ISSN 0909 6264

Trykt af IMM - DTU

Minimalist Software Engineering I: Definition

IMM, Numerical Analysis Group

Per Skafte Hansen

Abstract:

The Software Life Cycle Approach is a well-established and to some extent standardized method for managing programming projects. Its full version is primarily of use to larger groups of programmers, hierarchically organized and working to external specifications. The minimalist version presented here is aimed at the very small team (one or a few programmers) working on tiny projects.

Author's preface:

New readers can begin here: this report contains the second shortest description available of standard software engineering techniques. It also contains the shortest, in an appendix. The typical reader is anyone interested in developing better programming habits but too busy with specific programs to read a 500-page book.

A reader who is already well versed in software engineering will find no startling novelties is this pamphlet. The five pages (pp. 4-8) containing the definition of the Minimalist Software Life Cycle are, essentially, a digest of existing standards. Note, however, that a description of the concept of Configuration Management is placed *before* the description of the life cycle phases. This is done because I believe that Configuration Management is unduly neglected in the literature, while it is alpha and omega of programming practice, as evidenced by the frequent disasters caused by its equally frequent absence.

I originally intended to include a set of guidelines for the practice of Minimalist Software Engineering with this report, but changed my mind for three reasons:

1) This pamphlet ***must*** be kept so short that there is no excuse for not reading it

2) Extracting the guidelines from the literature and writing them down for others to use has already taken much longer than I anticipated, especially because the guidelines must be minimalist, too

3) I would like to have more hands-on experience with the minimalist software engineering approach before advicing anyone else on the subject

The guidelines (an estimated 20 pages) will appear in a separate report. Until then, read the next 8 pages a few times extra and/or produce your own guidelines.

1 Scope

The purpose of this note is to introduce a minimalist version of the "Software Life Cycle", a now classical approach to software engineering. The minimalist version is recommended for use in *tiny software projects*. Typical examples of programming efforts in this category are

M.Sc. thesis projects that involve programming
Development of small function libraries for in-house use
Proof-of-feasibility programs developed as parts of research projects

The note *defines* Minimalist Software Engineering by describing all elements of a minimalist Software Life Cycle. By adhering to the life cycle approach, the programmer can monitor the stepwise development of a program with every step well defined and all outputs verifiable against specifications, up to the level of rigour adopted for the project at hand. **Note**: the use of rigourous, formal methods is *not* required. The goal of software engineering is orderliness.

1.1 Definition of the Software Life Cycle

The minimalist software life cycle spans the activities from idea to finished program and hence pays little or no attention to delivery, operation and maintenance. The cycle itself consists of five distinct phases, each covering a very clearly defined job:

Software Configuration Management Planning
User Requirements Specification
Software Requirements Specification
Design
Implementation

It is of the utmost importance (and will be emphasised again and again) that each phase is carried through to completion before the next phase is begun.

Associated with each phase is a set of outputs ("*deliverables*", or "*deliverable items*"). The following sections describe — in fairly abstract terms — what the activities and outputs must be for each phase.

1.2 How to use this note

Read everything very carefully, and try to assign a specific meaning (and a level of importance) to every item. For some projects, certain items may be ignored altogether. Read the section on Software Configuration Management again and begin there. Then follow the prescripts for each phase.

Software Engineering is iterative and it is an integral part of the life cycle concept that each phase can be frozen at any stage if major specification or design flaws are discovered. If the Configuration Management is done correctly, it will in such a situation be possible to go back and re-start from an earlier phase and to do so in a disciplined manner.

