A Test-Driven Approach for Extracting Libraries of Reusable Components from Existing Applications

Elaf Selim, Yaser Ghanam, Chris Burns, Teddy Seyed, Frank Maurer

Department of Computer Science, University of Calgary
2500 University Dr. NW, Calgary, AB, Canada, T2N 1N4
{esselim, yghanam, ccburns, aseyed, fmaurer}@ucalgary.ca

Abstract. In agile approaches such as Extreme Programming, time is not spent on making sure that system components can be reused in similar systems. Therefore, there is a need to investigate whether reuse can be achieved by extracting reusable assets from existing applications. This paper presents an approach that relies on refactoring and testing practices for extracting reusable assets from existing applications. The approach creates reusable APIs in a bottom-up fashion, on demand when a new application might benefit from component in an existing application. The extraction process is guided and supported by the usage examples and the testing scenarios in the existing application and the new one. The paper presents a case study, where the approach was used to extract components from the user interface of an existing application, wrap these components in an API, and use this API in the existing and new applications.

Keywords: Extracting Reusable Components, APIs, Libraries, Test-driven Refactoring, Acceptance Tests, GIS, Multi-Touch Surfaces

1 Introduction

The lightweight software engineering practices of extreme programming (XP) were adopted to help develop software applications more quickly and so that changes during the entire software development life cycle could be accommodated more easily. XP and agile software engineering in general have become popular [5]. In XP, systems may be created using a "design-by-reuse" process by tailoring components previously extracted and added to a repository [8]. However, being committed to developing products quickly, time is not spent in a preliminary phase to design and create reusable APIs which can benefit future similar projects. "YAGNI – You Ain't Gonna Need It" is the battle cry in XP arguing against attempts to predict what might be needed in the future. In XP, software systems are developed without doing extensive research in the beginning or spending much time on requirement gathering [2]. These systems are developed simply to satisfy the current customer and what this customer really wants [16] without thinking much about the needs of future customers or how the systems can be reused.

Software reuse, on the other hand, promises to offer a number of advantages, and may require explicit support in the development process [16]. Despite the fact that reuse can reduce the development cost of new applications and also reduce the maintenance cost of these applications, XP does not aim at supporting developing software for possible future reuse or the creation of reusable components to be exploited by other applications instead of creating these applications from scratch.

In our case, our team had developed an application to support the control room operation of an electricity utility with Geographic Information System (GIS) maps on multi-touch digital surfaces. However, other applications are also in need of similar functionalities. Example applications include command and control, traffic management, analysis of agricultural and geological data…etc. Thus, our first application could provide components for similar applications designed to interact with GIS data on multi-touch devices. We found that a number of user interface elements in our application are general enough to benefit similar GIS multi-touch applications. The value of the effort spent in designing these components can be preserved by extracting them and collecting them in an API library which can support several other applications in other domains. When we started to develop a second application that supports visualizing and interacting with oil well GIS sensor data, we were motivated to start thinking about the best approach to extract this reusable API. The remainder of this paper describes a bottom-up, test-driven approach that we followed for extracting reusable APIs from the first application to support the development of the second software system.

In this paper, we propose and evaluate a systematic approach for extracting libraries of reusable components from an application and use them in new software products. The code around these components in the original application is changed as well to use the newly extracted and modified components. The approach relies on acceptance or functional tests which can help the refactoring process by planning the changes needed to be done and making sure that the resulting products match the requirements of the customer and behave as expected. Usage examples are also extracted from the original application since they help identify the appropriate use of the extracted components [23].

The remaining of this paper is structured as follows. The following section discusses some of the related work. Section 3 presents the extraction approach. In Section 4, the paper presents a preliminary case study carried out to assess the feasibility and usefulness of this approach followed by a discussion of the results in Section 5.

2 Related Work

There is a large body of literature on the concept of reuse, its approaches and techniques. Reuse approaches are commonly categorized under planned reuse [22] or opportunistic reuse [19]. Generally, planned reuse refers to the proactive treatment of reuse wherein the organization dedicates upfront planning activities specifically to plan for reuse - the software process defines what assets are to be reused and how to adapt them [22]. On the other hand, opportunistic reuse refers to the reactive treatment of reuse that happens only when the opportunity for reuse avails itself [19]. The idea we promote in this paper is an opportunistic approach in the sense that we deal with reuse only when there is a demand to provide a reusable asset to be used across a set of applications. Some work in the literature addresses the development of reusable assets as a proactive activity in which reusable asset are planned for upfront [7, 18]. Other approaches – like the one we propose in this paper – are extractive approaches that attempt to identify potentially reusable assets in existing applications and extract those assets [6, 4]. For example, Lanubile et al. [12] applied a program decomposition method to the problem of extracting reusable functions from ill-structured programs. Another approach was proposed by Ning et al. [17] which also relied on segmenting the programs into manageable pieces before the extraction process. While these methods are fundamentally different from our approach, which will be explained later, we use the same two general steps of focusing and then extracting, but we achieve the needed focus using user stories and tests.

