Advisor Agent Support for Issue Tracking in Medical Device Development

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

This case study concerns the use of software agent advisors to improve efficiency and quality in issue tracking activities of development teams at the world's largest medical device manufacturer. Each software agent monitors, interacts with, and learns from its environment and user, recognizing when and how to provide different kinds of advice and support to facilitate issue tracking activities without directly modifying anything or otherwise violating domain constraints. The deployed software agent has not only enjoyed regular and growing use, but contributed to significant improvements. Issue rejection was significantly reduced and more focused, yielding significant quality and efficiency gains such as fewer reviews by quality assurance. This success reflects the benefits of the underlying AI technology.

Application Domain

The medical device industry produces software-intensive devices such as implantable pacemakers, drug pumps, and neurostimulators that provide critical therapy for millions of people with a range of diseases from diabetes to chronic pain. In this domain, software is developed under strictly regulated processes to ensure patient safety. A hallmark example of why this is so important is the radiation therapy device called *Therac-25* for which poor software development practices were identified as the root cause of three associated patient deaths in the 1980s (Leveson and Turner 1993). To help prevent such catastrophic outcomes, issue tracking is used at the heart of development, management, and quality assurance processes.

The issue tracking process manages and documents each concern, its resolution, the review of that resolution, related artifacts, and its closure. This process converts issues identified by team members, including enhancements, defects, new features, and other items, into documentation and product. These are eventually delivered to such stakeholders as regulatory agencies, clinicians, and the host business itself.

In this paper we describe a novel software agent, called *PnitAgent*, used to support the issue tracking activities of teams of engineers (that we call *S team* and *P team*) working together to develop software for next generation neuromodulation therapies. The S team (n≈20) focuses primarily on drug delivery device software, while the P team (n≈15) focuses on platform software that supports communications, data management, and other functions common to drug delivery, spinal cord stimulation, and other therapies. Part of this software will be used by clinicians as their primary means of interacting with many different medical devices including nearly all recent and next generation implantable drug delivery systems. In these systems, delivering fluids too quickly or too slowly are among several hazards that can lead to severe harms such as overdose (e.g., with opiods for treating pain) or underdose (e.g., with Baclofen for treating spacticity). Improving software quality not only saves effort but also increases product safety.

The Production Neuromodulation Issue Tracking System (PNITS) and the supporting Software Quality Assurance (SQA) engineers tracked the essential activities of the S and P teams before PnitAgent was first used in early 2010 and more broadly after July 2010. Data prior to the introduction of PnitAgent reflect the mechanisms developed and refined by Medtronic for identifying quality related software issues and for tracking their resolution. The characteristics and performance during this *before* period establish the baseline against which we measure the impact of PnitAgent.

The S team had accumulated a number of issues over the previous 2 years. 537 issues had been closed, 560 issues had been reviewed by SQA and more than 1000 had not yet received review by SQA (typically issues received several reviews with SQA review just before closure review). Of the issues reviewed by SQA, more than 50% (268) had been rejected at least once, and more than 20% had been rejected at least once by SQA.

Several members of the team chose to co-locate in a laboratory-environment, held daily meetings, practiced pair programming and pair issue tracking, sat with the development lead that acted as the local process owner, and employed a TWiki website for distributed capture and review of detailed process description. However, detailed process alignment, scenario-specific application, and in-context recall of the relevant details remained elusive.

Near the beginning of April 2010 a team including the author of this work was formed to improve issue tracking. This team classified a sample of 120 rejections (30 SQA and 90 peer/self rejections) and abstracted these into themes. More than 75% of rejections related to the story told by issues (primarily problems with version, resolution description, related issues, and review information) rather than defects in the systems under development. Flaws in the product under development (source code, documentation, verification, and other artifacts) are often found in other related activities (development, code/design meetings, or test development), but are within the scope of issue reviews. These flaws were commonly referred to as "content issues". The major sources of rejection, along with concerns identified by interacting with potential users, formed the basis of PnitAgent.

The issue tracking system cannot be changed easily because of the associated costs and the need to maintain consistency. It is also not feasible to provide any direct replacement for user review or editing of issues.

These domain constraints naturally suggested the use of advisor software agents. Knowledge and expertise can be built into and collected by agents which monitor and interact with their environment and users and provide different kinds of advice and support without directly modifying issues, altering validated systems, or otherwise violating domain constraints. The agent must be appropriate for various degrees of user adoption under real conditions. This implies usability, performance, and other requirements such as the ability to be used and maintained by different users with low effort.

