Software Process and Quality

Alan Dix

11 April, 1999

This report was produced to guide software development in aQtive limited and is made available for teaching purposes.

Apart from anonymising a few references to individuals this is exactly as released internally in April 1999.

Note that Qbits are a type of component used in aQtive products.

The report does not prescribe a rigid process, but instead concentrates on the products and documents produced during software development and their interrelations.

Also important is the focus on decisions during coding, which may signal design issues that should be explicitly considered.

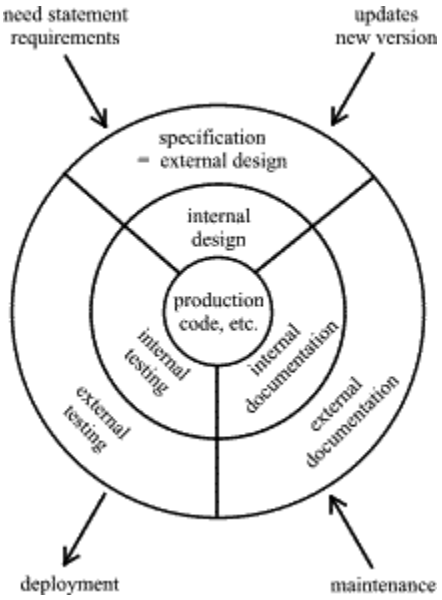
Alan Dix, 8/10/2001
http:www.hcibook.com/alan/teaching/CSC221/

Development Process

Outline

Depending on the nature of the deliverable the order of process may vary, but the following stages should have been carried out (possibly in parallel):

1. Procurement (initial need identification, requirements specification and subsequent upgrades)
2. Specification (external design)
3. Internal design
4. Coding
5. Documentation (internal and external)
6. Testing (internal and external)
7. Delivery (and subsequent maintenance)



These stages are shown in figure 1. There is general flow from top to bottom, but it is important that there are feedback loops, especially when using a more evolutionary development approach.

The stages are drawn as an onion skin. The specification, external testing and external documentation are the interface to the external world. They ensure the fitness for purpose and external quality. The internal design documents, internal testing and internal documentation are more concerned with the internal quality of the software. Although, the external quality is the most obvious concern, internal quality becomes and issue when maintenance and upgrades are required. Furthermore, poor internal quality typically 'leaks out' in the form of crashes, unexpected behaviour in unforeseen circumstances and poor performance.

Because of the nature of the software we are producing, a traditional waterfall development process is not appropriate. Typically product design will involve an iterative process with early versions of the code informing design choices.

To maintain quality in this process whilst allowing freedom to develop code and products in an innovative and appropriate fashion we concentrate on the recording of the documents that are produced by these stages rather than the precise order of these stages.

Types of Deliverable

The nature of the development documents and the aprovals required depend on the kind of deliverable.

At aQtive we have 3 main types of code deliverable:

- *Products* – named products that are sold/delivered to customers
- *Qbits* – which may be a product in themselves, but are usually part of one or more products
- *Java classes and packages* – forming part of one or more Qbits or included in libraries

The latter two are similar except that for Qbits one is dealing mainly with node types and behaviours, but for classes mainly methods.

Approval/Signoff Required

	Product	Qbit	Java class/package
Specification	XX/YY	XX/YY + Team leader	Team leader
Internal Design	XX/YY/Team leader	Team leader	Team leader
Documentation			
Internal	Team leader	Team leader	Team leader
External	XX/YY	Team leader	Team leader
Testing	YY/Team leader	Team leader	Team leader
Delivery	XX/YY + notify board	XX/YY + Team leader	Team leader

Storage and archiving

Where reasonable all agreed versions of documents (including specifications, design notes, etc.) should be archived electronically or in paper form.

For products and major qbits there should be a directory structure on the server with sub-directories for each kind of document. A similar paper filing system should be used for non-electronic documents. Typical directory structure:

```
MyQbit/
        specification/
                MyQbit-spec-v1.doc
                MyQbit-spec-v2.doc
        design/
                meeting-3-2-99.doc
                email-12-2-99.txt
        code/ ….        –       may be stored in central development directory
        doc/
                user-manual-v1.doc
                user-manual-v2.doc
                internal.html
        test/
                results-15-4-99.xls
        deliver/
                MyQbit.jar
```

Where there are several versions of the specification/code/documentation, the naming convention should make the correspondence clear (e.g. spec-v1.doc, code/v1/*.java; spec-v2.doc, code/v2/*.java).

