JOURNAL OF OBJECT TECHNOLOGY

Online at http://www.jot.fm. Published by ETH Zurich, Chair of Software Engineering ©**JOT, 2007**

Vol. 6, No. 3, March - April 2007

Agile Artifacts - Documenting, Tracking and Reporting

Trust The Source Luke!

Dave Thomas

1 SIMPLE TOOLS MEET THE NEEDS OF BOTH DEVELOPMENT AND MANAGEMENT

One of the obstacles to introducing Agile development into large software organizations is providing a means for the developers to work effectively using light weight practices and tools while at the same time ensuring that the management and company have the necessary visibility and documentation to ensure that they can manage and later maintain the software being developed. Unfortunately, many of the commercial tools are very draconian and inflict all sort of extra work and overhead on developers while at the same time not providing the promised benefits. This often results in tension between process and programmer [1] which can give the impression that the developers don't care about documentation or design and that the management cares only about the process artifacts rather than the code.

In any large scale Agile process, artifacts include not only source code but also Requirements, often bundled into Features, Use Cases and Stories, and their associated Unit and Acceptance Tests. Complex Features and Stories are composites of smaller ones. Additional supporting artifacts include Models/Prototypes, Teams and their Backlogs. We describe a simple Literate Programming approach for capturing artifacts and associated automated tooling to support tracking, reporting and traceability.

Traceability makes it possible to identify and understand the relationships between artifacts. For example, for a given requirement, one can answer questions like: Has the requirement been implemented? If so, by which developer(s)? Where in the code base will I find it? It is important to understand that the mappings between requirements, models, use cases and code are not one-to-one, nor are they always precise. They are a collection of statements about the software that can be used to model it for the purpose of understanding. In this article we describe a simple code-centric approach which we have found useful for many years. It can be implemented quickly using open source tools with

Cite this column as follows: Dave Thomas, "Agile Artifacts - Documenting, Tracking and Reporting, Trust The Source Luke!", in *Journal of Object Technology*, vol. 6 no. 3 March – April 2006, pp. 31-37 http://www.jot.fm/issues/issue_2007_03/column4

little imposition on developers. Perhaps surprisingly it provides much better information for management than many more complex and expensive approaches.

2 LITERATE PROGRAMMING AND WIKIS

The Agile Manifesto emphasizes the importance of code. Literate Programming [2] is a best practice style for writing programs which are intended to be read and evolved over time. Some naïve AgileXP purists insist that code needs no comments or other documentation since the code should be easily understood from its own clarity and associated unit and acceptance tests. They have a well founded concern about the tendency for a comment not to be updated or refactored when the code is changed. However, these concerns don't in our experience justify the omission of proper structured comments which improve readability and understandability of the code. In his book Domain Driven Development, Eric Evans provides guidelines for clearly separating the naming of domain representations from underlying technology ones.

Unfortunately, the early literate programming tools were not designed for today's interactive programming environments hence they are awkward for today's agile developers. Recently, however, code folding editors have been introduced into IDEs like Eclipse and Literate Outliners such as LEO [3] have appeared which encourage literate programming. Since it is tedious to write effective documentation in a programming IDE, especially for requirements, we have found Wikis to be a simple and effective mechanism for capturing such information, with annotations linking to the code and vice versa. A Wiki is a natural place to capture requirements and project artifacts since it is already in use for informal collaboration in and between teams as well as for capturing acceptance tests (FIT and Fitnesse). Wikis and annotations may also reference more extensive artifacts such as models, standards, and prototypes which elaborate the requirements, design intentions, etc.

3 ARTIFACT DOCUMENTATION

Comprehensive traceability requires coding standards as well as tool support and ideally should be integrated with the build and reporting environment. The most effective means to track and report important project information for large scale development is to instrument the code base within the configuration management (CM) system. Current best practice is to annotate the text associated with artifacts with small amounts of information to identify the artifact itself and its relationships to other artifacts.

The simplest, most effective way to do this is to use structured comments which are associated with the code. This is one of the reasons that meta-information through annotations is now a standard feature in languages such as Java and C#. If everyone associated with the definition and modification of artifacts follows a disciplined annotation policy for code, then all of this information will be saved in the CM system.

When direct annotation is not possible, the next best approach is to use a separate description that references the artifact. With either approach, during each major build simple tools such as fuzzy parsers can search through the code base to find the annotations and update a traceability website or database.

