1-1-2017

Impact Analysis in Web Applications

Drake Svoboda
Wayne State University, fv9838@wayne.edu

Impact Analysis in Web Applications

Drake Svoboda

12/11/2017

Contents

1 Introduction

Software change is a fundamental aspect of software development.

Rajlich describes a method for conducting software change in his book *Software Engineering: The Current Practice.* The method begins with two successive steps: first *concept location*, then *impact analysis* (2012).

Concept location is the process by which a developer locates the code unit that houses the concepts that will be changed. Dependency search is a method for conducting concept location. In dependency search, the developer first looks at the highest level clients of the software. From those clients, he follows dependencies downward from client to supplier. At each step, the developer chooses the supplier that is suspected to have the concept implemented in its combined responsibility. This process stops once the code unit which houses the relevant concept is located (Rajlich, 2012, p. 95-97).

Once the relevant concept is located, impact analysis is conducted. During impact analysis, the developer determines the code units connected to the concept that must also be updated. Iteratively, the developer chooses code units that are adjacent to impacted code units to determine if those code units either propagate the change or are also impacted by the change. The estimated impact set for the change is the connected graph of code units estimated as impacted during these two processes (Rajlich, 2012, p. 105-123).

The processes of concept location and impact analysis could be condensed into a single process if the developer were able to estimate *each* highest level client that has some impact in its combined responsibility. The developer would choose those highest level clients that are estimated to have impact somewhere in their supplier slices. From those clients, a method similar to dependency search would be conducted to estimate the entire impact set. At each code unit investigated during dependency search, the developer would also estimate the impact on the code unit. In such a method, the developer would only need to traverse dependencies in one direction.

Due to the *invisibility* of software described by Brooks (1987), it can be difficult to estimate if there is impact somewhere in the supplier slice of a given code unit. Concept location by dependency search addresses this difficulty by allowing the developer to backtrack up the dependency graph if a wrong turn is taken. It can also be difficult to estimate the impact on a given code unit before the concept is located. If a highest level client that has impact in its combined responsibility also belongs to the impact set, these difficulties would be minimized. Since the impact set for a change is a connected graph, the developer can guarantee that the impact is either isolated to only the highest level client or the highest level client will have some supplier that is either impacted by the change or propagates the change. For changes where those highest level code units belong to the impact set, this method would be viable.

In applications with a user interface, the individual widgets that make up the user interface are the highest level clients of the application. If a change request describes a clear impact to widgets in the user interface, the combined process of concept location and impact analysis could be used. This report defines such a method for web applications. The method described is

demonstrated in two sample changes conducted on an open source web application. The full source code for these two case studies can be found in an open access code repository (AScrum).

1.1 Widgets

A widget is any graphical element on an application's user interface.

Widgets belong to two categories: interactive and containers widgets. Interactive widgets support an interaction with the user (labels, buttons, inputs, etc.). Container widgets group a set of widgets added to them (tabs, windows, panels, etc.). A given widget could belong to both categories. Consider a table that contains several rows. When a user clicks the table itself, some functionality is performed. This table is both an interactive and container widget.

In web applications, each widget is the client of a single HTML element. A widget can be described by the HTML element that supplies it, and an HTML element can be described by the UI widget that it supplies. An HTML element that supplies a container widget can be visualized as a tree node with children that are HTML elements that supply widgets within the container. An entire web page can be visualized this way with the root node as the document's outermost container.

The combined responsibility of a container widget is the combination of the responsibilities of each widget within the container; therefore, any HTML element that is nested in a parent HTML element is a supplier to that parent. For example, consider a form that contains buttons and inputs. The function of the form is dependent on each button and input within the form. The tree representation of an HTML element can be extended to a dependency graph with directed edges from parent node to child node.

2 Method for Impact Analysis in Web Applications

The first step of this method is to determine the impact on the user interface. The impact on the user interface is the set of all widgets impacted by the change. A widget is impacted if the change request describes any change to the combined responsibility of the widget, i.e., there are some impacted units in the supplier slice of the widget. This method is only applicable to changes where the set of impacted widgets can be determined intuitively using only the change request and the user interface. The impacted widgets are the highest level clients that are used to begin the method.

Each widget in the impacted set is marked NEXT. For each of these widgets, the developer discovers the HTML element that supplies the widget.

The HTML element that supplies a widget can be discovered using web browser development tools. In Google Chrome, the developer can "inspect" the widget by right clicking on the widget in the UI and selecting the "inspect" option from the menu. Chrome will open the developer window with the HTML element highlighted.

The developer must then locate the suppliers to the discovered HTML element.

The “event listeners” tab of Chrome’s developer window displays each JavaScript method bound to any of the HTML element’s events (click, mouse over, etc.). The JavaScript code units where these methods are located are suppliers to the HTML element. These code units are marked NEXT.