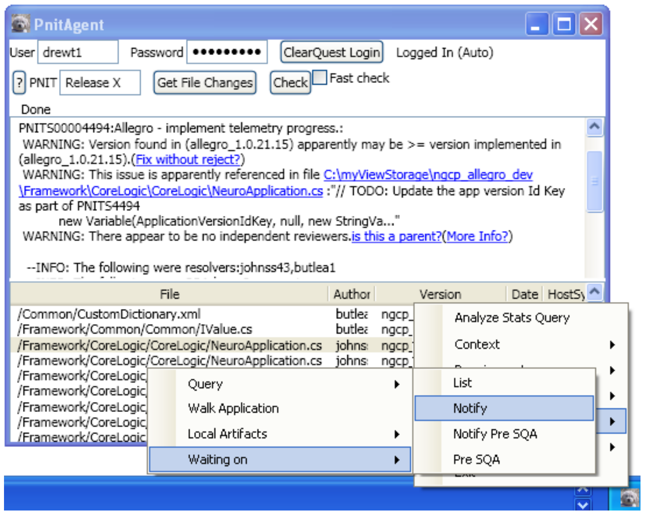
Example Use

This section presents an example use case to introduce the domain and agent. In this scenario, a developer was assigned an issue; has made changes to product software and documentation; and has described her initial work in the issue.

Intending to check over the issue, the user opens it in the issue tracking system. PnitAgent has been running in the background, recognizes that an issue has been brought into focus, and identifies the issue. It gathers information from the issue tracking system and presents a small bubble window in the lower right hand corner of the user's screen. The bubble window has links that allow the user to check the issue for problems or take other actions. The window fades away over a few seconds, but first the user clicks on a link in it to request *Automated Issue Analysis*.

The agent collects data about the issue from the issue tracking system, the version control system, its local memory files, and product artifacts. It uses text processing and information retrieval techniques to analyze the data (as described later in *Agent Design* and in (Drew and Gini 2012)). When it identifies some concerns, it presents them to the user in the form of warnings and related information. In this case, the agent has determined from the free-form English text entered in the issue description and resolution description that the user has made changes to fix a problem in the product software and that there is no indication that review of these changes should be deferred to another issue.

The agent analyzes the assigned reviewers and who has contributed to the resolution of the issue and its associated artifact changes. Based on this, it produces a warning that there does not appear to be an independent reviewer even though one is expected in this situation. It explains which reviewers appear to have been contributors based on issue


User drewt1 Password ••••••••• Logged In (Auto)

PNIT Release X

Done

```
PNITS00004494:Allegro - implement telemetry progress.:
 WARNING: Version found in (allegro_1.0.21.15) apparently may be >= version implemented in
(allegro_1.0.21.15).(Fix without reject?)
 WARNING: This issue is apparently referenced in file C:\myViewStorage\ngcp_allegro_dev
\Framework\CoreLogic\CoreLogic\NeuroApplication.cs :"// TODO: Update the app version Id Key
as part of PNITS4494
         new Variable(ApplicationVersionIdKey, null, new StringVa..."
 WARNING: There appear to be no independent reviewers.is this a parent?(More Info?)

  --INFO: The following were resolvers:johnss43,butlea1
```

File	Author	Version	Date	HostSy
/Common/CustomDictionary.xml	butlea	ngcp_		
/Framework/Common/Common/IValue.cs	butlea	ngcp_		
/Framework/CoreLogic/CoreLogic/NeuroApplication.cs	johns	ngcp_		
/Framework/CoreLogic/CoreLogic/NeuroApplication.cs	johns	ngcp_		
/Framework/CoreLogic				
/Framework/CoreLogic				
/Framework/CoreLogic				
/Framework/CoreLogic				
/Framework/CoreLogic				

Figure 1: Example agent window and tray menu

tracking and version control data, and which users do not seem suited to be independent reviewers of this issue because they are verification or SQA engineers.

The agent also warns of areas of vagueness in the resolution description, contradictions between a version mentioned in the resolution description and other fields in the issue, and that the issue has been referenced from comments about work that is not yet complete in the code and design documents. In addition to textual explanation, the agent provides a link to a relevant website that documents the review roles expected and required in common scenarios.

After resolving or deciding to ignore the warnings, the user asks the agent to get all the artifact changes associated with the issue by clicking the "Get File Changes" button. This triggers a related desire/event for the requested goal to be raised in the agent's primary control thread. The agent starts a short term background thread to act on the goal. The background processing is tied to the "Get File Changes" button, which the user can use to stop it prematurely, and is *safe* in the sense that abort or failure of, or in, the thread will not be treated as an error. The background thread checks that the issue context is valid and gathers information about it and the files referenced from the issue tracking system, dispatching status updates back to the primary control thread which presents them on the main UI window if it is open.

The agent extracts information on change sets from the issue tracking and version control systems before examining the text of the issue to identify any changes made to artifacts not kept in the version control system. This involves retrieving free form text fields within the issue, replacing noise characters, removing noise words, using a regular expression to match references to documents in the document control system, then doing some additional information retrieval (Manning, Raghavan, and Schütze 2008) to extract version information about any document references that were found. The agent eventually integrates, organizes and dispatches a list of all the associated file changes to the primary thread.