Within the context of agile software development [13], Sugumaran et al. [21] proposed the construction of a knowledge-based framework to enable agile teams make better decisions when selecting and customizing software components for reuse. The approach, however, focuses only on how to use such a framework in an agile context but it is not clear on how to build the proposed framework within an agile context. In XP, refactoring is an essential part of the development cycle [11]. In the approach we propose, we rely on refactoring as a key activity to enhance the reusability of assets [15]. Moser et al. [16] conducted a study to assess if refactoring in agile environments improves the quality and reusability of – otherwise hard to reuse – classes. Their results support the hypothesis that continuous refactoring improves quality metrics thus promoting ad-hoc reuse of object-oriented classes. Washizaki et al. [23, 24] proposed a refactoring approach for extracting candidate reusable classes from object oriented programs and modifying the surrounding parts of the extracted parts in the original programs.

Our effort is different from the abovementioned efforts in that it takes into consideration the new user stories in order to enhance the design or/and extend the implementation of the extracted asset. The approach also leverages tests as a focusing mechanism and as a safety net. We also show how APIs can be built to enable the reuse of such components guided by the original usage instances as examples.

3 Extraction Approach

Say we have a system named S_o that has already been developed to satisfy the requirements of a given customer. S_o has been developed based on a set of user stories

US_o. These user stories are translated to a set of acceptance tests AT_o. The tests interact – through a thin layer of code (aka. fixtures) – with the code units that compose the system.

The approach we suggest has the objective of maximizing reuse of the UI widgets defined in S_o to be utilized in a new application S_n such that:

(a) All ATs are passing for the system S_o. That is:
$R(AT_i) = \text{Pass} \ \forall \ AT_i \in AT_0$ (condition C_1)
where R is the result of running an AT against the system

C_1 should be maintained to be true throughout the whole process. This serves as a safety net for the old system.

(b) The new system S_n is described using the set US_n that translates to a set AT_n. The underlying functionality in S_n is initially missing (i.e. classes and methods are empty), therefore:
$R(AT_j) = \text{Fail} \ \forall \ AT_j \in AT_n$
where R is the result of running an AT against the system

This verdict is changed to become false given that a non-empty subset of AT_n will pass by reusing the code units. That is:

$R(AT_j) = \text{Pass} \ \forall \ AT_j \in AT`_n$ given $AT`_n \subseteq AT_n$ and $AT`_n \neq \emptyset$ (condition C_2)
where R is the result of running an AT against the system

We achieve this by executing the following procedure:

1. Write the ATs for S_n to verify the satisfaction of US_n.
2. Manually compare US_o with US_n to find potential reuse opportunities. Say $US_a \in US_o$ has found to be similar to $US_b \in$ to US_n.
3. Compare AT_a (that tests US_a) with AT_b (that tests US_b). This comparison yields two important pieces of information:
    a. What is common between AT_a and AT_b, since the potential API need not provide interfaces to configure common features.
    b. What is variable between AT_a and AT_b, since the potential API needs to provide a means of configuring the reused artifacts to satisfy this variability.
4. Refactor the artifacts in S_o (mainly code and possibly ATs) that are relevant to the code unit of interest based on the outcome of step 3.
5. Separate the refactored artifact into an API library. The objective of this step is to enable reuse of a single source code and avoid having redundant code in the two applications.
6. Refactor the code in S_o so that it utilizes the new design of the code (observe C_1)

7. Use the API library in the implementation of S_n (observe C_2)

To describe the extraction approach and demonstrate how it is executed, the following section presents a preliminary case study on the analysis process and the extraction steps involving two systems we built in-house.

