Presentation, Proposal, and Paper Program Permissions Tool  
Team P5T

Richard Laughlin       Alex Kelley       Jacob Siegel       James Fitzgerald

Department of Veteran’s Affairs: Center of Excellence for Suicide Prevention  
Nathan Claes

Faculty Coach  
Tom Reichlmayr

Project Overview
The P5T system is a computer system for the US Department of Veteran Affairs Center of Excellence for Suicide Prevention’s research program. The goal of the system is to translate the current paper and email-based tracking process of research projects and grants into an online version. The system will allow users to submit their project applications online and track the status and progress throughout every step of the approval process. Once a project has been approved, the system will allow users to upload research publications and project documents. The finished system also will implement an automated web crawler with the ability to find and map additional, external resources for automatically populating and/or linking to existing information in the system.

Basic Requirements

User Analysis
There are three anticipated users of the system:

- **Researcher** - Frequent users. Not assumed to be technologically savvy. Use the system to either store or gather information related to the research projects they are working on.
- **Administrative Staff** - Infrequent users. Not assumed to be technologically savvy. Use the system to gather bulk amounts of data associated with projects or work on travel requests.
- **System Administrator** - Very infrequent users. Assumed to be somewhat technologically savvy.

Inputs
The vast majority of system inputs come from these three users via the web interface. The only remaining input is the gathering of content from the Crawler database to populate “value-added” web functionality, such as citation tracking.
**Outputs**
The system will output data to users via a standard TCP/HTTP connection with a web browser. The system will also output data to the MSSQL database to persist information which is entered by users.

**Requirements**
We created a comprehensive requirements document which lists the entire set of project requirements. The following is an excerpt from this document which lists the primary functionality of the system:

- **3.2 - The system shall be composed of the following logical components:**
  - **Affiliation**
    - Title
    - Phone Number
    - Email Address
  - **Author**
    - First Name
    - Last Name
    - Middle Initial
    - Affiliations with exactly one primary affiliation.
  - **Artifacts**
    - Have the following standard set of properties in addition to those listed below:
      - A text description (max 128 characters).
      - A collection of Attachments.
      - A collection of URLs.
    - Have the following types:
      - Research Article
      - Journal Article
      - Conference Presentation
    - Has the following additional properties:
      - Presentation title (max 50 characters)
      - Presentation type (max 64 characters)
      - Conference Name (max 128 characters)
      - Conference Location (max 128 characters)
      - Conference Date
    - Media Interview
    - Has the following additional properties:
      - Interview Topic (max 128 characters)
      - Interview Outlet (max 128 characters)
      - Interview Location (max 128 characters)
- Interview Date
- Grant
  - Has the following additional properties:
    - Grant Source (max 128 characters)
    - Grant Number (max 128 characters)
    - Grant Title (max 128 characters)
    - Grant Start Date
    - Grant End Date
  - Grant Status of the following types:
    - Pre-review
    - Requires Review
    - Reviewed
    - Submitted
    - Approved
    - Approved, with feedback
    - Rejected, with feedback
    - Rejected, without feedback
- Other
  - File
    - The data associated with the attachment.
    - A text description of no more than 128 characters.
  - Program
    - Name
    - Description
    - A collection of collaborating Researchers
    - A collection of watching Researchers (see 3.7 )
    - A collection of artifacts
    - An IRB Status of the following types:
      - Pre-review
      - Requires Review
      - Reviewed
      - Submitted
      - Approved
      - Approved, with feedback
      - Rejected, with feedback
      - Rejected, without feedback
  - Reminder
    - Description
    - Date and Time of Reminder
    - Priority of the following types:
      - High
      - Medium
Constraints
The constraints placed on us for the project were largely in terms of technology choices. We were required by the sponsor to use ASP.NET with MVC 3, MS SQL 2008 and we had to support IE 8. This limited us to just these technologies with some minor frontend additions but overall it did not place any unreasonable constraints on our design choices as the system itself was fairly straightforward.

A major constraint that we experienced that was unrelated to technology choices was that of finding sources for the crawler that we could legally scrape for data. Since much of the citation data is behind paywalls or is copyrighted, it was very difficult to find sources that would allow us to use their data both for free and without infringing on any copyrights held by specific authors or journals. This led to a choice to simply build a framework for the crawler and provide a single module for the one source we found that provided mostly adequate data and functionality.

We did run into issues with time constraints, but only in terms of sponsor availability and team availability. Sometimes it was difficult to complete tasks when there were other more concrete tasks that needed to be completed for another class.