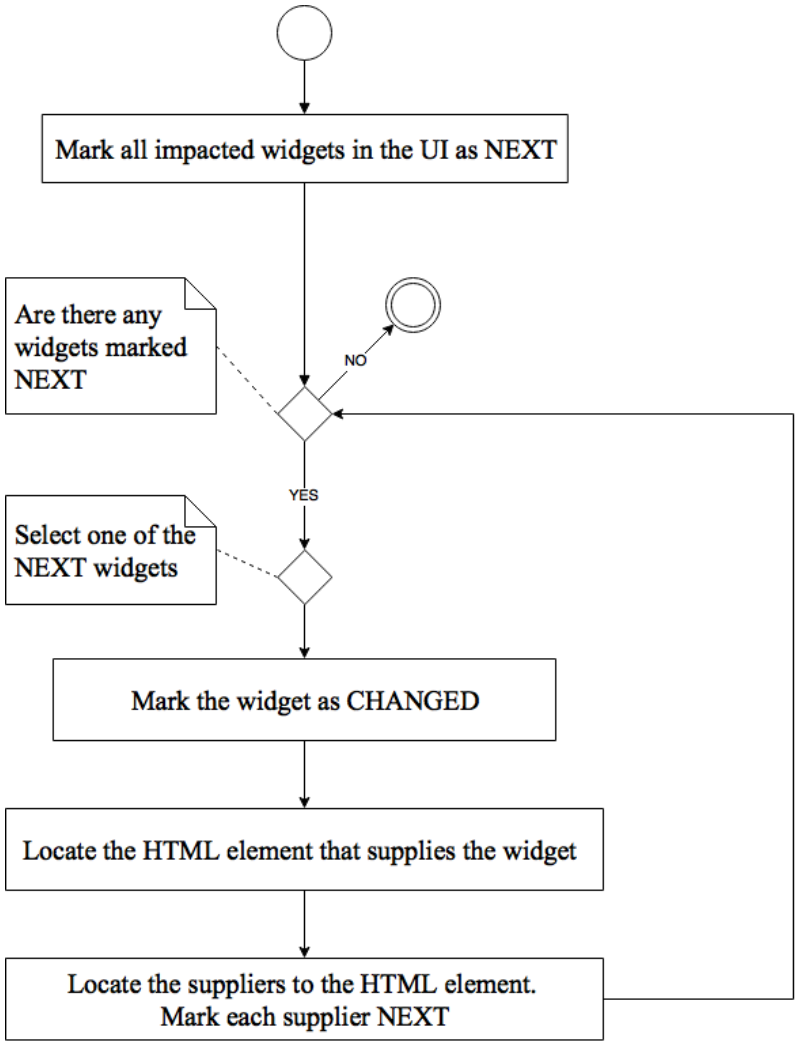

Figure 1. Activity Diagram for Phase 1 of Impact Analysis Using the User Interface

The code units marked NEXT in the previous steps serve as a foothold in the code from which further impact analysis can be conducted via dependency search. The activity diagram for the remaining steps of impact analysis is shown in Figure 2.

Iteratively, the developer selects a code unit marked NEXT and estimates the new mark for the unit. If the new mark for the unit is either CHANGED or PROPAGATING, each supplier to that unit is marked NEXT. Once every NEXT code unit has been investigated, the process is terminated.

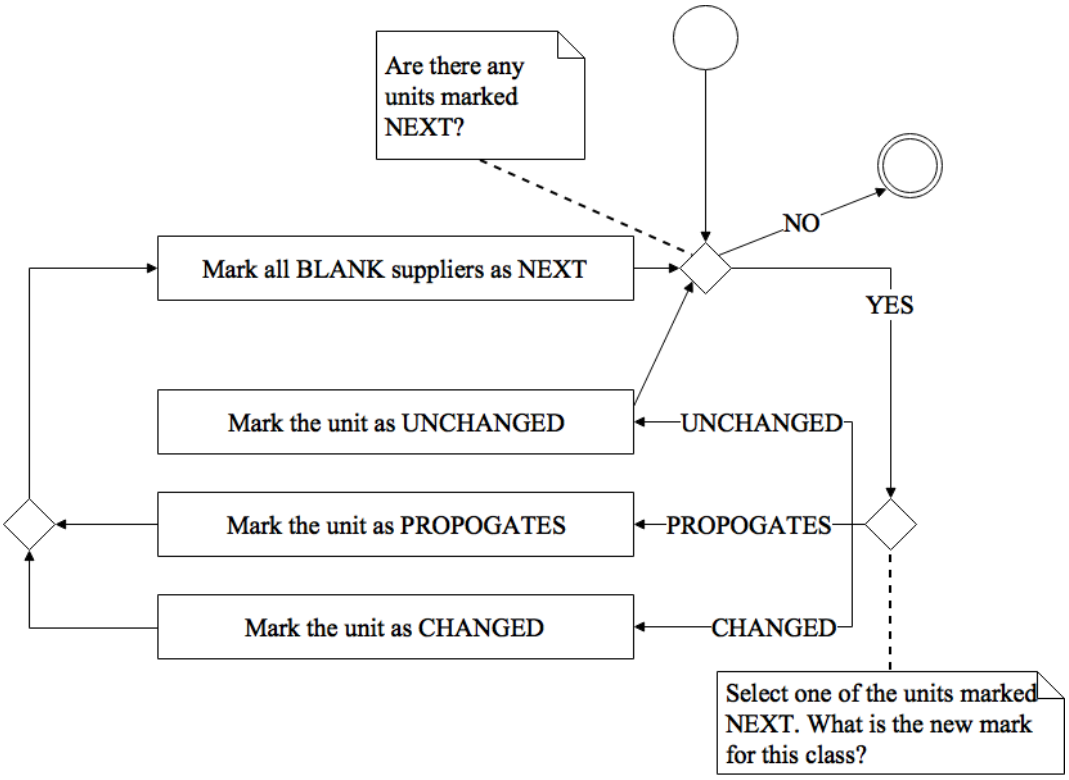

Figure 2. Activity diagram of impact analysis.

3 Overview of AScrum

AScrum, an open source web application, was chosen as the target of two case study changes. These changes were conducted using the method described in section 2. AScrum was written to be featured in the book *Mastering Web Application Development with AngularJS* by Kozlowski & Darwin. AScrum was built using AngularJS, a client side JavaScript web framework.