4 Case Study

4.1 Application Context

Utilities which are heavily reliant on geospatial data have been working with paper maps unfolded on tables for a long time. Using these traditional methods causes data management tasks to be asynchronous and very tedious. Delays in updating these maps pose a safety risk for field workers that may be guided based on outdated information. More recently, utility companies are starting to adopt Geographic Information Systems technology. A GIS is used for capturing, storing, managing, analyzing, and displaying all forms of geographically referenced information. Some electrical utilities now have their own GIS data which include details of their circuits, power-lines, switches, etc. Despite their use of GIS servers to store geospatial data and GIS software tools to analyze it, some teams in the control centers of electricity companies still prefer to use large printed paper maps to assist in data management tasks such as making changes to the data and discussing and analyzing snapshots of the data to find solutions for problems. Most GIS applications are focused on supporting a single user and these teams typically work in a collaborative mode, which is why they tend to gravitate towards using large paper maps. Paper maps on a table are convenient when multiple team members are working together to achieve a certain goal since they are large enough so that everyone can see the information comfortably, subgroups can work on different maps concurrently and everyone has concurrent access to editing these maps using pens and markers.

Using the technology of multi-touch digital tables can serve as a suitable alternative for using paper maps on regular tables. They can be used for displaying and managing these large digital maps for collaborative purposes. Since interactive digital surfaces combine both the input and visual output spaces, they have a number of potential benefits. Tabletop surfaces accept a number of new types of input, including using hand gestures and tangible objects. They encourage people to work collaboratively in co-located groups [20] and the large surface area is very convenient for displaying and interacting with information. The idea behind developing the first application; eGrid, was to provide an innovative and convenient digital tabletop environment to address the needs of co-located control center teams in electricity companies. By using eGrid, teams are able to browse through, analyze and annotate GIS maps, facilitating collaboration without the need to print the maps on paper. The second application, eWell, visualizes oil well sensor data which is captured and stored in GIS form. Starting the development of this second application motivated the extraction of the API from the first application.

4.2 The First Application: eGrid

The requirements behind eGrid reflect the needs of a local electricity company. The application was designed and developed using a simple, lightweight extreme programming process which relied mainly on effective feedback for iterative product improvement. The process started by creating low fidelity prototypes such as sketches and simple demos, since "a picture is worth a thousand words" [14]. Specifying the requirements of the application was done in a number of simple and informal ways including sketches, user story lists and interview notes. Whiteboards were usually used to help focus attention and promote collaboration in the design process. [3] The demos were useful in brainstorming interface ideas but they actually did not capture the design of the interactions in this gesture-oriented application, which is why these demos were of little use in getting valuable feedback from users.

eGrid is a very user interface intensive application. It sits on top of ESRI ArcGIS API for WPF 4.0 [1]. It is developed as a multi-touch user interface for a horizontal digital table with no specific orientation. The design of the user interface elements, touch events and hand gestures were done with this target hardware platform in mind. To capture the interactivity and multi-touch capabilities of the application, more advanced prototypes were developed next; allowing eGrid to gain increased customer interest. The prototypes were developed in multiple iterations guided by frequent demonstrations to industrial partners and other interested GIS professionals. The interface and features of eGrid have been changing continuously throughout these iterations to respond to the feedback received and to add more requested features. Training and observation sessions in the control center of the electricity company were also conducted to understand more about the actual environment in which the application will be deployed. In addition, several interviews with GIS professionals were also conducted to further understand the problem domain, the extra features needed in eGrid and the general features in eGrid that can be useful to other tabletop GIS applications.

In the process of analyzing the design of eGrid and brainstorming the extraction of reusable components, two important questions had to be in mind:

(1) Which of these components are general enough to be useful in similar GIS applications?
(2) How can the design of these user interface elements be flexible enough to accommodate different application needs?

The following sub-sections describe briefly some of the components in the UI of eGrid which will be extracted and added to the API and the reason behind certain design decisions.

Background Map. This map is displayed in the background to help users keep track of trouble report locations and provide an overview on the state of the electrical grid of the city. The location envelope, i.e. the coordinates of the map, the layers, as well as the ability to interact with the map by panning and zooming will have to be changed through options.

Corner Menus. To access common application tasks which are not related to any specific map frame, an alternative was needed for the ordinary menu designs of vertical screen applications. Since the target platform is a horizontal surface, for which there is no fixed orientation, quarter pie corner menus were added to the interface. This design consumes a small area of the screen and is accessible by any user no matter where he/she is standing. When the user taps on any of the corner menus, it expands to a slightly larger quarter circle menu showing a group of icons for different functionalities.

