Technical Report

Department of Computer Science and Engineering
University of Minnesota
4-192 Keller Hall
200 Union Street SE
Minneapolis, MN 55455-0159 USA

TR 12-010

Automation for Regulated Issue Tracking Activities

Touby Drew and Maria Gini

March 26, 2012
Automation for Regulated Issue Tracking Activities

Touby Drew
Medtronic, Inc.
7000 Central Avenue NE; MS RCE290
Minneapolis, MN, 55432-3576
touby.drew@medtronic.com

Maria Gini
University of Minnesota
4-192 EE/CSci, 200 Union St SE
Minneapolis, MN 55455
gini@cs.umn.edu

Abstract

We describe the application of automated support for issue tracking and related software engineering activities of development teams at the world’s largest medical device manufacturer. We present some challenges and classes of defects found in product software, related artifacts, and the issues which track them. We describe enhanced means for defect detection, data mining and analysis, and other novel support we provide at the time of issue review. Finally, we describe evidence of the positive impact of this support, its adoption, lessons learned and potential next steps.

1 Introduction

In the medical device industry, software is developed under strictly regulated processes to ensure patient safety. Issue tracking systems are essential tools at the heart of development, management, and quality assurance in this domain. The issue tracking process manages and documents each concern, its resolution, the review of that resolution, its related artifacts, and, ultimately, the closure of the issue. This process converts issues identified by team members, including enhancements, defects, new features, and other items, into documentation and products.

Within the medical device industry, companies produce a wide variety of software-intensive devices. Pacemakers, for example, were prolonging the lives of more than 2.4 million people in the United States in 2002 alone. Each pacemaker has medical device software within it and within the instruments that interact with it.

As described in [6] and [7], we developed an advisor software agent, PnitAgent, to support issue tracking related activities at Medtronic Neuromodulation. This agent was initially deployed to two teams of engineers that we call S team and P team. Formative research and agent prototyping started in 2009, release of the software occurred in July 2010 and it was kept largely unchanged until initial analysis of its impact was completed in March 2011. Dozens of users adopted the agent. The rate and nature of issue rejections it was designed to address improved significantly. Regular
users of the agent saw greater rates of improvement than those who did not use it regularly. Its adoption spread to other teams and roles including several managers who use it for status analysis and reporting.

Our previous work [7] broadly describes the development and application of AI technology in the agent design, natural language processing techniques, and ubiquitous user monitoring, focusing on the innovation and success of PnitAgent.

In this paper we describe detailed concepts and practical experience with automating software engineering activities. This includes the use of data mining with issue tracking, version control and other data sources to better understand on-going software development activities. This paper is primarily focused in two different veins: rich support at the time of issue review (Section 2) and broader visibility into the status of software development projects (Section 3).

Key contributions of this work include the articulation of new concepts and practical experience related to:

1. a novel approach to static analysis of source code at the time of issue review, which includes integration of data about the issue, process and related artifacts;
2. enhanced support for annotation (e.g. related to remaining work) including novel capabilities for detection, analysis and presentation;
3. enhanced role analysis and support for collaboration;
4. support for flexible, cross-data-source, batch query and automated analysis;
5. support for broad issue-focused change sets that works across system boundaries;
6. support for improved project status and change understanding which leverages issue tracking and requirements management systems.

2 Automation at Issue Review

Issue review is a particularly challenging and important time in the process of developing software in a regulated environment. It is a time when information from the issue under review is joined in the mind of the reviewer with information from other issues and associated changes in documentation, source code, and related artifacts. This provides a rich but fundamental mechanism for incremental review to support change management and quality assurance.

2.1 In the Context of Previous Work

Significant previous work exists in the area of automating software engineering activities which are relevant at the time of issue review. The term Software Configuration Management (SCM) has been used to describe what allows “us to keep evolving software products under control, and thus contribute to satisfying quality and delay constraints” [8]. Our work is built on commercially available tools and their basic customizations and integrations.

As an example of the sort of automation already in place before the advances which are the focus of this paper, it is useful to consider a basic flow of events just before and at the time of issue review. After a set of changes are completed by a developer, they are committed to a development stream in the version control system. Before the version control system will accept such a commit,
association of the changeset with an appropriately configured issue from the issue tracking system is required. After commit, a build server detects the availability of new changes and automates build, static analysis, testing, coverage analysis, snapshot creation, metrics collection, triggering of related builds, status communication and other activities. The user then finishes the description of the snapshot and other details in the associated issue, ensures it is ready for review, and marks it as such. Finally, the issue tracking system changes the issue state, preventing further promotes associated with it, and sends emails to the relevant reviewers.

The importance of process support within software configuration management has been broadly recognized and pursued. While our work touches on mature topics, such as version control and issue tracking systems, it is fundamentally focused on automating support for their use as tools and the insights they can provide for a more comprehensive regulated software development process. Static (code) analysis tools provide an example of this theme.

2.2 Static Analysis Re-Imagined

Commonly, static analysis tools contribute to the discovery of defects and there is an expectation that issue tracking tools (such as ClearQuest) will be used to manage the defects they helped find [9]. Static analysis has been shown to predict defect density [12] and is often integrated into programming environments. However, the use of static analysis related to issue tracking has not been fully explored.

