Why Programmers Don't Use Methods And What We Can Do About It

Trygve Reenskaug
Taskon, Oslo

At a recent meeting in the Object Management Group in Tampa, Florida, Ivar Jacobson stated that only 5% of the world's programmers use some method for analysis and design. The remaining 95% just write code. As a methodologist, my knee jerk reaction is one of horror. Can it really be that bad? In all mature engineering disciplines, formal design plays an essential part of their procedures. Isn't it time that software engineering grew up and made formal analysis and design a natural part of the software production process? In this column I will examine this question and suggest that the answer is not as obvious as we like to believe.

First a few words about the other engineering disciplines. Engineering design methods do *not* guarantee that they get it right the first time. Engineering documentation is *not* an infallible source of information that always reflects the true state of affairs. Consider a North Sea oil production platform. The basic technology of such a huge structure is fairly simple. Its complexity is due to size; its design can need 1000 engineerers working concurrently for more than a year. In paper form, the final design documentation weights several tons. This documentation is *not* 100% correct, it is particularly hard to avoid inconsistencies caused by modifications not being propagated to all affected drawings. And after a few years of normal maintenance operations, the platform can have been modified without a corresponding update of the documentation.

More importantly, the analogy to conventional engineering fails because the construction of an oil production platform is materially different from the production of a computer program. The detailed design specification of a program is its code; the actual building of the program is done automatically. One of the goals of almost all programming languages ever developed has been that the code shall be self-documenting. When we look upon it this way, 100% of all programmers do detailed design.

My favorite quote is due to C. A. R. Hoare: "*There are two ways of constructing a software design: One way is to make it so simple that there are obviously no deficiencies and the other way is to make it so complicated that there are no obvious deficiencies."*

The *other way* would have been my guiding star if my goal were to prepare for a future legal battle. But since my goal is to produce useful software, the *first way* must be my way. So the question becomes *"is my code so simple that there are obviously no deficiencies?"* (Remember that the readers are myself, my colleagues, and future program maintainers who will be charged with conserving the simplicity). The answer clearly depends on my chosen programming language, the problem, and my ability to solve it. If the answer is yes, all is well and good. If the answer is no, I need additional artifacts to help me create the required simplicity.

Let's get more concrete and study a simple example. Consider the *activity networks* that are used in project planning an control. An *activity* is an abstraction on a task. It is characterized by having a *duration*, a number of *predecessors,* and a number of *successors.* An activity can start when all its *predecessors* are completed. Its earliest finish is its *duration* after the finish of the latest predecessor. And its successors can start any time after its finish time. Figure 1 shows a simple example.

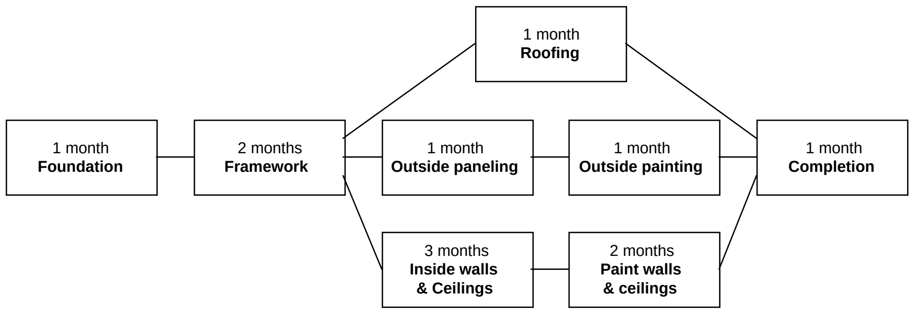

Figure 1. Example activity network

The network permits me to compute answers to questions such as: "If I start building 1/1, when is the earliest I can be finished?" (end of month 9). "When do I need to get painters?" (month 5 for the outside and month 7 and 8 for the inside). The computation used is *front-loading*: Start with the project start time and the initial activity, add its duration, and iterate through the activity successors. Similarly, *back-loading* computes the latest completion time for all activities starting from the end activities and the project completion time.