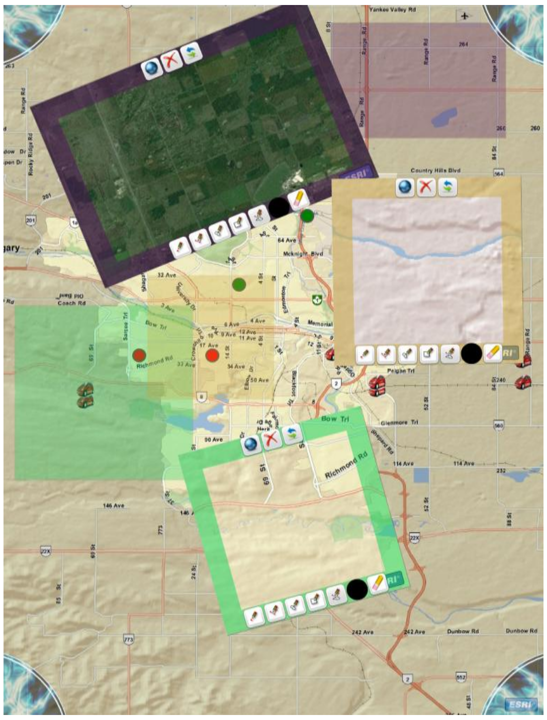

Fig. 1. An overall screen shot of eGrid with three map frames opened. The background map appears with trouble report pins.

Trouble Report Pins. Trouble reports define service outage locations and the details of the outage situation. Trouble reports are geo-referenced; each report has an exact map location and they have different severities and/or priorities similar to IT trouble tickets. The design of eGrid represents these reports as circular pins placed in their locations on the background map. Fig 1 is a snapshot of eGrid with some map frames opened. On the background map, trouble report pins are scattered in their correct locations.

Map Frames. Giving users the flexibility to work on multiple different maps at the same time or work on one big map was motivated by a number of factors. This design tries to mimic the environment of the control center in which users have the flexibility to work on multiple printed maps at the same time. In addition, collaborating teams often toggle between different modes of collaboration and this design is flexible and avoids imposing any specific mode on the users. For achieving this flexibility, individual map frames were designed. Each map frame has its own layers settings. The design also allows multiple users to interact with multiple maps at the same time. A map frame can be created in a number of different ways. One option is to use a gesture composed of two consecutive finger touches that define the corners of the rectangular part of the background map which will appear in the map frame. Another option is to use a lasso gesture to define this area. Inside the frame, the map itself can be navigated using hand gestures. The design of these navigation gestures is based on the work of Wobbrock el al in their paper about user defined gestures for surface computing. [12]

4.3 The Second Application: eWell

The second application eWell is an application for the oil and gas industry. Its main target is to visualize oil well sensor data which is captured and stored in GIS form. The requirements of this second application motivated the extraction of the API from eGrid. A certain question seemed fundamental and needed an answer: which features/user scenarios fall in the intersection of the requirements of eGrid and the requirements of eWell? Or which components of eGrid should be separated into the API to benefit eWell and other applications? We found that eGrid in its current state has a lot of user interface elements which can benefit all tabletop GIS applications regardless of the domain. By trying to answer this question and based on the user stories of eGrid and the user stories of eWell, we created a list of user stories for the API user (the developer of GIS applications on multi-touch tables). The user stories also helped in brainstorming the extraction approach chosen and the granularity of the interface functions exposed.

After implementing the API and changing eGrid into an example application built on top of the API, the next question would be: which new features are needed by eWell which were not part of eGrid and thus are not yet supported by the API? And which of these features are specific to eWell and which of them can be useful for other applications? An iterative pattern is revealed here in the approach used, where new applications built on top of the API participate in enriching the API while having their own flavours at the same time. In our case, for example, for Corner Menus to be reused, the API should provide the application developer with the flexibility of changing the contents of these menus. Another example is the concept of marking some locations on the map and having more information associated with each location. This notion can be very useful in other applications as well. Therefore, trouble report pins and their associated information are generalized in the API as bookmark locations and associated notes or comments.

After a few iterations of building products on top of the API, users of the API should be able to select from various options and tailor the components to meet their specific needs.

Our case study includes the process of extracting the user interface components of eGrid which can be reused in similar applications, creating an API to include these components and using this API in both eGrid and a new application eWell. The approach presented learns from the usage examples in eGrid to guide the extraction of the API and relies on the acceptance tests of eGrid and also new acceptance tests that capture the user stories of the second application eWell. Acceptance tests are used to determine the components which can be reused in eWell and the extra features and changes that need to be done to these components. The acceptance tests are also used as a safety net in order to make sure that the refactoring steps will not change the behaviour of the eGrid and also serve the user stories of the new application eWell. The approach is iterative such that the requirements from any other new application interested in using the API can change the design of the API and add more options or variability points to the components extracted.

