



SOFTWARE SPECIFICATION

Jonathan Tilbury
TESSELLA SUPPORT SERVICES PLC

Issue V1.R1.M1
September 1999



SOFTWARE SPECIFICATION

Introduction

As everyone knows, software is highly complex and often designed to fulfill a number of functions. The complete specification of a system is thus a very difficult task. It is however very important as according to many industry surveys, poor software specification is one of the most common sources of project failure.

This supplement aims to introduce you to some of the approaches used to specify software and give advice on the circumstances in which these approaches can be used.

Project Lifecycles

When producing software, the traditional approach is to specify your requirements, then design the system, write the code, test the system, and release it to the users. This is known as the **waterfall lifecycle** because there is a general one-way movement through the system. This approach enables the system to be fully specified before design and development are started. As you may appreciate, this is both a strength and a weakness. It keeps the developers to the point and allows a good estimate of the overall cost to be performed, but it means the specification must be complete and accurate.

Several variations on this approach can occur. For example, the requirements can be specified, a framework designed and then the functions implemented, tested and released as a series of iterations, starting with the most important. This **iterative development lifecycle** is good where all requirements are known in advance but a quicker delivery of key features is required.

The system architecture can be produced with full knowledge of all the requirements allowing for a gradual production of the features. This allows the users to provide feedback before development is complete, increasing their confidence and involvement. It is essential, however, that all of the requirements are known from the start and it is only suitable for systems that can be partially delivered. Due to repeated testing and delivery stages, the cost of the project will be higher.

Another approach is the **evolutionary development lifecycle**. This treats the project as a series of waterfall lifecycles, performing requirements, design,

development and delivery in each cycle. Each cycle builds on the documents and code produced in the previous cycle. This is a flexible approach that gets deliverables to the user quickly, but it runs the risk of needing to re-design the system if complex requirements are introduced at a later stage. Again, due to repeated testing and delivery and the need for re-worked development, the cost will be higher.

In many circumstances the business model is not fully defined or understood at the start of the project. In these circumstances a **Rapid Application Development (RAD)** approach may be used. This approach develops the system as a series of prototypes, some of which are finished and released to users. The RAD allows the specification to grow as the prototypes are produced, but means the final deliverable is not known at the start of the project. Although this is the most flexible approach, it makes budget definition difficult and is only suitable for certain types of products.

The Waterfall Lifecycle

As described above, the traditional specify-design-develop-release life cycle means the software must be carefully specified up-front. In practice, software specification can be broken down into three stages:

- **User requirements**, where the user says what they want in user terms.
- **Software requirements**, where the functions of the proposed software are described in general but clear terms.
- **Architectural Design**, where the software developers describe the overall system structure and may specify some of the critical features such as algorithms or the graphical user interface structure.

User Requirements

This stage allows users to define what they want the system to do. The best method for gathering user requirements is through a series of interviews with the users. It is important to review any existing systems currently in use as this will provide vital clues to the users expectations. Users should be encouraged to describe what they currently do and the problems they encounter.

This approach leads us to one of the most common difficulties with the user

requirements stage: users are often unaware of the opportunities new technology can offer. Encouraging users to think beyond their current experience is one of the challenges of successful user requirements specification.

There are several approaches that will steer user input towards what can be achieved with the new system. One approach is to produce a simple prototype. Although this is intended solely to stimulate ideas and is generally thrown away afterwards, it is very effective at getting users to think in new ways. Demonstrating similar systems can also help. This can be backed up by a brainstorming session, where users and software developers suggest possible options and ideas based on their experience of other systems.

The main output of this process is a User Requirements Document (URD), containing all the user's requirements in an ordered and controlled manner. It covers aspects of what the users want from the system and is written in their own terms. It is best represented as a table of requirements with each one kept short and distinct. It also provides a basis from which tests can be constructed to evaluate the final outcome against the user's initial requirements.