Let's write a program for *frontLoading* an activity network. I have chosen the Smalltalk programming language to avoid cluttering the code with computer-specific details. The program is shown in figure 2, and the result of running the test method *Activity test1: 1* is shown in figure 3. In the program, I have made a superclass for the general *NetworkNode*, and a subclass that specializes it to describe an *Activity*. The program is neither elegant nor efficient, but it serves our purpose. (And it worked, at least before I copied the code into this journal).

```
Object subclass: #NetworkNode
        instanceVariableNames: 'predecessors successors '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'ActivityNetworks'

setPredecessor: pred
        (predecessors includes: pred)
        ifFalse:
                [predecessors add: pred.
                pred setSuccessor: self].
setSuccessor: succ
        (successors includes: succ)
        ifFalse:
                [successors add: succ.
                succ setPredecessor: self].
initialize
        predecessors := OrderedCollection new.
        successors := OrderedCollection new.

NetworkNode subclass: #Activity
        instanceVariableNames: 'name duration earlyStart '
        classVariableNames: ''
        poolDictionaries: ''
        category: 'ActivityNetworks'

duration: aNumber
        duration := aNumber.
frontLoading: aMonthNumber
        (earlyStart notNil and: [earlyStart >= aMonthNumber])
        ifFalse:
                [earlyStart := earlyStart isNil ifTrue: [aMonthNumber] ifFalse: [aMonthNumber max: earlyStart].
                successors do: [:succ | succ frontLoading: earlyStart + duration]].
name: aString
        name := aString.
report
        Transcript cr; show: name printString , ' from start of month: ' , earlyStart printString
                , ' to end of month: ' , (earlyStart + duration - 1) printString.
        successors do: [:succ | succ report].
reset
        earlyStart notNil
        ifTrue:
                [earlyStart := nil.
                successors do: [:succ | succ reset]].
initialize
        super initialize.
        duration := 0.
        earlyStart := nil.

Activity class methodsFor: testing
test1: projectStart
        | act1 act2 act3 act4 act5 act6 act7 act8 |
        act1 := Activity new initialize name: 'Foundation'; duration: 1.
        act2 := Activity new initialize name: 'Framework'; duration: 2.
        act3 := Activity new initialize name: 'Roofing'; duration: 1.
        act4 := Activity new initialize name: 'Outside paneling'; duration: 1.
        act5 := Activity new initialize name: 'Outside painting'; duration: 1.
        act6 := Activity new initialize name: 'Inside walls & ceilings'; duration: 3.
        act7 := Activity new initialize name: 'Paint Walls and Ceilings'; duration: 2.
        act8 := Activity new initialize name: 'Completion'; duration: 1.
        act2 setPredecessor: act1. act3 setPredecessor: act2. act4 setPredecessor: act2.
        act5 setPredecessor: act4. act6 setPredecessor: act2. act7 setPredecessor: act6.
        act8 setPredecessor: act3; setPredecessor: act5; setPredecessor: act7.
        act1 reset; frontLoading: projectStart; report.
```

Figure 2. Smalltalk code

```
'Foundation' from start of month: 1 to end of month: 1
'Framework' from start of month: 2 to end of month: 3
'Roofing' from start of month: 4 to end of month: 4
'Completion' from start of month: 9 to end of month: 9
'Outside paneling' from start of month: 4 to end of month: 4
'Outside painting' from start of month: 5 to end of month: 5
'Completion' from start of month: 9 to end of month: 9
'Inside walls & ceilings' from start of month: 4 to end of month: 6
'Paint Walls and Ceilings' from start of month: 7 to end of month: 8
'Completion' from start of month: 9 to end of month: 9
```

Figure 3. Activity early start and completion

Let's read the code and remember the question: *"is this code so simple that there are obviously no deficiencies?"* If you feel the answer is yes, then I suggest you select a more complex example from your own experience and keep it in mind during the remainder of my argument. I will pretend that my example, simple as it is, hides its essence among a multitude of details so that we need additional documentation in order to convince ourselves and others that it is obviously correct. We then have to choose an appropriate idiom for what we need to express.

The simplest solution is to make what is known as "a strategic decision". It usually translates into finding out what everybody else are doing, and do the same. But what other people do is not necessarily a good guide. (We know that more than more than 30% of the population smokes cigarettes, but this fact doesn't make it more healthy). We have to understand in which way our program code is lacking in clarity, and what additional information is needed to make the program clear in our minds. If a given method helps us, it is a good method. If it doesn't, it is a bad method. Let's look at two alternatives. One is the *program-centered* approach, it focuses on the classes and their relationships. The other is the *system-centered* approach, it focuses on systems of collaborating objects.

Figure 4 illustrates the program-centered approach. It is a class diagram in the notation of the Unified Modeling Language (UML) [http://www.rational.com/]. Class *NetworkNode* has important relationships called *predecessors* and *successors*, the stars mean that both have cardinality MANY. Two methods are defined for the NetworkNode interface: *setPredecessor:* and *setSuccessor:*. Class *Activity* is a subclass of *NetworkNode* where I have defined additional instance variables and methods.