The purpose of AScrum is to manage the scrum software development process. Users can sign in to the application to see projects and tasks to which they are assigned. Users can also create, update or delete projects, backlog items, sprints, and tasks.

AScrum uses MongoDB, an open-source, document-oriented database, to store five collections of data: *users*, *projects*, *backlog items*, *sprints*, and *tasks*.

AScrum was selected because a large percentage of its source code is client side JavaScript. This quality facilitates the discovery of suppliers to HTML elements using Google Chrome. Further research is needed to determine the validity of this method using other frameworks and technologies.

4 Case Study One

AScrum allows users to update four different types of resource: projects, backlog items, sprints and tasks. A sample change was conducted on AScrum to allow a user to revert the changes to a resource to any point in that resource's revision history.

4.1 Impact analysis

The "Save" and "Revert Changes" button widgets were estimated as impacted for this change. These buttons appear on the task edit page, backlog item edit page, project edit page, and user edit page of AScrum. The user interface for the backlog item edit page is shown in Figure 3.

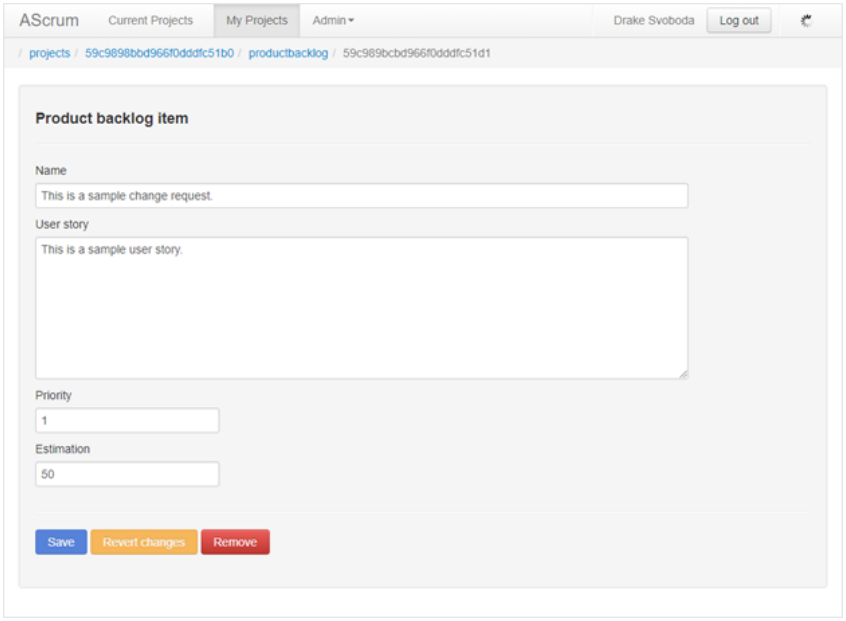

Figure 3. Backlog Item Edit Page

The "Save" button saves the changes the user makes to a resource. This change will extend the "Save" button to also update the resource's revision history on each save.

The "Revert Changes" button allows the user to revert the changes he or she made while on the edit page. This change will update the "Revert Changes" button to allow the user to revert to any point in the resource's revision history.

These two buttons make up the set of impacted widgets; both of these widgets were marked NEXT.

The "Save" button was selected first, marked CHANGED, and inspected in Google Chrome. Chrome retuned an HTML `button` element; this element invokes the `save` method supplied by the AngularJS module `directives.crud.edit`. `directives.crud.edit` has two members, `resource` and `original`, that store an instance of a resource from the database. The `resource` field is updated when the user makes changes in the form. The `save` method

saves these updates to the database. The `save` method should be updated to also save `original` to the resource's revision history; `directives.crud.edit` was marked CHANGED. The HTML `button` element was marked PROPAGATING.

`Resource` and `original` are each instances of the nested abstract type `Resource` from the AngularJS module `mongolabResource`; `mongolabResource` was marked NEXT.

The "Revert Changes" button widget was selected, marked CHANGED, and inspected in Google Chrome. Chrome retuned an HTML `button` element; this element invokes the `revert` method supplied by `directives.crud.edit`. The `revert` method restores `resource` to `original`. This method should be changed to restore `resource` to any point in the resource's revision history. `directives.crud.edit` was already marked CHANGED in the previous steps. The HTML `button` element was also marked CHANGED; this button should be updated to allow the user to select a point in the resource's revision history.

`mongolabResource` was investigated next to determine its new mark. This unit is responsible with querying and updating the collections in the database. This change will introduce new collections to store a revision history for each element in the database. New functionality should be added to the nested abstract type `Resource` from `mongolabResource` to query these collections. `mongolabResource` is marked CHANGED. The following modules extend `Resource`: `resources.tasks`, `resources.productbacklog`, `resources.users`, `resources.sprints`, and `resources.projects`. Each of these code units were marked NEXT, investigated, and determined to be UNCHANGED.

The HTML markup for the "Save" and "Revert Changes" HTML `button` elements is located in `directives.crud.buttons`. These HTML elements were already located in Chrome and marked. `directives.crud.buttons` contains the extensions of these HTML elements in the source code. `directives.crud.buttons` was located by a GREP search for a snippet of each HTML `button` element's markup. This snippet searched was retrieved from Chrome. Further research is needed to determine a reliable method for locating the extension of an HTML element in source code from an HTML element found in a web browser.

Both `button` elements are clients of `directives.crud.edit` and suppliers to widgets in the UI; `directives.crud.buttons` contains the HTML markup for these elements and is therefore also a client of `directives.crud.edit` and a supplier to the two widgets in the UI.