The user requirements are best broken down by type. The first major section is functional requirements. This covers what the system should do and how the user might use it. Example sub-headings include data to be maintained by the system, interfaces to other systems, communication with hardware, external files, required calculations, display of data, editing of data, backup and retrieval of data, user interface characteristics and installation.

The next major section should contain, what are generally called, performance requirements. This should cover the speed, capacity and reliability of the system. Users should be as explicit as possible and produce testable measures.

The final section should include design and implementation requirements. Safety and security issues should be categorically stated and their implications explained. Often there are in-house requirements for how the system will be produced and what it should run on. This section should include the target platforms, required development tools, and other project requirements such as documentation expected by the users.

It should be remembered that the user requirements are concerned with asking

the question “can we have this?”. The software requirements phase that follows will respond with what the software developers intend to deliver. Any omissions, corrections or impossible requirements should be identified in this next stage.

Use Cases

Users may be unfamiliar with documenting their requirements in such precise terms, finding it easier to write a set of scenarios or Use Cases. These are step by step documents giving examples of the systems in use. As well as clarifying the user requirements, these can be used as the basis for acceptance testing.

An example section from a Use Case might be:

1. Extract a named protein from the database.
2. Display the structure on the screen as a linear and circular display.
3. Search the protein for a particular sequence.
4. Show the sequences found on the graphical displays.

Software Requirements

The software requirements phase is the most important specification phase. The output is the Software Requirements Document (SRD). This is a description of the software that will be delivered and follows a significant amount of analysis into what the users have asked for and what is possible. Critically the requirements are written in system rather than user terms although presented in a similar format.

The key word that distinguishes the software requirements phase from the user requirements phase is *analysis* - it can be thought of as the “system analysis and specification” phase. Although the input represents what the user has asked for, it is analysed by an experienced software developer who will produce a high level logical architecture for the system. Each element can then be described in detail.

The mechanism for documenting the high level logical system description varies according to the type of system. Most commonly, it is a combination of a data flow diagram (DFD) and an entity-relationship diagram (ERD). These are very simple to understand and help illuminate the detailed requirements without going into too much detail. The notation can follow a standard such as SSADM or UML, or can be home grown. A simple example of a DFD is shown in figure 1. Whatever the approach, the aim is to break the system down into manageable, logical blocks allowing an ordered, understandable description of the system.