4.4 Example case study

This section includes an example of applying the extraction approach on the Map Frame widget from eGrid and reusing it in eWell.

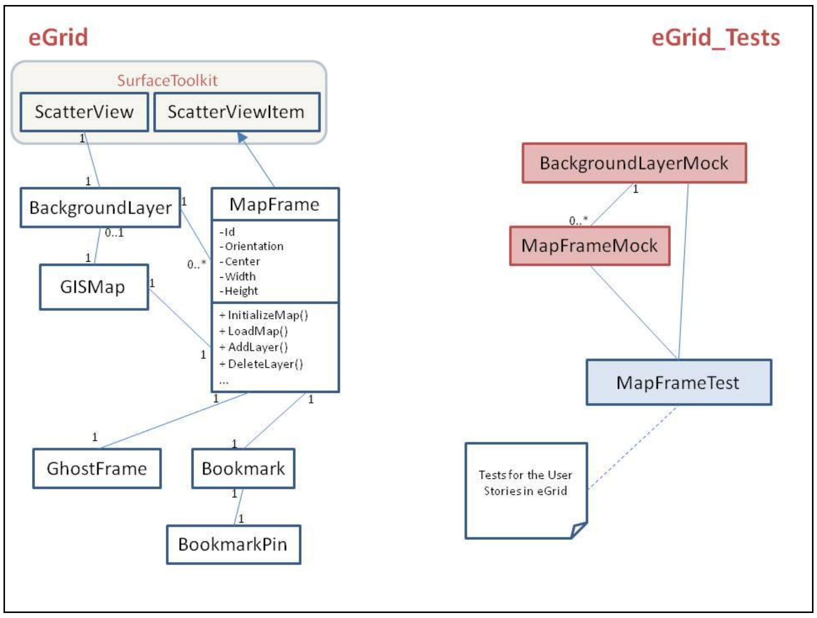

Fig. 2. A class diagram explaining the state of the design of eGrid before any extraction is done.

Fig 2 is a diagram representing part of the design in eGrid before any extraction is done. All the classes are in the user interface layer of eGrid. The implementation of the interface in eGrid was done in WPF 4.0 mostly in XAML files. Since part of the implementation was done using a markup language (i.e. XAML not C# code), it was difficult to implement the acceptance tests without creating mock versions of these classes. Mock classes have no XAML portions and they have mock implementations for the objects defined in the XAML files. The purpose of having these mock classes is to help create acceptance tests for UI code. The functionality is nearly perfectly preserved between the mock under test and the original user control. They are created only for the purpose of facilitating the creation of tests and they have to be in synchronization with the original classes at all times. The following steps are the same as the steps described in extraction approach section and will be applied iteratively on each component to be extracted.

1. Writing Acceptance Tests for eWell.
This step includes making sure that the acceptance tests of eGrid and eWell cover the scenarios related to the map frame widget. From the user stories of eWell, acceptance tests were created. Since there are no implemented classes for eWell, simple mock classes were created with no logic in them so that they can be attached to the tests. The created tests compile but fail against the incomplete mock classes. Fig 3 explains how the acceptance tests of eGrid and eWell interact with the class and mock classes in the implementation of eGrid.

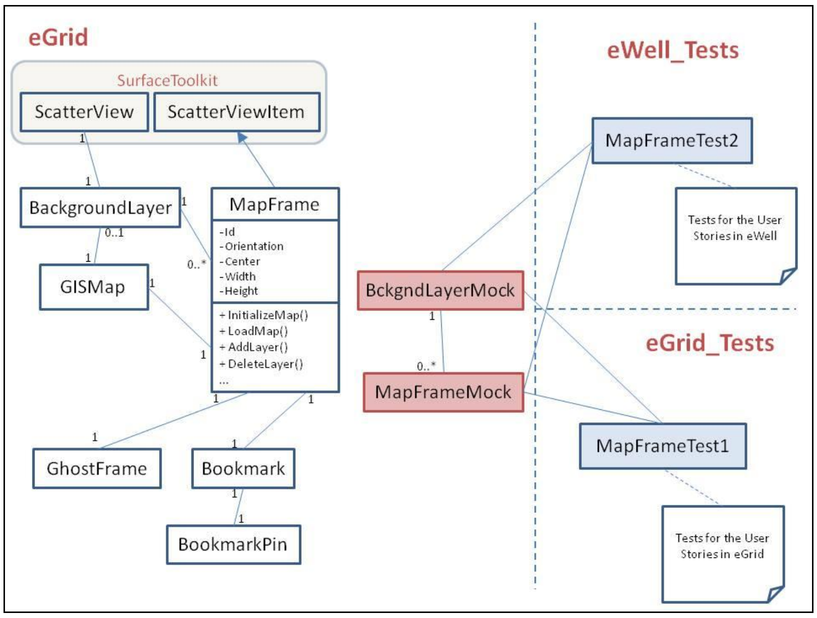

Fig. 3. A class diagram explaining how the tests of eGrid and eWell reference the classes in the implementation of eGrid.

