

The Solution First Programming Process—or “How Quick Fix is much more than a quick fix”!

Henrik Bærbak Christensen
Department of Computer Science
Aabogade 34
8200 Aarhus N - Denmark
hbc@cs.au.dk

ABSTRACT

Learning to program is hard and one major obstacle for students is getting to grips with the *programming process* itself: What do I do now, what are the next steps, and what end result do I want to achieve? Further complications arise as many, even simple, designs require quite a lot of code to be written that are just “stepping stones” on the path to a solution. Thus, instead of “writing the code that expresses my final design”, we have to “write some code that will eventually be used by the final design code that I hope to write soon.” However, modern integrated development environments have strong programming language support which allows us a programming process that starts by writing the solution code. In this paper, we will exemplify such a *solution first programming process* by a small example, and advocate that teachers take full advantage of the support offered by modern IDEs to teach a more natural programming process for our students.

Categories and Subject Descriptors

D.1.5 [Programming Techniques]: Object-Oriented Programming; D.2.3 [Software Engineering]: Coding Tools and Techniques; D.2.6 [Software Engineering]: Programming Environments; K.3.2 [Computers and Education]: Computer and Information Science Education

General Terms

Design, Human Factors

Keywords

Programming process, Development environment, Incremental Development, Pedagogy, Programming Education, Design

1. INTRODUCTION

Developing software is, by its very nature, always a process, whether we are formally aware of it or not [2, p. 4.2].

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ICSE/SEET 2014 Hyderabad, India

Copyright 2014 ACM X-XXXXX-XX-X/XX/XX ...\$15.00.

Programming entails a lot of skills. Curriculum (and indeed much teaching) have a tendency to focus on the “nouns” or “topics” in programming: language constructs, algorithms, data structures, objects, recursion, testing, etc. This topic focus is often prominent in text books in programming which use it as an organizing principle for the chapters. The benefit for the teacher is obvious as you may mix chapters from different books or use them out of order. However, the liability is that it neglects one of the hardest challenges of programming namely that of combining the proper set of techniques in the right order for the problem at hand [3, 4]. Basically, the topic based approach leaves the students with unconnected islands of skills, and little awareness of the process of developing software.

In many disciplines, the *process* is central in learning: The tools of a carpenter is of little use if not trained in the process of using them. Early attempts to define a programming process included *top-down programming* in the 1970s and 1980s and later *bottom-up programming* in the 1990s. Both provided a compelling conceptual framework but fell short in covering the actual complexity of developing non-trivial systems. In top-down programming (taken literally) only non-executable abstractions exists until the very last stage which means no real feedback is available for early detection and correction of mistakes. In bottom-up programming, the situation is essentially the same: while low level modules exist and compile, they are not integrated nor provide end-user functionality until the very last stage of development, much to late to detect misunderstandings in requirements and architecture. Thus, while the techniques were taught, successful development meant breaking the doctrine.

Agile methodologies and notably test-driven programming came with a refreshing new take and a very concrete programming process which includes a vocabulary of process steps (TDD patterns) as well as an overall process (the rhythm) [1]. These processes were real programming processes with a strong focus on the actual development of code. Caspersen and Kölling [2] combines aspects of the agile paradigm’s focus on working functional increments and iterative development process into a novice programming process called STREAM. Here, the agile idea of adding functional increments during sprints is reformulated in a teaching context as *growing islands of functionality* in a stepwise improvement cycle. Still, when it comes to the manifestation of the systematic programming process for novices, their process has a flavour of top-down programming, like “create stub classes for all interfaces in the project.”

These programming process descriptions are independent

of the actual tooling i.e. the actual development environment used. While this is appropriate from the point of generalisation and general applicability, it also misses an important practical point: Software *is* developed using strong development tools and they provide services that, as we will show, may drastically influence the actual programming process in a way that it becomes more efficient as well as more pedagogical.

The main contribution of this paper can thus be viewed as both a very simple tool trick as well as a major challenge to the educational research in programming process models. From the tool trick perspective, we will demonstrate how Eclipse’s “Quick Fix” feature can speed up development—for any teacher to quickly include as a nice feature in his or her teaching. From the educational research challenge perspective, we will outline how this innocent looking feature actually drastically changes the flow of the programming process itself in a way that is more pedagogical (“logical” in a sense) and leads to faster development. We term this process *solution first programming* as it starts with the actual programming problem to be solved—and solves it right away!

Our paper is structured with a concrete example of the process in section 2 where we present a small programming task, outline a classic programming process for solving it, and next in detail present how it is solved using solution first programming. We present some concrete measurements on the process in section 3 along with discussions and relations to other work on programming process, before concluding in section 4.

