Date

5.7 Implementing the Methods of Date

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 10 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 10. 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 11 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 11. 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 12.

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

Figure 12. 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 13.

```
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 13. Complete implementation of Date

5.8 Discussion of Date Implementation

The above development of a class implementing *Date* demonstrates the application of the techniques set out in section 4. 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.

6. A SECOND EXAMPLE: A SIMULATION

Our second example is the core of a simulation, in which actors move in a two-dimensional bounded world. The world is divided into a limited set of discrete locations, so that the position of the actors in the world can be specified as a coordinate pair in a grid (*row*, *column*).

For the purpose of this discussion, we examine a fairly simple version of such a simulation. The principles discussed here are, however, generally applicable.

We examine two classes that specify this application: a class *Actor* that represents the actors that live and act within the simulated world, and a class *Simulator*, the main class that holds and controls the collection of actors. We discuss this example to illustrate some additional points (while mostly skipping those parts that we have already covered above).

6.1 Specification of Simulator

Again, we give the specifications of *Simulator* and *Actor* in the form of Java interfaces (Figure 14). Alternatively, they may be provided as a UML diagram or informally as a list of required methods.

```
/** A simulator that manages actors in a 2D world.
    Locations in the world are specified by row and
    column number. */
interface Simulator {

  /** Add Actor a to this simulation */
  void add(Actor a, int row, int col);

  /** Return the actor at location specified by (row,col).
      Return null if there is no actor at the location. */
  Actor getActorAt(int row, int col);

  /** Let all actors act once */
  void act();

  /** Display a representation of the current world state
      to standard out */
  void display();
}

/** An actor in the world */
interface Actor {
  void act();
}
```

Figure 14. Specification of Simulator and Actor

6.2 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.

6.3 Alternative Representations for Simulator

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 *Simulator* and ignore class *Actor*.

The main task of the Simulator class is to hold a collection of all actors, and to manipulate and process this collection. One representation makes use of an unordered list of actors. Actors store information about their location in the world in their instance data; we will denote this representation R_1.

An alternative representation uses a grid (a two-dimensional array). Actors are stored in this grid according to their logical position – they do not need to store their position in the actor object itself; we will denote this representation R_2.

6.4 Evaluation of Alternative Representations for Simulator

For both R_1 and R_2, implementation of the *add* method is trivial (adding an element to the end of the list for R_1, or storing an actor object at a given grid position in R_2).

R_2 simplifies the programming task of *getActorAt*, since only a single access to the grid at a known location is required, whereas a linear search of the actor list is required with R_1. (This will also effect efficiency, with the time complexity of R_1 being $O(n)$, where n denotes the number of actors in the simulation, while R_2 is $O(1)$. However, our concern here is exclusively implementation difficulty – efficiency should not greatly influence our discussion at this stage.) We might assign an REM value of *Trivial* to R_2, while R_1 is slightly more work, but still *Easy*.

Implementation of the simulator's *act* method (invoking *act* on all actors) requires a simple sweep of all actors in R_1. Again, we classify a simple iterator loop as *Easy*. Using R_2, this method requires a traversal of the grid, which includes a nested loop. Since this is harder than a single sweep, we might classify this as *Average*.

The last method, *display*, is intended to print a representation of the current world to the screen. For the grid variant, R_2, this is similar to the previous method – a nested loop – and therefore classified as *Average* again. For R_1, the task is considerable harder, since the list has no particular order. We consider this to be *Hard*. The result of the analysis is summarized in the Effort REM for class *Simulator* (Table 3).

R_1: Use unordered list to store actors

R_2: Use 2-dimensional array to hold actors

IMPL. EFFORT	***R1***	***R2***
***add*()**	Trivial	Trivial
***getActor*()**	Easy	Trivial
***act*()**	Easy	Average
***display*()**	Hard	Average

Table 3. Estimate of required effort to implement class Simulator

We choose R_2 because it allows for the simplest implementation of *Simulator*.

6.5 Attributes

We are now ready to add the instance fields and constructor to our stub version of *Simulator* (Figure 15). Again, the work here is fairly straight forward: we create a version of our chosen representation in Java (field definition and setup in the constructor – this is a largely mechanical task.

