28TH ANNUAL
PACIFIC NW
SOFTWARE
QUALITY
CONFERENCE

OCTOBER 18TH – 19TH, 2010

ACHIEVING QUALITY IN A COMPLEX ENVIRONMENT

Conference Paper Excerpt from the

CONFERENCE PROCEEDINGS

Using Silverlight PivotViewer to make sense of the chaos

Max Slade, Frederick Fourie, Melinda Minch, Ritchie Hughes

maxs@microsoft.com

Abstract

Today's methods of organizing and searching information cannot keep pace with the exponential rate of growth. This information overload presents an opportunity to expose new value in the aggregate, where the data interactions provide greater ability to act on previously unexplored insights.

PivotViewer will change the way people perceive and explore the information that surrounds them by visualizing information in new ways, exposing hidden relationships, and making it easier to act on these newly discovered insights. This is possible because PivotViewer can simultaneously show thousands of items on screen at once while allowing you to effortlessly browse around the information by selecting properties that you want to explore. You are also able to zoom into a single item and get more information and then pivot on any property that the item has.

In an industry which generates vast quantities of data in the form of defect reports, test results, code coverage, and other metrics, Testers can use PivotViewer to quickly find trends and anomalies, visualize expected trends for consumption by decision makers, and focus our limited resources on the areas that matter most. We expect that PivotViewer can be used in ways that we haven't yet imagined.

Biographies

Max Slade is a Principal Test Manager at Microsoft's Live Labs. Max has worked at Microsoft in the test discipline since 1996. He has led the test teams for PivotViewer, Live Labs, Desktop Search, and NetGen. Max has a bachelor degree in Electronics Technology from Chapman University

Frederick Fourie has been developing software at Microsoft for nearly ten years on products ranging from Windows to Robotics. Frederick has a degree in Computer Science and Applied Mathematics from Rhodes University (South Africa) and is currently working on Microsoft Robotics Studio.

Ritchie Hughes is a software design engineer in test at Microsoft's Live Labs, located in downtown Bellevue, Washington. Ritchie's career at Microsoft started with building tools and test automation for the original Windows Mobile developer platforms, before moving on to an incubation project that become Microsoft Surface. Ritchie has a bachelor degree in Computer Science from the University of Cambridge.

Melinda Minch is currently a tester at Getty Images in Seattle, Washington, where she works on the main website. Melinda has an M.S. in Computer Science from Case Western Reserve University and a long-standing interest in multivariate data visualization.

1. Introduction

As testers, one of our main functions is to measure things that give visibility into the quality of the software we're building. We measure defect reports, functionality, performance, stability, stress, load, localizability, etc. and in doing so generate staggering amounts of information. There is so much information generated while testing a typical software project that it's difficult to derive the full value from all that is produced. We settle for generating charts, graphs, or tabular information that are adequate at answering directed questions. However, there is a wealth of untapped information trapped in those formats that are not easily viewed in aggregate. There are veins of information gold hidden in those repositories.

Now it is possible to prospect with PivotViewer, a powerful tool that allows you to view your test data in a way that's powerful, informative, and fun. With PivotViewer, you can visualize, explore, and understand large sets of data; we need not be in a state of information overload. Now we find that the whole is greater than the sum of the parts. PivotViewer makes it possible to truly understand the information. It is possible with PivotViewer to move from many items to one item, from many items to many related items, or from one to many related items.

PivotViewer is a Silverlight Control that can be used to view the data that you generate while testing your product. By using tools provided by the PivotViewer team, you can translate your data into the two formats that PivotViewer requires – CXML (an XML schema), and Deep Zoom images. Then using traditional web hosting or the Just in Time server that we provide sample code for, you can make your data available in PivotViewer. Detailed information about Deep Zoom can be found here: http://www.silverlight.net/learn/quickstarts/deepzoom/ .

PivotViewer has been used by several Fortune 500 companies to gain insights from Business Intelligence information that were previously hidden, required an expert to detect, or were difficult to explain to decision makers via tables or spreadsheets.

Live Labs at Microsoft released the PivotViewer experience with two different technologies. Last November the Pivot WPF client was released as an experiment. We expected that once we released the Pivot experience to the world, we'd see uses that we hadn't imagined ourselves. This happened in a big way, primarily around how different companies used it for Business Intelligence. Corporations found that putting the data that they use to judge business success into Pivot was amazingly powerful. The success of the WPF client led to many requests for a way to have the Pivot experience in any browser and on any Operating System. Our answer came at the end of June this year when we shipped the Silverlight PivotViewer control.

In this paper I will describe the application of PivotViewer to the test discipline by walking through three example sets of information: Bugs, test run results, and code coverage. Since PivotViewer is a data visualization tool, I'll be providing screenshots and a narrative to demonstrate the mechanics of using it. However, no matter how well I convey ideas in this paper, it will not be a substitute for seeing it yourself as the information architecture relies on animations during transitions of views on the data. The key points in this paper can be significantly enhanced by trying it yourself or seeing one of the videos of PivotViewer in action available at www.getPivot.com .

Following the examples, I will describe how to put your own data into a PivotViewer collection and provide a high level description of the architecture of the Viewer and hosted data. The term "collection" has special meaning that will be described in detail. The most basic definition of a collection is the full set of

items that PivotViewer can display at once without any filters applied. A collection is the set of items and their metadata described in an XML file and associated imagery stored in a Deep Zoom collection.

In conclusion, I will have a call to action asking how you imagine PivotViewer can be used by you and your organization or by others.

2. Bugs

In the following example I'll walk through an example of a collection of Product Studio data created to explore our defect tracking. Product Studio is a bug tracking database used extensively at Microsoft. I'll describe what the collection contains, how we selected the properties to include, and how we created the visuals for individual bugs. The figures below will show the entire data set (the forest), various slices of the set by *facets* (trees), and individual bugs (leaves). Facets are properties that the items within the collection share. The example below also walks through how I explored the data. There are thousands of ways the data could be viewed. I'm going to show a selection that I hope will inspire others to see ways that PivotViewer could be used with their own data.

