The ATLAS Software Process

Stephen M. Fisher <S.M.Fisher@rl.ac.uk>

Rutherford Appleton Laboratory, Chilton, Didcot, Oxon, OX11 0QX, UK

The ATLAS Software Process sets out the way in which software development for the ATLAS experiment will be organised. This process defines the sequence of activities that take place during the lifetime of the project, the roles that must be filled in order to implement the process and the deliverables produced in the course of software development. The model is evolutionary, with short cycles of development followed by a review of what has been produced, so that management gets rapid feedback on the progress of the project. A particular concern is to ensure that good quality and easily maintainable code is produced. For this reason, considerable emphasis has been placed on the necessity for documentation and coding standards. As there is no experience of a formal software process in HEP we had to design something which we felt would be acceptable in our environment. The process is currently being used by a small group. A part of the process is that the process itself can change–we have already made changes and expect to make many more especially in the early stages.

Key words: software process; quality assurance; ATLAS; OO

1 Introduction

About 9 months ago we defined the ATLAS Software Process (asp)[1] to try and bring order to some of our OO work and with the intention of recommending it to the experiment as the way of producing software. The software process and its improvement is *not* an end in itself but has been defined to assist in the production of a well documented system which does what is wanted and which is easy to maintain.

2 Important parts of the process

The process defines *cycles* of eight working weeks. At the end of each cycle a new release of the system, fully documented but incomplete should be ready. There is an initial phase, before the cycles begin, which has not gone as we had imagined as we had seriously underestimated the complexities of the human interactions in getting started.

The software is split into *domains*, some of which correspond to a detector. A domain team is responsible for the complete development of software within a domain. Each domain architect represents the domain in the *core team*.

The process is built around a set of *deliverables*. The critical deliverables are, requirements, design and code. These deliverables are reviewed. In each case reviewers are provided with the deliverable and a guide to what they should be looking out for in this kind of review.

3 Deliverables

In defining the process we have tried to cut down as far as possible on the number of deliverables and to make their maintenance easy. If we were to miss out the requirements or design documents, it would be impossible to assess the design. The design document we advocate is language specific so that different domains can produce code which will work together. It might seem preferable to have a more informal design document but it is hard to review an informal design and it cannot be reverse engineered from the code.

We are using Software through Pictures (StP)[2] to hold requirements and design. We can generate the requirements and design documents from a working system at any time. The requirements document simply lists requirements grouped by domain. For each domain we have functional requirements, system properties and constraints. The design document shows which class or operation offered by a class satisfies each functional requirement. During design work a requirement may be moved between domains, but some domain must undertake to satisfy each functional requirement. System properties (e.g. the efficiency of track finding) can only be checked when the system is built; some may be attributed to a single domain though many will not. Constraints (e.g. saying which external packages to use), may be general or upon domains. The document includes *Object Interaction Graphs* (OIGs) for *Use Cases* of interest. Each Use Case appears as an explanation of the OIG. We hope to see all operations exercised by an OIG and none which appear to have no use.

Design documents are produced both by the core team and the domain teams. The core team document defines the interfaces between domains. Its OIGs only shows messages between objects in different domains. These objects define the *interface classes* in each domain and the operations they need. A domain team starts with the interface class diagram for its domain and its set of requirements and does a similar job, but now showing all classes in the domain.

Each diagram, class and operation may be annotated within StP. A general introduction for the design document and introductions for each of the major sections may be written, and a design document generated automatically from the StP repository. As fragments of code can be stored in StP, everything can in principle be generated from it. However most people prefer to work on the code of a class directly once the skeleton of the class has been generated with the annotations for the class and operations appearing as comments in the code. We consider reverse engineering to be a legitimate way to proceed, especially for making small changes to the system, so we have developed a strategy for taking code and generating a revised design document from this. Class interfaces are extracted from the code and comments in the code used to refresh the annotations. Inconsistencies with OIGs and requirements will be spotted by the document generator, and it is the responsibility of those preparing the design document to reconcile the diagrams. We have successfully carried out code reviews of some of our prototype code, working electronically without meetings. We are about to try out a similar design review procedure and see if it is possible to assess a design with just the information in the generated document.

4 Conclusions

It has been slow to get started because we have been challenging many things as we learned how far to adopt and how far to adapt the process. We have found that the deliverables hold the process together. Both forward and reverse engineering require tool support.

References

[1] S. M. Fisher, K. Bos and R. Candlin, *The Atlas Software Process*, ATLAS Software Note 27 [1]

[2] *Software through Pictures*, A CASE tool from Aonix, California, USA [2]

[1] http://atlasinfo.cern.ch/Atlas/documentation/notes/SOFTWARE/note27/asp.html
[2] http://www.aonix.com/Products/StP/StP_OMT.html