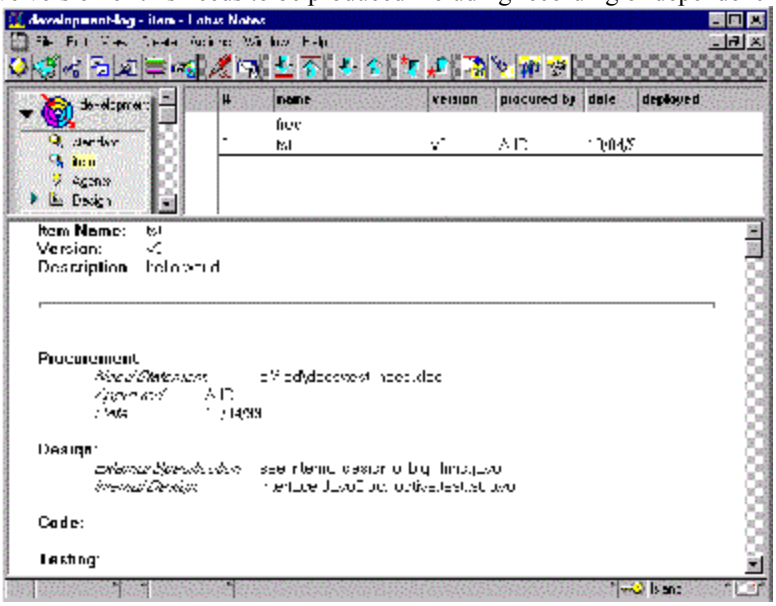
For smaller components this type of structure may be too heavyweight, and a substantial part of the specificatiom, design and documentation is likely to be included as comments and JavaDoc within the code.

However, for **all** software items, the existence and location of all documents must be recorded. A prototype Notes database for this is at:

….*directory*/`development_log.nsf`

This database should be used to store pointers to the location of all relevant documents. It can contain paths or URLs of relevant documents and comments such as "see JavaDoc".

[[A more extensive version of this needs to be produced including recording of dependencies.]]



Details of Development Stages

Specification

Appropriate Forms:

- Scenarios of use **[P/Q]**
  (primarily for user interface, but can also be used for other external interactions.)
  Should include account of user interface, simulated screen shots, network interactions, local resource use and interaction with other aQtive products. May include demonstrations using HTML/Powerpoint.
- Feature lists **[primarily P]**
- Text descriptions **[any]**
- Network interactions **[any]**
  Use of standard protocols and services, where appropriate specification of new protocols.
- Rigorous specification **[any]**
  If appropriate for complex or mission critical code.
- Definition of nodes and types **[Q]**
- Definition of public API **[J]**
- Interface class (with comments) **[J]**
- Use of internal model (not necessarily as implemented) **[J]**
- Legal ordering/protocol for commands/method calls/node invocation **[Q/J]**
  very often forgotten, but very important
  use timing diagrams, low-level scenarios, FSMs, STNs, process algebras
- Reference implementation **[Q/J]**
  e.g. in prototyping language, or using simple algorithms

Strength indicators

Not all parts of a specification carry equal weight:

- Some parts represent essential features
- Some are arbitrary
- Some may be not as desired, but chosen as a compromise,.
- Some of these may be expected to change in subsequent versions

It is important that this is recorded in the specification, together with reasons where appropriate, in order to aid future development.

Internal Design

This ranges from the architectural design of larger components to the choice of individual algorithms

Appropriate forms:

- Component diagrams
- Dataflows
- Major node/method lists for sub-components
- Specifications of sub-components (e.g. Java interface classes)
- Specifications of linkage code
- Refinement of scenarios
  to walk through component interactions
- Designs of 'private' methods and classes
  N.B. these may not be declared 'private' in the Java sense, but if not intended to be available to thord parties this should be recorded in JavaDoc and other documentation.

