Backstory: A Search Tool for Software Developers Supporting Scalable Sensemaking

Gina Venolia
Microsoft Research
One Microsoft Way, Redmond, WA 98052 USA
gina.venolia@microsoft.com

ABSTRACT

Software developers have many information needs which could be answered using the various repositories at their disposal, but they underutilize these knowledge resources for a variety of good reasons. Backstory is a search tool for software developers aimed at addressing those reasons, and so to improve knowledge flow among teammates. This paper presents as background the results of a survey of developers' current search habits and desires for a new tool. A case study of root-cause analysis is also presented, which informs the design of Backstory and adds detail to an accepted model of sensemaking. The Backstory UI is described with respect to the needs identified in the survey and user study. Finally, the Backstory UI design suggests that a tool can support sensemaking in such a way that it's not intimidating or distracting for simple investigations, yet has mechanisms that the user may employ incrementally as the complexity of an investigation increases – a characteristic referred to here as *scalable sensemaking*.

INTRODUCTION

Software developers maintain rich mental models of the systems that they create [4]. However these models are always incomplete and sometimes wrong. The source code is the primary external representation of the system, but, surprisingly, it too is incomplete – developers know much about the code that's not expressed in the code. Some of this knowledge is captured in written form, e.g. in email threads, check-in messages, dialogs attached to bug reports, specifications, and design documents.

Developers often need information about a system's behavior or design [3], such as, *What code could have caused this behavior?*, *Why was it implemented this way?*, and *What are the implications of this change?* When faced with such a need a developer first turns to his mental model. If that proves insufficient then he typically turns to the code. Should that fail to provide an answer he then turns to a coworker. If that fails then he may, as a last resort, turn to searching the various repositories at his disposal.

It is not difficult to understand why the written form may be a last resort. Information is spread across many repositories, each with its own search-and-browse UI. There is poor "information scent" to indicate whether a repository contains *anything* pertinent to the current information need, or if a particular search result contains relevant information. If information is found, it is difficult to assess whether it is hopelessly out-of-date or is the latest word on the subject.

In this paper I will describe a search tool for software developers, called *Backstory*, which aspires to address these problems. My goal is to help developers make better use of the written resources where knowledge may be lying fallow, and so reduce the need to interrupt teammates and increase knowledge flow within the team. Its core, Backstory is a federated search tool with excellent previewing. Additional features support small- and medium-scale sensemaking investigations. Ideally, Backstory will make simple information needs easy to satisfy, make complex needs possible to satisfy, and scale gracefully between these extremes.

Before describing Backstory's UI, I will describe two formative field studies of developers' search behavior.

SURVEY OF DEVELOPER SEARCH NEEDS

I (along with Rob DeLine and Eric Price) recently surveyed developers in a product division of Microsoft Corporation regarding their current use of search tools and their needs for future search tools. We deployed the survey to 300 developers and received 97 completed responses. Of the respondents, 77% were individual contributors, 18% were leads, and 5% were managers. Respondents' median tenure at Microsoft was 4.4 years and the in the division, 4.0 years.

We asked about the various reasons for searching over the source code (see Figure 1), developers responded most strongly to *How do I use this function?* and *How does the code work?* They responded less strongly to *Why is the code written this way?*, and least strongly to *Who has worked on this code?*

The source code control system that the division used provided no tool for searching over the check-ins. We asked how often they would search over various aspects of the check-ins, and they responded somewhat strongly, and, interestingly, similarly to all three categories of information in the check-ins (see Figure 2).

We asked them to imagine a search tool that could search over code and other information sources, and asked which

Gina Venolia. Backstory: A search tool for software developers supporting scalable sensemaking. Microsoft Research Technical Report MSR-TR-2008-13. January 2008.