2. Comparing the User Stories of eGrid and eWell.

By comparing the user stories of eGrid and eWell concerning the map frame widget, reuse opportunities can be identified. In eGrid, the main user story which relates to the map frame is: “As a user I can open a map frame using a two-point gesture. The map frame is positioned on the screen such that the two touch points are on top of two frame corners and the map displayed inside of the map frame has the same coordinates as the map area underneath the map frame”. In eWell, the user story is: “As a user I can create a map frame using a single touch point. The map frame is positioned on the screen such that its center point is underneath the touch point and the map displayed inside of the map frame has the same coordinates as the map area underneath the map frame”. In addition, another eWell user story is also related to the map frame widget: “As a user I can define a different default layer to be loaded into the map frame when it is first created”.

3. Analyzing the Acceptance Tests of eGrid and eWell.

The next step is to determine the changes to be done to the map frame widget based on the acceptance tests of eGrid and the new tests of eWell. Comparing the acceptance tests resulted in understanding the features which the map frame widget should provide. In addition, we also understood the options or variability points that the interface should provide to accommodate the differences between the scenarios in eGrid and in eWell. For example, after separating the map frame widget, changes will have to be done to handle the distinction between one and two map point instantiation methods and to handle the ability to change the default map layer.

4. Refactoring the Map Frame Widget Based on the Analysis.

After analyzing the common features and the variability between the use of the map frame in eGrid and eWell, changes have to be done to the implementation of the widget. For example, to handle two different methods for creating the map frame, we defined multiple constructors in the refactored map frame class and we added a new property to handle the URL address for the map when created.

5. Moving the Map Frame Widget to the API Layer.

In this step, the refactored map frame classes were moved into the new API layer. eGrid will use the map frame widget from the API layer just like it uses other third party components. eWell, as well as any other similar application, can use these classes from the API layer. The extraction of the component and the refactoring changes needed are not complete unless the acceptance tests in eGrid and eWell pass. These tests were combined to run against the new extracted component in the API after slightly modifying the implementation to run against the refactored map frame. The scenarios tested are semantically identical to the scenarios captured by the original acceptance tests from both applications.

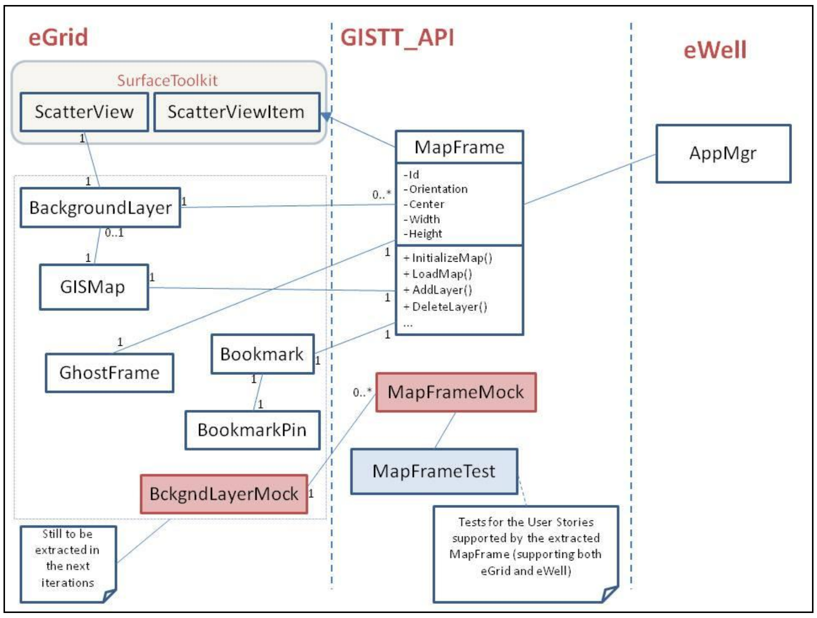

Fig. 4. A diagram explaining how the final state after extracting the MapFrame component and adding it to the API.