Finally the agent presents a list of changes to the user, who re-sorts them by clicking on the column headers. The user then double clicks on the change-items and the agent takes its default support action for each. This generally takes the form of retrieving the changed and previous versions (e.g., from the version control system or the Medtronic Revision Control System) and displaying them in a graphical differencing tool to the user, converting them if needed.

Occasionally, the user uses a context menu to do other types of review activities, such as start static analysis, view associated commit comments, or review approval status. Finally satisfied, the user marks the issue as ready for review.

With her last issue resolved, the user wants to urge her colleagues to complete any remaining work for the upcoming release. Having already created a query named "Release X" in the issue tracking system to return the relevant issues, she types "Release X" into the agent's entry box. Then she uses a menu item to notify the outstanding users (see Figure 1). The agent sends an email to each relevant user letting them know which issues are waiting on them, for what, and who asked that they be notified.

Agent Design

Key design challenges in the design of PnitAgent included:

1. Monitoring the user activities in a way that is non-intrusive, has acceptable performance, is ubiquitous, and is easy for users to control.
2. Abstracting the complexity of user and environment specifics to ensure robust and appropriate operation with different user credentials, preferences, levels of experience, and access to tools, data and services.
3. Building a model of each user, with history and preferences, and creating appropriate analysis tools to enable the agent to support each user.
4. Presenting information, intentions, status, and suggest next actions in a way that is helpful and acceptable to users.

This work builds on previous research into highly abstract monitoring of the user's actions as the user shifts focus between windows and widgets within those windows (Drew and Gini 2010). This previous work described the agent's probabilistic modelling of sequences of user focus. The agent observes the user and builds a Markov-chain model of the sequence of contexts it observes over a period of time. It then uses the model to recommend next user actions. The model leverages an efficient trie-based data structure and an algorithm for training and prediction that is similar to the Prediction by Partial Match algorithm proposed in (Cleary and Witten 1984) and further analyzed in (Begleiter, El-Yaniv, and Yona 2004). In this way the agent can guide users though complex and unfamiliar user interfaces.

This previous work also described how the agents support ubiquitous annotation by leveraging a simplified form of programming by demonstration in which the agent observes the user as she demonstrates UI focus contexts and associates annotations with them. The agent later recognizes that the user (or someone they have distributed their annotations to) enters one of these contexts and presents the associated annotation content. This capability was simplified in PnitAgent to remove the need for a browser plugin, and was seeded with content from previous training to make it immediately useful without requiring users to train the agent. Advanced users may selectively train, save, combine, and load alternate models as may be appropriate for their team or other specific use. This proved useful for new users, but for most use it was so pedantic as to be annoying.

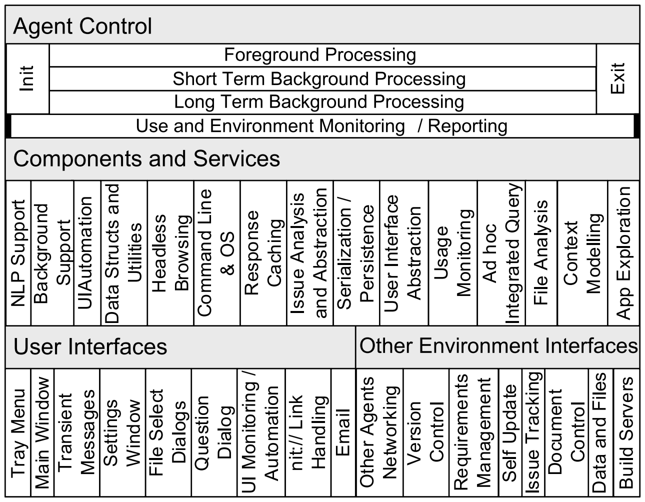


Figure 2: Logical architecture diagram

To address these shortcomings, PnitAgent gathers information directly from the various tools and systems available to it in addition to working through the graphical user interface (see Figure 2). The fact that the agent gathers information directly from other systems creates a greater coupling to existing software that made deployment and maintenance more difficult, but enables additional capabilities. For instance, access to data, such as textual, unstructured, user-entered descriptions of concerns and resolutions requires the ability to effectively process the contents of the text. The agent does not do full grammatical analysis of the English free form text, but uses a series of application-specific pattern matching and basic language processing techniques (e.g., stemming, proximal text extraction, and replacement of noise words and characters). An example of simple stem matching is given in Listing 1.

Listing 1: Simplified stem match for target modifier

```
if(lowerQuery.Contains("clos"))
  modifiers(ModifierTargetType.ToClose);
```

To increase performance, the agent exploits information about the context to guide its process of elimination in extracting and classifying information.

The agent is built around dynamic dialog with its environment. The agent control layer of the logical architecture (see Figure 2) reflects the control and flow of its processing. The agent performs its primary processing on a main thread onto which some key event processing is dispatched and from which other background threads are started and managed. Ongoing monitoring and reporting provide support and input events even during initialization and shutdown. Much of the agent's architecture and processing is structured around interfaces with the user and the environment, that are supported and abstracted by services and components.