Development Process
We used a combination of the Spiral model and Evolutionary Prototyping. This enabled us to determine next steps for each release based off of risk analysis and provide a working prototype to the project sponsor for user acceptance testing and evaluation. The process on an outward facing level (providing the prototypes to the sponsor) was approved by the sponsor, but details like how long each iteration was or when each release would be was up to us. As far as team roles are concerned, we picked up tasks as they were created based on what seemed to be the best fit for us. There were some minor recurring roles like managing documentation or keeping track of metrics but beyond that there were no formalized roles.
Project Schedule: Planned and Actual

Early on we identified that our project had two distinct sections: website and crawler. With the help of our sponsor we determined that we would build a solid foundation for the website and then begin to alternate between web releases and crawler releases. Our project schedule was devised individually for each release. To accomplish this, we broke the release tasks down into their “atomic” pieces and then created a weekly goal of these atomic pieces which we wanted to have completed. The following table illustrates the high level releases:

<table>
<thead>
<tr>
<th>Release Name</th>
<th>Start Date</th>
<th>End Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>R1 - Core Web Functionality</td>
<td>9/25</td>
<td>10/30</td>
</tr>
<tr>
<td>R2 - Crawler Design / Risk Mitigation</td>
<td>10/31</td>
<td>11/13</td>
</tr>
<tr>
<td>R3 - Web Functionality Refinement</td>
<td>11/14</td>
<td>12/11</td>
</tr>
<tr>
<td>R4 - Website Interface Refinement</td>
<td>1/27</td>
<td>2/12</td>
</tr>
<tr>
<td>R5 - Crawler Research &amp; Prototype</td>
<td>2/13</td>
<td>3/14</td>
</tr>
<tr>
<td>R6 - Website Final Features</td>
<td>3/15</td>
<td>4/27</td>
</tr>
<tr>
<td>R7 - Documentation &amp; Bug Fixes</td>
<td>4/28</td>
<td>5/21</td>
</tr>
</tbody>
</table>

Each release listed had its own set of tasks to be completed before the next release. For example here is the breakdown for Release 1:

<table>
<thead>
<tr>
<th>Task Name</th>
<th>Anticipated End</th>
<th>Actual End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Database Tables &amp; Domain Model Classes</td>
<td>10/7</td>
<td>10/13</td>
</tr>
<tr>
<td>HTML Layouts for CRUD &amp; “Glue” Pages</td>
<td>10/14</td>
<td>10/16</td>
</tr>
<tr>
<td>Form Usability &amp; Accessibility</td>
<td>10/16</td>
<td>10/18</td>
</tr>
<tr>
<td>Layout &amp; Functionality Refactoring</td>
<td>10/21</td>
<td>10/30</td>
</tr>
<tr>
<td>Cleanup &amp; Bug Fixes</td>
<td>10/30</td>
<td>N.A.</td>
</tr>
</tbody>
</table>

A lot of slippage occurred in R1, and subsequently happened in later releases as well. This slippage was due to a variety of factors:
● Underestimation of the difficulty of tasks.
● Overestimation of the time available to complete tasks.
● Unexpected assignments from other classes which interfere.
● Incomplete information which required feedback from the sponsor.

We eventually became better at estimation, but time was the most difficult factor to manage through the course of our project.

**System Design**

**Database**

Our database specification and design centered around the agreed-upon logical components of the system as described in the Requirements section. These components themselves are an extension of the artifacts and information currently held in the non-electronic system, and were simple enough to build.

Initially, each table was built one-to-one with the provided specifications, with link tables set up where appropriate. This has been maintained in the final design and implementation, with the addition of certain logging and security tables: namely, Activity and Notification tables for each of the different artifacts represented in the system. Whenever a user is to make a change that would affect the database in any way, a notification is created that links said user with a specific action in the activity table. This way, we are able to keep track of the actions of every user in the system and maintain an available log of their behavior.

**Main System (Web Application)**

Like all C# MVC3 systems the general flow of the application is structured according to the following diagram:
Incoming requests from users who are not validated are automatically filtered and redirected to the authentication page. All other requests pass through to the controller function which handles a particular action within the system. These actions may touch the database one or more times and prepare the data which will then be used to generate the HTML based view which is sent back to the client.