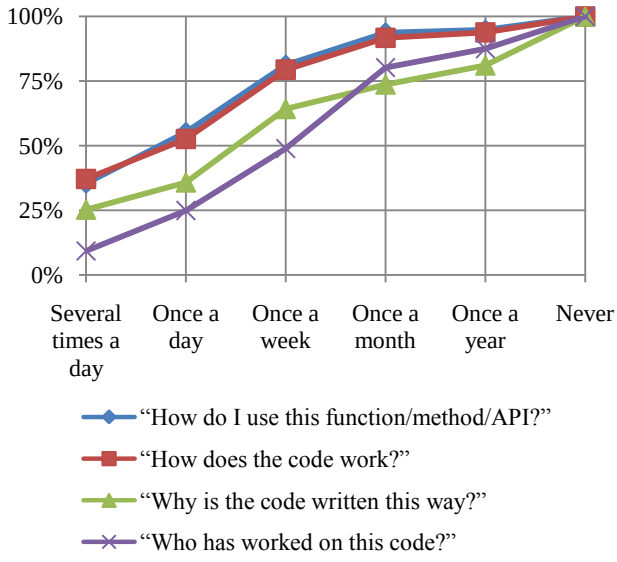

Figure 1: Cumulative survey responses to, "How often do you use search tools to answer the following kinds of questions?"

repositories they would use (see Table 1). A majority of respondents would search over the bug database and the specifications.

Together these results suggest the importance of search to developers and their interest in multi-repository (e.g. federated) search. But the survey results say little about *how* developers search.

CASE STUDY: ROOT-CAUSE ANALYSIS OF SOFTWARE DEFECTS

One task that relies heavily on in-depth exploration is *root-cause analysis*, or *RCA*, which is the process of finding the reasons for critical failures [1]. I interviewed one of the people at Microsoft responsible for RCA of key software defects, in hopes of finding a model for how developers might search.

The RCA process began when a particular incident was identified as worthy of investigation, e.g. a build break or a critical bug regression. He chose some keywords from the incident description and searched for them over the various repositories at his disposal, including the bug database, the

Table 1: Survey responses to, "What other information sources would you be interested in searching?"

Repository	% "Yes"
Bug database	74%
Specs	61%
Online crash dumps	48%
Tests	39%
Dependency data	37%
Team email	30%
Blogs	30%
Intranet sites	29%
Internet	21%

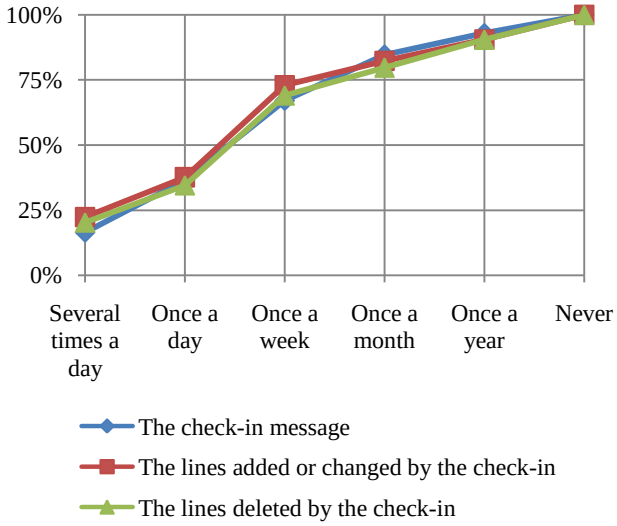

Figure 2: Cumulative survey responses to, "If you could search over check-ins, how often would you look for check-ins by..."

product-support knowledgebase, email repositories, project-specific databases, code check-ins, etc. Each repository had its own unique search interface, so this process was quite tedious. He refined his query to improve the result set. He then scanned the result list to find potentially-relevant documents, which he opened and examined.

For each relevant document he created an entry in a structured Microsoft Word document, clearly identifying its date, the document type and name, often a quotation from the document, and often his own notes about the document and/or questions for further investigation. These entries were sorted chronologically, forming a timeline. The Word document was his working "notebook," and was generally not shown to others.

In reading the source documents he sometimes encountered additional terms, people, date ranges, etc. which lead to more searches, which in turn lead to more documents, which then lead to more entries in the timeline. During this process he also created and discarded hypotheses, which he retained in another section of his "notebook" document.

In addition to his electronic investigations, he interviewed key participants in the incident. These interviews sometimes resulted in the addition of entries to the timeline and clarifying comments for existing entries.

His final report was a separate document, in which he presented a narrative of the incident and a summary of its root causes. The former was based on the timeline and the latter on the hypotheses.

One of his investigations was of a particular build break. The timeline for this investigation contained about 50 entries representing documents or events that spanned 48 days. The final report identified 21 issues and three root causes. He reported that the investigation, including the

electronic sleuthing, interviews, and writing, took him "a couple of weeks" of full-time work.