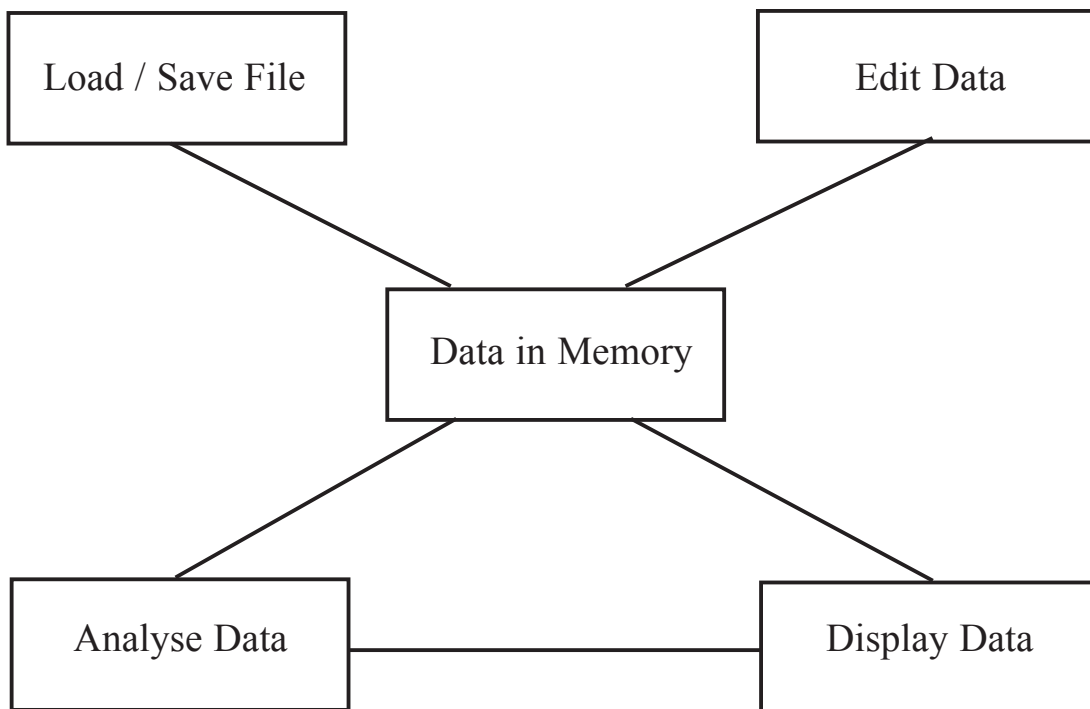


Figure 1: A simple data flow diagram

Once this basic system decomposition has taken place, the requirements for each of the components must be documented. In the example above, separate sections should be written for each block in the diagram. The requirements are documented in system terms as a series of short, testable statements. This should cover the functions of each block, the speed and capacity requirements and the external interfaces.

Once each of the major functional blocks has been identified, the system wide requirements must be documented. This should cover requirements for any user interface, delivery platform, documentation, functionality, installation, security and safety, quality reliability and maintenance requirements. It should also detail the functionality to allow testing to occur.

Judging the level of detail of the requirements is one of the most difficult aspects of this phase and a skill required by the author. The level of detail should be flexible enough to enable the developers to write the system without constraining the design stage too much.

One of the major pitfalls of this stage is to over specify any graphical user interfaces (GUI). This is particularly true of windows based systems where the

system functions are held within the interface - it becomes very difficult for the user to distinguish between the interface and the functions. Disciplining the author to document the functions, and not how they are accessed, is often very difficult, but at this stage very important. The SRD should only contain high level principles of the GUI, and not a detailed specification.

Other pitfalls arise due to lack of precision and clarity. This mostly occurs when describing the delivery platform or the system performance.

Examples of poor requirements include the following:

- The system must read any comma separated file.
- The system must calculate the mean value within 2 seconds.
- The system must run on any PC running MS Windows.

Does this mean the system must load even a multi-Gigabyte file, and that the mean of this must be calculated in 2 seconds on a 486MHz PC running Windows 3.0?

Better requirements would be:

- The file size that can be loaded must be limited only by the physical memory of the PC and the configuration of Windows. There must be no hard file size limit.
- The reference test file has 5 columns and 2000 rows.
- The system must calculate the mean of one column of the test file in 2 seconds.
- The minimum PC specification that the system must be run on is a 200 MHz Pentium PC with 100 Mb of spare disk space and 60 Mb of memory.
- The system must run under MS Windows NT 4.0 (Service pack 3).

Another pitfall is conflicting requirements. These may be functional or non-functional and lead to difficulties in testing and delivering the system. Often conflicting requirements are in separate sections of the SRD. Examples are

- The system must be accessible from Netscape version 4.04
- The system must be written in Java version 1.1

In this case Java 1.1 does not work in Netscape 4.04, so this leads to a conflict.

The above examples show the level of detail required and should avoid some of the many other undeliverable requirements software engineers are presented with. I challenge anyone to devise a test to prove the following (true) examples!

- The system must be fast
- The system must be portable
- The system must be user friendly

The final section of the software requirements cross references each requirement back to the initial User Requirements Documents. Another key feature of this phase is that software requirements do not have to fulfill every user requirement and could even introduce new requirements. However, both of these situations must be justified. Some user requirements might be satisfied by hardware or other external systems, or the system analysis performed might show up problems. User requirements may be shown to be impossible, superfluous, or missing!

The best way to perform the system requirements phase depends on the type of system, level of detail provided by the users and the size of the project. Typically it is performed by a small number (usually one) of senior software engineers. Interviews with users and other stakeholders are performed and a draft is produced. An early review of this draft will show if the system requirements phase is on target.