```
/** A simulator that manages actors in a 2D world.
    Locations in the world are specified by row and
    column number. */
class Simulator {
  private Actor[][] world;

  /** Create an empty Simulator */
  public Simulator(int rows, int columns) {
    world = new Actor[rows][columns];
  }

  /** Add Actor a to this simulation */
  public void add(Actor a, int row, int col) {
    // FixMe
  }

  /** Return the actor at location specified by (row,col).
      Return null if there is no actor at that location. */
  public Actor getActorAt(int row, int col) {
    return null;  // FixMe
  }

  /** Let all actors act once */
  public void act(){
    // FixMe
  }

  /** Display a representation of the current world state
      to standard out */
  public void display(){
    // FixMe
  }
}
```

Figure 15. Partial implementation of Simulator

This is indeed a very small step toward a complete implementation of *Simulator*, 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.

6.6 Implementing the Methods of Simulator

Having decided upon a representation of the simulator, we have decoupled the four sub-tasks of implementing the methods of the *Simulator* interface. This is an instance of the principle *separation of concerns* – Dijkstra's mantra and primary instrument of thought [Dijkstra 1976, pp. 211].

The *add* method can be implemented simply by storing the new actor at the specified location in the grid (assuming replacement of possibly existing actors is the intended behaviour). This gives us the implementation for the first of our four methods (Figure 16).

```
/** Add Actor a to this simulation */
public void add(Actor a, int row, int col) {
  world[row][col] = a;
}
```

Figure 16. Implementation of add()

The *getActorAt* method has a similarly simple implementation. All that is required is a direct access at the specified world location, and a return of that value found at that position (Figure 17).

```
/** Return the actor at location specified by (row,col).
    Return null if there is no actor at that location. */
public Actor getActorAt(int row, int col) {
  return world[row][col];
}
```

Figure 17. Implementation of getActor()

The *act* and *display* methods are a little more interesting – they both involve traversing the whole grid. The implementation for both methods is quite similar – we discuss the *display* method here, and leave the *act* method as an exercise to the reader.

Implementing *display* involves traversing the grid structure and displaying the contents of every grid location. For this implementation, we assume our specification requires that the output is in the form of ASCII characters arranged in lines and columns, with a dot (".") for an empty location and the letter "A" for an actor. A first step towards implementing *display* is shown in Figure 18.

```
/** Display a representation of the current world state
    to standard out */
public void display() {
  for(Actor[] row : world) {
      // display actors in the current row
      // go to new line
  }
}
```

Figure 18. Partial implementation of display()

Here, we iterate over the rows of the grid, and note the remaining work to be done informally.

It is obvious that displaying the actors in each row involves an iteration within that row, 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 displaying a single row. We name the method *displayRow*. Following the *Mañana Principle*, we also define a method for the second task, starting a new line on screen, named *newLine* (Figure 19). Note that the *Mañana Principle* can be used independently of method complexity: the *newLine* method will be very simple – we can see that already. However, following this principle still has value, leading to readable, decoupled code that lends itself to modification more easily (in this case, for example, output to a different medium).

```
private void displayRow(Actor[] row) {
  // FixMe
}

private void newLine() {
  // FixMe
}
```

Figure 19. Specification of displayRow() and newLine()

With methods *displayRow* and *newLine* to serve us, we can now finish the loop body of method *display* (Figure 20).

```
/** Display a representation of the current
    world state to standard out */
public void display() {
  for(Actor[] row : world) {
      displayRow(row);
      newLine();
  }
}
```

Figure 20. Implementation of display()

Now we need to implement the new private methods *displayRow* and *newLine*. The *newLine* method is easy to do (Figure 21).