New HTML markup should be added to `directives.crud.buttons` for a select box that allows the user to select a point in the resource's revision history; `directives.crud.buttons` was marked CHANGED.

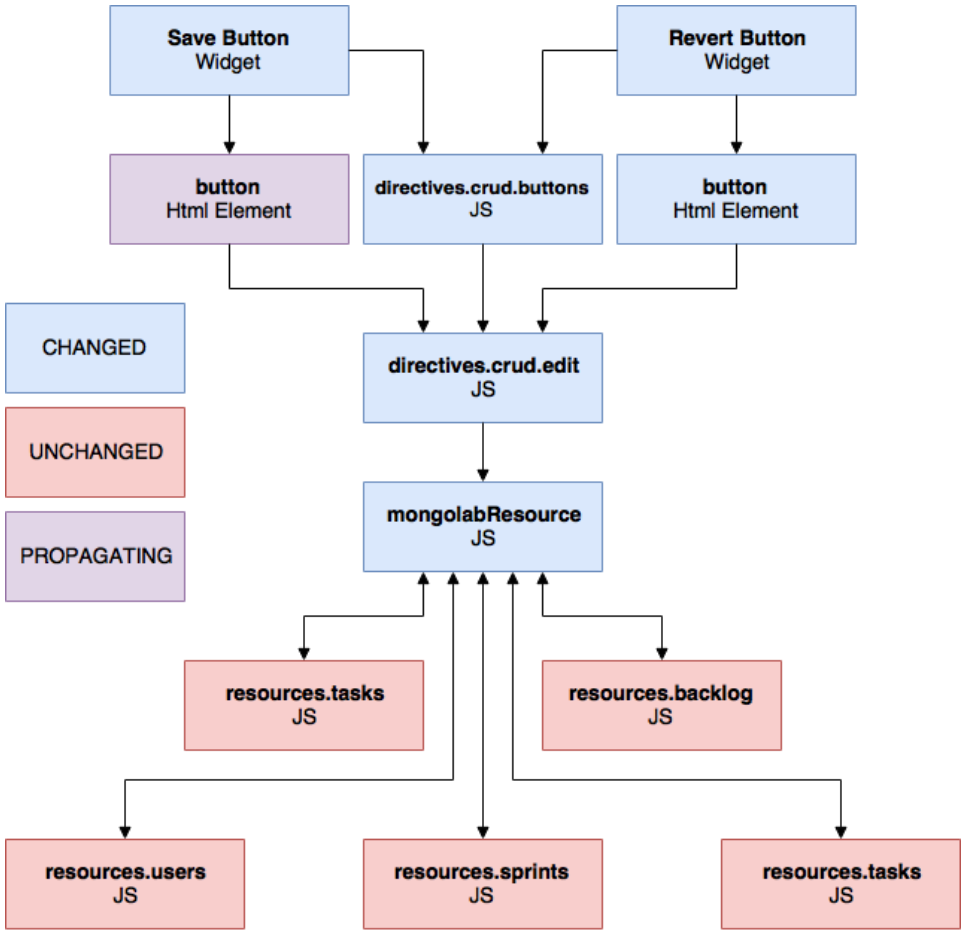

Figure 4. Estimated Impact Set

4.2 Actualization

When a type extends the abstract type `Resource`, it supplies `Resource` with the name of a collection in the database; `Resource` is then responsible with querying that collection. For each collection in the database, a corresponding history collection was added. The new collections added were *usersHistory*, *projectsHistory*, *backlogitemsHistory*, *sprintsHistory* and *tasksHistory*. An element in a history collection has the foreign key `historyFor` that matches the id of an element in the main collection. Any type that extends `Resource` must now also have an associated history collection; new functionality was added to `Resource` to facilitate querying that history collection.

`Resource` has both static and instance methods. An instance of `Resource` has variable members that match the fields of an element in the database.

The static method `queryHistory` was added to `Resource` to facilitate querying a history collection. This method takes a query as input then returns the result from the appropriate history collection.

The instance method `$saveAsHistory` was added to `Resource` to insert an element to the appropriate history collection. `$saveAsHistory` assigns the instance's `historyFor` member to the instance's id then inserts the instance into the history collection.

The instance method `$allHistory` was added to `Resource` to retrieve the entire revision history for an instance of `Resource`. This is accomplished by querying the appropriate history collection for elements whose `historyFor` member matches the id of the give instance.

`directives.crud.edit` has two members, `resource` and `original`, that store an instance of `Resource`. The `save` method from `directives.crud.edit` was extended to save both `resource` to the main collection and `original` to the resource's revision history by using the `$saveAsHistory` method.

`directives.crud.edit`'s `revert` method was replaced by the method `revertTo`; `revertTo` takes an instance of `Resource` as a parameter and assigns the `resource` member to that instance.

`directives.crud.edit`'s `canRevert` method was replaced by the method `canRevertTo`; `canRevertTo` takes an instance of `Resource` as a parameter and returns false if the passed parameter matches the `resource` member. This method is used to prevent the user from reverting to the revision that is currently being displayed.

The `directives.history` AngularJS Module was created to house the HTML markup for a new history select box. The history select box lists the date and time of each revision; the user can select a time and revert to that revision. The user interface for the history select box can be seen in Figure 5.