A prototype written during the software requirements phase may help software engineers and the users agree on what will be delivered. It also allows the engineers to explore what is possible using the expected solution routes. Formally the prototype should not be used to specify the interface as this can waste a significant amount of time and divert the team from the real goal which is to specify the software functions. Realistically, many users can only visualise the system through a user interface, so giving them a GUI to focus on enables them to specify the functions with more accuracy.

It is sometimes difficult to write software requirements without steering too far towards a particular solution. This again depends on the skill of the author who will include a lot of his own ideas - these must be checked by the users and the

person paying for the project - but should not compromise the selection of the development technology unless this is one of the requirements.

The final Software Requirements Document must be reviewed and accepted by a cross sample of users and any decision makers responsible for authorising the budget (or their nominated representative). This process can be eased by a formal review meeting, where the author helps those unfamiliar with SRD's to understand the document's full implications, preventing any potential misunderstandings. The author must also ensure that approval is given at this stage.

Architectural Design

The user and software requirements phases defined what the users want and what the system can do; the architectural design determines “how” the system will be produced. The architectural design phase aims to identify the major system components, define the form they should take, state how they interact and define their external interfaces. Certain aspects of the system, especially the graphical user interface and any complex algorithms are defined much more closely at this stage and this can be thought of as “specification”.

The architectural design takes the logical decomposition of the system produced earlier and extends it to correspond with the physical software components (and interfaces) that together will provide a feasible solution to the requirements. The term “components” is deliberately vague, depending on the type of problem at hand. It can cover practically all types of software entities, such as processes, programs, classes, subroutine libraries, databases, etc.

In a complex system there may be many levels of architectural design. For example, there may be an overall specification of the links between components and then individual designs of the database, user interface, analysis package, etc.

Architectural design is often performed using a standard design methodology such as SSADM or UML. These provide a common way of representing the system and are usually backed by software tools to make the documentation of the system easier. However in Tessella's experience a full implementation of a design methodology is only practical for large, many man-year projects. For smaller projects, taking a more flexible approach to the tools on offer helps speed the design without becoming bogged down in an overblown approach.

Each type of component is defined differently. The following examples show some approaches to this:

- Class library. This can be described using a simple description plus a diagram, or with a full class hierarchy plus a relationship diagram.
- Subroutine library. This may be a simple textual description plus a list of the main routines, a simple tabular description of each routine and its arguments, or a complete formal specification of each major routine.
- GUI. This is usually performed at a high level identifying and describing all the major components and the navigation between them. It will cover any style issues (often by reference to a style guide). A detailed description of each component can be done at this stage but is usually left until development.
- Programs/Processes/Threads. This can contain a textual description of each component, its interaction with data and other processes and any priority information. It may contain more low level information such as algorithms, internal data processing and flow of control.
- Files. This might be a simple description of the contents of the file or a full specification of the file format.
- Relational Databases. At its simplest this can be an entity-relationship diagram plus a description of the entities. However, it usually extends to table and column names, indexing strategies and entity life histories.
- Third-Party Software/External Systems. This covers a summary of the features of the system being used and the interface to it.

In addition to the individual components described above, there are many system-wide features that are specified in the architectural design. This includes the development environment to be used, the approach to error handling, any online help system to be implemented (although this may be treated as a separate component), any debugging or automated testing information required and the identification and approach for the critical speed and capacity bottlenecks.

The architectural design is cross referenced back to the Software Requirements

Document to show how each requirement is handled by the design. This is critical in making sure no requirements have been omitted.

How These Documents Are Used

There is a natural flow from the user requirements stage (“this is what I want”) to the software requirements phase (“this is what can be done”) to the architectural design phase (“this is how we are going to do it”). The documents are used in the following way.

The User Requirements Document is used as a source for the software requirements phase and is used as the basis of the user’s acceptance tests (often based on the Use Cases document). It is not however used during system design and development.