RCA AS SENSEMAKING

Figure 3 shows an idealized schematic representation of the RCA workflow. (Real-world activities, such as interviews, can provide new keywords, documents, notes, and hypotheses, but are not depicted in the schematic.) The idealized schematic shares many features with Pirolli and Card's schematic of the notional sensemaking loop for intelligence analysts [5]. The schematic follows directly from the previous section, so I won't describe the whole thing, but will call out some interesting features.

- The Search/Refine loop is another way of saying "query refinement" [7].
- As there are multiple searches executed, the collection of search results is, conceptually, the union of many query result sets.
- The box labeled *Exploration* corresponds to the Pirolli and Card's *Foraging Loop*. The *Search Results* box corresponds to their *Shoebox*, and the *Relevant Items + Annotations* box corresponds to their *Evidence File*.
- Triage is a notable part of the RCA process but is not an explicit part of their model.
- Items may be found both directly through search and by browsing from search results. Items may contain additional query terms, so there is loop formed between searching and browsing.
- In the RCA case study, I did not identify any schematization beyond a simple chronological list of evidence.
- Annotation of evidence was an important aspect of the RCA process.
- The hypotheses, with their supporting and refuting collections of evidence, correspond to Pirolli and Card's *Hypotheses* box.
- The final report corresponds to their *Presentation* box.

So RCA for software defects seems to parallel Pirolli and Card's model of sensemaking loop for intelligence analysts, with only slight deviations. Figure 3 adds a few details to the operationalization of the process, most notably the query refinement loop, the triage steps, annotations, and the explicit interplay between searching and browsing.

The extent to which this model applies to typical developers search behaviors is not known. Perhaps some fraction of information needs can be satisfied with a single pass through the *Exploration* loop. Looping in the exploration process may be required only when the correct query keywords are not known *a priori*. The explicit representation of hypotheses and creation of a report might be required only in special circumstances. It is likely that a developer doesn't initially know the extent of his search needs, so being able to gracefully engage the tool to support these more-complicated functions as needed. I call this graceful engagement of features *scalable sensemaking*.

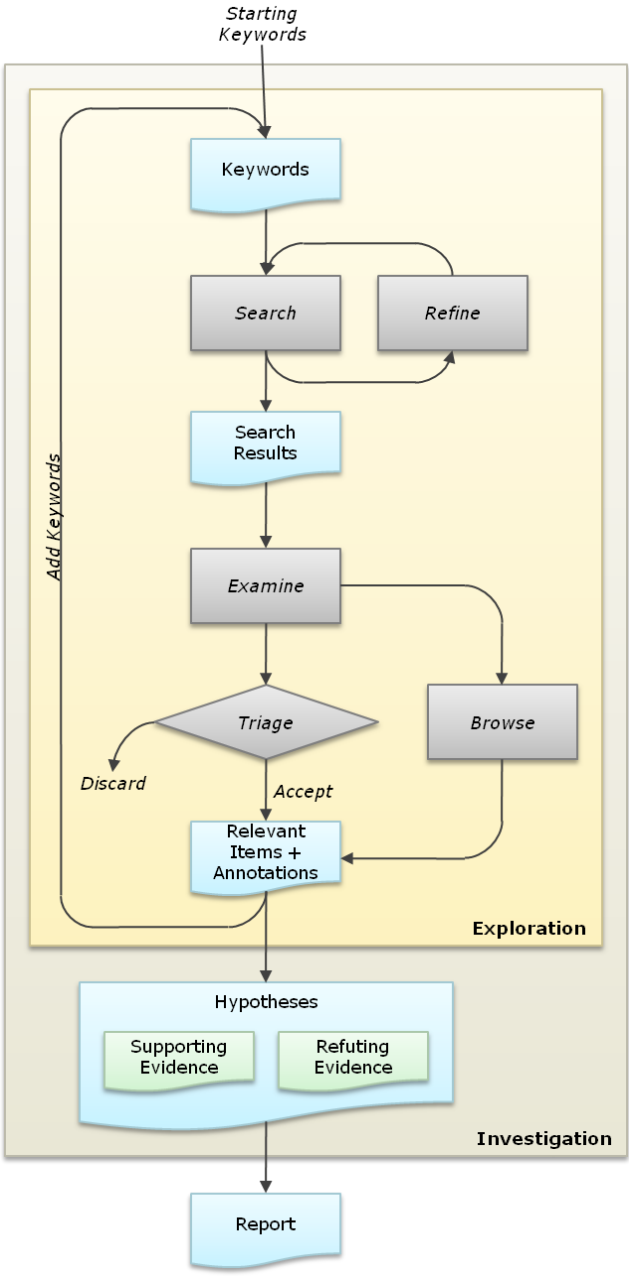

Figure 3: An idealized schematic of the root-cause analysis workflow.