We have separated controller functionality into 11 different classes: Auth, Home, ResearchArticle, JournalArticle, Interview, ConferencePresentation, Grant, Other, Programs, Researcher, and Notifications. Auth handles all login and registration functionality. Research Article, Journal Article, Interview, Conference Presentation, Grant, and Other handle the functionality associated with each of these different artifacts. Programs handles the creation, update, and deletion of Programs. Researcher handles the creation, and update of Researcher. Finally Notifications handles the distribution and viewing of notifications tied to particular system actions taken.

After receiving a request, controller actions commonly interact with the database via Database Model classes. These classes provide simple Create, Read, Update, and Delete functionality for the associated database tables. Most importantly, these classes make use of the Lazy and LazyList class wrappers to simplify common actions and minimize the number of database calls for a given request.

Controllers then use a variety of cshtml views to render HTML for the users. Many of the controllers only respond to requests given to the server from AJAX requests. The results of these requests are then inserted into the content of modal windows which pop-over the rest of the page content on the client web browser.

The construction of the views uses a combination of basic HTML elements, bootstrap elements, and jQuery elements. The most interesting of these is the construction of the program view page. When a button is pressed an AJAX request is fired which fetches the content of a particular modal window. Part of the content returned is a javascript section which can override global function variables. Once the content of the sub-page is returned, the modalCallback() javascript function is executed which gives the sub-page the opportunity to execute necessary functions for setting up complex widgets such as date and time pickers. After the modalCallback is executed the any <form> elements contained within the sub-page are modified so that they also perform AJAX requests upon submission.

However, before the AJAX request is fired, an additional callback function, validationCallback(), is used to determine if the form on the page is ready for submission, or requires changes. This allows for the user to be given specific, instant, network-independent validation. Additional validation is performed on the server side to prevent issues or security vulnerabilities as a result of invalid inputs.
Crawler System
In design of this system, we had to consider the application of a web crawler that would be run either manually or semi-autonomously at certain scheduled times. Further, a necessary consideration was in the fluid nature of the web itself: Sites change over time, and no site is the same; thus, a generic crawler cannot be created. These two considerations had the largest impact on the design of the crawler.

Flow and System Interaction
Upon startup, the crawler reads in a root configuration file that describes the necessary class files which fulfill roles as configuration readers, database connectors, data miners, etc. Then a list of sub-configuration files are loaded using the requested configuration class. Afterward, a list of database connections are created using the requested database connector class. Finally, the data mining classes are loaded and given the ability to access configuration and database class instances.

After startup is complete, each data mining module is given access to the thread pool. It inserts tasks to the thread pool which are executed concurrently to other thread pool tasks. This allows multiple data mining modules to gather data from different sources simultaneously. Because the crawler is primarily IO bound this allows for massive performance gains.

Kernel and Threading
The Kernel class stores collections of all configuration, database, data mining, and thread pool instances. This allows data mining modules to access any data which was loaded via configuration files that they see fit to access during their operation. This is important because the interfaces for Configuration, Database, and Data Miner only supply the functions to determine the type of the object being accessed. This allows for the crawler system to be extended with additional configuration, database and data mining types in the future if it is necessary.

The thread pool used is very simple: It simply maintains a queue of tasks and executes them on a set number of threads which is configurable. Other possible thread pool options are possible, but this implementation is significantly simpler and does a good job. Because the crawler as a whole is IO bound a high number of threads should be made available in the thread pool to allow the crawler to “queue up” a large number of network and database IO requests simultaneously.

Data Mining Modules
As stated previously, no web site is the same. Thus, no crawler can be designed that is able to function across multiple pages in a generic faction. Because of this, crawlers must be specifically designed to certain website specifications and layouts -- but even then, the site may change in the future and break the crawler. Finally, due to various constraints and legality issues in pulling from scholarly databases, there are simply not that many sources we are able to use out the gate.
The design we decided upon to address these problems was actually a simple, modular approach. Rather than build a large system around crawling one or two specific sites, we developed a system that could take site-specific modules and run them as data miner threads. Each mining operation could run in conjunction, and we could run multiple crawlers as part of a singular system. This also allows for easy extension and modification, should future users decide that they want to use more sources or drop certain ones.

Through this smaller class diagram can be seen the setup for our data mining modules. The interface IDataMiner connects each module to our Kernel so that we can decide how it enters the thread pool and evaluates itself, while specific interfaces for the type of crawler exist to decide upon its functionality. IWebCrawler, for instance, denotes a module that is able to parse HTML documents using the public `HtmlAgilityPack` library.