Each agent maintains information that is unique to it and its users, including observations, user responses, context associations, and use information. It acts differently based on that information and a number of environmental factors such as who the user is, what folder it was started from, who logged in with it last, what version its software is, what systems it can access and the like. The agent keeps track of its past and current users with separate information associated with each. Unless a user has been explicitly identified to the agent, it typically assumes its user is the one currently logged into its host operating system (OS).

Multiple agents can act on behalf of the same user. Different instances started from the same location share persisted information. If these are started by the same user (or as link handlers on their behalf), they will heavily share information, but may operate on different tasks and observe different user interfaces (e.g., if started on different machines from a common network folder). Multiple instances may be started simultaneously by one user on one or several machines for various reasons. For example, the OS starts new agent instances to handle links activated outside of an active agent and, depending on what is requested, a new agent instance may start, complete the task, and exit automatically.

A key focus of the design was enabling the agent to dynamically alter how it observes and interacts with the user and electronic resources around it. For example, when the agent experiences an unanticipated exception while trying to access the COM interface to the version control system, it keeps a note that it is unavailable and tries the command-line interface. More generally the agent identifies the availability of the interfaces and systems by attempting to find and test them, validating their responses and recognizing failures in this process and during use. If a system is completely unavailable, the agent overrides and hides access to any conflicting settings or actions and explains the situation both immediately and as needed later on (e.g., indicating that reviewer independence can only be partially checked without version control access).

The agent uses information from the OS to identify the local user and, based on that, searches for other information about them. For example, if a user asks the agent to notify others about what is waiting on them (through a link or as shown in Figure 1), the agent will not notify all users about everything waiting on them. Instead, it asks the user for context (or retrieves it from the PNIT field if it is not already in focus on the screen). Responses in forms including "4494", "Release X", "everything related to 4494", "waiting on my closure", and many others are understood by the agent. This is accomplished by processing summarized in Algorithm 1.

The algorithm uses application-specific techniques such as the regular expression shown in Listing 2 to match, classify, and extract information from the user's response.

Algorithm 1 Get issue context from user response R

```
if $R$ is an issue id then ...                                   ▷ "4494", "PNIT 4494"
else if $R$ is a publicly defined query then ...
else if $R$ is a $currentUser$ defined query then ...            ▷ "Release X"
else if $R$ is a list of issues ids then ...      ▷ "1,2" "Pnit1 Pnit2"; see Listing 2
else                                      ▷ $R$ is a natural language query or invalid
    if $R$ is a 'waiting' type query then               ▷ "the stuff waiting on doej1"
        $type \Leftarrow waitingOn$
    else if $R$ is a 'related' type query then         ▷ "everything related to 4494"
        $type \Leftarrow relatedTo$
...
    end if
...
    if $R$ has a specific issue target then            ▷ "everything related to *4494*"
        $target \Leftarrow$ issue id from $R$
        $targetType \Leftarrow issue$
    else if $R$ has a specific user target then         ▷ "the stuff waiting on *doej1*"
        $target \Leftarrow$ user id from $R$
...
    end if
    if $R$ has current user target then                       ▷ "... my ...", "... me ..."
        $target \Leftarrow currentUser$
    end if
    if $targetType = specificUser$ or $targetType = currentUser$
then
        $targetMods \Leftarrow$ modifiers from $R$         ▷ "... my *closure*"; see Listing 1
        if count of $targetMods$ ¡ 1 then
            $targetMods \Leftarrow resove + review + close$
        end if
    end if
...
end if
```

Listing 2: Regular expression identifying lists of issue ids

```
new Regex(”(,|;|([0−9]+\\s[0−9]+)|([0−9]+\\spnit))”,
```

The use of domain-specific information retrieval techniques is possible because, despite the large variety of information the agent needs to find and process, the domain is constrained and structured around the issue tracking interface and the issue context.

Agent Capabilities

PnitAgent is a true assistant, which supports the user in a diligent, proactive, and non-intrusive way. The support is provided through a variety of capabilities as described next.

Annotation and Sequential Support

The agent supports display, capture of annotations, and next step recommendation by constantly monitoring the window and widget in focus. This nearly ubiquitous capability was seeded with content to make it immediately useful to every user. This included a set of annotations that are shown automatically as users visit an issue's fields and selections to provide usage information. For instance, while selecting the *Fixed* value for the Resolution field shown in Figure 3 the agent provides a link to more information and indicates that *Fixed* is used only when a defect has been fixed under the *current* issue.

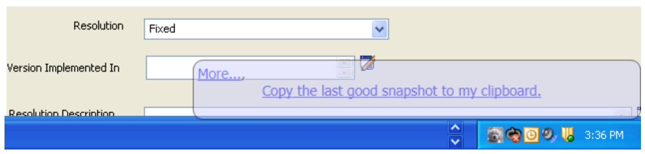
Automated Issue Analysis

This capability provides automated analysis of issues and their associated artifacts, with the aim of giving warnings to help users identify problems and context information. This requires access to issue tracking and other data, which the agent accesses through command line, COM, dynamically linked library, web, and other interfaces on the users behalf.