With PivotViewer, we can expose the important fields in the bug database as parameters (facet categories plus facets) in the filter pane – the left side of the PivotViewer (Figure 1a). We chose Status, Priority, Opened By, Resolved By, Closed By, Issue Type, Triage, Resolution, Node Name, Opened Date, Closed Date, and Days Active. All but the last facet category are pulled directly from the database. The Days Active facet category is calculated by doing a simple calculation (Date Resolved – Date Opened). This was an important factor for the PivotViewer development process. Minimizing the time between code creation and defect fixing reduced the cost of the developer having to context switch from new tasks to older tasks giving a higher velocity of feature development (Figure 1b).

Figure 1a. Six months of bugs from one of our projects. The left pane is referred to as the Facet pane. For example, Priority is a Facet Category and 0, 1, 2, 3, and 4 are facets. The counts for each facet are displayed just to the right of the facet. The current view shows all bugs opened during this time period. The colors of the items in view represent the Status of the bug. Red for Active, green for Closed, and

yellow for Resolved. Active means the bug hasn't been fixed yet. Closed means the bug has been fixed and verified by a tester in the product. Resolved means the developer believes the bug to be fixed but not yet verified by a tester.

Figure 1b. Distribution of bugs over an entire ship cycle showing number of days bugs were active before being resolved.

In general, to create a great collection we've found that seven to ten facet categories are the right number. Usually fewer facet categories is a shallow set of data and more than ten appears too complicated. However, with a database as rich in metadata as a bug database and considering that the audience using the collection will be familiar with these fields we made an exception by including more than ten. Another best practice is to order the facet categories in order of importance. With our bug database collection we included the important field values as facet categories in an order that made finding the most commonly used fields easy.

Fields can be shown in the pane on the right (info pane) once zoomed in on a single bug (Figure 2) and each one of those is a clickable link to pivot to see all bugs sorted by that value. In this collection, the info pane is only providing another way to transition to other facet categories. In other collections, there is value in exposing facets that aren't important enough to include in the Filter pane (left pane) yet are interesting to expose while looking at a single item.

Once you've decided what metadata to include for each item, the next step is to create a Deep Zoom image for a visual representation of a bug (Figure 2). We have used a declarative language that allows us to create visuals at query time that map the data to a visual representation. We have used color, text, and

shape to represent our data. We've found that a dominant background color tied to Status (Fixed, Resolved, Closed, etc), the Title at the top of the bug, a secondary color for Priority including the number in one box and a second box containing the Resolution work very well in conveying information in this context.

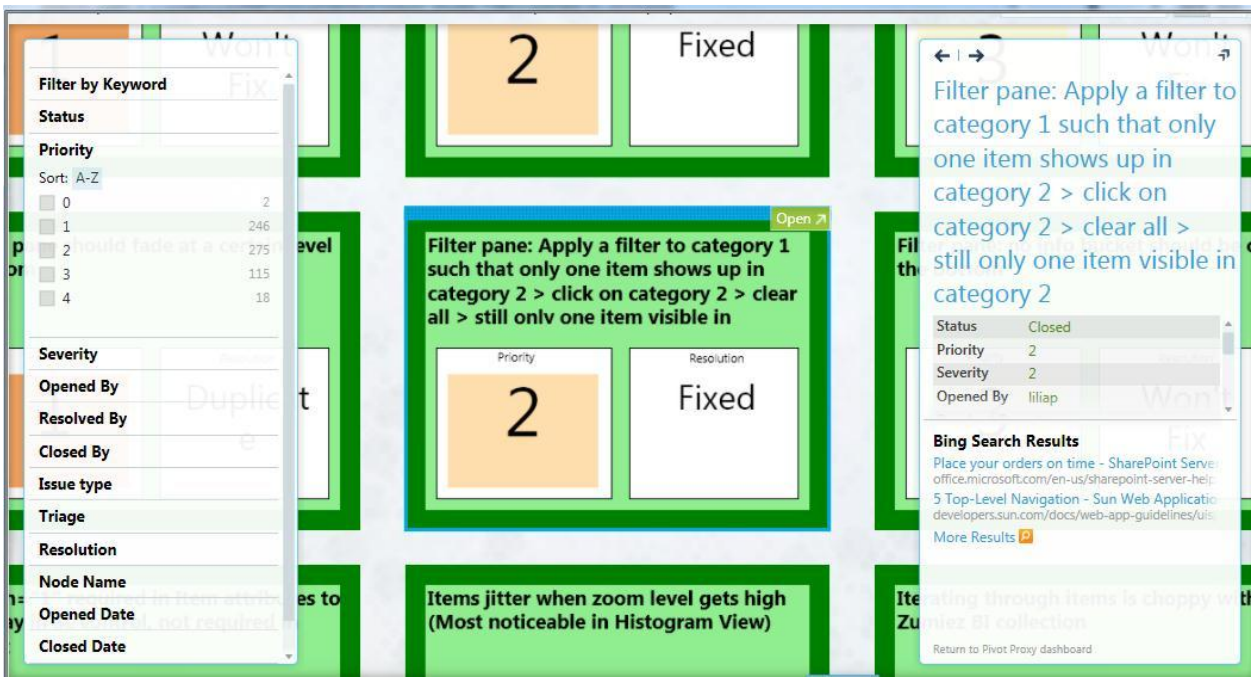

Figure 2. *Here is an example of zooming in on a particular bug – note that more detail emerges in the bug itself and the info pane on the right as you select an item. Items can be zoomed into by clicking on them or by using the mouse wheel to zoom. Once an item is selected it is possible to iterate through the collection by using the arrow keys.*

Since it's not possible to read the text when viewing the entire collection of items at once, we have a transition to just the dominant color when zoomed out (Figures 1 and 3).