In particular, this work explores two related topics:

1. the potential for focusing static analysis given an issue or set of issues from the issue tracking system and
2. the potential for performing something like static-analysis on issues themselves (retrospective, non-run-time, external analysis of issues aimed at identifying potential errors in those issues and their related artifacts).

Consider the review of an issue before its closure in which a reviewer examines both the issue and the changes made to software code, documents and related artifacts. Despite the best efforts of all involved to keep mistakes out, some automated support for identifying them helps to reduce errors. It would be best if this was provided only, or at least primarily, within the scope of the issue(s) being reviewed. Within this area, there is not only a timely opportunity to look at the contents of the issue and related artifacts with a more focused and appropriate lens, but also a unique opportunity to leverage combined data about the issue, process and related artifacts.

2.2.1 Enhanced Support for Analysis Exclusions

On the S and P teams process requires that all analysis exceptions be justified and all blocks of analysis exclusion must be explicitly defined before they are reviewed under the issue associated with their addition or modification. This makes explicit the decisions related to suppressing analysis coverage and simplifies the review of those decisions. Generally it is impractical to expect analysis tools to check all aspects of user requested exceptions to their analysis. Traditionally, these concerns must be checked manually (in the code or tool output) and can be forgotten or missed. To better support these activities, PnitAgent automates the detection of such concerns, for example by using regular expressions like that in Listing 1 which can be applied to identify unjustified exemptions to the FxCop static analysis tool, which was one of the tools used by the S and P teams.
This expression basically identifies where it sees the term “SuppressMessage” followed by anything ultimately leading up to a `]' character without a `[` character or the word “Justification” in between. Another example is in Listing 2 for detecting when analysis of a certain rule by the Tiobe ClockSharp¹ static analysis tool has been disabled and remains disabled until the end of the file because it is not explicitly re-enabled. This regular expression matches the multi-line rule disable comment and then (in a non-capturing group) everything up to the end of the file that is not followed by a reference to the same rule number as would be required to re-enable checking of that rule in the current file.

Similar techniques can be applied to support exclusions or suppressions for other static and dynamic analysis tools and extended with further reasoning support. For example, unjustified coverage exclusion attributes as used with the NCover tool can be identified and only raised as a significant concern if (in aggregate) the resulting additional uncovered lines of code cause dynamic analysis coverage to fall short of preset thresholds.

### 2.2.2 Realizing Advantages of Issue Focus

In large, complex software there can be a great number of exemptions without justifications that are left unnoticed and/or uncorrected. Furthermore, although this checking is important, checking it alone or on a single file is not ideal. It is expected that this and a number of other concerns are checked on many files and reported together. When the goal is to review an issue, this can lead to a significant amount of extra unrelated output to wade through (see for example Figure 1). A better way to support such concerns is to organize and present them in the context of the issue in review and as appropriate for their relationship to it.

There may be no file changes or a specific changeset associated with an issue. In the former case, no associated artifact analysis concerns need to be considered and in the latter, only those which are in the changeset are important to the review at hand. Concerns with artifacts not associated with an issue may be de-emphasized, made available upon request, or ignored as shown in Figure 2. This greatly reduces the volume of output a user needs to focus on to complete an issue review. Furthermore, although the risk of false positives is extremely low in this case, by narrowing the

---

¹www.clocksharp.com
scope of concern, not only are there fewer irrelevant true positives, there are fewer irrelevant false positives.

### 2.2.3 A Better Time for Analysis

In addition to providing support that reduces the volume of concerns to consider during review, appropriate automation can help to address timing concerns as well.

Often examples of how timing contributes to review complexity relate to the fact that there may be several people and issues associated with changes to a single file over the same period or that problems could have been introduced after the issue in question. Furthermore, they may have been resolved after the issue in question and, while worth noting, may not be worth rejecting an issue. To clarify that the problems found were introduced as part of the changes tracked by the issue under review and have not been resolved since, Pnit.Agent supports analyzing the latest version, the basis, a locally modified, or other specific versions, of an associated file or group of files. Furthermore, automation of such analysis and related reasoning can help to identify the more interesting static analysis concerns which were introduced by this issue (not present in the basis) and have not been subsequently addressed in the latest version.

As another example, the automated build processes of the S and P teams enforce many static analysis checks automatically, but do not address certain coding standards. These include security
standards which are not supported by the static analysis tools and for which compliance is not required until the software is ready for review. An example regular expression used by the agent to support recognition of exposed classes that do not have expected security attributes is shown in Listing 3.

```csharp
static readonly Regex NoSecurityDemand =
    new Regex(
        "namespace (.\s|\S\n|(?!))\s+(SecurityAction|UnitTest)\s+\"\n        "(\s+generate|Test\(Fixture\))\{0,500\}"\n        "\n[\s]*\([public|protected]\)\.*\ class" ,
        RegexOptions.Compiled | RegexOptions.ExplicitCapture );
```

Listing 3: Missing security demand regex for C#

Though some may desire such concerns to be enforced immediately, consensus within the S, P and other teams is that some should only be enforced at the time of issue review (after some degree of integration and automated build but before more formal integration). This is driven in part by the desire to not delay initial integration too long, but to make sure the standards are high and enforced, automatically where possible, before review.