4 PROJECT REPORTING

Such an approach enables the Continuous Integration and Test Environment to compile project reports, traceability data, metrics, and other information about the software being developed. This in turn provides valuable feedback to the developers and management. Using such instrumentation it is possible to *automatically* produce all of the following reports as part of the build:

Tracking – Burn Down/Up and Velocity

Backlog Status

Team and Developer Progress/Productivity

Variance between Actual and Estimates for Continuous Improvement

Development, Test and Integrate Rhythm

Our recommended approach is to use an IDE (in our case Eclipse) together with a Wiki (in our case Fitnesse Wiki) for capturing the information. Using IDEs and Wikis substantially reduces or even eliminates the need for other more tedious manual tools such as Excel, MS Project[1] etc. It is well known that leveraging the tools that are actually used by developers, and not forcing them to learn new ones, greatly increases the potential for obtaining and maintaining information about the software.

That said, the generic approach is independent of any specific tools, or any particular annotation method, but simply requires that all assets be self-identified and versioned in the CM system or equivalent. It is also important to note that specific projects or companies may necessitate different annotations or artifacts, which can easily be accommodated

5 EXAMPLE ANNOTATIONS FOR CODE ARTIFACTS

The approach we currently used in our environment is to annotate the code artifacts explicitly using structured comment conventions such as JavaDoc for Java (Doxygen is a popular tool for C/C++).

All new methods, fields and constructors added by Java developers get the tag **@story** inserted explicitly by the developer using the following tag convention:

[1] Note: it is straight forward to generate the appropriate MS Project or Primivera files if these are needed by corporate reporting.

```
@story {story-name} {time-and-date}
```

If a field or method is modified more than once, then there is a separate tag added for each update. Tags are added to both implementation and test code, with naming conventions used to distinguish between implementation code, Unit Tests and Acceptance Tests. This allows the build utility to assemble a fine-grained trace of all the development activity undertaken to implement any particular story. A report can be generated indicating what user stories are implemented where, by whom, which have Unit Tests or Acceptance Tests, and so forth. Using this basic traceability information, one can compute metrics such as velocity, burn down/up charts, etc.

The default naming convention uses *AT{artifact name}* and *UT{artifact name}* to denote acceptance and unit tests, respectively, associated with a particular artifact. For example, the Unit Tests for a **GameConsole** Story would be named **UTGameConsole**. Since tagging relates Stories to code, and Stories are related to other artifacts like Features and Use Cases through the metadata stored in the Wikis, there is sufficient information to create and maintain all the traceability relationships.

The relationships between Acceptance Tests, Unit Tests, and Stories can sometimes become too complex for the naming conventions to handle. This does not present a problem however, because Acceptance Tests and Unit Tests have **@story** tags as well, which will override the naming conventions in such a case.

NOTE: Alternatives to this approach are to reference the associated classes/methods from the story artifact, or to use properly versioned non-code artifacts as task markers, and automatically infer the information from the version history. The latter is in many ways the preferred approach since it imposes minimal demands on the developer, but requires more sophisticated tool (semantic diff) or IDE support such as Mylar [6]. Tagging has the virtue of simplicity.

6 EXAMPLE WIKI TEMPLATES FOR NON-CODE ARTIFACTS

The following are some example Wiki templates that may be used to annotate the code base with important information about non-code artifacts. The same approach can be used with defect trackers such as JIRA [5], project management such as Trac [4], and/or requirements tools but Wikis are our personal choice. The essential requirement is a process to synchronize the development history of the code with the associated assets. The following templates can be applied with minimal changes to most Agile development projects. We stress that this is an example from a particular client and that terminology and artifacts will vary from one organization to the next, although, in our experience, the contents are roughly similar.

Team

Team pages have the header TeamName with the following elements:

Team: TeamName;

Member: member name; -- repeated for each team member
Project: ProjectName; -- added to allow easy navigation

Story

Story pages have the header StoryName with the following elements:
Story: StoryName;
Status: (None, Done, In-Progress, Deferred);
Author: name; repeated if more than one
Description: descriptive text;
Fullfills: requirement; -- reference to requirement, use case, defect list
Clarity: (Clear, Needs-Clarity, Un-Clear);
Team: TeamName ; -- team assigned the story, Un-Assigned otherwise
Developer: name; -- repeated for each team member implementing the story;
Sprint: release, sprint; -- the release and sprint in which this story is planned, e.g. 2.1
Ideal-Days: (best, avg, worst); -- estimated effort in ideal days, (0,0,0) means no estimate;
Classes: (best, avg, worst); -- estimated number of classes, (0,0,0) means no estimate
Points: (best, avg, worst); -- estimated effort in story points, (0,0,0) means no estimate
Unit-Prefix: name; -- Unit Test prefix, default is UTStoryName
Accept-Prefix: name; -- Acceptance Test prefix, default is ATStoryName
Composed-Of: other StoryNames; -- blank if task (non-composite) story.
Part-Of: parent StoryNames; -- parent story or blank if none.