The real power of PivotViewer comes from being able to easily swim around in this collection of bugs. Instead of relying on pre-authored queries, charts, or graphs, one can visualize this typically non-visual data in many interesting ways. Figure 3 is a view that shows several nodes in our bug database using histogram view. Nodes are a structure defined in the Product Studio database that is a hierarchical definition of how we categorize bugs. Consider nodes trees within the forest by way of analogy. This is powerful because with this one view I can gather quite a bit of information (See Figure 3). I can see the relative number of bugs in each node, the resolution counts in the filter pane on the left side, and the status as indicated by the color of the items. I can also easily learn about the distribution of any of the other filter pane groups by selecting them – the information is now relevant to the filtered set of information, not the entire collection.

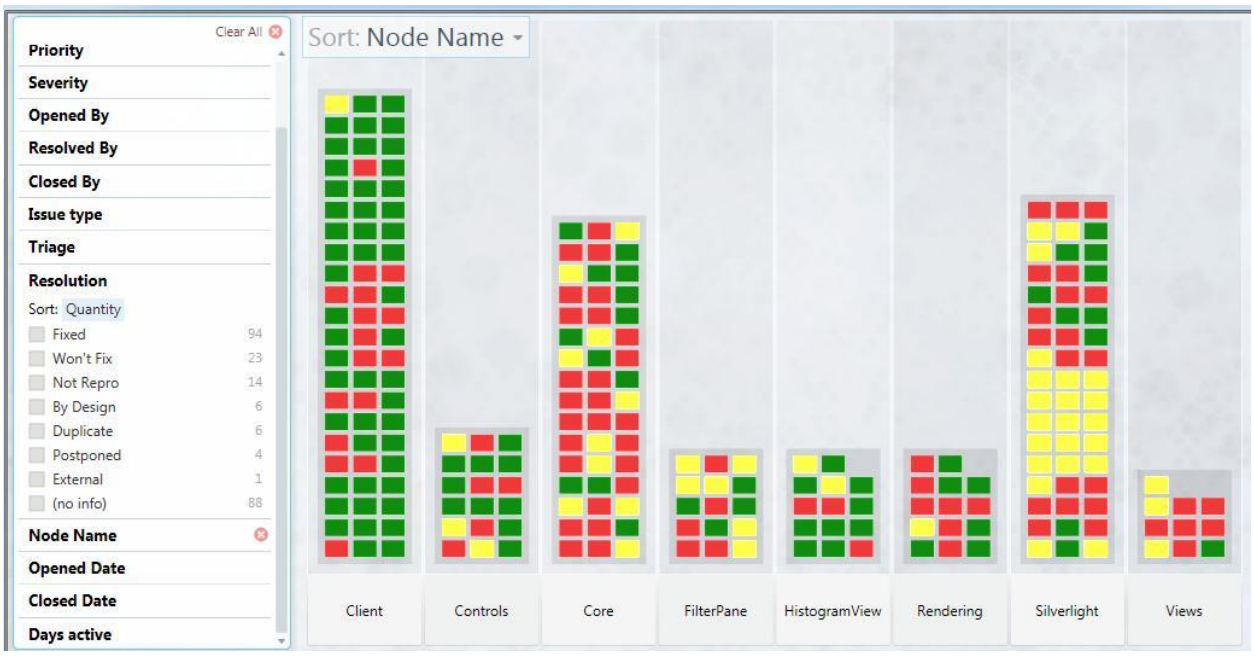

Figure 3. Histogram view of various nodes in a Product Studio bug database. In this view I've selected a number of functional and technical areas (nodes in the Product Studio database) by selecting them in the Filter pane (left side). The view is then sorted by Node Name (see "Sort: Node name" above the view). This particular view showing a large number of bugs in "Core" is interesting since we want core components to be solid so the features built on top are on a more stable code base.

PivotViewer can be used to easily find missing information. For example, when we first put our bug database information into PivotViewer, I quickly learned that some of the information we want people to enter was incomplete. The Severity field wasn't always being given a value (Figure 4) because of all of the bugs that had (no info) for that value. If Severity was being used by the team to improve prioritizing work or as a way of providing better visibility into the quality of the project, then this would be a useful discovery and could be corrected with the team.