```
private void newLine() {
  System.out.println();
}
```

Figure 21. Implementation of newLine()

The *displayRow* method involves two aspects: an iteration over the actors in the given row, and the display of each of those actors on screen. To separate those two aspects, we use the *Mañana Principle* again, and assume we have a *displayActor* method. The *displayRow* method then becomes quite simple to write (Figure 22).

```
private void displayRow(Actor[] row) {
  for(Actor a : row) {
    displayActor(a);
  }
}
```

Figure 22. Implementation of displayRow()

The last thing to do is to create the *displayActor* method, which is shown in Figure 23. Since this is our last method, we do not need to create a stub, but can proceed straight to the implementation.

```
private void displayActor(Actor a) {
  if (a == null)
    System.out.print('.');
  else
    System.out.print('A');
}
```

Figure 23. Implementation of displayActor()

This completes the development of an implementation of *Simulator* based on R_2. We have seen that by carefully choosing the simpler representation overall, and repeatedly applying the *Mañana Principle*, each method becomes reasonably easy to write and understand.

6.7 Discussion of the Development of Simulator

The discussion of the simulation 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 *Simulator*. In the general case, this does not always have to be the case. If, for instance, we were dealing with a number of different actors which are to be displayed differently depending on their type or state, we might introduce a *getDisplayChar* method in the *Actor* class as an application of the *Hard Problem* rule while implementing the simulator's *displayActor* 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.

7. A PRELIMINARY EVALUATION OF TEACHING STREAM

In this section we report on a small, preliminary study evaluating the learning outcomes of teaching STREAM.

The STREAM process has been taught in the introductory programming courses at the authors' home universities (Aarhus University and the University of Kent) for the past three years and the results are very encouraging.

In order to conduct a preliminary evaluation of process competence, we set up an experiment just prior to the final examination at Aarhus University two years ago. We designed a programming task similar to our final examination. No guidance was provided with respect to the overall programming process. The task description consisted of a class model and functional specifications of methods in the model, and students were told to implement the specified model.

We first designed and carried out an evaluation based on "think aloud": We asked the students to think aloud while solving the given task. For each student, the screen and the student's voice were recorded. This experiment largely failed because the students often did not think aloud; they were preoccupied with the programming task and did not have mental resources to also speak about what they were thinking.

After this, we designed and carried out an evaluation based on observations. 38 students took part in the evaluation (they were representative of the whole population of approximately 400 students). 13 teaching assistants (TAs) helped monitor the students while they solved the programming assignment. The experiment lasted one hour.

Our goal was to evaluate the students' programming process when no process guidance is provided in the phrasing of the assignment. A group of TAs examined the students and took notes of their behaviour; the student/TA ratio was 3/1.

The TAs were instructed to take notes of the students' programming process. A form was designed and used to record the notes. TAs were instructed to record student activities, in particular noting whenever a student violated the 'standard process' that had been taught in the course.

The form consisted of columns for recording the time from the start of the task and the nature of observed activity. Figure 24 shows the form structure and some typical entries. A mark in a column indicates activity in that category.

T	**TC**	**PC**	**R**	**S**	**F_1**	**F_2**	**...**	**F_n**	**E**	**C**
...										
:11		x		x		x				fixing indentation
...										
:27		x	x		x					handling special case
...										
:41	x				x				x	can't find last element
...										

T: elapsed time (in minutes); TC: test code; PC: production code; R: refine; S: restructure; F_i: functionality; Error handling; C: comment (free text)