The module that we decided to include as a functional example was one for PSU’s CiteSeerX public scholarly database. Initially, we wanted to use CSX because it implements the Open Archive Initiative’s method to programmatically pull metadata information from the site’s databases. However, in designing around the OAI, we discovered that CSX had disabled the ability to receive citation data over a year ago, with no current intention to bring it back. While we still wanted to use CSX, we could no longer use OAI. The only solution to this was to implement an HTML crawler for CSX.

Using the `HtmlAgilityPack`, developing a crawler was fairly simple. The library was based on the XmlDocument traversal classes already present in C# so all we had to design for was the specific structures in HTML that we wanted to look out for. To retrieve the URL for the citations page for an article, for example, we scan the document for instances of `"//a[@class='citation remove']"` and retrieve the href from the anchor. After this, it becomes even
easier: citation pages have XML-based RSS formats, so we then use the XmlDocument scanner to pull the information we require. After running out of citation sources, we would finally compile everything and insert it into the Crawler database.

This example module serves to inform future developers how to implement and use both the HtmlDocument scanning portion of the HtmlAgilityPack library, as well as XmlDocument scanning, and as such was designed only to run on document names specified when the constructor is called. It may be necessary to extend this design to work with author name and surname search, but that is left up to a future party.

**Process and Product Metrics**

The metrics we decided to employ for our project were as follows:

**Product Metrics**

- Comment Density
- Avg. LOC per feature
- Avg. Population(count of all classes and methods) per feature
- Total number of features
- Software Maturity Index(SMI = \([M_T - (F_a + F_c + F_d)]/M_T\))
  
  where
  
  \(M_T = \#\text{features in the current release}\)
  
  \(F_a = \#\text{features in the current release that have been added}\)
  
  \(F_c = \#\text{features in the current release that have been changed}\)
  
  \(F_d = \#\text{features from the preceding release that were deleted in the current release}\)

**Process Metrics**

- Avg. Time between cycles
- Avg. added features per release
- Avg. added LOC per release
- Avg. Comment density
- Delta Risk = (Risk Exposure - Avg Risk Exposure) / Avg Risk Exposure * 100
- Work Breakdown of time spent in Requirements, Design, Construction, Testing & Maintenance
- Avg. time spent per feature
- Delta SMI

We chose to use these product metrics because we believe that the results would give a pretty good idea of the size of our product, and some of its complexity. We believe that the process metrics we chose will be useful in illustrating how our process went throughout the lifetime of
our project and where our time was spent. The product metrics listed mainly pertain to the final release of our product as these numbers reflect the most accurate state of our product in its current state. The Process metrics use numbers throughout the whole life cycle of our project as this is the best way to document our process throughout the project. Regarding metrics that use lines of code (“LOC”), this only includes lines of C# in our project files, this does not include line of HTML or JavaScript in our views as we could not find a metric tool to help calculate these metrics. As a result of not being able to find a metric tool to help us parse our project and calculate LOC, we created our own tool using a python script. The script counts empty lines, comment lines (including multi line comments) and lines of source (these lines are considered a “Line of code”). Because we use a strict definition of what constitutes a “Line of code”, counting is consistent to give us the most accurate results.

Product Metric Results

The final comment density of the product was about 4.26%. This result is much lower than we were hoping to get in the beginning. This result reflects that we needed to use better commenting practices when documenting our code. The final average lines of code per feature of our final product was 363. Considering that the final feature count of our product was 32, we think that this result reasonable reflects the size of our product also taking into account that this does not include the code from our views. The average population of our features was 15. we think that this number accurately depicts the complexity of some of our features. Our final Software Maturity Index was 0.875. Because our index is below 1, this shows that our product is still not fully “stable”, however, this index has steadily increased throughout the lifetime of the project so we think that this is a reasonable result.

Process Metric Results

Starting with the average time between releases was around 3.75 weeks. We planned on making releases every 2 weeks, however due to decreased contact with our sponsor in the second semester and some technical issues between releases, we were either unable to complete some of our releases in this time frame, or we thought it unnecessary to have a release within that time frame due to lack of visible progress. That being said, though this number is higher than what were originally planning for, it makes sense that we got such a result. The average number of features added per release was 11. Though this number is technically accurate, we think that it is also a little misleading. This is because for our first release we implemented the core functionality for the system which contained the majority of our features, while each future release implemented considerably less features numerically, as they focused on more complex and robust features as well as changes to the existing system as a result of sponsor feedback. Our average for added lines of code per release at the conclusion of the project was 2,032. Like the last metric this number is also a bit misleading for the same reason, though to a lesser degree than the last metric as the features we implemented in later releases were more complex and more robust, there were just less of them. Our average comment density per release was 3.27%.
As stated above, we think that this number is too low and that we should have used better commenting practices when documenting our code.