BACKSTORY UI

From this background, I identified a set of requirements for a search tool for developers. The user should be able to:

- *Search* across multiple repositories,
- *Combine* multiple queries without losing context,
- *View* results with high-quality summaries,
- *Triage* to separate the wheat from the chaff,
- *Preview* and *browse* in-place,
- *Transition* seamlessly between searching and browsing,
- *Organize* using tags and annotation,
- *Suspend* and *resume* an investigation, and
- *Refresh* existing queries.

Figure 4 shows the Backstory user interface, which consists of three panes: query editor (top-left), search grid (top-right), and preview (bottom).

Search. To begin, the user types the keywords into the box at the top of the query editor, selects the data sources to search, specifies any query parameters particular to the selected data sources using the controls boxed under the checkbox (such as CodeCOOP in the figure), and then clicks the *New* button to execute the queries. Each query is represented by a column in the search grid. As results are returned asynchronously by the queries, they appear as rows in the search grid. A checkmark appears in a cell if the row's result was returned by the column's query. A row may have multiple checkmarks if the result was returned by multiple queries.

The first 100 results are fetched for each result, and the first ten are actually displayed. The 90 hidden results may decorate other queries' results with checkmarks. The user may display more results for any query by right-clicking on the column header and clicking *Get More Results*, whereupon the next 100 are fetched and the next 10 are shown.

There is an underlying plug-in model for data source types, and the user can add a new data source at any time by clicking the *Add…* link in the query editor pane and then going through a configuration wizard. Some data sources types have parameters which can be configured using the wizard. For example a particular bug database must be specified, and internet or intranet search may optionally be scoped to a particular site.

Combine. The user may initiate follow-up queries by typing different keywords into the box at the top of the query editor and then clicking *New*. Unlike traditional search engines, where the previous result set would be lost, Backstory creates additional columns to represent the new queries and additional rows to represent the new results. If a new query returns a preexisting search result, a new checkmark appears on the existing row. The user may perform query refinement by clicking the *Modify* button instead of *New*, and so the new queries replace the old instead of adding to them.

View. Each result is shown with a brief textual summary inspired by the ubiquitous web search experience. If not provided by the data source, a summary is generated from the text contents of the document referenced by the search result. All query keywords are highlighted in the title, summary, and location fields of the search result rows. Highlight colors are determined by a hash, so a given word is always highlighted in the same color.

Triage. The user may remove an irrelevant search result by clicking the "thumbs-down" button; the result immediately disappears from the normal view. Even if a subsequent query returns that result, it won't appear again. Clicking the "thumbs-up" button marks the result as relevant to the current investigation. The left and right cursor arrow keys allow the user to quickly mark the selected result relevant or irrelevant and then advance to the next result, enabling rapid triaging of search results.

Borrowing from the visual language of email triage, there is an "unread" status bit which is reflected by title of the result being rendered with a bold font. Unlike the global link coloring used in web browsers, the unread bit is per-investigation, and may help the user triage items even when not using the "thumbs" buttons.

The rows of the search grid may be sorted by a number of factors. The default is to sort so that new items appear at the bottom of the list, which provides a predictable order much like sorting the email inbox by date. The user may sort chronologically by the item's creation date, which mimics the chronological ordering of evidence found in the RCA case study. The items may be sorted by textual similarity to the thumbs-up set (and dissimilarity to the thumbs-down set), based on the words in the title and summary; this order is useful for triaging a large number of items from multiple queries. The user may sort by one or more queries. As a natural consequence of sorting by multiple Boolean columns, items are sorted by the Boolean logic table combinations of the columns. For example if sorting by one query and then another, the items that result from both queries appear at the top, followed by those that came from the second but not the first query, then the first but not the second, then the items that came other queries.