Once the agent has logged into the issue tracking system, it provides a text field where the user can identify one or more issues of interest by typing an issue id or the name of a query stored in the issue tracking system. This is flexible (as described earlier), but requires the user to know and enter this information. The user then activates the “Check” button to begin full analysis (taking a minute or more per issue) or “Fast Check” for faster but less thorough checking (taking only a few seconds per issue). While checking, the agent reports its status in a transient text block and prints out the results of its analysis and explanations into a larger area.

Hybrid Interactive Support

Support is also provided in a hybrid form, which combines *Annotation and Sequential Support* and *Automated Issue Analysis*. This requires the agent to monitor the user’s focus to see when the user enters a different context, and provide context-appropriate dynamic content or links to capabilities. For example, the agent recognizes in the background that the user is looking at an issue when they select or open it in an issue tracking client. The agent proactively analyzes the issue, and, only if concerns are identified, presents those concerns to the user in a transient message. In another example, the agent recognizes that the user is modifying the “Version Implemented In” field of an issue. The agent helps identify information about the appropriate build and copies it to paste buffer, as seen in Figure 3.



Resolution	Fixed
Version Implemented In	
Resolution Description	

More...

Copy the last good snapshot to my clipboard.

Figure 3: Example of hybrid support for version recommendation

Associated Artifact Review Support

The agent provides two primary modalities of support for reviewing the artifacts associated with an issue under review:

1. When the *Automated Issue Analysis* is enabled, the agent examines the contents of the issue itself, and does additional analysis to identify concerns in related artifacts.
2. When asked to identify artifacts directly associated with an issue, the agent finds the artifacts and supports useful activities related to those artifacts.

In case 1, the agent uses a cache of results from the last time it looked through the contents of the files in the associated version control workspace(s). It identifies which files have changes directly associated with the issue, updating the cache first if necessary. Using this information it raises relevant concerns. For example, it issues warnings about any case in which the issue under review is referenced from one of the cached files (e.g., Microsoft Word documents, or product source code files). The agent intentionally excludes from its search certain files that commonly retain references to issues (such as change summaries or files that appear to be design or code review records). The remaining references reflect work on the current issue that remains to be addressed before the issue is ready for review. The agent also warns about similar concerns where issues that are already closed appear to be referenced from such reminders. In another example, the agent checks that all static analysis exclusions are bounded and justified and that the security coding standards (not enforced by other tools) have been met.

In case 2, the user can ask the agent to list all the artifacts that appear to have been changed as part of the issue under review. This includes parsing the change package information automatically associated with an issue by the version control system as well as examining the free form text entered by users into various fields of the issue that often describe related artifacts controlled by systems other than the version control system. The list is presented in two forms, one as text that the user can copy and modify, and a second as a spreadsheet in which users can sort (by attributes such as file name, version, date of change, or author of change) and interact with each item by double-clicking or working through a right-click context menu. In this way, users can rapidly identify differences, promote comments, contributing authors, and various versions of the affected artifacts without having to work through the interfaces of the different systems in which they are stored.

Analysis and Collaboration Support

Finally, the agent supports related analysis and collaboration tasks. An agent can be asked to save baselines and identify changes in requirements and issue tracking systems or just a specific subset (e.g., defined by a query or list) for individual users. The agent can provide detailed information rapidly and transparently. The agent can identify who and what an issue, document or group of items are waiting on and notify users of their pending work with user-specific emails. The agent also registers itself as a link-handler to be started by the OS when a specific link is clicked. This allows the agent to retrieve artifacts from different systems or take actions such as navigating through items on the user interface (e.g., to reproduce a bug). Its support for saving baselines, calculating statistics and providing notifications can be left running automatically in the background, for example to create weekly status updates. Certain analysis tasks, such as the number of open “product code” issues in a baseline, involve asking questions to users (e.g., as to whether they think an early stage issue will affect product code). Agents save those answers, referring to them later to avoid repeat questions.

Other Capabilities

The agent has additional capabilities that are useful for its support. For example, it logs its usage, automatically attempts to update itself, and emails use, and exception information back to its administrator. Users can also ask their agent to start an email to the administrator filling in useful log and version information that the user can send or modify.

Development, Deployment and Maintenance

PnitAgent was developed by a single developer outside of their full time job, consuming an estimated 400 hours of time during the fall of 2009 and spring of 2010. Until April 2011, the agent was intentionally left unchanged to reduce variables that could complicate use and impact analysis. After this, about 4 hours a month from the original developer have been devoted to maintenance and enhancements. In the aggregate, a few hours of implementation, dozens of hours of beta testing, several hours of discussion, and a few hours of documentation and validation exception support were provided by others within Medtronic. No direct funding or planning was provided by Medtronic, but company resources including meeting facilities, computers, servers, and proprietary software were used for development, deployment and maintenance of the agent.

