Software Engineering Group Projects – Design Specification Standards

Author: C. J. Price, N.W.Hardy and B.P.Tiddeman
Config Ref: SE.QA.05A
Date: 29th September 2013
Version: 1.7
Status: Release

Department of Computer Science
Aberystwyth University
Aberystwyth
Ceredigion
SY23 3DB

CONTENTS

1 INTRODUCTION

1.1 Purpose of this Document

The purpose of this document is to describe the format of, and information which must be supplied in, design specifications produced in software engineering group projects.

1.2 Scope

This document specifies the standards for writing software design specifications. It describes the necessary layout and content of a design specification.

This document should be read by all project members. It is assumed that the reader is already familiar with the QA Plan [1].

1.3 Objectives

The main objective is to aid the production of a design specification which is a complete and accurate translation of the client's requirements into a description of the design elements necessary for the implementation phase. These will generally include the software structure, components, interfaces, and data. In a complete design specification, each requirement must be traceable to one or more design entities.

2 RELEVANT QA DOCUMENTS

The Design Specification must be produced in accordance with the quality standards specified in the QA Plan [1]. In particular, it must be produced as part of a specified task in the Project Plan [2] and maintained within the Configuration Management System according to the appropriate procedures [3].

The basic layout and information content must conform to the general documentation standards [4], which, amongst other items, specifies that there must be an introductory section and a references section. The remainder of this QA document (i.e., SE.QA.05), is concerned with the other sections which should be included in a design specification.

An example design specification following this format is provided in the web site for the group project module. There is also a requirements specification for the same example, so that they can be compared.

3 OUTLINE STRUCTURE

An outline structure for a design specification is presented in Table 1. All design specifications should have this general structure. Since there is a wide variation in the kind of systems that could be developed in group projects (e.g., games, simulations, database applications), this outline structure is only a recommendation, and is not mandatory. The design specification should address the same issues as those indicated in table 1, but in an order or format appropriate to the particular system being specified. By default, the suggested outline structure should be used.

1 INTRODUCTION (structure as defined in QA Document SE.QA.03)

2 DECOMPOSITION DESCRIPTION
- 2.1 Programs in system
- 2.2. Significant classes in each program
  - 2.2.1 Significant classes in Program 1
  - 2.2.2 Significant classes in Program 2 .....
  - 2.2.n Significant classes in Program n .....
- 2.3 Modules shared between programs
- 2.4 Mapping from requirements to classes

3 DEPENDENCY DESCRIPTION
- 3.1 Component Diagrams
  - 3.1.1 Component Diagram for Program 1
  - 3.1.2 Component Diagram for Program 2
- 3.2 Inheritance Relationships

4 INTERFACE DESCRIPTION
- 4.1 Class 1 interface specification
- 4.2 Class 2 interface specification . . .

5 DETAILED DESIGN
- 5.1 Sequence diagrams
- 5.2 Significant algorithms
- 5.3 Significant data structures

6 REFERENCES (structure as defined in QA Document SE.QA.03)

APPENDICES

Table 1: Outline Structure for Design Specification

A description of each element mentioned in the outline structure is presented in the following sections.

The terms *class* and *package* are used to describe the logical and syntactic structuring of data and procedures within an application. Programming languages and environments may support such scoping concepts in a variety of ways and to varying extents. Structuring the design at this level is an important discipline. The term *method* may be taken to refer to function, procedure, subroutine etc. as supported by the target language.

4 DECOMPOSITION DESCRIPTION

Extract from document structure:

2 DECOMPOSITION DESCRIPTION
- 2.1 Programs in system
- 2.2. Significant classes in each program
  - 2.2.1 Significant classes in Program 1
  - 2.2.2 Significant classes in Program 2 .....
  - 2.2.n Significant classes in Program n .....
- 2.3 Classes shared between programs
- 2.4 Mapping from requirements to classes

The decomposition description records the division of the software system into programs and the modules which make up those programs. It describes the way the system has been structured and the purpose and function of each program and significant module, giving an overview of, and justification for, the design. Obviously, if there is only one program in the design, then the headings can be simplified.

4.1 Programs in system

If the system being implemented is made up of more than one program, then this section should describe each of the programs, and the relationship between the programs (e.g. that they will run on different machines and that program 2 will take output from program 1 as its input).

If there is only one program, then this section should just give a brief summary of what the program does.

4.2 Significant classes

For each program, the main classes should be named and a short description of the purpose of each class should be given. The statement of purpose should be an English language description just a few sentences long. A specification for each module should *not* be given here – that level of detail is provided later in the document.

If there are a set of classes shared between several programs, then they should be described in a separate section. In fact, a tidier solution is to make them into a Package, and have a section describing each package.