Preview and browse. The preview pane shows a view of the selected item, whose implementation varies by the type of document indicated by the search result. Source code is previewed with syntax coloring. Many of the preview implementations support query keyword highlighting, using the same colors as in the search grid. Buttons at the top of the preview pane allow the user to advance to the next or previous highlighted query keywords, making it easier to find the salient parts of large documents.

Transition. For HTML documents the preview pane is a limited web browser, so links may be followed *in situ*. The user may use the thumbs-up and -down buttons in the HTML preview pane to mark the current URL, adding it to the investigation if it's not already there.

Organize. The user may annotate any search result by invoking a dialog from the context menu and typing some text. The user may add any number of *tags* to the investigation. A tag acts as a mathematical set. Each tag is represented by a column of checkboxes in the search grid. The checkboxes control membership in the tag's set. The user may sort by tag set membership.

Suspend and resume. The user may save the entire state of the investigation (queries, results, "thumbs" marks, "unread" state, tag membership, etc.) to a file on disk, which may then be shared or opened later.

Refresh. If substantial time has elapsed since the queries were executed, the state of the repositories may have changed. The user may issue a command to *Refresh All Queries*, which re-executes all the queries and then merges the new result sets into the investigation. New items in the results are marked "unread" and so are highlighted in bold; items with no thumbs-mark that are not present in the new result set are removed from view; items marked thumbs-up or are present in the new and old result set are retained unchanged. This feature enables the user to create complicated, multi-query and multi-repository investigations and then poll them to stay informed of "breaking news."

STATUS

Backstory has been available for download on the Microsoft intranet for several months now. Thirty-five people have tried it more than once. I have not yet done any analysis of these early adopters. I plan to do a major "marketing" push soon, which will include a predeployment survey. I plan to interview and survey a sampling of the users to understand how Backstory is used (or why it is abandoned). With these techniques, as well as analysis of usage logs, I hope to address the core research questions that I introduced at the beginning:

- Does it help developers make better use of the textual resources where knowledge may be lying fallow?
- Does it reduce unnecessary interruption of teammates?
- Does it increase knowledge flow within the team?

DISCUSSION

Backstory scales from quick, single-query searches to deep sensemaking tasks involving multiple queries across multiple repositories taking place over long periods of time. The UI mechanisms that support deep sensemaking do not interfere with performing quick searches, suggesting that Backstory may achieve the ideal of supporting scalable sensemaking.

Backstory was developed with software developers in mind, and so I have concentrated the implementation effort on search over code-related repositories and preview of code-related artifacts. There is nothing about Backstory that is limited to developers. Future work will examine its applicability to general web search tasks and to other job roles such as academic, legal, or medical workers.

There is much that can be done to improve Backstory as a UI design. In particular it provides only one view of the data, whereas industrial-strength tools such as Jigsaw [6] and Entity Workspace [2] take great advantage of multiple synchronized views. I am interested in extending Backstory with appropriate complementary views.

My goal primary goals in the project relate to improving knowledge flow among developers, making use of existing knowledge resources, and reducing unnecessary interruptions. As a side effect to my primary goals, I am learning interesting things about the sensemaking model, such as the importance of query refinement, triage, annotations, and the explicit interplay between searching and browsing. The notion of *scalable* sensemaking is, I think, an important and novel contribution, which may help make tools built from the sensemaking model approachable to the general population.

REFERENCES

1. Andersen, B. Root cause analysis: Simplified tools and techniques. ISBN 978-0873896924. ASQ Quality Press, 2006.
2. Bier, E., E. Ishak, and E. Chi. Entity workspace: An evidence file that aids memory, inference, and reading. In *Proc. ISI 2006*.
3. Ko, A., R. DeLine, and G. Venolia. Information needs in collocated software development teams. In *Proc. ICSE 2007*.
4. LaToza, T., G. Venolia, and R. DeLine. Maintaining mental models: A study of developer work habits. In *Proc. ICSE 2006*.
5. Pirolli, P. and S. Card. The sensemaking process and leverage points for analyst technology as identified through cognitive task analysis. In *Proc. International Conference on Intelligence Analysis 2005*.
6. Stasko, J., C. Görg, Z. Liu, and K. Singhal. Jigsaw: Supporting investigative analysis through interactive visualization. In *Proc. VAST 2007*.
7. Vélez, B., R. Weiss, M. Sheldon, and D. Gifford. Fast and effective query refinement. In *Proc. SIGIR 1997*.