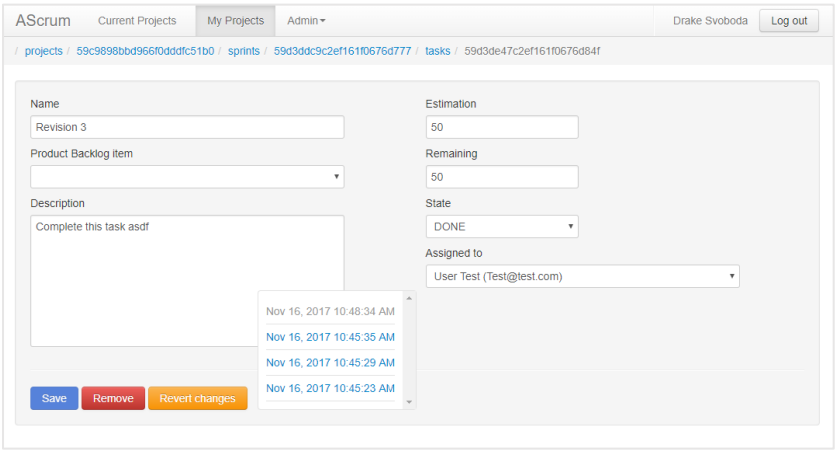

Figure 5. History Select Box

The "Revert Changes" HTML `button` element was moved from `directives.crud.buttons` to `directives.history`. This HTML `button` element was updated to invoke a new method, `toggleHistory`, supplied by `directives.history` that toggles the visibility of the history select box.

Each link within the history select box invokes the new `revertAndClose` method supplied by `directives.history`. This method calls `directives.crud.edit`'s `revert` method then hides the history select box. Since `directives.history` contains the "Revert" HTML `button` element and supplies that element with the `revertAndClose` method, `directives.history` is a supplier to both the "Revert" button widget and the "Revert" button HTML element.

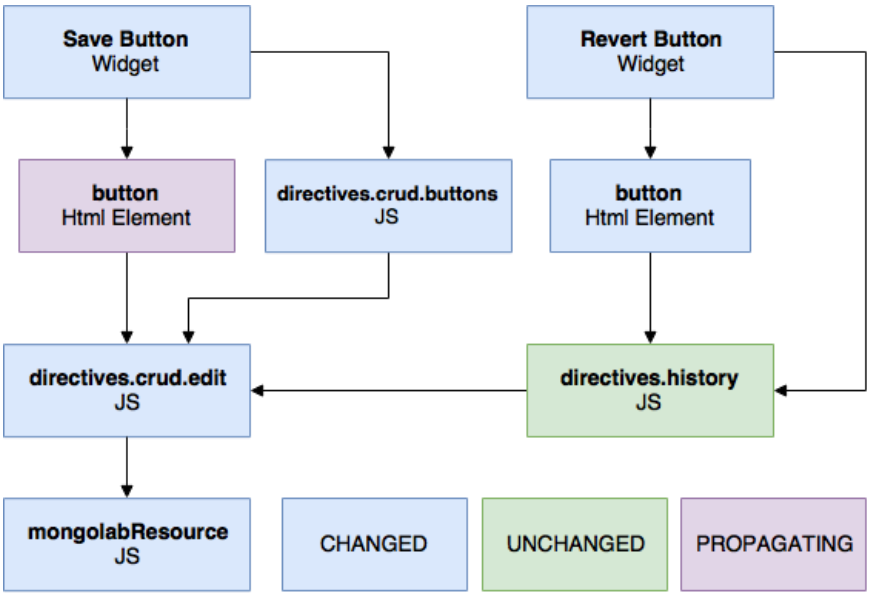

Figure 6. Dependency Graph after Actualization

5 Case Study Two

A second sample change was conducted on AScrum. This change added the ability for a user to watch a task. Each task that a user is watching is displayed on the user's dashboard. Two widgets were estimated as impacted for this change: the task edit form and the user dashboard. The estimated impact on the task edit form is the addition of a new button to allow the user to watch the task. The estimated impact on the user dashboard is the addition of a list of tasks the user is watching.

5.1 Impact Analysis

The task edit form is a container widget that contains buttons and inputs. The user interface for the task edit form is shown in Figure 7. The task edit form was inspected to reveal an HTML `form` element. The only suppliers to the `form` element are the HTML elements nested within the `form`. Each of these HTML elements were marked NEXT.

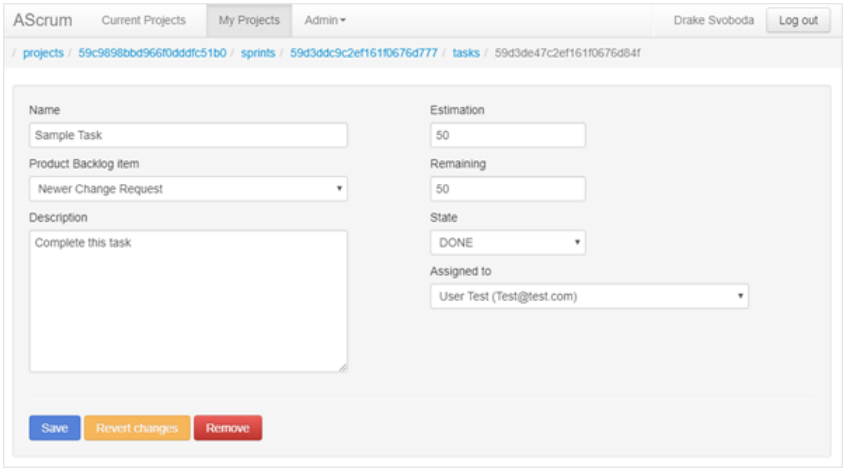

Figure 5. Task Edit Form

The HTML `button` element that supplies the "Remove" button widget was selected. This `button` is not impacted by the change, however, its suppliers were located and investigated to determine if the `button` should be marked PROPAGATING. This `button` element invokes the `remove` method supplied by the AngularJS module `directives.crud.edit`. The new mark for `directives.crud.edit` was estimated as UNCHAGED. The HTML `button` element was also marked UNCHANGED. The analysis of `directives.crud.edit` revealed that the "Save" and "Revert" HTML button elements are also supplied by `directives.crud.edit`.