The Software Requirements Document is used as the basis for the architectural design and for the system tests written to prove the system performs as required.

The architectural design allows a team of developers to produce software knowing that it will all link together correctly. It keeps them focused on the requirements without doing their job for them. The architectural design is usually converted into a system maintenance guide at the end of the project.

Formal Specification Languages

There are a wide variety of formal specification languages available. These are often the result of research projects and are used in a wide range of systems. Examples are Z, VDM, VDM++, VVSL, RSL, Larch and Obj. Although all are slightly different, they seek to formally specify the system in mathematical terms so that it can be proved at testing time.

Typically these languages are based on model theory or abstract algebraic operations. For example a model based language such as Z, RSL, or VDM consists of a mathematical model built from simple data types like sets, lists and mappings, along with operations which change the state of the model. For example, a specification of a hotel reservation system would contain a mapping from room numbers to names and addresses of occupants (modeled as character strings), along with operations to add and remove guests and rooms, occupation dates etc.

Many formal languages of this type are supported by software tools. The

resulting specification can be carried through to design and development to allow the system to be formally ‘proved’, i.e. to prove that the source code produced actually fulfills the requirement. Some tools even provide source code generators, taking the specification language and generating source code that performs the functions specified.

Formal specification languages have been used successfully on a variety of systems, including compilers, databases, fault-tolerant storage systems, graphics software, medical warning systems, novel computing architectures, and security-critical message processing systems.

As might be expected, the use of formal specification languages is generally restricted to large complex systems where the specification stage is expected to take a significant length of time. They have a steep learning curve and the results are difficult to explain to users. However, when used for suitable systems they are provable and provide a direct link into design, development and testing.

What To Use When

The waterfall lifecycle is particularly suitable for reasonably sized projects where all the users are available and able to support the software developers and there is plenty of time to produce the system. Often however this is not the case. Whilst it is recommended to use the full approach whenever possible, shortcuts may be performed if they are likely to save money and time without increasing the risk of the project failing to deliver what was required.

The user requirement phase can be reduced in scale if the software requirements are to be produced in full. Where the users are unwilling or unable to provide a fully documented set of requirements, it is still very useful if they produce a set of Use Cases. This can be very illuminating to the software developers and can be used as the basis for the acceptance tests.

However small the system, it is always recommended that some sort of Software Requirements Document is produced. However, where the system is small, the solution obvious, and the development environment straightforward and intuitive (for example Visual Basic or MS Access), the Software Requirements Document can be extended to incorporate the architectural design. This can be done by mapping the physical components (forms, functions etc.) onto their logical descriptions. This can save time and allow a single simple specification and design document to be produced.

Rapid Application Development

The approach described above assumes that the requirements are fully understood at the start. What happens if the business model is not known yet? This is where Rapid Application Development (RAD) comes in. This assumes you only have a rough idea of what you require, and that you need to see the system in use before refining your specification.

The system is developed as a series of short stages. There are three types: a prototype stage that is used to demonstrate key features and to explore their functions, a development stage that finishes off the prototype functions and produces a tested if incomplete system, and finally a delivery stage that rolls the system out to users for immediate use.

The project runs using a small team comprising the software developers and empowered users. The stages can be ordered in any appropriate way. Typically, there are a series of prototype phases as the functionality is explored. A development stage may be followed by a delivery to the users, or more prototyping of another part of the system.

A RAD project is managed by breaking the project duration down into a series of time boxes and assigning stages to these. The order and type of stages are controlled to make sure the key business benefits are produced within the budget available.

In general there are several deliveries throughout the project. This is what makes the process “rapid” - the first release of the software is generally much earlier than with the waterfall approach.

Specification on a RAD project is achieved using the prototype. In essence the prototype *is* the specification. At significant milestones (for example when making a release to users) a system description document can be updated to ensure a record is kept of what the system should do to enable testing to be performed.

Obviously only certain types of projects can benefit from RAD. It is especially suitable to projects where the Graphical User Interface is the major component of the system. Systems with complex algorithms or real time control are not suitable for RAD development.