4.3 Table mapping requirements onto classes

In order to understand the implications of any changes in requirements or in the design, and to check that all requirements have been met, you should produce a table mapping functional requirements onto the classes which contribute to those requirements. For example:

Requirement	*Classes providing requirement*
FR1	HeatingClass, DiaryClass, DiaryObject
FR2	LogClass, DiaryClass, DiaryObject
etc	

5 DEPENDENCY DESCRIPTION

Extract from document structure:

3 DEPENDENCY DESCRIPTION
- 3.1 Component Diagrams
  - 3.1.1 Component Diagram for Program 1
  - 3.1.2 Component Diagram for Program 2
- 3.2 Inheritance Relationships

The dependency description specifies the relationships and dependencies between modules. It helps the reader to understand how the parts of the system described in the decomposition description fit together. It describes the general architecture, such as Model-View-Controller. UML provides a formalism called *Component Diagrams*. These are an appropriate way of describing the relationship between modules for Java programs. The specification should contain a subsection containing a component diagram for each program, showing the method links between modules.

The other types of dependency which can be shown are compilation dependencies between modules, and inheritance dependencies between classes. The group project team may choose to include or omit these things depending how appropriate they are for the particular application.

6 INTERFACE DESCRIPTION

Extract from document structure:

4 INTERFACE DESCRIPTION
4.1 Class 1 interface specification
4.2 Class 2 interface specification . . .

This section should provide everything designers, programmers and testers need to know to use the facilities provided by a module. It should include an outline specification for each *class* or *interface* in the system. The specification should include:

1) The name and type of the class or interface (e.g. public or private, abstract for a class).

2) Classes or interfaces which it extends (and why).

3) Public methods implemented by the class or interface. The parameter names and types for each method should be specified, and a short summary of the purpose of the method given.

The simplest way of specifying each class might be as a Java code outline, following the Java coding standards [5]. Certainly it makes sense for all names to follow the Java naming standards at this point. Several tools are capable of generating the code automatically from UML class descriptions. The Java specification is preferable in this case, and should follow the Java coding standards [5] and should compile together into a system (that will do little as it only contains outlines of classes).

Where it is possible and practical to do so a Java *interface* should be specified rather than a *class*, together with one or more factory classes containing static factory methods which return an instance of the interface type. This allows changing of the implementation at a single point (the factory method(s)), allowing easier modification, experimentation etc.

The ordering of class descriptions in the document has not been specified here, but it should follow a sensible convention. One possibility is all classes in alphabetical order. Another choice would be to order by program, to match the decomposition section.

For target languages which do not support class-based scoping, these guidelines must be adapted accordingly.

7 DETAILED DESIGN

5 DETAILED DESIGN
5.1 Sequence diagrams
5.2 Significant algorithms
5.3 Significant data structures

Extract from document structure

It is unnecessary and inappropriate in the group project to provide all of the internal details of each module. However, you should have considered the difficult parts of the design, and the way in which all of the classes work together. For any module whose internal workings are not self-evident there should be detail to demonstrate the feasibility and coherence of the overall design. Specifying the interface for an intracable module leaves the whole design intractable.

UML *sequence diagrams* are one good way of documenting how the classes work together for the major operations of the program (identified by use cases, described in the user manual for the group project). *State diagrams* and *activity diagrams* may also be of use.

During design, the team will have performed experimental programming and theoretical investigation to reduce the risk associated with difficult parts of the implementation. Decisions on difficult parts of the system should be documented in this section. The most appropriate way of doing this for algorithms is likely to be as a textual description of what is to be done. Where code exists, it might be referred to, but probably should not be included.

Significant data structures occur when a number of objects are linked together in a more complex structure. Class diagrams showing the entity relationships between classes should be drawn where appropriate, along with object diagrams which show how the static relationships in class diagrams work out in real examples.

The mechanism to support any persistant data must be described. Language specific mechanisms may be cited and need not be explained (for example the JPA for Java [5]).

REFERENCES

[1] QA Document SE.QA.01 - Quality Assurance Plan.

[2] QA Document SE.QA.02 - Project Management Standards.

[3] QA Document SE.QA.08 - Operating Procedures and Configuration Management Standards.

[4] QA Document SE.QA.03 - General Documentation Standards.

[5] QA Document SE.QA.09 - Java Coding Standards.

DOCUMENT HISTORY

Version	*CCF No.*	*Date*	*Changes made to document*	*Changed by*
1.0	N/A	05/10/01	Document given complete rehaul	CJP
1.1	N/A	18/07/02	Minor changes after review with Graham Parker.	CJP
1.2	N/A	12/08/03	Got rid of compilation dependencies. Added sequence diagrams.	CJP
1.3	N/A	26/09/04	BN10 - more explicit about sections 6 and 7.	CJP
1.4	N/A	14/09/06	Renamed as SE.QA.05A as 5B now created	CJP
1.5	N/A	12/09/08	Changed document template to be Aber Uni	CJP
1.6	N/A	05/11/10	Updated to Docbook; minor style changes; non-OO target languages referenced; %B material added; ref. persistence mechanisms.	NWH
1.7	N/A	29/09/13	Reverted to word doc. Added a reference to using interfaces and factory methods.	BPT