Now regarding our time related process metrics, the average time it took to implement a feature was 3.67 hours. We think this is a pretty good representation considering hours spent on implementing features ranged from 1 hour to 15 hours, and that there were a fair amount of features that took around the average amount of time to implement. The average change in our Software Maturity Index over the lifetime of our project was around 68%, this shows that were were making steady progress in completing our project and constructing a stable system. Our breakdown in hours spent doing requirements, design, construction, and testing/maintenance ended up being: 14% for requirements, 6% for design, 70% construction, and 10% testing and maintenance. These results show that we spent considerable more time doing the construction of the project than we did any other part of the project.

Though these numbers may not be great, they do make sense to us because we did spend the majority of our time constructing the system. We spent about the first month eliciting our requirements from our sponsor to make sure that we knew what exactly we were supposed to be making, from there we designed our system which did not take as long as requirements because the overall system is not very complicated and we had a good grasp on what the system should do because of our thoroughness with requirements elicitation. these numbers also show that we were a bit lackluster in our testing, which we know is true as our main form of testing was user testing from our sponsor, as mentioned in our final presentation.

Regarding our risk exposure, from the beginning of the project, our risk exposure decreased from 16.01 to 11.38. Though this does not seem like a very big change, we did mitigate a fair amount of our risks, however, the risks we could not mitigate were weighted heavier than those we mitigated. It is unfortunate as well because a couple of the high risk issues we could not mitigate ended up affecting us. For example, our highest rated risk(regarding being able to provide sufficient info for our web crawler) was rated at a 4.5 and though we acted tentatively to mitigate the risk by starting planning, research, and development of our crawler early, we could not mitigate the risk because through our research we found legal issues with collecting the data we were supposed to supply. this resulted in us having to redesign our crawler and redefine its scope.

**Metric Interpretation**

The metrics tell us that we spent the majority of our time in development and that our release schedule was more spread out than we planned in terms of release frequency. The results also point out that we needed to implement better code documentation practices as our comment density was consistently low. Also, that we did not spend as much time testing as we could and probably should have. The results do however showed that we were consistently making progress with our system development as our SMI continued to increase.
Product State at Time of Delivery
At the time of delivery our project is mostly completed but functional. Users can perform most of the operations requested by the sponsor, but there are a few important features that are missing and a few UI improvements to be made. This is a good foundation for another team to build upon to bring the project to a completion.

Most importantly is the status of the crawler. As mentioned before, we ran into trouble in finding sources for the crawler so we provided the architectural framework to add more source modules when suitable sources are found. We also provided a module for one source so that future developers have an example module to be followed when building new source modules.

Overall the project is completed enough to be useful for the sponsor, albeit without some of the nice features that would make it exceptionally useful. A future team should be able to complete the remaining features and polish the project into a complete state in a relatively short period of time.

Project Reflection

Process
Our choice of spiral model helped us throughout the course of the project. We were able to identify key risks early and help limit their impact on the project as it progressed. Our planning of releases was flexible and responsive to changes in plan or requirements. However, it forced us to create arbitrary deadlines which were often not realistic in terms of the amount of work we would be able to complete through the course of the release cycle. Once one deadline slipped the rest were bound to slip in a cascade.

During the second term this effect was compounded by a sponsor who was often distant. Our deadlines seemed less rigid because the sponsor would not see the results of the release immediately anyway. This isn't to say it is our sponsor's fault - he has important work to do - but it is an effect on our outlook that we did not notice until it was too far into the project to correct effectively. It was just far too easy to give a release a few extra days instead of doing the proper rescheduling.

If we were to do it again we would definitely do less detailed planning of individual features per release. This would lessen the psychological impact of letting some inevitable slippage take place because high level features are seem to be just taking a longer time to complete than initially expected, but we're not “behind”. As a result our deadlines would have not become increasingly less rigid over time.

Product
The website portion of our product is quite far along in its construction. There still remains a small number of features which require implementation, but overall the website is completed to a point at which it may be useful for the sponsor to use in some capacity to manage
their artifacts. The only features for the website portion that are not completed are related to travel requests, and editing of researchers.

The crawler implementation is not quite as advanced. The CiteSeerX module works as intended, but it is not integrated with a database schema and the data gathered is not currently used by the website portion of the project. However, the work required to change this is minimal due to the crawler architecture.

References