Other important examples of concerns that are often not addressed before initial integration, but are expected at issue review, are opportunities related to code annotation and understanding of references to related and remaining work.

### 2.3 Code and Other Artifact Annotation

Storey et al [15] examine issues related to how to manage annotations and use them to improve tool support. Specifically, they note the relationship between comments such as TODO or FIXME and the links to code in issue tracking systems, as well as automation support provided by development environments and frameworks. However, they stop short of describing the potential for automating checking for TODO comments at the time of issue review/closure.
We have implemented guidelines to relate TODO code annotations with the related issue(s) and an automated capability to present related TODO comments (from code as well as documents\textsuperscript{2}) which reference the issue under review or are within the scope of the issue under review (and either reference another issue which could be important for understanding the context or do not reference an issue despite guidelines contraindicating this). We believe that the following capabilities are valuable and novel extensions of automation to support concerns at the time of issue review:

1. identification and presentation of references to remaining work within an issue;
2. identification and presentation of references to the current issue from other artifacts;
3. identification and presentation of references to work remaining related to (e.g. in files referenced by) issues which are in a post-resolving state;
4. identification of references to work remaining which is not inside of and does not directly reference a tracking issue;
5. identification and presentation of references to nonexistent items across data sources;
6. integration of data and presentation including issue contents retrieved from the tracking system, software sources, documentation in spreadsheets and other formats, associated requirements retrieved from a requirement management system.

The identification of remaining work from annotations naturally put into artifacts such as software source code, documents, and the internal fields of one or more issues under review, is not well supported by current tools. Editors and integrated development environments have begun to support this kind of concern, but only in a limited fashion (without regular expressions support, lacking support for association with tracking issues and for several common artifact types such as Microsoft Word documents).

Two key challenges for richer support in this area are the ability to access and support artifacts in different formats and systems (e.g. such as the issue tracking system), as well as to correctly detect not just remaining work and issue references, but the association between the two.

An example regular expression for detecting references to remaining work in C-style code (relevant to C, C++, C#, Java and other languages) is shown in Listing 4. Line 3 matches what is considered to start a comment and text preceding indication of remaining work. In this domain, comments almost always start with “/\*” or use “MessageBox.Show.” With other development languages or domains this matching may need to be changed or replaced (e.g. with “/\*\*” multiline commenting where even newlines may precede the text indicating remaining work, but “/\*” cannot).

```csharp
static readonly Regex TodoCommentRegex =
new Regex(
    "\(MessageBox.Show\[//,.*\]+"+
    "(todo|tbd|hack|revisit|fixme.xxx)\)+
    "\[\s\S\n\]{0,120}\",
    RegexOptions.Compiled |
    RegexOptions.IgnoreCase |
    RegexOptions.ExplicitCapture);
```

Listing 4: TODO regex for C-style code

\textsuperscript{2}The detection, reasoning about, and integrated presentation of TODO-type comments even extends to the contents of the related issues in the issue tracking system.
In xml-based files, which are also used for several purposes, comments are started with the “!−” string and an expression like that in Listing 5 is more appropriate.

"<!−−.*\(todo|tbd|hack|revisit|fixme|xxx\)"+

Listing 5: Partial TODO regex for xml-style code

In Microsoft Office documents references to remaining work are generally not called out by any special characters and may be highlighted, put in a special font, or, more commonly, not distinguished in any fashion (see Listing 6).

"\(todo|tbd|hack|revisit|fixme|xxx|pni ts\)"+

Listing 6: Partial TODO regex for documentation

In Listing 4, line 4 identifies the group that matches the text that indicates remaining work. Line 5 matches additional context that can be displayed to the user.

This context can save the user time with false positives. One can imagine how long and prone to false positives line 4 would have to be if the (not unheard of) practice of using only initials to indicate remaining work were the only common way of indicating remaining work. On the other hand, in the relevant artifacts for the S and P team, searching only for the remaining work indicator “TBD” actually does not yield any false positives. Interestingly, it is preferred to use, and some teams have standardized on, the indicator “TODO” which leads to false positives with relatively simple search expressions and techniques such as seen with “…ToDouble…” This was commonly used in part due to default support provided by major integrated development environments. Consensus is that broader checking for different types of remaining work at issue review is best.

Put directly into practice, this kind of work can provide useful, but limited indication of all the remaining work associated with an artifact (e.g. see lower portion of Figure 1). While this can be organized by and linked to files, shown with context, and presented with varying degrees of finesse, it is not particularly well suited for use at the time of issue review. We submit that, in addition to identifying a reference to remaining work, it is much more useful to also search for and utilize information about an issue which is tracking the resolution of that remaining work.