1.3 Acronyms

A number of acronyms are used in the text. They are defined on their first occurrence but are also collected here for ease of reference:

ADD	Architectural Design Document
ImD	Implementation Document
LC	Life Cycle
SCM	Software Configuration Management
SCMP	Software Configuration Management Plan
SR	Software Requirements
SRD	Software Requirements Document
UR	User Requirements
URD	User Requirements Document

1.4 References

The main sources of inspiration are two well-established sets of standards:

ANSI/IEEE Std. 729-1983 Glossary of Software Engineering Terminology
ANSI/IEEE Std. 830-1984 Software Requirements Specification
ANSI/IEEE Std. 828-1983 Software Configuration Management Plans

ESA PSS-05-0 Software Engineering Standards
ESA PSS-05-02 Guide to the User Requirements Phase
ESA PSS-05-03 Guide to the Software Requirements Phase

Also, the following books must be mentioned:

General Electric Software Engineering Handbook,
McGraw-Hill 1986

R.S. Pressman: "*Software Engineering*",
European Adaptation, McGraw-Hill 1992

John J. Rakos: "*Software Project Management for Small to Medium Sized Projects*",
Prentice-Hall 1990

2 Configuration Management

The name Software Configuration Management (SCM) is dreadfully pompous, but the actual activity is simple. The purpose of SCM is to make the Life Cycle approach work (—!—), essentially by setting up rules and making sure that they are followed. In abstract terms Configuration Management consists in

identifying *configuration items* in a system, and
controlling the release and change of these throughout the production period

Software Configuration Management for the Life Cycle approach must

identify the deliverable items of each phase
specify means of monitoring the state of a deliverable item
ascertain that each item is finished before the production of a dependent item is begun
provide clear-cut *stop-and-backtrack* mechanisms
provide documentation lay-outs

To the one-man project team this may seem too much; but notice that almost all programming errors, at all levels, arise from attempts to do two things at the same time or from having done one thing in two different ways (or only half-way), or from having two parts of a program built on the assumption that the other part does the thing. Hence, Configuration Management is absolutely mandatory and a few plain words will be added here:

> Every programmer knows the effect of accidentally using an old version of an otherwise finished module along with a new module under development. The discipline of avoiding this mistake is known as Version Control, and many of the better programming tools automate this to some extent. SCM is Version Control in the large.
>
> In every phase, at every stage, an item will undergo iterative improvement until it is considered ready for use. SCM must tell (by specifying rules, lay-out etc.) how to signal that work on an item has begun, where to store intermediate versions, how to tell these apart, how to signal that the work is done, where to store the final version. This goes for documents as well as for code.
>
> In particular, SCM takes care of *change control*. See description below.

Some key words need explanation before the formal definition can proceed:

A *project data base* may be as simple as three directories containing old versions, version under development, finished versions; and a binder with two different-coloured pieces of cardboard separating old documents, document under preparation and finished documents.

While a piece of code or text is being developed, anything goes, no questions are asked. As soon as it is added to the project data base for use in the development of other items, it becomes a *baseline*. A baseline is a finished version. It can only be changed through a formal procedure of specifying (and writing down) the changes needed, placing a copy of the item with the collection of old versions, removing the item from the collection of finished versions, performing the changes, verifying them, and adding the item to the data base again.

SCM must specify what the baselines are for each phase of the life cycle, and what procedure to follow when a baseline needs changes.

2.1 The Software Configuration Management Plan

The SCMP is built during the life cycle. An overall plan is worked out in sufficient detail to allow work to begin, and each phase of the life cycle ends with the writing of a specific SCMP for the next phase. An outline of a typical SCMP is as follows:

Introduction:

Purpose: name of project, purpose of this (part of) SCMP
Scope: items, participants, activities
Definitions and acronyms
References

Management:

Names of agreed-upon requirements to be managed by this SCM
Milestones, baselines, tests and reviews
Policies: the SCM procedures or policies (not specific methods yet) for
- naming conventions
- version control
- document formats and processing

Activities:

Configuration identification:
- items that make up a base line
- acceptance criteria for base lines
- titling, labelling and numbering of code and documents

Configuration control:
- rules for change proposal, approval, documentation, back-up

Tools, etc.:

"Third party" tools, techniques and methods can be described here

Records:

Details of decisions concerning SCM documents to retain, where and how.