Challenges during development were identified and addressed with short development cycles directed by ethnography and beta test feedback. This iterative, user-focused development was uniquely possible with the AI-based approach employed. The advisor agent paradigm allowed the capability to be deployed without requiring the overhead of formal validation, which would have slowed or stifled further change after it was first used. Furthermore, the autonomous capabilities of the agent to dynamically adapt, facilitate user and agent feedback, and update itself transparently when appropriate made its evolution more rapid.

The agent was beta tested in more than 50 versions that, anecdotally, largely went unnoticed by beta users. Its basic functionality was kept unchanged after its release to allow for its impact to be analyzed without additional variability.

Evidence of Success

Information collected from agent emails and the issue tracking system itself provide evidence for the success of PnitAgent. Dozens of users adopted the agent and the rate and nature of issue rejections it was designed to address improved. In fact, regular users of the agent, saw greater rates of improvement than those who did not use it regularly. Additionally, the agent may have dampened the negative effect of issue complexity on rate and repeat of rejection. Finally, as a result of this success there have been changes in business practice such as reduction in reviews required by SQA in groups using PnitAgent.

In evaluating the impacts of the agent, we compare the period before its use with a period afterwards. Unless noted, we consider the period *before* to include issues that were reviewed by the S and P teams before January 2010 when a couple of users began to provide feedback on an early prototype. Similarly the period *after* includes only issues first reviewed after announcement of the agent and broader use following a presentation on 16 July 2010 but before 14 March 2011 when data analysis for impact evaluation first began.

Improvement in Rates and Timing of Rejection

When comparing these periods, the overall rate of rejection decreased significantly as shown in Table 1. Table 2 shows how the number of reviews before a rejection and the reject rates at all levels decreased. All the differences are statistically significant (at 95% or better confidence level).

Table 1: Overall before & after

	Issues	Reviews	Rejects	Rate
Before	355	1954	336	17.3%
After	517	2668	267	10%

Table 2: Decrease in reviews before rejection and reject rates

	Avg #	SQA reject rate	Closure reject rate
Before	4.42	19.8%	11.9%
After	3.95	7%	3.2%

For a project of this size with a total of approximatively 7,000 review cycles a conservative estimate of the savings due to the reduced time spent on SQA reviews and rejection rework is $350,000.

Greater Improvement with Regular Agent Use

To examine the relationship between use of the agent and issue tracking improvements in the *after* period, we divide the populations of issues (and their reviews) based on how much their resolvers had used PnitAgent. About half of PnitAgent users qualify as regular users, because their agents reported more than 5 days of use and 12 issues checked. Although not all regular users were developers, most S and P team software developers were regular users. For all but one of them, there was an email reporting use within the last nine days (between March 10, 2011 and March 18, 2011). Their median reported days of PnitAgent use was 84 *work days* (where a new work day of use was defined to start with any detected use more than 12 hours after that of the start of the last work day of use). For these regular users the primary use involved *Automated Issue Analysis* (directly and/or through *Hybrid Interactive Support*) with a median rate of 2.04 issues analyzed per work day. All of them also used the agent for issue related artifact review and all but three for *Annotation and Sequential Support*.

Table 3: Decrease in reject rates due to agent use

	Overall Reject	SQA reject	Closure reject
Regular users	7.9%	4.0%	1.6%
Improvement	50.9%	78.9%	
Others	12.3%	9.9%	4.3%
Improvement	38.2%	57.5%	

Table 3 shows how reject rates at all levels decreased and were lower (at 95% or better confidence level) for regular users versus others, which include both non-users and users that are not regular users. Those that never used the agent had even higher rejection rates.