Next, the HTML `input` element that supplies the "Name" input widget was investigated. This `input` is not impacted by the change, however, its suppliers were located and investigated to determine if the `input` propagates the change. The `input` element updates the `name` attribute of the variable `task`; `task` is a member of the AngularJS controller `TaskEditCtrl`. `TaskEditCtrl` was estimated as CHANGED. The estimated impact to `TaskEditCtrl` is the addition of a new method to allow the current user to watch the task. The `input` element was marked PROPAGATING.

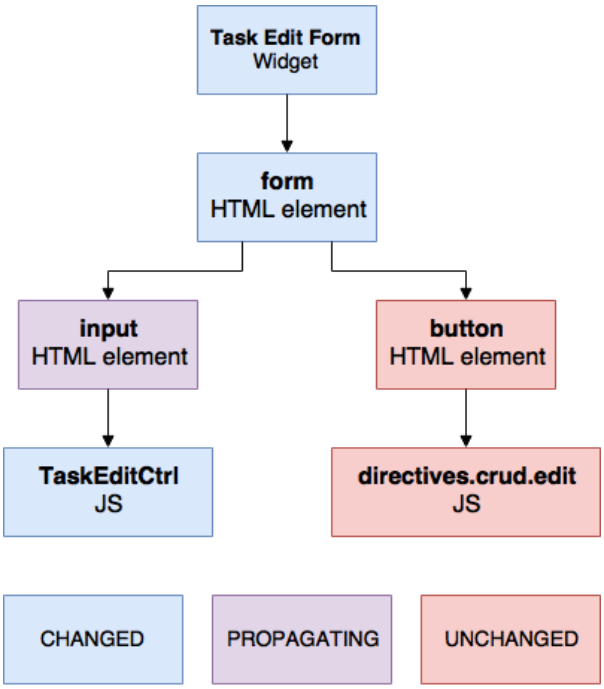

Figure 8. TaskEditCtrl Located from the Task Edit Form.

`TaskEditCtrl` is a client of the following code units: `resources.tasks`, `resources.productbacklog`, `resources.projects`, `resources.sprints`, `resources.users`, `services.crud`, and `services.crudRouteProvider`. Each of these units were marked NEXT.

The user dashboard is also a container widget. The user interface for the user dashboard is shown in Figure 9. The user dashboard was inspected to reveal an HTML `div` element. The only suppliers to the `div` element are the HTML elements nested within the `div`; these element were marked NEXT.

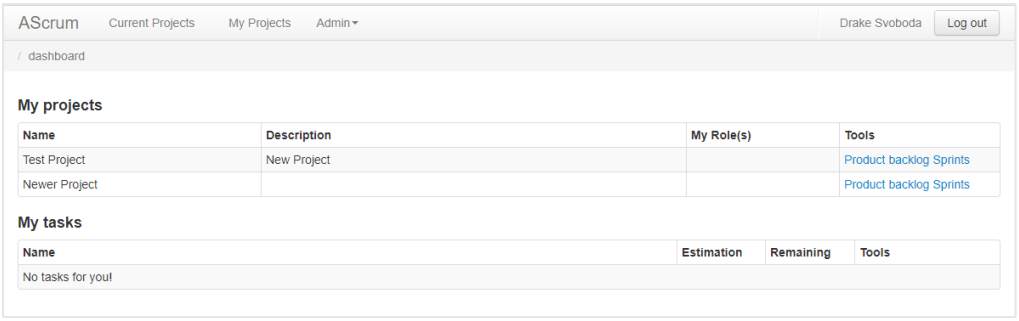

Figure 9. User Dashboard

The HTML `anchor` element that supplies the "Sprints" link was investigated. This element is unchanged, however, its suppliers were located and investigated to determine if the element propagates the change. The `anchor` element invokes the `manageSprints` method supplied by the AngularJS controller `DashboardCtrl`. `DashboardCtrl` was estimated as

CHANGED. The estimated impact on `DashboardCtrl` is the addition of a new member to store the list of tasks the current user is watching. The `anchor` element was marked PROPAGATING.

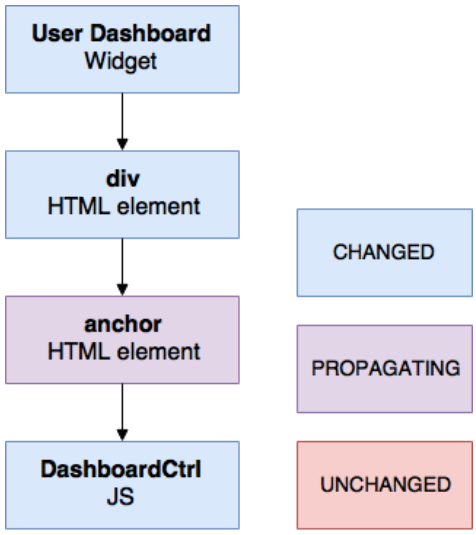

Figure 6. DashboardCtrl Located from the User Dashboard

`DashboardCtrl` is a client of `resources.projects`, `resources.tasks`, and `security.service`. These code units were marked NEXT.

Each of the code units marked NEXT during the previous steps were investigated. Only `resources.tasks` was estimated as CHANGED; the rest were estimated as UNCHANGED.

