Debugger Canvas: Industrial Experience with the Code Bubbles Paradigm

Robert DeLine, Andrew Bragdon, Kael Rowan, Jens Jacobsen
Microsoft Research, Visual Studio Ultimate
Microsoft Corporation
Redmond, WA, USA
{rdeline, anbrag, kaelr, jensj}@microsoft.com

Steven P. Reiss
Department of Computer Science
Brown University
Providence, RI, USA
{spr}@brown.edu

***Abstract*—At ICSE 2010, the Code Bubbles team from Brown University and the Code Canvas team from Microsoft Research presented similar ideas for new user experiences for an integrated development environment. Since then, the two teams formed a collaboration, along with the Microsoft Visual Studio team, to release Debugger Canvas, an industrial version of the Code Bubbles paradigm. With Debugger Canvas, a programmer debugs her code as a collection of code bubbles, annotated with call paths and variable values, on a two-dimensional pan-and-zoom surface. In this experience report, we describe new user interface ideas, describe the rationale behind our design choices, evaluate the performance overhead of the new design, and provide user feedback based on lab participants, post-release usage data, and a user survey and interviews. We conclude that the code bubbles paradigm does scale to existing customer code bases, is best implemented as a mode in the existing user experience rather than a replacement, and is most useful when the user has a long or complex call paths, a large or unfamiliar code base, or complex control patterns, like factories or dynamic linking.**

***Keywords*—integrated development environments; user interfaces; human factors; experience report**

I. Introduction

At ICSE 2010, two groups presented novel user experiences for integrated development environments (IDEs), designed along similar lines. Code Bubbles[*] [1] [2] from Brown University and Code Canvas [3] from Microsoft Research both replace the IDE's typical set of tabbed documents and tool windows with a pan-and-zoom surface that hosts a software project's code and related artifacts. Both designs present the code as a collection of individual definitions, called "bubbles" or "fragments", annotated with relationship information, like call lines between methods. The goal of both designs is to gather all the information a programmer needs to complete a task into a single, spatially stable display. The main difference between the designs is that Code Bubbles displays a programmer's working set of definitions, for example, the set of definitions visited during a debugging session; whereas, Code Canvas presents the complete set of a project's definitions, using a map metaphor.

Given the similarity of the designs, the researchers from both groups formed a collaboration, along with members of the Microsoft Visual Studio Ultimate team, to produce an industrial version of this new user experience. Our team of eight, including three members of the Code Bubbles and Code Canvas teams, spent nine months designing, building, testing, and user testing an extension to Visual Studio. In June 2011, we publically released the result, called Debugger Canvas, which has had over 14,000 downloads and many daily users. This paper presents an experience report on the design and implementation decisions behind Debugger Canvas and the lessons learned so far from public adoption.