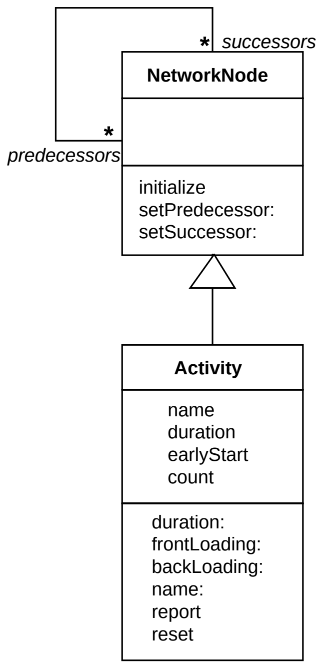

Figure 4. Class diagram illustrating the program-centered approach.

In figure 5, I have chosen the OOram idiom and notation to illustrate the system-centered approach [http://www.sn.no/taskon]. In this approach, a *role model* shows how a system of objects satisfy one or more goals. Figure 5(a) shows that an object playing the role of *aPredecessor* knows about any number of objects playing the role of *anActivity*. The shown role represents one of them. *aPredecessor* can send the *frontLoading:* message to *anActivity*, and *anActivity* can send *backLoading:* to *aPredecessor*. Figure 5(b) illustrates a typical sequence of messages for the goal of performing a frontLoading operation, and figure 5(c) shows the corresponding sequence for backLoading.

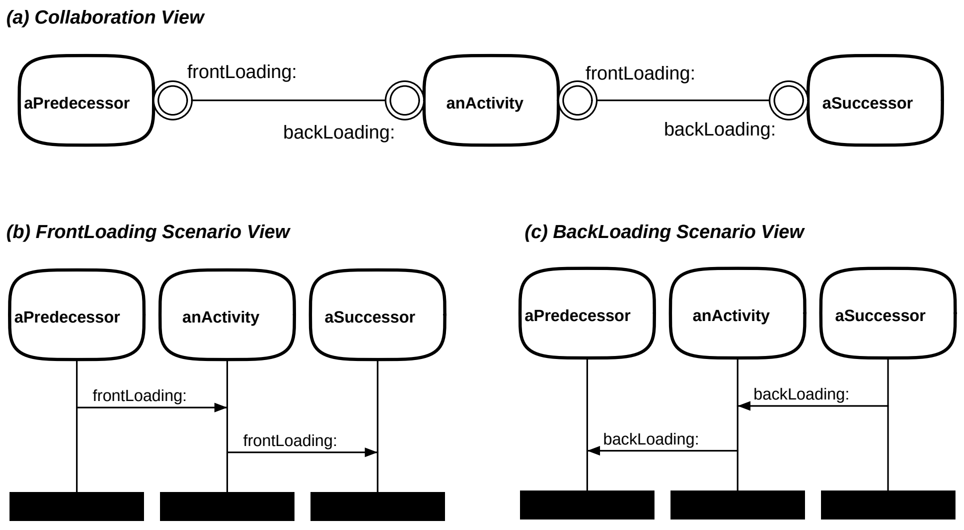

Figure 5. Three views on a frontLoading role model

Comparing the two idioms, we see that the class model of figure 4 highlights the classes and their relationships, while the role model of figure 5 highlights the objects and their collaboration patterns. The class model shows the messages that need be implemented as methods, while the role model identifies both the senders and receivers of the messages, why the messages are sent and when they are sent.

Which is the better idiom? If your programming language and tools do not clearly exhibit the structure of the code, you need the program-centered approach. If you need additional insights into what happens at runtime, the system-centered approach will be your choice. If you need both, your programming environment may be less than ideal.

Doing design before implementation can be very rewarding. Our company recently assisted in a project writing a fairly large, distributed system in Smalltalk. We got involved some time after project start, and our client had already begun coding some parts. We did the remaining parts, starting with a role model design of the essential functionality before turning to the implementation. Our parts were the only ones completed on time, in spite of the head start of the 'pure coders'. The project is now completed, and we are currently assisting the client in producing a design for the whole system in preparation for future, kindred projects.

It is not always feasible to complete the design before coding. In the mid-sixties, when the idea of top down analysis and design was gaining momentum, I tried to persuade a number of Master students to apply top down methods in their work. To my disappointment, I found that they could never do it. I then began to observe my own development work, and found that frequently, I couldn't work top down either. The reason seemed clear: *I can only work top down if I know at least one and preferably several solutions to every detail.* The students, with their lack of experience, never knew how to do all the details. I, with my bent towards research, often try to tackle problems that I don't know how to solve. In those cases, I have to clear away the low-level details before I can see the top-level design.

This experience is still valid. Whenever I know what I am doing, I do a system-oriented analysis, design, and implementation. In other cases, I frequently find it expedient to make preliminary role models of critical systems on the back of an envelope before I embark on exploratory programming. This results in a breadboard program that works most of the time. A serious design of the critical subsystems follows; I use reverse engineering on the prototype program to make sure the architecture leaves room for all important details. Finally, I do a serious implementation that includes cleaning up the class hierarchy.