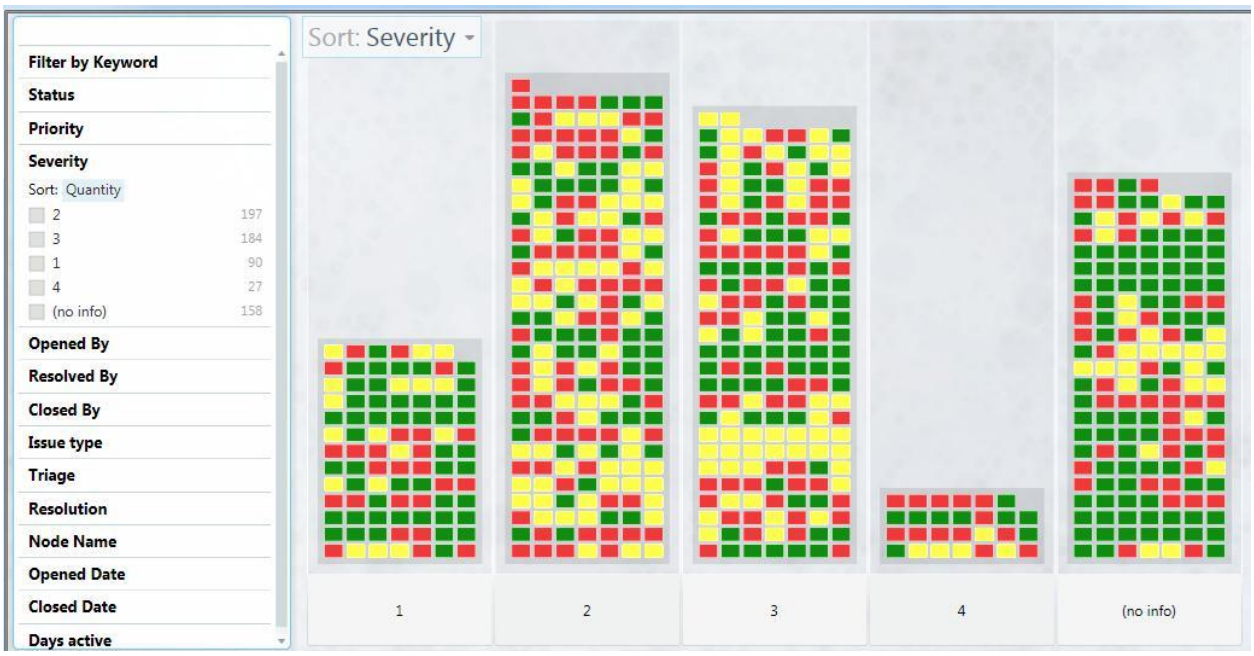

Figure 4 – Note bugs on the right side of the view with "(no info)". The color coding in this view for Status can be ignored.

Next, I graphed the bugs over time with the Opened Date field. This revealed the trend that I expected given the releases we had in that timeframe (See Figure 5a and 5b with defect issues graphed over time). This is valuable for a release manager to verify trends that are expected in an intuitive visual way. These views also facilitate showing others on the team the quality of the project in an intuitive easily understood way. The combination of the filter pane and the histogram with the visual display of data works with both the left and right brain. This combination brings information understanding that has not yet been available in a generalized way before. One can find better visual representations of information for any give dataset but PivotViewer provides a consistent and intuitive way to see multi-dimensional data representations for any data.

For example, in Figure 5b I can see what days of the week tend to have the most bugs opened in a given set of functional areas. The color coding tells me the status which may be important if a milestone is approaching. More importantly, this view can show the trend of bugs being opened over time for a particular feature area. Instead of having a holistic product view, I can drill into a feature area or set of features to gauge the health of this area. Besides getting a deeper understanding of the quality of a feature area, I can share this view with others and it is easily understood due to the intuitive graphic display. Each view state has a unique URL so it's easy to share any interesting views you find with others on the team.