Searching for such a connection can be difficult, but can be bounded to make that search more expedient and practical. If a reference to an issue occurs in the text near the reference to remaining work, the search can be constrained to the same comment block or taken a bit farther, even within the context captured by the previous regular expressions. This can be further bolstered by best practices in documentation and coding standards. For example, the standards used by the S, P and related teams require that known remaining work be tracked in the relevant artifact with an issue number. Additionally, some of them require that in software source materials the term “TODO” be within a few characters before a reference to the issue identifier.

As alluded to in Section 2.2, this is even more interesting when organized and presented in the context of the issue under review, for example as shown in Figure 2. This helps minimize unrelated output and by organizing, reasoning about and presenting remaining work information according to its relationship with the issue under test, it can be prioritized and explained in meaningful ways. For example, references to the issue being reviewed and those with no tracking issue are generally more important than references to other issues. Similarly, references to other issues which are in areas changed as part of this issue’s changeset are generally more important than references to other open issues outside of the areas associated with this issue. Also, while not directly relevant
to the issue under review unless introduced by it, references on the tips of files associated with the
issue under review to remaining work tracked by issues which are no longer in a resolving state
reflect another sort of concern which should be raised and addressed.

This can be extended to not only have a focusing and prioritizing effect, but to have an integrat-
ing one as well. For example, analysis of indications of remaining work occurs on the text present
in the issue under review and these concerns are presented alongside references to remaining work
in associated documentation providing a focused and prioritized, but more complete view of the
relevant remaining work. Furthermore, in addition to integration across multiple sources, this is
integrated with any other related concerns (such as those described in Section 2.4).

Put into practice in PnitAgent (e.g. as shown in Figure 2), this checking has lead to the
appropriate creation of new issues and the rejections of others at the time of issue review. This
may also have further improved understanding of the context of the issue and related artifacts
under review in other ways as well.

2.4 Role Analysis for Process Support

A key opportunity and area of focus for our automation efforts relates to detailed process under-
standing, alignment and application within this domain. Significant process definition exists in tools
for Software Configuration Management (such as ClearGuide [11] or more recently IBM’s UCM [4])
and there are best practice guidelines and process definition for software development within the
medical device industry (e.g. [1, 2]). However, it is not fully understood how best to support process
in issue tracking and associated activities. More generally, detailed process support provides value
but comes with added complexity. As a result, general process support tools (like ClearGuide) that
provide fine control swing out of fashion and are replaced by more simplified process support (like
UCM). Delguach, for example, noted that better explicit role support for issue tracking systems
is needed “if we want to reason automatically about roles and their appropriateness or legitimacy
[5].” This is addressed by the capabilities we created as part of PnitAgent.

2.4.1 Reviewer Support

Improved role support is of particular importance as an enabler for automation of the process
of ensuring the appropriateness of reviewers. The Food and Drug Administration, for example,
requires “formal documented reviews” and that related “procedures shall ensure that participants
at each . . . review include representatives of all functions concerned with the design stage being
reviewed and an individual(s) who does not have direct responsibility for the design stage being
reviewed, as well as any specialists needed” [3]. These functional, independent, and specialist
reviewer roles are important in the review of issues, but not well supported by issue tracking tools.

Based on formative classification of rejections sampled in March 2010, problems with missing
or inappropriate reviewers of issues accounted for 11% of the overall and 13% of the (later and
more process-focused) software quality assurance (SQA) rejections. Addressing this involves un-
derstanding the situation of the issue at hand (e.g. duplicate, bug in test, requirement change,
. . . ) and the appropriateness of its reviewers. Resolving issues with reviewers, once identified, is
generally straightforward, often requiring the addition, removal, change or clarification of one or
more reviewers and their roles. However, it can be a serious problem if reviewers are not correct,
as appropriate review is essential to ensuring compliance, quality, communication, completeness,
objectivity, cross-pollination, and so on. For example, depending on the nature of an issue, it
can be important to ensure any changes to product have been independently reviewed or that the problem described by the submitter is the same as that addressed by the resolver.