6. Refactoring eGrid Artifacts which Interact with the Map Frame.
Since the constructors of the map frame have changed to accommodate the extra new scenarios, changes will have to be done to all the code pieces in eGrid which interfaced with the map frame widget. For example, changing the constructors of the map frame necessitates changes in the code parts of eGrid which used these constructors. Other changes also needed to be made in order to make sure that the map frame classes absolutely encapsulate the behavior of the widget and to make sure that the interface is as simple and intuitive as possible. Needless to say that the acceptance tests of eGrid have to pass in order to make sure that the behavior has not been changed.

7. Using the Extracted Map Frame in eWell.
In this step, the extracted component will be used in the new application. The usage examples found in eGrid will make it simple to use the component in eWell as well. Fig 4 explains how the MapFrame class and its associated Mock class and test is moved into an API layer and used from there in both eGrid and eWell.

5 Discussion

The proposed approach for extracting components and creating API libraries offers a number of potential benefits. First, the approach aligns better with the iterative and incremental nature of XP since it handles components on-demand, one at a time – as opposed to developing all the components in one phase. Moreover, the approach supports reuse reactively which minimizes investment upfront, and guarantees that component extraction happens only when there is a real need for a component's functionality by other applications. The opposite would be a proactive treatment where all the possibilities of reuse have to be examined and handled in the beginning facing the risk of losing this investment if the components were not actually used in other applications. Furthermore, this approach makes use of the usage examples and the acceptance tests, which are existing assets in the original application, to inform the decisions made about exposing and hiding certain features in the API.

A challenge exists, however, when the reusable components that are to be extracted are mostly user interface components. This is due to the fact that testing user interface components is sometimes tricky especially if they are designed using scripting or mark-up languages. In particular, creating acceptance tests for user interface related stories is far from trivial [10]. Usually, in user-interface intensive applications, tests do not cover the entire portion of the user interface layer which makes refactoring code in this layer risky. A partial solution has been used in this case study where mock classes have been created for user interface classes which have scripting parts. The purpose of having these mock classes is to help create acceptance tests for the user interface classes. This served as a solution for creating acceptance tests for user interface classes, however, these mock classes have to be synchronized with the original classes whenever any changes are made to them. Although we emphasized the use of acceptance tests as a safety net throughout the whole process, the proposed approach can leverage any type of tests (e.g. unit tests) to do the job. The most critical aspect is that these tests need to be automated so that the abstraction process could be done with minimal risk.

Our next step is to repeat the process by developing other applications based on the new API using the same approach. The benefit of repeating the process is to further assess the approach of using acceptance test driven extraction of user interface APIs, as well as enhance the API by generalizing it more according to the options needed by the new applications. Usability studies of the API will follow to assess how effective, valuable and useful the API is for developing GIS tabletop systems and thus how useful is the approach used in designing the API and extracting it from the scenarios of other applications.

6 Conclusions

In XP, little time is spent in gathering requirements and doing upfront design to facilitate the creation of reusable components. Refactoring can be used to extract the reusable components from existing applications. This paper presents an approach for structured refactoring to extract reusable user interface components guided and supported by the acceptance testing framework and the usage examples found in the original application.

The contribution of this research paper is twofold. First, we proposed an iterative technique for extracting reusable components and usage examples from existing applications guided by their acceptance tests. Second, we presented a case study in which we apply this technique to extract components from an existing application based on the requirements of a new application and try to assess its value and usefulness. The main conclusion of this research can be summarized as follows: extracting reusable components from existing applications can be done in an XP environment using a systematic iterative test-driven approach for refactoring. The usefulness and limitations of this approach will further be assessed through more case studies in the future.

Acknowledgments. We would like to extend our thanks to TRLabs and NSERC SurfNet for supporting the projects of eGrid, eWell and the API.