2.2 Some "mandatory practices"

Every single item in a project shall be subjected to Configuration Management
Every single item shall have a unique identifier
Every finished item shall be documented/labelled with at least
- identifier
- description of purpose
- author name
- creation date
- creation history (dates of changes)

The project library shall consist of at least
- a static library (old versions for reference)
- a master library of baseline versions
- a dynamic library of items under development

The SCMP shall specify
- back-up procedures
- change request, approval and verification procedures

3 User Requirements Phase

To the impatient programmer, the life cycle approach will probably seem never to get to the fun part. Also, the early phases may be difficult to tell apart. The User Requirements Phase (UR phase) defines the *problem* to be solved in the project, ***as it appears to the user***, specified into

- capabilities needed by the user
- constraints in the form of requirements of adaptability, portability, security etc.

If the UR phase is not carried through, parts of the program produced will contain too many features, others too few. — So there you are...

3.1 Output from the UR phase

User Requirements Document (URD)
Plans for (user) acceptance tests
Software Configuration Management Plan for the Software Requirements Phase
User's outline of the User Manual for the projected software

3.2 User Requirements Document, outline

Introduction:

Name and purpose of product
Definitions and acronyms used in URD
References for URD
Overview/Abstract of URD

Description:

User characteristics (who is the user)
Assumptions made in the project
[Operational environment]

User Requirements:

Capabilities
Constraints

3.3 Some "mandatory practices"

Essential requirements shall be marked as such
Source of each requirement shall be stated (for reference in cases of doubt)
Each requirement shall be verifiable
The URD shall always be produced before the Software Requirements Specification phase begins

3.4 Comments

Another, perhaps more suggestive term for "capabilities" is *features*. The UR phase may take the form of a mild brain storm that collects all conceivable features of relevance, followed by a careful organization of these into a hierarchy, possibly with priorities attached.

4 Software Requirements Phase

The aim of the SR phase is to build a ("logical" or "conceptual") model of *what* the software will do, without implementation details, but specified into

Functionality: purpose of each functional element
Interfaces: to hardware, other software, data bases
Operations: how to run the software, screen lay-out, error messages, commands
Verifications: how to verify that the final software matches user's requirements
Documentation

4.1 Output from the SR phase

Software Requirements Document
SCMP for the Design Phase
Reviewed outline of the User Manual (incl. recommended lay-out)

4.2 Software Requirements Document, outline

Introduction:

Name and purpose of product
Definitions and acronyms used in SRD
References for SRD
Overview/Abstract of SRD

Description:

Relationship(s) to other software products
Constraints derived from URD
Top-down description of model

Requirements:

Functions (*i.e.* functionalities, not program elements)
Interfaces
Operations
Verification procedures
Documentation, incl. verification of SR against UR

4.3 Some "mandatory practices"

The SCMP for the SR phase, as developed in the UR phase, must be followed
Each function or item in the SRD must be traceable to the URD
Each requirement in the SRD shall be verifiable
The SRD must be produced before the Design Phase is entered

4.4 Comments

The SR phase is the most difficult part of the SLC, simply because conceptual modelling is the most difficult part of programming. If the UR is specified in terms of features, the SR may be expressed in terms of natural objects having these features and in components of an abstract machine managing the objects: an interactive graphics program can be discussed in terms of screen images and the items shown on them, a mathematical/numerical library in terms of mathematical abstractions such as matrices, equations, relations etc. As in the UR phase, the aim is to establish a hierarchical structure in the requirements.

5 Design Phase

Still no coding — the Design phase (sometimes Architectural Design Phase) is where the code-to-be is specified in minute detail.

A method must be chosen (pseudo-code, JSP, JSD, OMT, SADT, VDM or whatever) and strictly adhered to. Design proceeds top-down as far as at all possible, but with the option of completing some part of the design (in another fashion) where this is feasible and absolutely necessary, for instance to answer a design question arising elsewhere in the design.

NB: After the Design phase, the software **must** be so completely specified in terms of (public) data structures, function names and interfaces, that the actual coding could — almost — be left to a third party.