`resources.tasks` is responsible with querying the *tasks* collection in the database. The estimated impact on `resources.tasks` is the addition of a new method to return all of the tasks that a given user is watching. `resources.tasks` extends the nested abstract type `Resource` from the AngularJS module `mongolabResource`; `mongolabResource` was marked NEXT, investigated, and estimated as UNCHAGED.

The HTML markup for the task edit form's HTML `form` element is located in the HTML template `tasks-edit.tpl.html`. The `form` element was already located in Chrome and marked CHANGED. `tasks-edit.tpl.html` contains the extension of this `form` element in the source code of the application. `tasks-edit.tpl.html` was located by a GREP search for a snippet of the `form`'s HTML markup. This snippet was retrieved from Chrome.

The `form` element is a supplier to the task edit form widget and a client of `TaskEditCtrl`; therefore, `tasks-edit.tpl.html` is also a supplier to the task edit form and a client of `TaskEditCtrl`. This change will introduce a new "Watch This Task" HTML `button` element to `tasks-edit.tpl.html`; `tasks-edit.tpl.html` was marked CHANGED.

The HTML markup for the user dashboard's `div` HTML element is located in the HTML template `dashboard.tpl.html`. This `div` element was already located in Chrome and

marked CHANGED. `dashboard.tpl.html` contains the extension of this `div` element in the source code of the application. `dashboard.tpl.html` was located by a GREP search for a snippet of the `div`'s HTML markup. This snippet was retrieved from Chrome.

The `div` element is a supplier to the user dashboard widget and a client of `DashboardCtrl`; therefore, `dashboard.tpl.html` is also a supplier to the widget and a client of `DashboardCtrl`. This change will introduce a new HTML markup to `dashboard.tpl.html` for the list of tasks the user is watching; `dashboard.tpl.html` was marked changed.

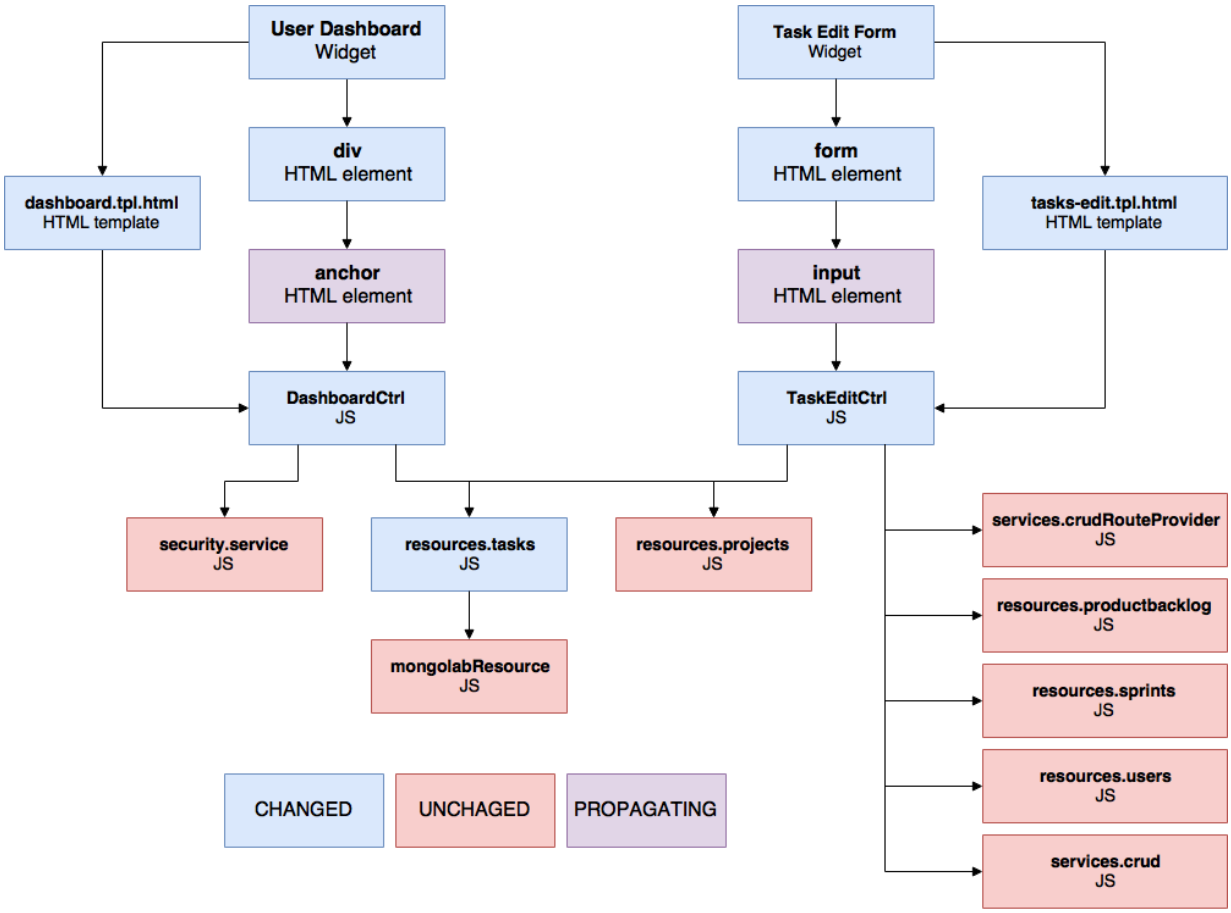

Figure 7. Estimated Impact Set

5.2 Actualization

The list `usersWatching` was added to each task in the *tasks* collection. This list stores the id of each user who is watching the task.