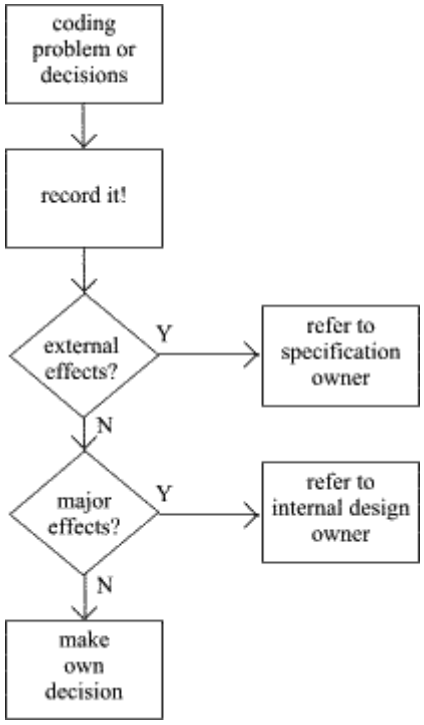
Coding

For products and larger components, the main effort of coding will often be decomposed into component coding. However, some 'driver' and linkage code will be needed and various forms of install scripts may also be required.

Recording design choices

Many design decisions are made during coding. Some of these are local to the code, others are either major software design choices (e.g. substantial algorithm redesign) or affect the user interface or other parts of the external specification (e.g. should an IO exception produce a dialogue box). These decisions should always be recorded and where appropriate refered to the appropriate authority.

In particular, it is very important that choices affecting the external behaviour are recorded and approved. The component specification does not belong to the development team.



Incremental delivery of code

It is normal for code to develop incrementally rather than in a waterfall fashion.

This leads to four main types of code versions:

1. Internal versions produced as part of development
2. Stable versions for use by other parts of aQtive
3. Pre-release versions of products (alpha/beta testing, demos) – *only for products*
4. Release versions of products – *only for products*

Of these:

(1) is totally under the control of the development team/individual developer although where reasonable source code control should be used;

(2) can be produced when appropriate by the team (or when needed by others), but once produced cannot be altered in any major way without approval as others may be relying on the code (although new stable releases can be produced);
(3) needs approval of XX/YY
(4) needs the above and notification to the board

N.B. *** need version naming convention ******

Documentation

Internal Documentation – Appropriate Forms

- Architectural diagrams
- JavaDoc
- Rationale of coding choices
- Tidy forms of internal design documents

External Documentation – Appropriate Forms

- Tutorials **[P/Q]**
- End-user documentation **[P/Q]**
- On-line help **[P/Q]**
- Node descriptions **[Q/J]**
- JavaDoc **[Q/J]**
- Ordering information (as in specification) **[Q/J]**

Note

If the documentation is getting hard to write it probably means something needs fixing in the product.

Testing

Internal testing verifies that the code behaves as the internal design says (verification) and checks internal functions and robustness. It also tests whether the internal design choices were appropriate (validation).

External testing ensures that the code meets the external specification (verification) and also whether the external specification was appropriate (validation).

Appropriate Forms

- informal or formal user testing
- internal use within aQtive
- stress testing with automatic harnesses (if possible)
- test scenarios
- extreme value/bad input testing
- multi-platform and multi-JVM testing

Testing Outcomes

Test success is easy – what happens when it fails?

Possible feedback paths:
1. If the code doesn't behave as expected – change the code
2. If the behaviour is as expected, but isn't right – change the specification/internal design

3. If getting it right is going to be too expensive –
    3.1 change the specification/internal design
    3.2 note restriction/workaround in documentation

The last of these would be appropriate if, for example, a queue class 'next()' method returned a bad value when applied to an empty queue. The class documentation could then be amended to say that all code using the class should test the queue length before using next().
It would only be used in very exceptional circumstances as a solution for a released product.

Delivery

Appropriate forms for products

- full install files
- all appropriate documentation
- CD-ROM production
- on-line deployment in download area

Internal deliverables (Java classes)

A source code control system (to be selected) will be used to ensure synchronisation of versions. Individual developers and teams can work at will on branches of the source tree, but each package will have a single gatekeeper (initially YY for all code, but to be devolved later) and the main development branch will only be updated through that gatekeeper.

In addition, a copy of the 'current' version of all .java and .class files in the com.aqtive domain will be held on the server (in jar files as appropriate) for inclusion of the class paths for debugging etc.