2. PROCESS WALKTHROUGH

What better way to demonstrate a process than by doing it? Below we will define a small programming task, that of refactoring an existing piece of code to prepare the design for handling more variants. First we will shortly outline how this may traditionally be done, and next walk through the solution first programming process. As a textual presentation required in a paper is a mediocre medium for unfolding a process, you will find a link to a video showing the process at the end of the section.

2.1 The Task

The programming task is taken from the textbook[5] and used in class for introducing the Strategy pattern[9] at the conceptual level and used for demonstrating refactoring[8] at the programming level.

The context is a pay station at a parking lot. Our customers from Alphatown want drivers to pay a flat rate of 5 cent for every 2 minutes parking time. Test-driven development (TDD)[1] has been used to implement the domain code and in particular the `addPayment` method which validates the coin, accumulates inserted money, and calculates total amount of parking time bought:

```
public void addPayment( int coinValue )
    throws IllegalCoinException {
    switch ( coinValue ) {
    case 5: break;
    case 10: break;
    case 25: break;
    default :
        throw new IllegalCoinException (
            "Invalid coin: "+coinValue);
```

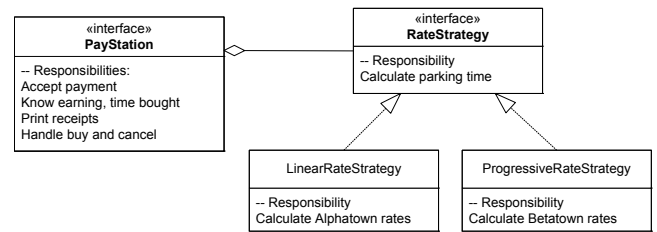


Figure 1: A Strategy pattern based design

```
}
insertedSoFar += coinValue;
timeBought = insertedSoFar / 5 * 2;
```

The interesting issue arises when Betatown wants to buy our pay station product but requires another algorithm for rate calculation. That is, the functionality captured by the statement marked in gray in the above code needs to vary for our two customers.

A sound solution is to introduce a Strategy pattern: a `RateStrategy` interface with two implementing classes, one for Alphatown’s flat rate calculation, and another for Betatown’s algorithm, as outlined in Figure 2.1.

Thus the present method `addPayment`’s code must be refactored so the rate calculation statement instead becomes:

```
[...]
insertedSoFar += coinValue;
timeBought = rateStrategy.calculateTime(insertedSoFar);
```

To bootstrap this change and ensure no defects are introduced, we first have to make all Alphatown’s test cases pass on the refactored code. This leads to the issue of the paper: *Which programming process should we use to refactor the pay station system?*

2.2 Traditional Programming

We dare call the programming process outlined below for the “traditional” process to distinguish it from our proposed solution first process. We use that term because it is in line with the TDD principles of “taking small steps”[1] and avoids long periods in which the code does not compile or pass its tests, it is seen in text books, and finally because we ourselves have practiced this process and seen colleagues do so for years.

A feasible path entails the following actions (Full details to be found in [5, Chapter 8]):

- ① Introduce the `RateStrategy` interface with the `calculateTime` method. Compile everything to ensure no syntax errors are introduced.
- ② Introduce an implementing class `LinearRateStrategy` that implements the `RateStrategy` interface, and copy the simple rate calculation algorithm into its `calculateTime` method. Compile to validate.
- ③ Introduce an instance variable, `rateStrategy` of type `RateStrategy` in the `PayStation` class, and assign this variable a new instance of `LinearRateStrategy`. Compile to validate.
- ④ Finally, do the refactoring of the rate calculation: Change the `timeBought = ...` into the final form `= rateStrategy.calculateTime(insertedSoFar);`. Compile and run the all tests to ensure no defects have been introduced.

This programming process has the merit that the code compiles and all tests run after each step, and of course solves the task. However, note that it is the very last step that actually solves the core issue here: to delegate the rate calculation instead of doing it in-place. Alas, all the previous steps are “just” overtures to the final crescendo which *is* the solution. But it requires that we have the design of the final solution “in our head” before committing to all these actions, as the final step is a “keystone” that finally validates all the previous steps.

From a learning perspective, this process seems “backwards”. In class, we describe the goal we want to achieve, namely replacing the in-place calculation of `timeBought` with a delegation to a strategy object. But our programming process doesn’t start there and is further obscured by implementation effort which isn’t used until the very last step. Of course, our strong students know exactly what goes on. However, our weaker students may become confused.