<table>
<thead>
<tr>
<th>Situation</th>
<th>Reviewer Role</th>
<th>Field</th>
<th>Responsibility</th>
</tr>
</thead>
<tbody>
<tr>
<td>Normal SW (image)</td>
<td>Self</td>
<td>None (informal)</td>
<td>Self Review (informal)</td>
</tr>
<tr>
<td></td>
<td>Independent</td>
<td>Outstanding Reviewers</td>
<td>Independent Pntr Review</td>
</tr>
<tr>
<td></td>
<td>Test</td>
<td>Outstanding Reviewers</td>
<td>Test Impact Pntr Review</td>
</tr>
<tr>
<td></td>
<td>SQA</td>
<td>Outstanding Reviewers</td>
<td>SQA Pntr Review</td>
</tr>
<tr>
<td></td>
<td>Closure</td>
<td>Closure Reviewer</td>
<td>Closure Review</td>
</tr>
<tr>
<td>Duplicate</td>
<td>Self</td>
<td>None (informal)</td>
<td>Self Review (informal)</td>
</tr>
<tr>
<td></td>
<td>Submitter(1)</td>
<td>Outstanding Reviewers</td>
<td>This pnt was understood correctly and is clearly covered by the it is claimed to be a duplicate of. This pnt is not believed to be referenced anywhere in the curr example, there should not be “TODO: PNTXXX” in code or docun is this issue we’re about to mark a duplicate.</td>
</tr>
<tr>
<td></td>
<td>SQA*</td>
<td>Outstanding Reviewers</td>
<td>SQA Pntr Review</td>
</tr>
<tr>
<td></td>
<td>Closure</td>
<td>Closure Reviewer</td>
<td>Closure Review</td>
</tr>
<tr>
<td>(SDD)Design or Code Review</td>
<td>Self</td>
<td>None (informal)</td>
<td>Self Review (informal)</td>
</tr>
<tr>
<td></td>
<td>Independent</td>
<td>Outstanding Reviewers</td>
<td>Independent Pntr Review</td>
</tr>
<tr>
<td></td>
<td>SQA</td>
<td>Outstanding Reviewers</td>
<td>SQA Pntr Review</td>
</tr>
<tr>
<td></td>
<td>Closure</td>
<td>Closure Reviewer</td>
<td>Closure Review</td>
</tr>
<tr>
<td>Requirement Proposal</td>
<td>Self</td>
<td>None (informal)</td>
<td>Self Review (informal)</td>
</tr>
<tr>
<td>See also</td>
<td>Systems</td>
<td>Outstanding Reviewers</td>
<td>Systems review.</td>
</tr>
<tr>
<td>Sm3ChangingRequirements</td>
<td>HFE (UI</td>
<td>Outstanding Reviewers</td>
<td>HFE review.</td>
</tr>
</tbody>
</table>

Figure 3: Excerpt from a summary of expectations for issue reviewers

As another example, it can be important to ensure that changes or new additions are visible to those in other roles that may have to create tests for them, consider their broader system impacts, validate their use, and so on. As part of ensuring detailed understanding and alignment on such concerns, we created the Table in Figure 3 and updated PntrAgent to recognize the situation and reviewers associated with the issue at hand and warn if a certain review role appears to be unsatisfied.

Automating recognition of the situation associated with an issue involves natural language processing (NLP) of free form text fields and rules based reasoning including checking the resolution classification selection, related documents, related issues, and other values associated with the issue. For example, the issue resolution classification of a issue may have had “Duplicate” selected identifying the issue as a duplicate and allowing for the expected reviewer roles to be identified and
other scenario-specific concerns (such as an associated changeset without redirection of review) to be raised. In another example, an issue whose resolution was classified as “New Work Completed” or “Other” might reflect one of several different scenarios that would have different associated reviewers and need further analysis of the headline, description, related issues and associated documents to identify it as a design review. In addition to identifying it as a design review, the settings associated with its related project would be accessed to determine if a software quality assurance review was needed (depending on their history and other factors, required reviewers vary slightly with teams). More generally, in each scenario a different set of expected review roles might be required and different scenario specific concerns evaluated and potentially raised.

A particularly important aspect of avoiding false positives in this endeavor is using natural language processing techniques to detect when review is deferred to another issue or the role of reviewers is clarified (for example explaining that a certain reviewer is acting as the independent reviewer in addition to, or despite, their normal role as a test impact reviewer). To support this, the agent finds strings that appear to be redirects and then identifies the issue they appear to redirect to so that it can be presented to the user. This is done separately for each of the relevant fields in the issue to allow the user to be referred specifically to what field the apparent redirect appears in (this is done for remaining work references as well based on similar logic; see reference to the field “Resolution Description” in Figure 2).

Once the agent has classified the situation of the issue in question and that it has not redirected its review to another issue, it identifies the expected review roles and attempts to identify a reviewer that satisfies each of these roles. This can involve integrating information from several sources including a database of the jobs of certain individuals, the current project settings, the issue itself, the requirements management system, the document control system, and the version control system. For example, PnitAgent might identify the situation as a product bug fix, use current project settings (or a conservative default if not set by the user) to determine that a software quality assurance reviewer is needed, load the list of reviewers from the issue (e.g. joining a list of identifiers from the Completed Reviewers and Reviewers Outstanding fields) and then use the jobs database to ensure at least one reviewer is a software quality assurance engineer. Checks like this are relatively simple for human reviewers, but often overlooked.

To ensure an independent reviewer is present, PnitAgent takes similar steps, but cannot simply use the jobs database to identify appropriate reviewers. It eliminates software quality assurance engineers and others based on their jobs, but that alone would lead to many false negatives. So it also checks the issue contents to rule out those involved with the issue resolution. When configured to do more thorough checking it takes additional steps, for example, working with the version control system to identify the author of each change and eliminate them from the list of potential independent reviewers. As seen in Figure 4, whenever the agent warns about a problem with a reviewer, in addition to briefly identifying its concern, it provides a link to a website containing related guidance including the table excerpted in Figure 3 and a significant amount of additional content. In more complex cases, such as with a missing independent reviewer, the agent adds further explanation of its classification of reviewers (e.g. as resolvers, test engineers, or systems engineers).