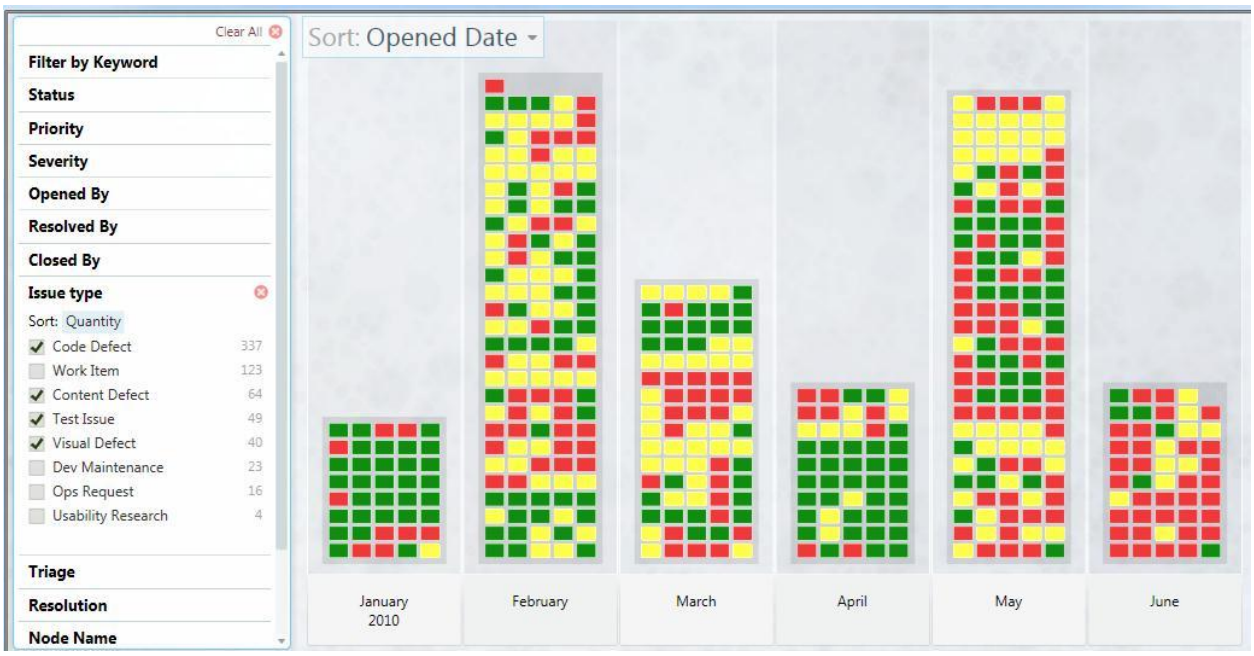

Figure 5a – Defects displayed over six month period

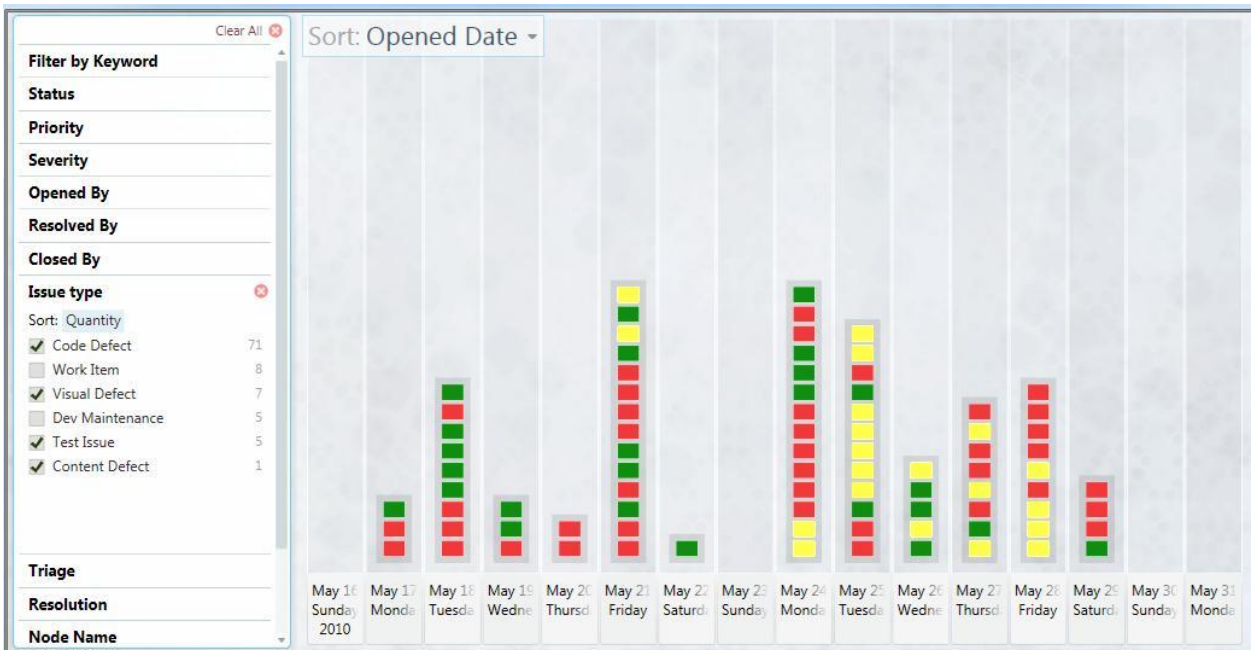

Figure 5b – Defects displayed over a two week period

Since it's easy to browse around the information I discovered some information that I hadn't known to look for. I realized that we have fixed bugs that were not approved by Triage. This informed me of a flaw in our process and prompted me to follow up on why we were working on bugs that may not have been required for release (see Figure 6).

Figure 6 – Bugs that were fixed, yet not Triage approved. Every bug in this view has been resolved fixed and the Triage counts show that 21 bugs are marked "Needs Triage" and 10 are marked "Investigate". This is a breach of our processes and a result of seeing these bugs lead me to find out why this occurred and find a way to prevent this in the future.

One thing that is difficult to explain in a paper with static screen shots like this is the powerful and fun transitions that occur when slicing and dicing the information with the filter pane or by clicking on columns. The animations that occur are powerful because they give the user a sense of context between a subset of the information and whole of the data, or from one subset to another. We aimed for a frame rate of 30-60 frames per second which gives a really smooth appearance. The fun part is that it's simply amazing to have the power to arrange information so easily and in such a pleasing way. You will find that it's fun to explore information that you cannot currently imagine is fun to peruse today.

I've given a single data source as an example but PivotViewer is designed to be a ubiquitous information visualization tool. We've also created collections with test run results and code coverage results.

3. Test run results

The collection found below in Figure 7 and Figure 8 contains around ten thousand test run results. This collection is interesting because it can decrease time spent debugging test failures. With PivotViewer, you can easily see common elements between the failures which will help you quickly determine where the root problem lies. In the test run data for Seadragon, PivoViewer was used to see test trends over time and on different hardware. It was now possible to see the commonalities in failures over multiple runs. This was particularly interesting because Seadragon did multiple runs with different hardware and it was much easier to spot the root cause within a pattern using PivotViewer. One helpful visual that was added to the visuals was the set the tics (sparkline) next to the Pass/Fail text in the item to indicate if this test has been flakey in the past or has just failed for the first time in a while (Figure 8).