Feature

Feature pages have the header FeatureName with the following elements:
UseCase: FeatureName;
Summary: one or two sentence brief overview;
Owner: name; -- who owns the feature; repeated if more than one
Fullfills: requirement; - reference to requirement, defect list;
Description: descriptive text;
Clarity: (Clear, Needs-Clarity, Un-Clear);
Team: TeamName ; -- team assigned the Feature, Un-Assigned otherwise
FeatureUseCase: UseCaseName; repeated for each associated feature use case;
IdealDays: (best, avg, worst); -- estimated effort in ideal days, (0,0,0) means no estimate;
Classes: (best, avg, worst); -- estimated number of classes, (0,0,0) means no estimate
Points: (best, avg, worst); -- estimated effort in story points, (0,0,0) means no estimate
AcceptPrefix: name; -- Acceptance Test prefix, default is ATFeatureName
ComposedOf: other FeatureNames; -- blank if non-composite feature.
PartOf: parent FeatureName; -- parent feature or blank if none.

UseCase

UseCase pages have the header UseCaseName with the following elements:
UseCase: UseCaseName;
Summary: one or two sentence brief overview;
Owner: name; -- who owns the use case, repeated if more than one
Fullfills: requirement; - reference to requirement, defect list;
Actors: List of actors participating in the use case. Possible actors are system components or applications, the network, and of course the user;
Preconditions: What must be true about the state of the system before the use case can start;

Postconditions: What must be true about the state of the system after the use case "expected path" completes. A testable state of the product.;
Description: in point form or descriptive text;
Exceptions: Name of exception - how handled (brief description or reference to a requirement or supporting use case);
Variations: Different ways to accomplish steps. Variation name - description (could be alternate list of steps or reference to supporting use case that contains the description);
CompliesWith: Important requirements to be aware of while implementing the use case (may reference standards, other documents);
Clarity: (Clear, Needs-Clarity, Un-Clear);
Team: TeamName ; -- team assigned the use case, Un-Assigned otherwise
IdealDays: (best, avg, worst); -- estimated effort in ideal days, (0,0,0) means no estimate;
Classes: (best, avg, worst); -- estimated number of classes, (0,0,0) means no estimate
Points: (best, avg, worst); -- estimated effort in story points, (0,0,0) means no estimate
AcceptPrefix: name; -- Acceptance Test prefix, default is ATUseCaseName
Composed-Of: other UseCaseNames; -- blank if non-composite use case.
Part-Of: parent UseCaseName; -- parent use case or blank if none.

Backlogs

The ProductBacklog page has all of the use cases for the current product release. It also has all of the release backlogs for this product release. The ReleaseName page has all of the use cases/stories for the current release. The ReleaseTeamName page has the release backlog for team TeamName. The SprintTeamName page has the sprint backlogs for each sprint for team TeamName. Each of these pages is a simple list of other Wiki pages.

Models

Model pages have the header ModelName with the following elements:
Model: ModelName;
Description: Descriptive text;
Owner: name; -- model owner, repeated if more than one
Kind: model kind;
Fullfills: UseCaseName or StoryName;
Model-Ref: URL to model;

7 SUMMARY

We have found that by leveraging popular best programming practices of literate programming, Wikis and configuration management high performance teams are able to easily capture and share all of the information management needs to manage the life cycle and evolution of its product assets. Furthermore, through simple automation of the continuous build all of the project tracking, management and metrics can be generated each build. This approach, which clearly can be easily tailored for different environments, provides all of the benefits of CMM, Six Sigma without the pain of draconian tools, special status meetings, reports, etc. It provides management and developers with a true visibility into product development from requirements through estimates, backlogs and on to unit and acceptance testing. Both developers and management get to have their cake and share it while all artifacts are captured in the common CM system!

REFERENCES

1. Dave Thomas: "The Unnecessary Tension between Process and Programmer", in Journal of Object Technology, vol. 5, no. 1, January-February 2006, pp. 7-11 http://www.jot.fm/issues/issue_2006_01/column1

2. Christopher Lee, Literate Programming – Propaganda and Tools, Carnegie Mellon University http://vasc.ri.cmu.edu/old_help/Programming/Literate/literate.html

3. LEO http://personalpages.tds.net/~edream/frontMatter.html#anchor871644

4. Trac http://trac.edgewall.org/

5. JIRA http://www.atlassian.com/software/jira/

6. Mylar http://www.eclipse.org/mylar/

About the author

Dave Thomas is cofounder/chairman of Bedarra Research Labs (www.bedarra.com), www.Online-Learning.com and the Open Augment Consortium (www.openaugment.org) and a founding director of the Agile Alliance (www.agilealliance.com). He is an adjunct research professor at Carleton University, Canada and the University of Queensland, Australia. Dave is the founder and past CEO of Object Technology International (www.oti.com) creator of the Eclipse IDE Platform, IBM VisualAge for Smalltalk, for Java, and MicroEdition for embedded systems. Contact him at dave@bedarra.com or www.davethomas.net.