2.4.2 Collaboration Support

Issue tracking systems are fundamentally systems to support complex collaborative work by teams of users. They provide basic support for this collaboration but allow users to safely and concurrently
work with shared issue data. Additionally, they often support sending users notifications via email about issues that may be of interest to them. It is not uncommon for a user to receive several, or even several dozen of these emails a day without any clear prioritization or indication of who and what are dependent on them.

While these are helpful to some extent, the task of remembering and prioritizing reviews requires diligence and continual attention. To support this, users often create and save queries to identify different sets of issues they are interested in and use those to recall and browse through issues or report directly on who owns them, their state, their outstanding reviewers, and other information which is needed to tell who they are waiting on and for what. Users often integrate their own knowledge of related process and their colleagues jobs to determine more precisely for each issue who is responsible for the next action(s) so that they can then gently encourage, or perhaps just monitor, progress in those areas. Sometimes, users give up on this analysis and simply poll anyone they think may be relevant.

Similar challenges exist with other work, such as the approval of documents in the document management system. Indeed, more generally, research on the information needs of developers concludes that their “most frequently sought information included awareness about artifacts and coworkers . . . [and] developers often had to defer tasks because the only source of knowledge was unavailable coworkers”[10]. With research [13] suggesting that such activities occupy more than half of a developers day and are often unplanned, the broader significance of these needs and their effects on interrupting the work of teammates can hardly be understated. Furthermore, an
additional challenge is that desktop-based communication is “too slow” with the need to “type out a coherent description of the problem.” These suggest that a context-aware solution could reduce context capture and communications time enabling more focused collaboration.

To better support these needs, PittAgent allows users to view, generally in a matter of seconds, an up-to-date list of the issues returned by a certain query. This includes a summary of who and what each issue is waiting on next. To initiate this, the user selects an appropriate item from the agent’s tray menu (see Figure 4). The agent executes the query and, for each issue returned by it, identifies what the issue is waiting on (e.g., resolution, review, closure). It identifies who this action is waiting on and, for review, more specifically what people and roles are next such as independent review, test review, or SQA review (which is only expected after other non-closure review). This involves similar processing to that for checking reviewers.

Additionally the user can ask the agent to notify those for whom issues in the query are waiting. In this case, the agent generates a separate email to each person that one or more issues are waiting on (after generating a unique list of people from the relevant set of needed issue-action-person tuples). It identifies their email address before creating and eventually sending the email telling them who asked that they be notified and exactly what issues are waiting on them for what. This reduces the number of people receiving emails and the burden on each of those people to parse through extraneous information to determine what they need to do.

Finally, the agent supports more flexible and complex batches of queries (through the “Query” menu which is not expanded, but visible in Figure 4). This allows users to rapidly gather and at least partially process data from multiple queries potentially involving a number of different systems. This has been used by some users to support relatively sophisticated metrics gathering. In a cross-team development leadership meeting, a principal software quality assurance engineer described her use of it to support issue tracking metrics saying “This will save us at least 3 person-weeks over the next year.” Another common use of this is to identify the status of design history file documents of interest to a particular project. Information on (typically a hundred or more) project-specific documents being worked on, reviewed and eventually approved by dozens of individuals is important (and previously time consuming) to track in the days and weeks leading up to major phase reviews.

![Figure 5: Example of integrated document status query](image)

In addition to system specific queries, this capability allows users to rapidly integrate information from several systems, working on their behalf to collect, analyze, and format data for them in the background. More interesting combinations and reasoning are possible. For example, the agent allows the user to identify a list of documents from the document control system and in addition to getting their approval and other status from that system, it searches the issue tracking system to determine if any open issues currently reference them so that those related issues can be closed.
appropriately. This allows for a more integrated understanding of document readiness and for that to be tracked more automatically over time as seen in Figure 5.

2.5 Change Sets Without Boundaries

Committing atomic changesets affecting several files under a version control system, and association of such changesets with an issue is important and supported by basic software configuration management. Similarly, associating each set of issue field changes, requirements changes, changes within documents in a document management system, and so on with the same issue is desirable and supported to varying degrees in traditional systems. Often, an issue tracks multiple version control changesets as well as changes made to other items such as requirements and documents. The resulting superset of changes (that we will also call a changeset) is generally not atomic (which is a problem left for other work), but is generally meant to be a highly cohesive set that is reviewed together at the time of issue review.

This is problematic because of the lack of integrated support to review all the changes together in as common a fashion as possible. Document changes from the document management system are often best reviewed alongside related code changes from the version control system. Similarly, for both items, although the steps involved with showing differences is slightly different, it is generally useful to view the differences between each changed document or file version and its predecessor version. Likewise, it can be useful to view the comment associated with the commit of each document or code changeset. For example, if one is trying to determine independence manually, seeing a list of who made changes to documents in the same list as those who made changes to code under the same issue is desirable.