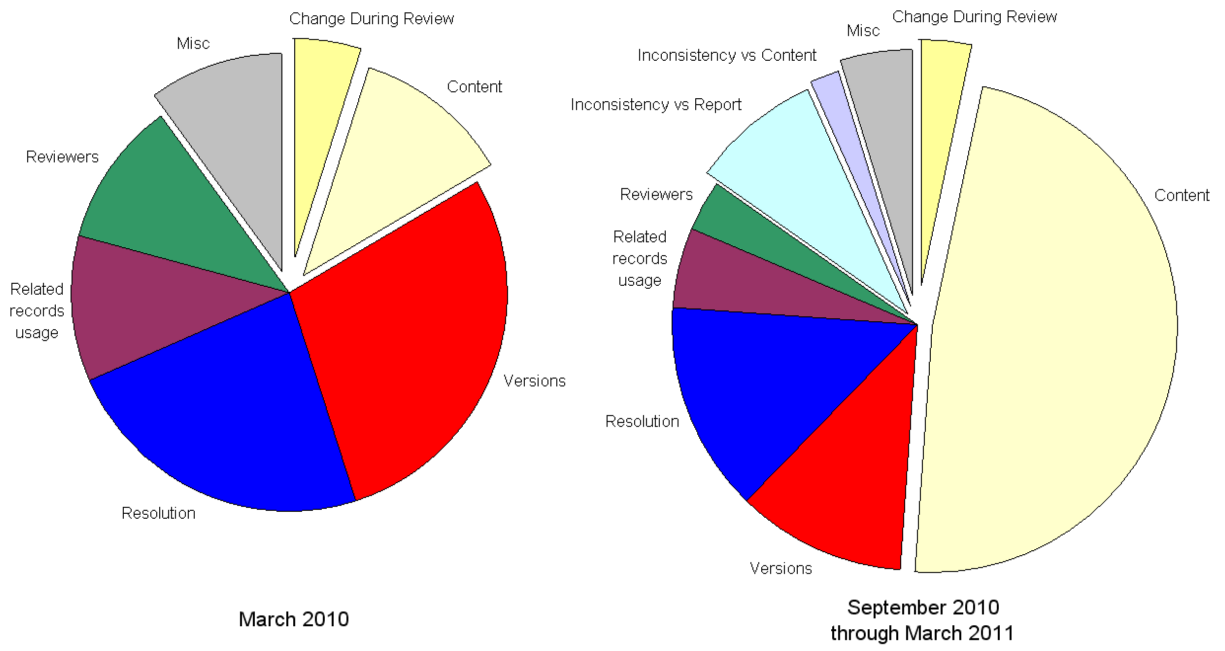


Figure 4: Rejections by classification before and after. Relative pie size reflects the total number of issues in the sample period.

Improvement in the Nature of Rejection

In addition to the reductions in their quantity, the nature of rejections has changed for the better. Based on formative classification of rejections sampled in March 2010, the agent was primarily designed to address the four major non-content sources of rejection that were identified as Versions, Resolutions, Related Records, and Reviewers. As seen in Figure 4, these sources of rejections shrunk significantly (both in number and as a percentage) in the sample from September 2010 through March 2011 (shown together in the bottom left side of the charts). So there was not only an overall reduction in the rate of rejection, but there was a greater reduction in the areas that the agent was designed to address. Note that the size of the pie charts reflects the total number of issues.

The rejection due to Change During Review (typically higher level design or scope change) dropped both in number and percentage, presumably due to easier, more rapid completion of issue reviews. Finally, the part that increased to become almost half of the rejections was that related to the content associated with the issues (typically product software and documentation).

Resulting Changes In Business Practice

Though more difficult to quantify, important positive changes were seen in business practice that were attributed by many involved to the positive effects of the agent.

The most clear and direct change was apparent in the use and users of the agent itself. Though purely optional, most of the software developers on the P and S teams were reported by the agents as users, many of them regular users. Several users emailed their managers and peers positive comments and recommendations about the tool such as: "I found the PnitAgent tool . . . 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. I also found it helpful in analyzing my own issues prior to routing them for review, which should help lower my rejection rate. . . "

For the teams where the agent was used most heavily, SQA engineers stopped reviewing the majority of issues. Directly citing dropping rejection rates and the positive effects of the agent software to project leadership, they praised the progress and indicated they no longer expected to be added as reviewers of most issues addressed near the end of 2010. This has lead to greater efficiency, improved quality, and according to at least one of the SQA engineers simplified analysis of the product for the purpose of regulatory review.

AI Technology and Related Work

The PnitAgent software meets many of the criteria and completely satisfies some of the notions set for agency in AI tradition and literature such as those put forth in (Wooldridge and Jennings 1995) and others. It is more specifically an "advisory agent" (Lieberman 1998), making it uniquely suited to operate and evolve in its heavily regulated environment semi-autonomously and free of stifling overhead. More fundamentally, it realizes AI and agent concepts and reflects the advantage of designing with these perspectives in mind.

Each agent acts to support its local user as an expert advisor analyzing, warning about, and helping to standardize issue tracking practices. Agents are designed to (and in some cases are allowed to) operate continuously for days or weeks (with, for example, close to tray and fade-away message behavoir, automatic re-login, and daily rebaselining and artifact caching), but may be stopped and started by the user at any point. At the same time, they retain information that shapes their activities regardless of how long their current incarnation has been running. Their support involves at times acting on the file system, retrieving data from various sources, starting and interacting with other programs and windows, displaying information to and soliciting information from users, and performing what might be called domain oriented reasoning, but never altering the issue tracking system or other product-related artifacts.

PnitAgent falls into "a class of systems that provide advanced UI functionality on top of existing applications" (Castelli et al. 2005). Relevant work in this area includes various attempts to support task modeling (Armentano and Amandi 2009), groupware procedures (Bergman et al. 2005), and automation after pattern recognition through programming by demonstration (Lau et al. 2004; Lieberman 1998). PnitAgent has also been shaped by concepts such as caching into a world model (Castelli et al. 2005); dynamic negotiation of protocol between agent and application (Lieberman 1998); how to adaptively monitor (Castelli et al. 2005); usability concerns such as graceful failure, partial automation, perceived value and other "critical barriers to widespread adoption" (Lau 2008), and work on focused crawling (Menczer, Pant, and Srinivasan 2004).