Tessella's Approach

Tessella's approach to software specification has been based on many years experience of software development, encountering both good and bad examples of how it is done, and the effects it can have on subsequent software development. This experience is now incorporated into Tessella's ISO-9000 accredited Quality Management System and is applied to all appropriate Tessella projects.

A good source for general information on the traditional software development life cycle is "Software Engineering Standards" (C.Mazza et al, Prentice Hall). This is a straightforward and accessible book covering the complete software development life cycle. Much of Tessella's software development procedures use principles described here.

Our Rapid Application Development procedures are based on a subset of the Dynamic System Development Methodology (DSDM). [DSDM Consortium, ISBN 1 899340 02 5]

Tessella Support Services plc
Creating Software for Science and Engineering

Tessella's services range from feasibility studies, through system design, development, implementation and ongoing support. Our expertise includes:

Data Analysis Software
Data Capture
Simulation Software
Advanced Graphics
Systems Support
Database Applications

Other Technical Supplements available include:

- | | |
|---|--|
| <input type="checkbox"/> Archiving of Electronic Info | <input type="checkbox"/> Object Oriented Programming |
| <input type="checkbox"/> Active Server Pages | <input type="checkbox"/> Pocket PC |
| <input type="checkbox"/> Automated GUI Testing | <input type="checkbox"/> Portable GUI Development |
| <input type="checkbox"/> Bayesian Statistics | <input type="checkbox"/> Printer Technology Guide |
| <input type="checkbox"/> Beowulf Clusters | <input type="checkbox"/> Real Time Systems |
| <input type="checkbox"/> C++ | <input type="checkbox"/> Regression Testing |
| <input type="checkbox"/> Client-Server Technology | <input type="checkbox"/> Security and the Internet |
| <input type="checkbox"/> COM | <input type="checkbox"/> Simulation |
| <input type="checkbox"/> Computational Fluid Dynamics | <input type="checkbox"/> Soft Computing |
| <input type="checkbox"/> Computer Image Processing | <input type="checkbox"/> Software Design Methodologies |
| <input type="checkbox"/> Decision Support Systems | <input type="checkbox"/> Software Development Cycle |
| <input type="checkbox"/> Electronic Data Capture | <input type="checkbox"/> Software Documentation |
| <input type="checkbox"/> Electronic Lab Notebooks | <input type="checkbox"/> Software Portability |
| <input type="checkbox"/> Excel | <input type="checkbox"/> Software Re-engineering |
| <input type="checkbox"/> Extending the Life of Software | <input type="checkbox"/> Software Specification |
| <input type="checkbox"/> Federal Drug Administration | <input type="checkbox"/> SQL |
| <input type="checkbox"/> FORTRAN 90 | <input type="checkbox"/> UNIX Inter-Process Comms |
| <input type="checkbox"/> Grid Computing | <input type="checkbox"/> UNIX Systems Performance |
| <input type="checkbox"/> High Throughput Screening | <input type="checkbox"/> UNIX Workstations |
| <input type="checkbox"/> Instrumentation | <input type="checkbox"/> Visual Basic 6 |
| <input type="checkbox"/> Integrated Lab Systems | <input type="checkbox"/> WAP |
| <input type="checkbox"/> J2EE | <input type="checkbox"/> Web Services |
| <input type="checkbox"/> Java | <input type="checkbox"/> Windows 2000 Services |
| <input type="checkbox"/> Lims | <input type="checkbox"/> XML |
| <input type="checkbox"/> Linux | <input type="checkbox"/> X Windows |
| <input type="checkbox"/> Microsoft Net | |



INVESTOR IN PEOPLE

Tessella Support Services plc

3 Vineyard Chambers, Abingdon, Oxon, OX14 3PX, England

Tel: (+44) (0) 1235 555511 Fax: (+44) (0) 1235 553301

E-mail: info@tessella.com Web Address: <http://www.tessella.com>