![Image](image.png)

Figure 6: Example PnitAgent showing integrated changeset listing

Review of the entire changeset of an issue is supported by PnitAgent primarily in two different ways. The first involves providing and focusing warnings and presenting them to the user based on integrated data from various sources as described previously. The second takes the form of an integrated change listing which shows a single comprehensive list of changes regardless of what content type or host system tracked this change. This includes not only obvious, and automatically associated changes like those made in associated changesets in the version control system, but also those from the Medtronic Revision Control System (MRCS), where design history documents are kept and approved, but changes cannot automatically be associated with an issue (e.g. see Figure 6).
To enable this, certain text fields of the issue under review are parsed for references to MRCS documents. Double clicking on any change item in the list causes the agent to take the default review support action for that item, which generally means retrieving the changed and previous version of the files and displaying them in a graphical differencing tool, converting them if needed. Right clicking on any item brings up a context menu which exposes additional support options including those seen in Figure 6 and several others. To some extent these actions and options vary, for example with the host system and change item. As shown in Figure 7 when the change item is an MRCS document identified by analyzing the issue content, it is possible that some essential information, such as the version number, can not be found. For such items, very limited actions are supported by the agent. Though this often reflects a problem with the documentation of an issue and it may be appropriate to make that apparent, when automating support for systems with natural language inputs, flexibility of this sort is essential.

![Figure 7: Example changeset with missing data](image)

Another challenge with data from multiple sources and software that operates on behalf of varied users is the need for robustness despite the complexities of resource availability and environment. In addition to other details described in [7], flexibility in the face of partial failure and well thought out error handling have proven to be useful in PnitAgent. For example, the agent prefers to access the version control system and issue tracking system through COM and other low level interfaces which results in greater performance, but these are dependencies that are not met on some users computers. When these interfaces are unavailable, the agent briefly explains the situation to the user and attempts to use command line clients or web interfaces through a headless browser. Depending on its level of success, the agent alters its user interface removing options, alerting the user, or taking other actions such as changing the color of user interface elements (see Figure 6), to indicate that the associated actions will have slow performance or other limitations.

This may seem like a great deal of flexibility and integrated functionality to provide changeset support with as few boundaries as possible, but there is always a desire for more. Important areas of current and future work include improved, integrated support for looking at changes in the content of issues themselves and at sets of changes in requirements. For example, after an issue is rejected and further addressed, it typically returns to the same reviewers. In such cases, users often want to see exactly what is changed in the content of the issue as well as in the related artifacts. In addition to support “since the last time I reviewed this,” it is desirable to look over longer periods with more regular measurements to understand the status and flow of work through the issue tracking, requirements management and other systems tracking the work of the development teams. Some topics related to these concerns are discussed next.
3 Automation to Increase Project Status Accessibility

In this section we describe our experience with automating support for data mining and analysis to provide a better understanding of general software development status. We describe some of our efforts to facilitate the mining and refining of information about recent and longer term changes in issues and requirements.

3.1 Effort Management

This work involves improved support related to the analysis, tracking and management of effort. This goes beyond the details tracked by annotations as described above and into what might be traditionally associated with project management. Prior to the last decade, Medtronic used almost exclusively plan-driven effort management. Recently, more agile methods of estimation and effort tracking have played an increasing role with many coming to the conclusion that more empirical effort management is a key part of “a better way” to pursue software development in this domain [14].

Though some teams in this domain have attempted to exclusively use empirical approaches to effort management, ultimately most teams working on cross-functional projects return to some hybrid form of empirical and defined/plan-driven effort management processes. This appears to be consistent with the more general SCM literature: “Traditional project management systems are not ideal for software projects . . . In contrast, workflow systems [are based on] the request backlog, and the average time to process a request. Some aspects of software projects are best modeled by workflow systems, for example, incoming defect reports [11].”

How to most effectively find this balance between empirical and defined (or workflow and project management) systems and the complexity and overhead of maintaining and pursuing them both effectively in practice is often not clear. As the domain in question requires that all or nearly all work done on all major artifacts (including code, review, and documentation) be tracked by the issue tracking system, we submit that the issue tracking system provides an existing mechanism and data repository for most of the major items associated with effort in our area of focus.

Furthermore, while the issue tracking system contains a mix of issues associated with scheduled and unplanned tasks and has not been designed to most effectively support effort management activities, it can be used for this purpose in combination with our software. In particular, PnitAgent can support effort analysis, tracking and management effectively with relatively low overhead for its users.

PnitAgent supports a range of activities from tracking resolution and review velocities within areas of interest to identifying changes and imbalances in team and individual effort backlogs, to reporting accomplishments at a high level of detail, to identifying who and for what a body of work is waiting, to supporting collaborative team work through communicating requests and priorities and so on. This involves, for example, allowing users to provide effort estimates in the free-form text of their issues and parsing this information out with the agent and, when such information has not been provided, having the agent estimate remaining effort. Agent estimation can involve attempting to determine the scenario of the issue, who owns it, who is expected to review it and so on. It this way the agent allocates effort for the resolver if no estimated effort has been identified and for each outstanding reviewer regardless. The agent then allows for aggregates of these efforts to be reviewed across certain groups of issues or the people who currently own (or it expects will own) the relevant activities.
3.2 Open Product Code Issues

Although issues provide a measure of the known work remaining to be completed as part of a project, it is often valuable to examine only the portion of those issues which are still open and have caused, are causing, or will cause changes to the product. These reflect the amount of known change left for the product as well as the visible risk of further defect introduction through resolution of these issues. Viewed as a time series, for example as shown in Figure 8, this may be a helpful indicator in gauging the convergence of software before release. This is not trivial to automate.