PnitAgent further involves AI-related technologies. These include association by demonstration of annotations, Markov modeling for learning and recommending next actions (Drew and Gini 2010), application-specific information extraction from free form text and conversational interaction with users (more details in (Drew and Gini 2012)). This work also draws on other parallels with AI ideas and challenges. For example it borrows from expert system design and knowledge representation the idea of using a separate and partially externalized rule base with rules, such as those on the roles of reviewers and how to classify requirements, provided in default reasoning/settings which the user (or team lead) may override or extend.

The agent is a culmination of research and ideas from AI and other communities, such as software engineering (Osterweil 1987; Storey et al. 2008) and business (Choo, A. et al. 2007), applied to software development issue tracking.

Conclusions

This work presented an agent, built on AI concepts and technology, designed as an advisor for, and adopted by, those involved in issue tracking activities that are part of software development for the medical device industry. As summarized previously, there is significant evidence of the broad and positive impact of the agent in improving the efficiency and quality of issue tracking activities that in turn enrich medical products. In the year after we completed the collection of data reported here, the agent has continued to be used successfully, in new ways, and by additional users. In addition to being suited for and achieving unprecedented success in this important, but narrowly defined new domain for AI, this work reflects an interesting and practical use of AI technologies and agent perspectives. This exemplifies a dynamic blend of coupling from automated and autonomous to interactive and tightly integrated. The agent was adopted, and found useful in a complex development environment. Several of those involved with this work commented that it helped the team to align on terminology and their broader understanding and execution of issue tracking activities.

References

Armentano, M. G., and Amandi, A. A. 2009. A framework for attaching personal assistants to existing applications. *Comput. Lang. Syst. Struct.* 35(4):448–463.

Begleiter, R.; El-Yaniv, R.; and Yona, G. 2004. On prediction using variable order Markov models. *Journal of Artificial Intelligence Research* 22:385–421.

Bergman, L.; Castelli, V.; Lau, T.; and Oblinger, D. 2005. DocWizards: a system for authoring follow-me documentation wizards. In *Proc. 18th Symp. on User Interface Software and Technology*, 191–200. ACM.

Castelli, V.; Bergman, L.; Lau, T.; and Oblinger, D. 2005. Layering advanced user interface functionalities onto existing applications. Technical Report RC23583, IBM.

Choo, A. et al. 2007. Method and context perspectives on learning and knowledge creation in quality management. *Journal of Operations Management* 25(4):918–931.

Cleary, J. G., and Witten, I. H. 1984. Data compression using adaptive coding and partial string matching. *IEEE Trans. on Communications* 32(4):396–402.

Drew, T., and Gini, M. 2010. MAITH: a meta-software agent for issue tracking help. In *Proc. 9th Int. Conf. on Autonomous Agents and Multiagent Systems*, 1755–1762.

Drew, T., and Gini, M. 2012. Automation for regulated issue tracking activities. Technical Report 12-010, Univ. of MN.

Lau, T.; Bergman, L.; Castelli, V.; and Oblinger, D. 2004. Sheepdog: learning procedures for technical support. In *Proc. 9th Int'l Conf. on Intelligent User Interfaces*, 109–116.

Lau, T. 2008. Why PBD systems fail: Lessons learned for usable AI. In *CHI 2008 Workshop on Usable AI*.

Leveson, N. G., and Turner, C. S. 1993. An investigation of the Therac-25 accidents. *Computer* 26(7):18–41.

Lieberman, H. 1998. Integrating user interface agents with conventional applications. In *Proc. 3rd Int'l Conf. on Intelligent User Interfaces*, 39–46. ACM.

Manning, C. D.; Raghavan, P.; and Schütze, H. 2008. *Introduction to Information Retrieval*. Cambridge Univ. Press.

Menczer, F.; Pant, G.; and Srinivasan, P. 2004. Topical web crawlers: Evaluating adaptive algorithms. *ACM Trans. Internet Technol.* 4(4):378–419.

Osterweil, L. 1987. Software processes are software too. In *Proc. 9th Int'l Conf. on Software Engineering*, 2–13. IEEE Computer Society Press.

Storey, M.; Ryall, J.; Bull, R. I.; Myers, D.; and Singer, J. 2008. TODO or to bug: exploring how task annotations play a role in the work practices of software developers. In *Proc. 30th Int'l Conf. on Software Engineering*, 251–260. ACM.

Wooldridge, M., and Jennings, N. R. 1995. Agent theories, architectures, and languages: A survey. In *Intelligent Agents*, volume 890/1995 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. 1–39.