Another issue is that we detect faulty designs late. What if we had made a bad initial plan only to find out our mistake in this last step? Then the numerous previous steps potentially had to be undone or at least modified. While this “does not happen in class” because we as teachers are carefully prepared (we took all the wrong paths in our offices and only take the correct one down to the lecture room for presentation), it *will* happen to the students when they struggle with their exercises.

2.3 Solution First Programming

But—why this backwards approach when we know what we want right from the start? Why not just change the statement in focus into

```
[...]
timeBought = rateStrategy.calculateTime(insertedSoFar);
```

right away?

This is possible because *Eclipse will fill out all the missing parts in an interactive session with us!* The strong understanding of Java in modern IDEs saves us both from a lot of typing as well as language mistakes. In Eclipse this feature is called “quick fix” [7].

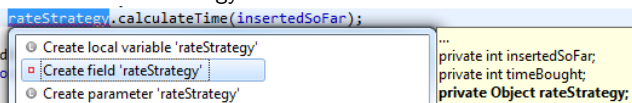
The faster and much more direct route to the exact same solution unfolds like this. (We recommend watching the video presentation found at <http://www.cs.au.dk/~baerbak/research/seet2014.html> as it is a better medium for showing process.)

① Introduce the solution

```
41     insertedSoFar += coinValue;
42     timeBought = rateStrategy.calculateTime(insertedSoFar);
```

Eclipse will mark the line with the Quick Fix icon, the red icon on the left of line 42.

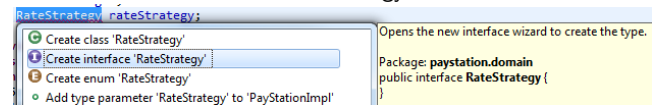
② Clicking the quick fix, we can select the action required: create the `rateStrategy` field.



which creates the field. However, the type is `Object` which is wrong.

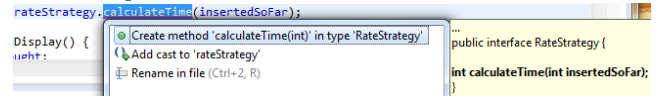
```
30     private int timeBought;
31     private Object rateStrategy;
```

③ Thus, we change the type to `RateStrategy`. This type is unknown and therefore another quick fix icon appears, and we choose to create the `RateStrategy` interface:



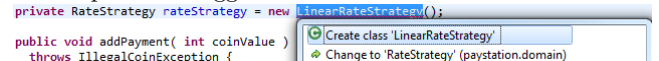
The “New Java Interface” pop-up appears (not shown here due to space) but all default values are correct so it is just a matter of hitting “Finish” to create the interface. This will remove the ‘quick fix’ on the `rateStrategy` field declaration line but not on the line in the `addPayment` method.

④ The problem is that method `calculateTime` is unknown. We click ‘quick fix’ and select to create the method.



At this moment, everything compiles but the tests fail because the `rateStrategy` is a null reference.

⑤ The fastest solution is to assign `rateStrategy` a new instance of `LinearRateStrategy` directly. This leads to yet another ‘quick fix’ suggestion:



This opens the “New Java Class” pop-up, and again all default values are correct and we can just hit “Finish” to create the class with a stub implementation.

```
1 package paystation.domain;
2
3 public class LinearRateStrategy implements RateStrategy {
4
5     @Override
6     public int calculateTime(int insertedSoFar) {
7         // TODO Auto-generated method stub
8         return 0;
9     }
10 }
11 }
```

⑥ The final step is trivial as we introduce the proper rate calculation.

```
5 @Override
6 public int calculateTime(int insertedSoFar) {
7     return insertedSoFar / 5 * 2;
8 }
```

All tests pass. Done.

3. DISCUSSION

To compare the two programming processes we registered keyboard activity using WhatPulse [12] on three sessions, solving the above refactoring exercise. In session 1 we solved it traditionally (section 2.2) and used Emacs as editor and Ant in the shell for compilation and testing. We tried to minimize effort as best possible by copying and pasting (copy “RateStrategy.java” to “LinearRateStrategy.java” and just changing the code) as well as by using Emacs’s Dynamic Abbrev Expansion [6] for name completion. In session 2 we again solved it traditionally (section 2.2) but used Eclipse’s generation facilities as much as possible (like clicking ‘New Interface’ and fill in the dialog instead of creating an interface source file from scratch), and finally in session 3 we

solved it using the solution first programming process (section 2.3). The table below shows statistics on key strokes and mouse clicks. They are averages over doing the same exercise five times in a row for each session.