![Pre-Review Product Code Issues](image)

**Figure 8:** Example plot of open product code issues from two related projects

Issues which are sufficiently far along in their lifecycle can be readily analyzed to determine if they have had associated code changes. Identifying which of these code changes are in the product and which are not can be leveraged from the scenario determination analysis described previously. With less mature issues (such as those which have just been submitted or are largely incomplete), it is more difficult, even though these typically more recent issues are of great interest to current decision making. To allow for this additional analysis, the agent seeks advice from the user while attempting to keep related user burden to a minimum. To do this, the agent gets the latest data on each issue and classifies all sufficiently mature issues and those which are not mature but already have product code changes made against them. It then looks for a cached answer to whether the issue is expected to affect product code. If found, that is used to classify the issue until it is more mature. If no answer is found, the agent prompts the user and caches their response.

In other situations, the agent caches answers to questions that come from intensive processing or interactions with other systems (e.g., forensic audit trail access and parsing). In addition to persisting simple answers, the agent applies the same principle with somewhat more complex data and techniques to capture “baselines” of issue tracking, requirements and other information. This has several advantages including the ability to be more efficient or robust in the face of resource access challenges and even bridge between tools due to its abstraction and persistence of relevant information. Additionally, it can later use baselines to provide capabilities which are difficult to reconstruct with the underlying systems alone. This includes support for rapidly determining what has changed in a group of issues or requirements week to week as described next.
3.3 Tracking Change in Issues

Using the agent’s support, high level information such as that shown in Figure 9 can be synthesized to support intuitive measures of the growth rate and volume of overall issue tracked work, unresolved issue or review debt a project has, and how many reviews and rejections are being performed by a team (even when that doesn’t immediately translate into changes in high-level issue state). Used at an even higher level of abstraction and aggregation, these are useful for not only consistent presentation of related concerns across projects, but for resource levelling and insight into the impacts of project priority calls. Such information is also further extrapolated to determine if a team is keeping up in different areas or even predict a rate of convergence (e.g. based on recent rate of change or more complicated models of issue discovery and completion).

![Figure 9: Example plot of review rates each week](image)

These are often intuitive and useful, but not trivial to automate. For example, rapidly analyzing and sub-classifying a set of issues over time can be difficult, such as when considering issues related to a feature which is not well classified by the system or when working to observe only Open Code Issues. The agent supports rapidly and retrospectively constructing such information and associated visualization.

Similarly, the agent supports much more detailed analysis of change in a set of issues. The agent is used weekly by the S and P teams to classify and summarize rejections which occurred and major changes in issues. This has been particularly useful when concerns are less focused on issue debt (which the team may be fast enough to keep at zero) than the rate of incoming product code issues which may keep the team busy and the schedule slipping even when the total count of issues is always low. Similarly keeping up with recent rejections allows software leadership to recognize themes and respond to process concerns in a timely fashion.
3.4 Tracking Change in Requirements

In much the same way that it supports tracking change in issues, PnutAgent automates analysis of change in requirements. This involves identifying, classifying and quantifying different sorts of change over time. This is applied across all traceable items managed by the requirements management system, which can be viewed overall such as seen in Figure 10.

![Chart showing change in requirements over time]

Figure 10: Example change in all requirements

This is decomposed into groups of interest such as inputs, system-level requirements, low level requirements (see Figure 11), design, and test records. These can be used to observe scope change, churn, the ripple of changes, convergence at different levels, and more generally the contribution of a more specific set of traceable items to the overall picture. At a more detailed level, the agent helps to support review of change in tracing, attributes and requirement content for a specific requirement or group of requirements.

4 Conclusions

This paper describes automation to support regulated software development activities, specifically those relevant at the time of issue review and for improved project understanding. Based on usage reports send by instances of PnutAgent as of March 2011, this kind of support was used regularly
by most of the software developers on the P and S teams and to support weekly status activities of their management. One developer wrote “I found the PnitAgent tool you created extremely useful when doing reviews. Not only have I been able to complete them faster, but the tool found problems I may have otherwise missed.” As described in [7], there were a number of qualitative and quantitative improvements after the introduction of PnitAgent including reduced rates of rejection overall (from 17.3% to 10%) with rejection rates of issues resolved by regular users lower and more improved from before the impact of the agent.

This paper elucidates advantages of issue focus; thoughtful integration of data from artifacts and configuration management systems; and a variety of forms of change analysis support. Although grounded in detailed examples, challenges and topics from their adoption at Medtronic Neuromodulation, these may provide an interesting and practical starting point for automation in other domains.

5 Acknowledgments

We would like to thank Mats Heimdahl and Mike Whalen at the University of Minnesota, Leigh Drew, and our many colleagues at Medtronic for their feedback and support.

References