Figure 7 – View of entire result set broken down by test run

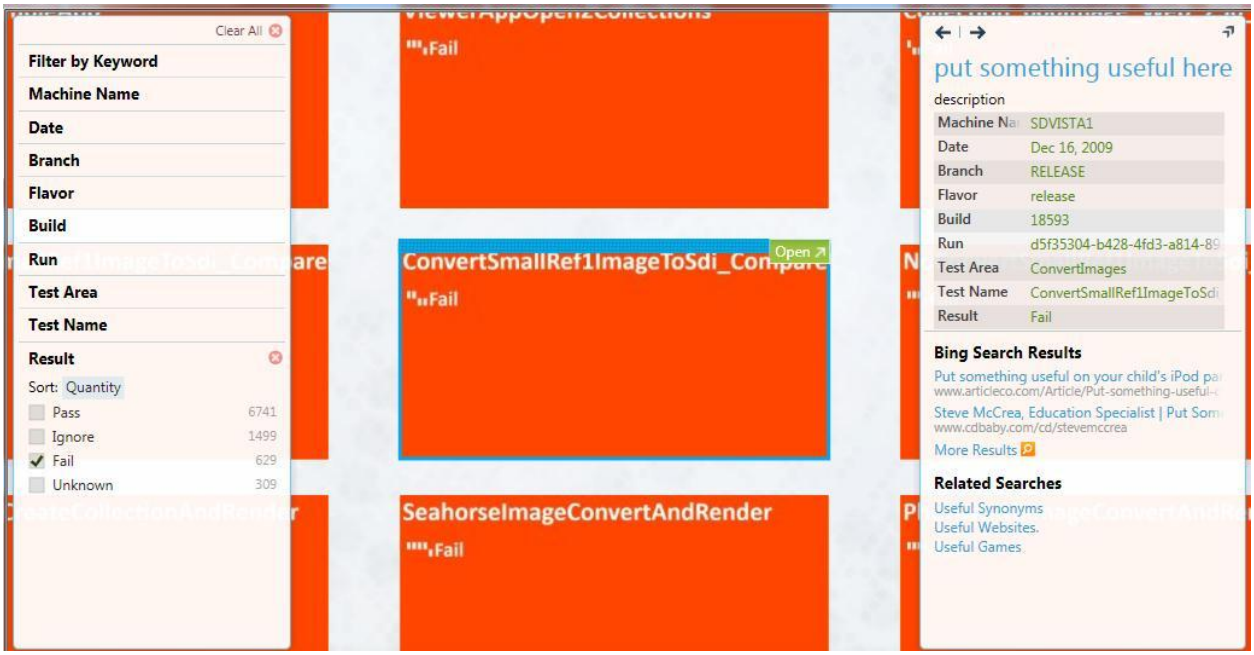

Figure 8 – Single test run result – note sparkline indicating trend of pass/fail over the last several runs. Each tic represents an individual test run for that case. The test case in the center has two passes, then two fails in a row.

4. **Code Coverage**

Code coverage analysis is often cumbersome to interpret. With the information put into PivotViewer and a simple complexity calculation, it becomes apparent very quickly where test coverage should be added.

The code coverage collection's utility came from the fact that people not used to a domain-specific tool such as Sleuth (Sleuth.exe is the executable for Microsoft's Magellan code coverage toolset) could immediately access the information. This removed the learning curve for interpreting code coverage information. Also, the ability sort and filter without authoring complex queries really made a difference in how quickly people could explore the data from various angles.

The reason we wanted a graphical representation of the data is because we wanted testers to be able pull up data quickly to determine where to focus their testing. Historical data shows that Sleuth was so cumbersome in this regard that testers we were deferring analysis and reporting to select individuals who could extract that information instead of using it themselves on a daily basis.

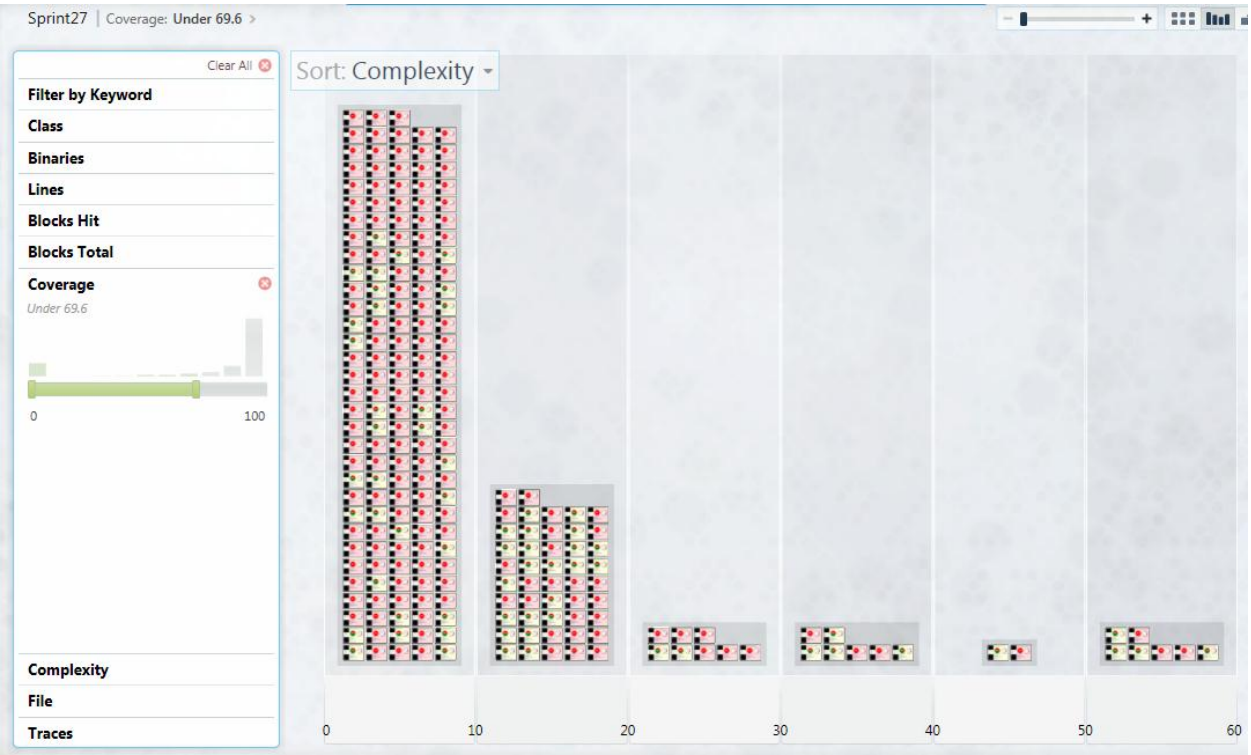

Figure 9 – Code coverage – what code should I write automation for next? The code on the right side of this histogram! Green, yellow, red (tested, partially tested, not tested) blocks distributed roughly in a ratio of 5:3:2 shows we're on track for hitting 70-80 code coverage goals. When sorted by descending complexity, we expect to see green in the high ranges and red mostly in the low ranges, indicating that risky functions are mostly tested as those would be high priority targets for testing. In this figure the view is already drilled down to the functions that have less coverage than we'd like, sorted by "Complexity".

Since the collection is color coded and uses a gauge to represent complexity, it is immediately apparent when new functions were added which were not tested (seen as clusters of red) and how much risk those represented (if the red blocks' gauge needles were high, we had a problem). See figures 9 and 10.