Two methods were added to `resources.tasks`: `watchedBy` and `isWatchedBy`.

The instance method `isWatchedBy` takes a user id as input. This method returns true if the given user is watching the task and false otherwise.

The static method `watchedBy` takes a user id as input. This method returns the list of tasks from the *tasks* collection that the given user is watching.

`TaskEditCtrl` has the member `task` that stores the task that is being edited by the task edit form.

The member `user` was added to `TaskEditCtrl` to store the user who is currently logged in.

The method `watch` was also added to `TaskEditCtrl`. This method adds `user`'s id to `task`'s `usersWatching` list then saves the changes to the *tasks* collection.

An HTML `button` element was added to `tasks-edit.tpl.html`. This button invokes the new `watch` method from `TaskEditCtrl`. The `isWatchedBy` method is used to disable the button if `user` is already watching the task.

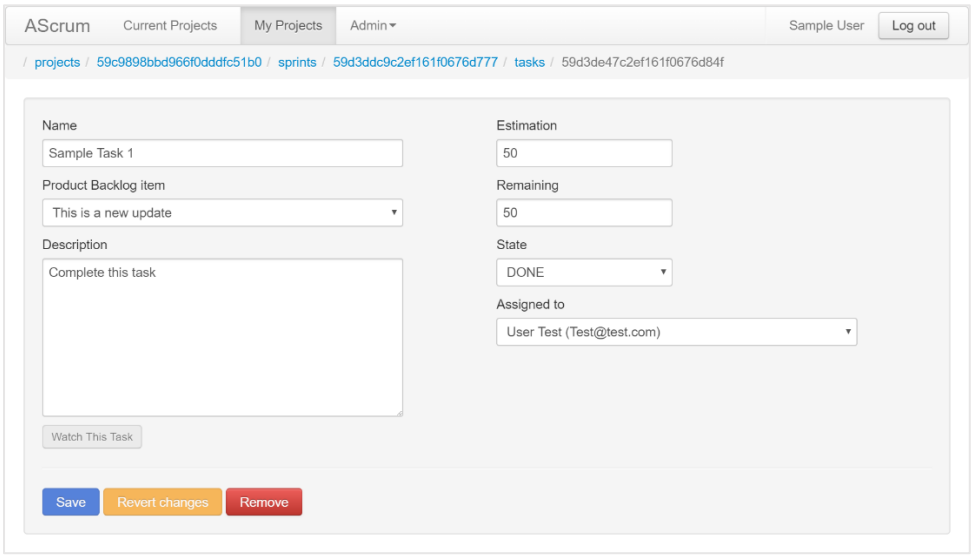

Figure 12.8 Task edit form after actualization

The member `watching` was added to `DashboardCtrl` to store the list of tasks the current user is watching. This variable is populated using the `watchedBy` method from `resources.tasks`. HTML markup was added to `dashboard.tpl.html` to display each task in `watching`.

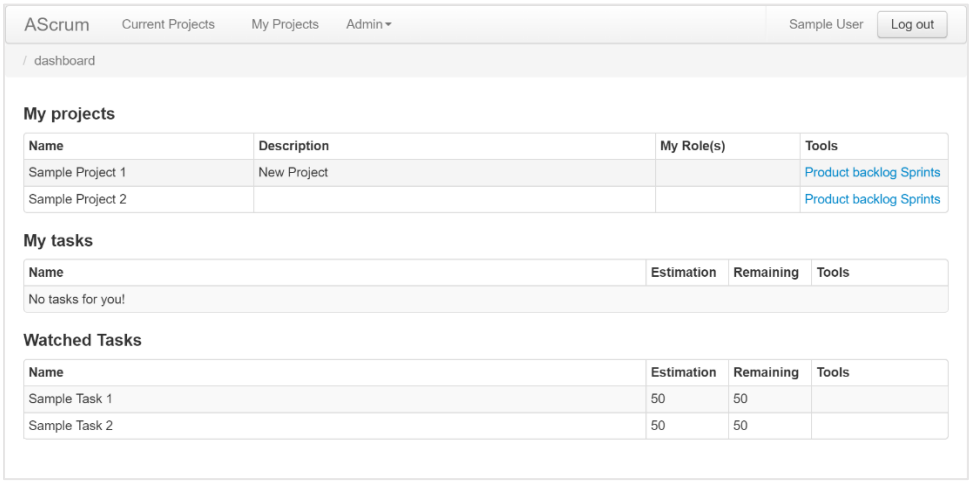

Figure 13. Dashboard after actualization

6 Conclusion

The two case study changes conducted for this report demonstrate the validity of the method described in section 2.

The widgets that make up a web application's user interface are the highest level clients of the application. If a change request describes an impact to a set of widgets and those widgets make up the set of all highest level clients that have some impact in their supplier slices, then a combined method of concept location and impact analysis is feasible.

7 Literature

AScrum, (2017), GitHub repository, doi:10.5281/zenodo.1100988

Brooks, F. (1987). No Silver—Bullet Essence and Accidents of Software Engineering. *Computer, 20*(4), 10-19. doi:10.1109/mc.1987.1663532

Google Chrome [Computer software]. (2014). Retrieved from https://www.google.com/chrome/browser/desktop/index.html

Kozlowski, P., & Darwin, P. B. (2013). *Mastering web application development with AngularJS*. Birmingham, U.K.: Packt Pub.

Rajlich, V. (2012). *Software engineering: the current practice*. Boca Raton, FL: Chapman & Hall/CRC.