The conclusion? I believe there could be several reasons for not using a method for analysis and design:

1. The program code itself is self-documenting so that additional descriptions are redundant. This probably occurs even less frequently than we like to believe.
2. Using a method in a top-down fashion can have failed due to insufficient understanding of solution details.
3. A chosen method can prove uninteresting because it supports an irrelevant modeling idiom.

If you find that your software is not *so simple that there are obviously no deficiencies,* you could consider

1. to choose the modeling idiom that best describes the hard parts of your problem. A program-centered approach will give overview of the code; a system-centered approach will give overview of how the system works
2. use an iterative approach to help get both architecture and details right. Do not let the first iteration code become the end product. (The first iteration can be done much faster if you know you will be rewriting it.)
3. do not over-document, but try to maximize self-documenting code.

In my own work, I always *think* in terms of goals, subjects, systems, responsibilities, and roles; and try to structure and comment my code to make it reflect my thoughts as clearly as possible. This is sufficient for simple programs and for simple parts of large programs. But when I cannot make the code *so simple that there are obviously no deficiencies*, I augment it by documenting its critical subsystems with the appropriate role models. For the long term, I am augmenting the programming language with role model concepts so that the system aspects become an explicit part of the code. I confidently expect that this will take us another step towards the goal of self-documenting code.

TO THE EDITOR: REPETITION OF THE FIGURE TEXTS IF YOU WANT TO TYPESET THEM YOURSELF

THE TEXT OF FIGURE 2

```
Object subclass: #NetworkNode
    instanceVariableNames: 'predecessors successors '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'ActivityNetworks'

setPredecessor: pred
    (predecessors includes: pred)
    ifFalse:
        [predecessors add: pred.
        pred setSuccessor: self].
setSuccessor: succ
    (successors includes: succ)
    ifFalse:
        [successors add: succ.
        succ setPredecessor: self].
initialize
    predecessors := OrderedCollection new.
    successors := OrderedCollection new.

NetworkNode subclass: #Activity
    instanceVariableNames: 'name duration earlyStart '
    classVariableNames: ''
    poolDictionaries: ''
    category: 'ActivityNetworks'

duration: aNumber
    duration := aNumber.
frontLoading: aMonthNumber
    (earlyStart notNil and: [earlyStart >= aMonthNumber])
    ifFalse:
        [earlyStart := earlyStart isNil ifTrue: [aMonthNumber] ifFalse: [aMonthNumber max: earlyStart].
        successors do: [:succ | succ frontLoading: earlyStart + duration]].
name: aString
    name := aString.
report
    Transcript cr; show: name printString , ' from start of month: ' , earlyStart printString , ' to end of month: ' , (earlyStart
+ duration - 1) printString.
    successors do: [:succ | succ report].
reset
    earlyStart notNil
    ifTrue:
        [earlyStart := nil.
        successors do: [:succ | succ reset]].
initialize
    super initialize.
    duration := 0.
    earlyStart := nil.

Activity class methodsFor: testing
test1: projectStart
    | act1 act2 act3 act4 act5 act6 act7 act8 |
    act1 := Activity new initialize name: 'Foundation'; duration: 1.
    act2 := Activity new initialize name: 'Framework'; duration: 2.
    act3 := Activity new initialize name: 'Roofing'; duration: 1.
    act4 := Activity new initialize name: 'Outside paneling'; duration: 1.
    act5 := Activity new initialize name: 'Outside painting'; duration: 1.
    act6 := Activity new initialize name: 'Inside walls & ceilings'; duration: 3.
    act7 := Activity new initialize name: 'Paint Walls and Ceilings'; duration: 2.
    act8 := Activity new initialize name: 'Completion'; duration: 1.
    act2 setPredecessor: act1. act3 setPredecessor: act2. act4 setPredecessor: act2.
    act5 setPredecessor: act4. act6 setPredecessor: act2. act7 setPredecessor: act6.
    act8 setPredecessor: act3; setPredecessor: act5; setPredecessor: act7.
    act1 reset; frontLoading: projectStart; report.
```

THE TEXT OF FIGURE 3

```
'Foundation' from start of month: 1 to end of month: 1
'Framework' from start of month: 2 to end of month: 3
'Roofing' from start of month: 4 to end of month: 4
```

```
'Completion' from start of month: 9 to end of month: 9
'Outside paneling' from start of month: 4 to end of month: 4
'Outside painting' from start of month: 5 to end of month: 5
'Completion' from start of month: 9 to end of month: 9
'Inside walls & ceilings' from start of month: 4 to end of month: 6
'Paint Walls and Ceilings' from start of month: 7 to end of month: 8
'Completion' from start of month: 9 to end of month: 9
```