Figure 24. Form for note-taking during experiment

The completed forms were analysed to produce a condensed characterization of each student's programming process with special focus on deviations from the prescribed process.

The somewhat surprising conclusion of the experiment was that all 38 students followed the process they had been taught even though no process guidance was provided. The students developed one part of the program at a time, separating the different concerns of the task. There was some variation in the frequency of students swapping between writing test code and writing production code and in writing the test code before or after the production code. STREAM suggests writing test code before the production code, but almost all the students wrote the production code first.

Immediately after the experiment, we conducted informal interviews with groups of students. When asked about their testing behaviour (less frequent than prescribed and after the functionality to be tested was implemented), they responded that they did not feel the need for the test in order to implement the requested methods. They wrote the tests because they had to, not because they needed it to understand the task or to ensure that the production code worked. It is hard to blame students for this since their behaviour mirrors expert behaviour [Caspersen 2007, section 6.3.2].

We refrain from drawing overly strong conclusions from this experiment, since it was not a strictly formal study with well-defined research questions. However, the observations are encouraging and suggest that students under the right conditions can learn the process we teach – at least when they are exposed to familiar tasks. Again, this reflects expert behaviour. Winslow puts it this way: “Experts, when given a task in a familiar area, work forward from the givens and develop subgoals in a hierarchical manner, but given an unfamiliar problem, fall back on general (opportunistic) problem solving” [Winslow 1996, p. 18].

This preliminary study shows that our teaching of STREAM had a positive effect on the development of our students’ process competences. To draw more general conclusions, further and more thorough investigations are needed.

8. 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. 40 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 [Dijkstra 1968, Dijkstra 1969, Wirth 1971, Wirth 1973]. 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 [Dijkstra 1976]. In continuation of Dijkstra’s seminal work, Back developed a refinement calculus [Back 1978, Back 1998] while Gries and others produced text books based on the methodology (e.g. [Gries 1981, Cohen 1990, Morgan 1990]). 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 [Wirfs-Brock 2003]. 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 [Fowler 1999]. 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 [Beck 2000, Martin 2003]; 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. Extreme programming typically manifests itself in the classroom as pair programming [Williams 2001, Bergin 2004, Hanks 2008]. Agile software development in education is covered by a special issue of Computer Science Education [Williams 2002]; practical software engineering education was the topic for another special issue of the same journal in 2001 [Saiedian 2001].

Test-driven development. The strategy of test-driven development [Beck 2003, Hunt 2003] 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. Several educators promote rethinking of the introductory programming course in terms of test-driven programming [Edwards 2004, Jones 2004, Janzen 2006].

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.

An obvious third direction of future work concerns a more thorough evaluation of the learning effects of teaching STREAM. Different methods can be used for such evaluations, e.g. "think aloud", observation, instrumentation of the programming environment, and stimulated recall. We particularly welcome third party adoption and evaluation of STREAM.

9. 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 briefly presented a model and a conceptual framework of incremental software development called Stepwise Improvement that characterises programming development as an explorative activity of discovery and invention taking place in the three-dimensional space of extension, refinement, and restructuring.