When sorted by file or function name, red blocks (functions not adequately tested) showing up in clusters indicates inadequate test coverage across an entire feature or component. In practice on the Seadragon project, this has actually identified new code which was not described in the check-in mail and hence was completely off the radar. This visualization raised red flags for the test team and resulted in features not relevant to the sprint being reverted from the source tree.

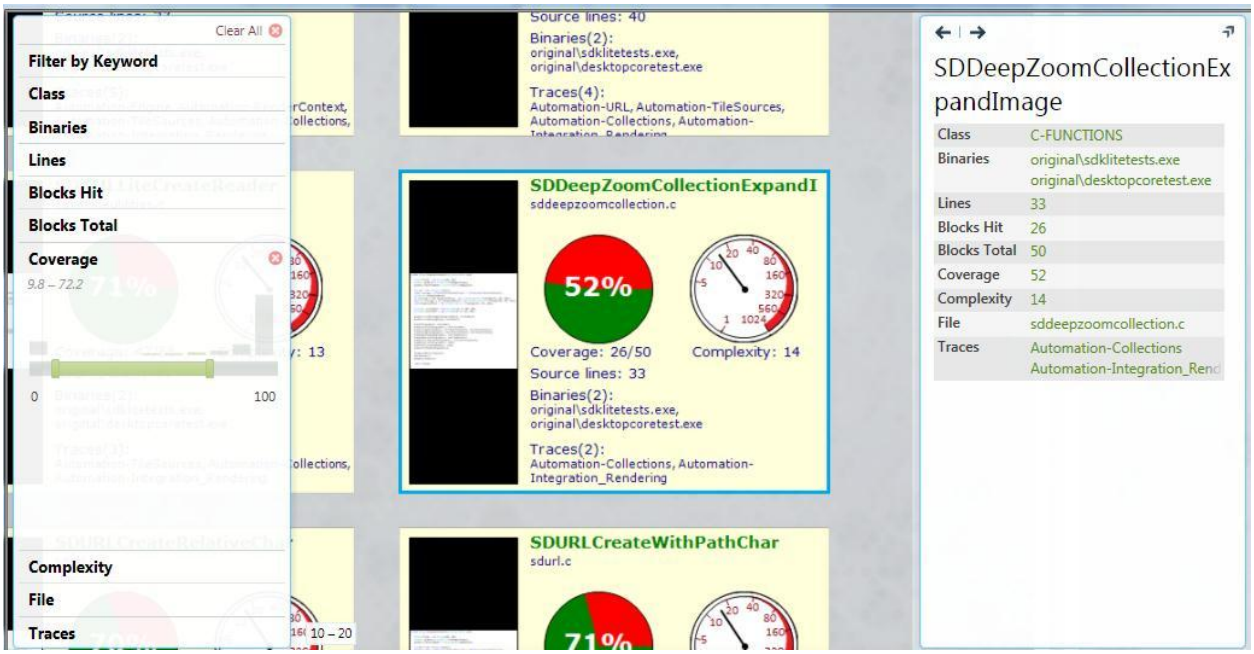

Figure 10 – Code Class with Complexity meter and code displayed

Once the PivotViewer code coverage collection was created and automated, new collections required approximately twelve minutes to be generated. Initial, one-time costs for setting up the system was approximately three days of work by one test engineer. The bug database collection had a one-time cost of five days for one tester to develop and either a few minutes or a few hours to create new collections depending on the size of the result set. The largest set of test results was 40,000 items.

In these three example collections, we've gone through three stages of learning.

1. Understanding of data quality. You can visually see where there are holes in the data. You can quickly identify what information your team needs to gather. For example, in the Bugs collection above, it was quickly obvious that my team wasn't using the Severity field consistently.
2. Visualizing trends you expect. Easily viewing any number of trends that you expect to see. For example, you may expect to see a certain shape for incoming defect reports – you can view this visually in a histogram easily and either see what you expected or be surprised by a shape you didn't expect to see. In the Bug collection, the incoming rate matched what I expected to see for the releases we had during that time.
3. Discovering trends you don't expect. With faceted browsing and searching and a visual display, it's easy to "swim around" in the data. While doing so, it's very common to discover interesting trends that you never thought to look for. An example of this was finding bugs that were resolved Fixed without

going through the proper Triage process. Another example given above was discovering code that was checked in that wasn't part of the scheduled work and therefore, wasn't yet tested.

4. How to put your own data into the PivotViewer experience

Since the following information is already covered in great detail at http://www.silverlight.net/learn/PivotViewer/, I'll include just enough here to allow the reader to understand the scope of building their own collection and refer the reader to the above URL for more detailed information. The requirements to host your data in a format that is viewable by PivotViewer are variable and depend on the type and more importantly the size of the dataset as well as whether the information needs to be real time.

4.1. Kinds of Collections

There are three primary kinds of collections. They differ in size and their ability to respond to custom user queries. Structurally, they are composed of either previously generated (or *static*) XML, or XML generated *dynamically* in response to a query. Simple collections are relatively easy to build and are recommended for collections with 3000 or fewer items and fairly static dataset that doesn't need to be updated very often. Linked collections are larger collections that are built by linking many simple collections together. It is useful for datasets that are static in nature because it's fairly complicated to update these collections. I don't recommend creating Linked collections and I'll explain just two types in the paper. Just in Time collections are recommended for large datasets that change often. A sample Just in Time server has been made available to the public. The examples in this paper represent both simple collections and Just in Time collections. The bug database collection is a Just in Time implementation. The test run results and code coverage results are simple collections because they are experiments to test the experience before investing more to incorporate them into our team's workflow. In practice, however, these collections should all be Just in Time implementations. For more detailed information on building collections, please refer to the documentation here: http://www.silverlight.net/learn/PivotViewer/collection-design/

For a Just in Time server implementation, like the Bug Database collection presented here, the focus is on creating the code that connects to your data-source and translates it to a set of items containing properties. If your data is already in tabular form, then this is straight-forward as each row maps to an item and each column maps to a facet value. However, you might want to combine multiple tables or perform aggregate calculations on multiple pieces of data.