References

1. ArcGIS API for WPF and Silverlight can be found at: http://resources.esri.com/arcgisserver/apis/silverlight/. Last accessed on 29th Nov 2010.
2. Beck, K.: Extreme Programming Explained: Embrace Change. Addison-Wesley (2000).
3. Brown, J., Lindgaard, G., Biddle, R.: Stories, Sketches, and Lists: Developers and Interaction Designers Interacting Through Artefacts. In Proc. of the Agile Conference 2008, Toronto, Canada, 39-50. (2008).
4. Burd, E., Munro, M., and Wezeman, C., Extracting reusable modules from legacy code: considering the issues of module granularity, *Proceedings of the Third Working Conference on Reverse Engineering, 1996,* pp.189-196.
5. Cao, L., Mohan, K., Xu, P., Ramesh, B.: How Extreme Does Extreme Programming Have to Be? Adapting XP Practices to Large-Scale Projects, Hawaii International Conference on System Sciences, p. 30083c, Proceedings of the 37th Annual Hawaii International Conference on System Sciences (HICSS'04) - Track 3, (2004).
6. Charles W. Krueger. 2006. New methods in software product line practice. *Commun. ACM* 49, 12 (December 2006), 37-40.
7. Clements, P., and Northrop, L.: Software Product Lines: Practice and Patterns, Addison-Wesley, US. (2001).
8. De Antonellis, V., Castano, S., Vandoni, L.: Building reusable components through project evolution analysis, Information Systems, Volume 19, Issue 3, April 1994, Pages 259-274, (1994).
9. Evoluce Table Technology available at: http://www.evoluce.com/en/ Last accessed 18th November 2010.
10. Finsterwalder, M.: Automating acceptance tests for GUI applications in an extreme programming environment. In Proceedings of the 2nd International Conference on eXtreme Programming and Flexible Processes in Software Engineering (Villasimius, Sardinia, Italy). Addison-Wesley. 114--117. (2001).

11. Fowler, M., *Refactoring: Improving the Design of Existing Code*, 1999, Addison-Wesley, Longman Publishing Co., Inc., Boston, MA, USA.
12. Lanubile, F., and Visaggio, G., Extracting reusable functions by flow graph based program slicing, *IEEE Transactions on Software Engineering,* 23(4), pp.246-259.
13. Manifesto for Agile Software Development, http://www.agilemanifesto.org, last accessed on December 13, 2010.
14. Memmel, T. and Reiterer, H.: Model-Based and Prototyping-Driven User Interface Specification to Support Collaboration and Creativity, in Journal of Universal Computer Science, 14 (19), (2008).
15. Mens, T., and Tourwe, T., A survey of software refactoring, *IEEE Transactions on Software Engineering*, 2004, 30(2), pp. 126-139.
16. Moser, R., Sillitti, A., Abrahamsson, P., Succi, G.: Does refactoring improve reusability?, Ninth International Conference on Software Reuse (ICSR-9), Turin, Italy, 11-15 June (2006).
17. Ning, J.Q., Engberts, A., and Kozaczynski, W., Recovering reusable components from legacy systems by program segmentation, *Proceedings of Working Conference on Reverse Engineering, 1993,* pp.64-72.
18. Pohl, K., Böckle, G., and Linden, F., *Software Product Line Engineering: Foundations, Principles and Techniques*, Springer, Germany, 2005.
19. Prieto-Díaz, R., Reuse as a New Paradigm for Software Development, In *Proceedings of the International Workshop on Systematic Reuse*, London, 1996.
20. Shen, C., Ryall, K., Forlines, C., Esenther, A., Vernier, F.D., Everitt, K., Wu, M., Wigdor, D., Morris, M.R., Hancock, M., Tse, E: Interfaces and Interactions for Direct-Touch Horizontal Surfaces. IEEE Computer Graphics and Applications, vol. 26, no. 5, pp. 36-46. (2006).
21. Sugumaran, V., Tanniru, M., and Storey, V., A knowledge-based framework for extracting components in agile systems development, *Inf. Technol. and Management,* 9(1), 2008, pp. 37-53.
22. Wartik, S., and Prieto-Diaz, P., Criteria for comparing reuse-oriented domain analysis approaches. *International Journal of Software Engineering and Knowledge Engineering*, 1992; 2(3): 403–431.
23. Washizaki, H., Fukazawa, Y.: A technique for automatic component extraction from object-oriented programs by refactoring, Science of Computer Programming, Volume 56, Issues 1-2, New Software Composition Concepts, April 2005, Pages 99-116, ISSN 0167-6423, DOI: 10.1016/j.scico.2004.11.007. (2005)
24. Washizaki, H., Fukazawa, Y.: Automated extract component refactoring. In Proceedings of the 4th international conference on Extreme programming and agile processes in software engineering (XP'03), Michele Marchesi and Giancarlo Succi (Eds.). Springer-Verlag, Berlin, Heidelberg, 328-330. (2003).
25. Wobbrock, J. O., Morris, M. R., and Wilson, A. D.: User-defined gestures for surface computing. In Proceedings of the 27th international Conference on Human Factors in Computing Systems (Boston, MA, USA, April 04 - 09, 2009). CHI '09. (2009).