5.1 Output from the Design Phase

Architechtural Design Document
SCMP for the Implementation Phase, including lay-out and naming conventions
Verification of design against SRD
User Manual, except for implementation-dependent details

5.2 Architectural Design Document, outline

Introduction:

Name of product
Definitions and acronyms used in ADD
References for ADD

Design method:

Choice of method: motivation, brief description

Overview:

Top level model description
Data structures
Control flow

Components:

For each component:
- Type
- Purpose
- Public data structures used
- Function description, incl. private data structures
- Dependencies

Verification of Design against SR

5.3 Some "mandatory practices"

The method chosen must permit top-down design. The components may be divided into disjoint *sets*, to be treated separately.
Only one method can be used *per* component set. Preferably, only one method overall.
The ADD must completely specify the code, leaving only elementary implementation issues
The ADD shall be thoroughly reviewed before coding begins

6 Implementation

This is where the coding is done. The process is as follows:

1. Using the ADD, write out a detailed plan of order of implementation
2. To supplement the conventions of the SCMP, produce, and adhere to a set of conventions for
   Detailed error handling
   Flagging of heuristics in code
3. As far as possible, proceed strictly top-down
4. Always work on one unit at a time and finish it before beginning work on the next
5. Where absolutely necessary, proceed strictly bottom-up:
   low-level i/o, device-drivers etc. may have to be produced first
6. Adhere strictly to the SCMP
7. Test program bottom-up:
   functions
   integration of functions to modules
   integration of modules

6.1 Output from the Implementation Phase

Code
ImD
Finished User Manual
Verification of ImD against ADD
Test report(s)

6.2 Implementation Document, outline

Introduction:

Name of product
Definitions and acronyms used in ImD
References for ImD

Standards and conventions used:

Detailed design
Algorithms
Naming
Low-level documentation

Components:

For each component (as for ADD, but giving algorithmic details where necessary):
Type
Purpose
Public data structures used
Function description, incl. private data structures
Dependencies

[Verification of implementation against ADD]

7 APPENDIX: Novice programmers

For introducing novices to programming discipline, a further compacted version of the definition on *pp.* 4-8 is needed. The text below is an edited translation of a set of *hints* given to civil engineering students in a C programming course.

Tiny programming tasks can be divided into four phases:

Specification of user requirements
Analysis of the operations to be performed by the problem
Design of the architecture of the problem
Implementation

User Requirements

Every program supplies a service to some group of *end-users*, so:

- The program must solve specific problems
- The program must appear in a specific way

The first phase of programming must clarify these two requirements to the point where they can be met precisely: what are the queries and expected answers; how does the user start and stop the program, handle it while it runs and obtain the answers as they are generated.

Analysis

When the user requirements are established, they can be split and recast into single processes to be performed by the program. In this phase, one does not use words or concepts belonging to a programming language, and no attention is payed to the possible appearance of the final *code*. Instead, the goal is to build a *mental model* of the program

Design

In the design phase, the architecture of the program is planned so carefully that the actual coding could (in principle) be left to a third party. Programming language may be used, but algorithmic details are omitted. The typical design phase starts from the processes described by the analysis and: 1) specifies data formats; 2) organizes the processes in a logical fashion so that each process gets the relevant input and delivers the intended output.

Implementation

Only now does coding begin: technical programming problems are solved one at a time; the logically separate parts are implemented and tested one at a time; co-operation between elements is likewise tested; then the program is gradually built from the parts.

Some remarks:

Program development is an iterative process. The fundamental idea of *The Life Cycle Approach* is to keep a clear view of how far the programming has progressed. Therefore:

- Each phase must be *completely finished* before the next begins
- When errors are detcted *all work from the point in error and onwards is re-done*
- One ***never*** skips a phase, not even when correcting an error
- After each phase, a *walkthrough* (*i.e.* a simulation on paper) is done to check that the results meet the requirements formulated in the previous phase.