Here are the steps to necessary:

1) Write the code to generate a list of items, each of which will have multiple facet values. If your source is a database, these might map to rows and columns in a particular table. However, you might want to join table or use a web-feed or middleware API to generate these items. Item facets might map directly to fields from your data source, or they might be calculated from aggregated data from multiple fields or tables. For example, a "Customer" item might have a "Number of sales" facet that is actually the sum of their entries in the "Orders" table in your database.
2) Create a XAML template that will visually represent your data. This should include template entries for text that you want to appear on the card, perhaps a customer's name or the title of a

bug. It can be visuals that vary based on the source data – perhaps a different color for different facet values, or an image embedded from a URL taken from a hidden facet value.

Once these two steps are performed, you should be able to deploy them to your JIT server instance and test. The JIT server requires ASP.NET and requires very little setup.

This snipped from the white paper is also pertinent: http://www.getpivot.com/developer-info/jit-tools.aspx

4.2. Collection Design

As with simple and linked collections, the PivotViewer allows you to quickly visualize the information within each collection view returned by a just-in time collection server. With this in mind, choose an appropriate number of facet categories and design your facet values so that applying them slices the view in ways that highlight interesting aspects and trends in the data. Select visual representations that are appropriate to the level of resolution displayed at any one time and which consistently support the message and insights your collection application is attempting to deliver.

With a just-in-time collection server, typically each collection view represents just a fraction of the items available in the larger backing data source. With this in mind, design your collection to include hyperlink facets that enable someone to launch a new collection view related to the current item in some way. For example, a collection view of household products containing an item for “table cloth” may provide multiple hyperlinks in the Info Pane that allow navigating to collection views associated with this item, e.g. “dining room”, “linen”, “matching items”, “blue”, “$10-$20”, or even provide actions such as “Add to cart” or “Send as email”.

With the authoring tools we have available, one can create a prototype collection that’s viewable in the Pivot client (www.getPivot.com) within an hour or so. More sophisticated collections that require a Just in Time server can take a few days to create and some backend infrastructure.

Creating a simple collection has four distinct steps:

1. **Pick your data** - First, pick a set of data to turn into a collection and decide how you want to present it. For tips, see Collection Design*. Any test data you collect is potentially interesting to put into PivotViewer. My team has created collections for Bug, Test Results, and Code Coverage. Other collections could be built with Performance, Stress, Load, or practically any data you generate while testing.
2. **Create XML and images** - Once you have your data sources, you’ll need to describe it in Collection XML (CXML) and transform your images to the Deep Zoom format. We’ve built a variety of tools* from an Excel plug-in to an open-source software library, or you could build your own. See Collection XML Schema*, Collection Image Content*, and Collection Design* for detailed information.
3. **Host it** - To share your collection with others, host it on a web server. For more information, see Collection Hosting*.
4. **Share it** - Download our PivotViewer SDK* in order to build a Silverlight control to host the collection on your webpage!

*Links in this section can be found here: http://www.silverlight.net/learn/PivotViewer/

4.3. Architecture

Conceptually, a collection is just like any other web content. There's a set of files on a server, and a local client that knows how to display them. In the current web, the files are traditionally HTML and images. In the collection case, the files are CXML and Deep Zoom-formatted (DZC) images. When the user browses the collection from a web page, the PivotViewer will use the Silverlight Control to display the files. See the following figure.

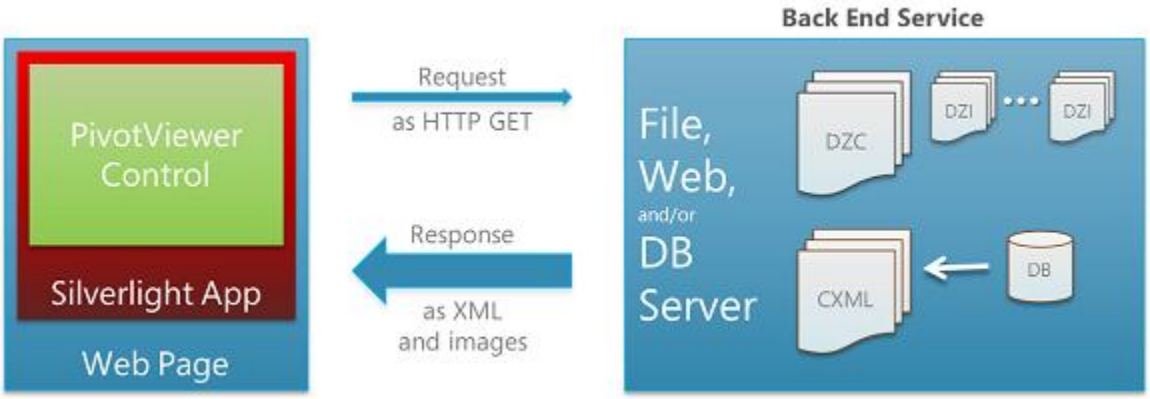

Figure 11. Basic architecture

5. Summary

I've offered a variety of ways we've thought to use PivotViewer in our testing with different sources of data as examples. Then I gave an explanation of how we built these collections and about how long it took us to create them. There are also developer documents that I've provided links for in case you need more technical information to build your solution.

PivotViewer Silverlight control and the Pivot client are available free today and you can build solutions on top of the control. I believe that the use of PivotViewer by testers throughout the industry will raise the bar for quality and give users better products.

I'd love to hear of any solutions you come up with on Getsatisfaction.com.

http://getsatisfaction.com/live_labs_pivot/products/microsoft_pivot_pivot