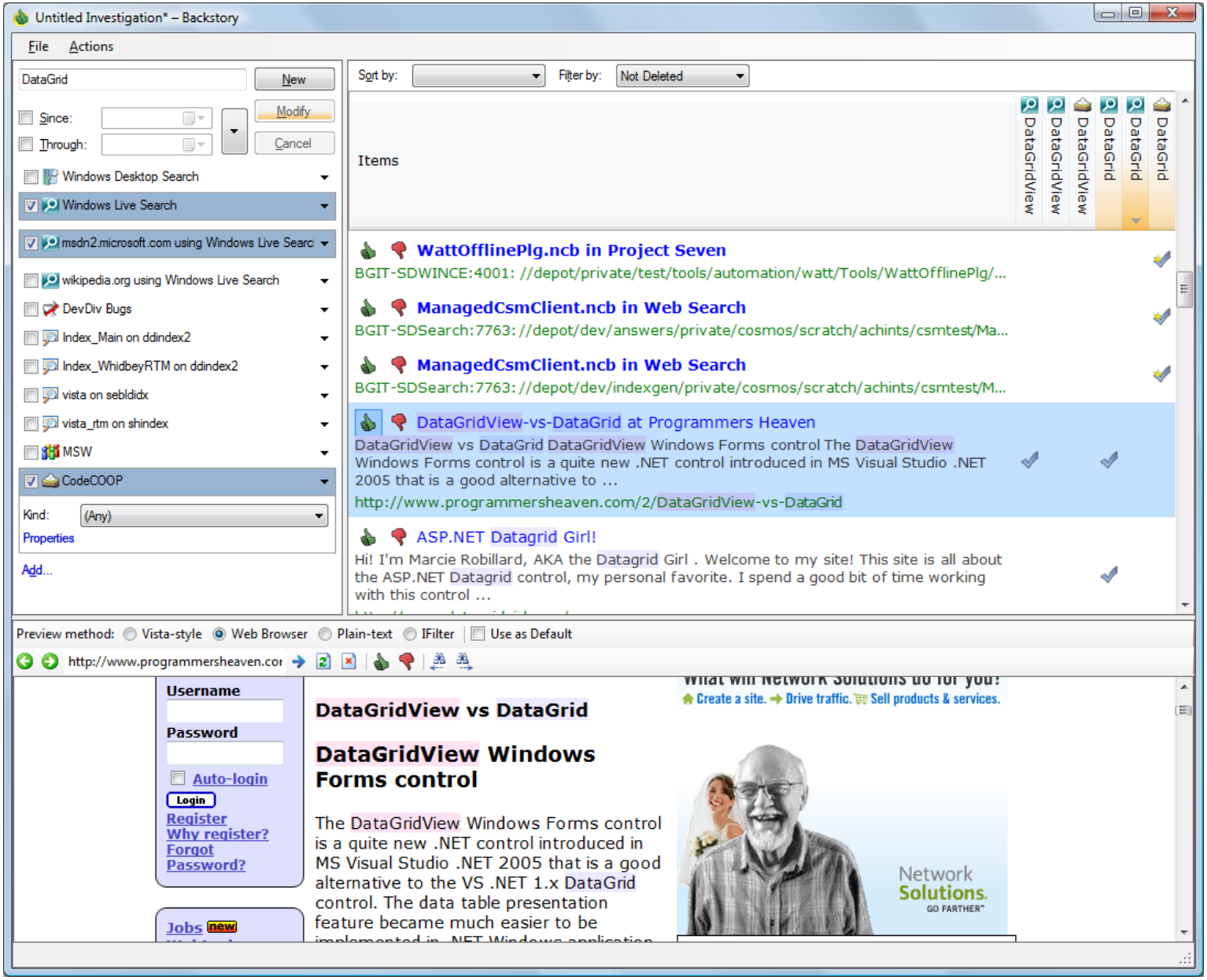

Figure 4: A screen shot of Backstory showing the three major panes of the UI and an investigation, which consists of two independent queries ("DataGridView" and "DataGrid") each federated across three repositories (Windows Live Search, Windows Live Search on a particular site, and a Microsoft-internal repository called CodeCOOP). Checkmarks show which queries produced which results.