Session	keystrokes	mouse clicks
1 (Emacs+Ant)	387	17
2 (Eclipse/Trad)	178	20
3 (Eclipse/Sol. First)	101	28

While these numbers cannot claim strong statistical validity, they never-the-less confirm our informal impression. First, though Emacs is an excellent editor, it is not a full IDE with a deep knowledge of Java, and this shows up in more typing, and also more time spent fixing typos and compilation issues. Second, the solution first programming process trades more mouse clicks for less typing. Indeed, the amount of keystrokes used in session 3 is much less than the 309 characters that define the source code of `RateStrategy` and `LinearRateStrategy`.

While saving time and effort as hinted from the above experiment is of course interesting, it is not the essential aspect. The essential aspect is that

the programming process steps now align with our design intention

Thus, when we want a solution that delegates to a strategy object, we do it right away. We solve the problem as the first move in our IDE—and essentially we are finished at this point. The “rest” is then an interactive session, guided by quick fix suggestions of Eclipse, but directed by us by providing minimal but semantical strong information: Naming interfaces and classes, indicating package membership, changing visibility, etc. The chore tasks are handled by the IDE. Thus we do not really “code” ourselves—Eclipse does this under our supervision.

Note that of the six steps in section 2.3 only three are concerned with semantics: in step ① we write the “final” code; in ③ we have to change the type of the field to `RateStrategy`, and finally in step ⑥ we enter the rate calculation code. The other steps are automatic remedies to make the code compile again, and the Eclipse suggestions are all correctly guessed.

On the downside of this process is the fact that it is strongly tied to a particular IDE. We have demonstrated it using Eclipse and know similar features in other main stream IDEs. But, of course, if you teach using a more classic setup of editor and compiler, or if you teach in programming languages that does not have this kind of support, you are out of luck.

4. CONCLUSION

We have proposed to further explore a new programming process which we denote *solution first programming*. Traditionally, many utilize a process that is rather bottom-up, programming code pieces that are deemed necessary for a solution to come later, as it avoids long period of code that cannot compile and cannot pass tests. In a learning context, this is unfortunate, as it introduces a long delay between a session is started and the actual solution to the task is put in place. In addition, it is also unfortunate as we may make wrong decisions about the nature of the final solution and this is discovered late in the process.

In solution first programming, the statements that represent the final solution is written first. Such a solution will

refer to unknown interfaces, classes, instances, and methods, but current main-stream IDEs have strong support for suggesting code additions that will quickly “fill in the blanks” of the solution. We have demonstrated the technique using a refactoring task, and we claim it is more natural and better in a learning context. We have also shown initial data to support that it is also a faster programming process and the code produced is less prone to errors. The solution first programming process, however, requires strong support by the IDE, as it is a premise that the IDE can propose and generate high quality code for the missing parts.

While the process is interesting by itself in practical teaching of programming, we also invite the teaching community to revisit the research concerning programming processes. Solution first programming have strong relations to top-down programming as well as the test-driven programming style of approaching abstractions from their “outside” i.e. use the methods before *implementing* the methods. The traditional disadvantage of these approaches such as long period where the code cannot compile or intensive use of stubs and mock objects [11, 10], is mitigated by the code generation facilities of modern IDEs.

5. REFERENCES

- [1] K. Beck. *Test-Driven Development by Example*. Addison-Wesley Signature Series, 2003.
- [2] M. E. Caspersen and M. Kolling. Stream: A first programming process. *Trans. Comput. Educ.*, 9(1):4:1–4:29, Mar. 2009.
- [3] H. B. Christensen. Systematic testing should not be a topic in the computer science curriculum! *SIGCSE Bull.*, 35(3):7–10, June 2003.
- [4] H. B. Christensen. A story-telling approach for a software engineering course design. In *Proceedings of the 14th annual ACM SIGCSE conference on Innovation and technology in computer science education*, ITiCSE '09, pages 60–64, New York, NY, USA, 2009. ACM.
- [5] H. B. Christensen. *Flexible, Reliable Software—Using Patterns and Agile Development*. CRC Press, 2010.
- [6] Emacs dynamic abbrev expansion. http://www.gnu.org/software/emacs/manual/html_node/emacs/Dynamic-Abbrevs.html. Accessed Oct 2013.
- [7] Eclipse documentation. <http://help.eclipse.org/kepler>. Section: Java development user guide, Concepts, Java Views, Quick Fix and Assist. Accessed Oct 2013.
- [8] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison-Wesley, 1999.
- [9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reuseable Object-Oriented Software*. Addison-Wesley, 1995.
- [10] T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: unit testing with mock objects. *Extreme programming examined*, pages 287–301, 2001.
- [11] D. Thomas and A. Hunt. Mock objects. *IEEE Software*, pages 22–24, May/June 2002.
- [12] <http://www.whatpulse.org>. Accessed Oct 2013.