Stepwise Improvement is specialised into an informal but systematic development process, STREAM, designed to be applied by beginners. As part of STREAM we have identified and described principles and systematic programming techniques particularly suited for novices learning object-oriented programming. Through two examples we have demonstrated the application of STREAM.

STREAM 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.

We have reported on a small, preliminary study indicating that teaching STREAM can have a positive effect on the development of the students' process competences.

REFERENCES

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

BACK, R.-J. 1998. *Refinement Calculus: A Systematic Introduction*. Springer-Verlag.

BECK, K. 2000. *Extreme Programming Explained: Embrace Change*. Addison-Wesley.

BECK, K. 2003. *Test-Driven Development by Example*. Addison-Wesley.

BERGIN, J., CARISTI, J., DUBINSKY, J., HAZZAN, O., AND WILLIAMS, L. 2004. Teaching Software Development Methods: The Case of Extreme Programming. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, Norfolk, Virginia, USA, 2004, 448-449.

CASPERSEN, M.E. 2007. *Educating Novices in the Skills of Programming*. PhD Dissertation PD-07-4, Department of Computer Science, University of Aarhus.

COCKBURN, A. 2002. *Agile Software Development*. Addison-Wesley.

COHEN, E. 1990. *Programming in the 1990's*. Springer-Verlag.

COLLINS, A.M., BROWN, J.S. AND HOLUM, A. 1991. Cognitive apprenticeship: Making thinking visible. In *American Educator*, Vol. 15, 3.

DAHL, O.-J., DIJKSTRA, E.W., AND HOARE, C.A:R. 1972. *Structured Programming*. Academic Press.

DIJKSTRA, E.W. 1968. A Constructive Approach to the Problem of Program Correctness. In *BIT* 8, 1968.

DIJKSTRA, E.W. 1969. *Notes on Structured Programming*, EWD 249, In [10].

DIJKSTRA, E.W. 1976. *A Discipline of Programming*. Prentice-Hall.

DR. JAVA. 2009. http://drjava.org. Accessed 8 February 2009.

EDWARDS, S.H. 2004. Using Software Testing to Move Students from Trial-and-Error to Reflection-in-Action. In *Proceedings of the 35th SIGCSE Technical Symposium on Computer Science Education*, Norfolk, Virginia, USA, 26-30.

FOWLER, M. 1999. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley.

GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J.M.. 1995. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley.

GRIES, D. 1981. *The Science of Programming*. Springer-Verlag.

HANKS, B. 2008. Problems Encountered by Novice Pair Programmers. *Journal on Educational Resources in Computing*, Vol. 7 (4), Article No. 2.

HUNT, A. AND THOMAS, D. 2003. *Pragmatic Unit Testing in Java with JUnit*. The Pragmatic Programmers.

JANZEN, D.S. AND SAIEDIAN, H. 2006. Test-Driven Learning: Intrinsic Integration of Testing into the CS/SE Curriculum. In *Proceedings of the 37th SIGCSE Technical Symposium on Computer Science Education*, Houston, Texas, USA, 254-258.

JONES, C.G. 2004. Test-Driven Development Goes to School. *Journal of Computing Sciences in Colleges*, Vol. 20 (1), 220-231.

JUNIT. 2009. http://www.junit.org. Accessed 8 February 2009.

KÖLLING, M. 2009. *Unit Testing in BlueJ*. http://www.bluej.org/tutorial/testing-tutorial.pdf. Accessed 8 February 2009.

KÖLLING, M., QUIG, B., PATTERSON, A., AND ROSENBERG, J. 2003. The BlueJ System and its Pedagogy. *Computer Science Education*, Vol. 13 (4), 249-268.

MARTIN, R.C. 2003. *Agile Software Development: Principles, Patterns, and Practices*. Prentice-Hall.

MORGAN, C. 1990. *Programming from Specifications*, Prentice-Hall.

SAIEDIAN, H. 2001. Practical Software Engineering Education. *Computer Science Education*, Vol. 11 (1), 3-5.

WILLIAMS, L.A. AND KESSLER, R.R. 2001. Experiments with Industry's "Pair-Programming" Model in the Computer Science Classroom. *Computer Science Education*, Vol. 11 (1), 7-20.

WILLIAMS, L.A., TOMAYKO, J. 2002. Agile Software Development. *Computer Science Education*, Vol. 12 (3), 167-168.

WINSLOW, L.E.. 1996. Programming pedagogy – a psychological overview. *SIGCSE Bulletin*, Vol. 28 (3), 17-22.

WIRFS-BROCK, R. AND MCKEAN, A. 2003. *Object Design: Roles, Responsibilities, and Collaborations*, Addison-Wesley.

WIRTH, N. 1971. Program Development by Stepwise Refinement. *Communications of the ACM*, Vol. 14 (4), 221-227.

WIRTH, N. 1973. *Systematic Programming*, Prentice-Hall.

Received November 2008; revised January 2009; accepted February 2009.