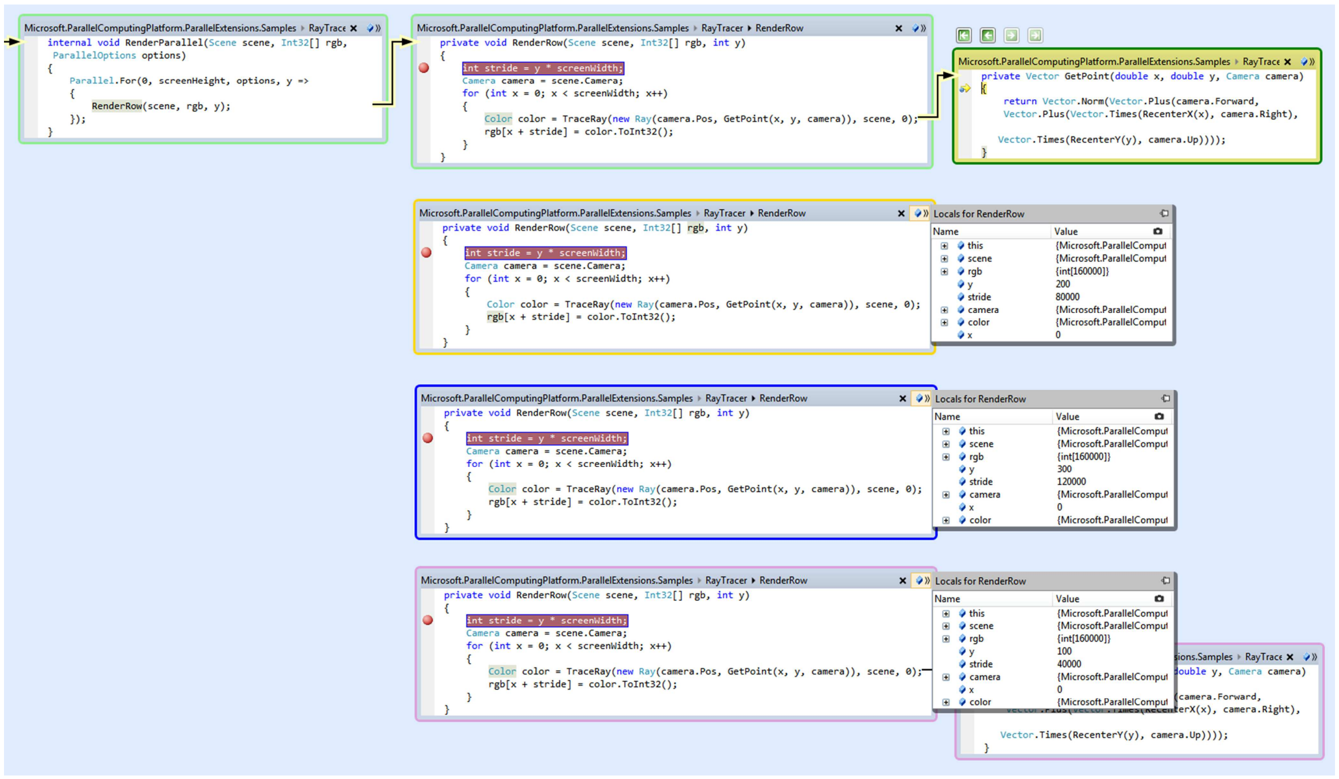
Our main goal in releasing Debugger Canvas was to gather public feedback on this new style of user experience. Our strategy to encourage adoption was to strike a balance between allowing the user to experience the bubble design for enough time to form an opinion while also allowing the user to remain comfortable in the existing user experience most of the time. We focused on use of the debugger since a debugging session often lasts for several minutes and is a separate experience from the rapid edit-compile-run cycle that many developers are hesitant to change. Debugging is also the kind of cognitively intense and navigation-heavy activity that the bubbles design is intended to help.

Figure 1 shows a screen shot of Debugger Canvas. With Debugger Canvas, when a programmer starts to use the debugger—for example, by hitting a breakpoint—Visual Studio creates a canvas and displays the executed method in a bubble on the canvas. As the programmer steps through the code, Debugger Canvas opens each executed method in its own bubble and draws arrows to represent method calls. Each bubble has a pop-up that shows the current value of the local variables, which can be snapshotted for comparison over time. Because the task of debugging also involves code exploration and trying potential bug fixes, Debugger Canvas also supports code navigation features, like go-to definition, and editing with the bubble. Each bubble is a full-fledged Visual Studio editor, with all the typical features like tooltips and code completion.

Our goal of implementing and releasing this product was to answer several questions:

- Can the code bubbles design be implemented with sufficient robustness and performance to scale up to the projects of Visual Studio's customers?
- To what extent would practicing programmers accept the new user experience? Would they prefer it as a mode during some tasks or use it most of the time?

[*] *Code Bubbles* is a trademark of Brown University.


Microsoft.ParallelComputingPlatform.ParallelExtensions.Samples ▸ RayTrace
```
internal void RenderParallel(Scene scene, Int32[] rgb,
 ParallelOptions options)
{
    Parallel.For(0, screenHeight, options, y =>
    {
        RenderRow(scene, rgb, y);
    });
}
```
Microsoft.ParallelComputingPlatform.ParallelExtensions.Samples ▸ RayTracer ▸ RenderRow
```
private void RenderRow(Scene scene, Int32[] rgb, int y)
{
    int stride = y * screenWidth;
    Camera camera = scene.Camera;
    for (int x = 0; x < screenWidth; x++)
    {
        Color color = TraceRay(new Ray(camera.Pos, GetPoint(x, y, camera)), scene, 0);
        rgb[x + stride] = color.ToInt32();
    }
}
```
Microsoft.ParallelComputingPlatform.ParallelExtensions.Samples ▸ RayTrace
```
private Vector GetPoint(double x, double y, Camera camera)
{
    return Vector.Norm(Vector.Plus(camera.Forward,
    Vector.Plus(Vector.Times(RecenterX(x), camera.Right),

   Vector.Times(RecenterY(y), camera.Up))));
}
```
Microsoft.ParallelComputingPlatform.ParallelExtensions.Samples ▸ RayTracer ▸ RenderRow
```
private void RenderRow(Scene scene, Int32[] rgb, int y)
{
    int stride = y * screenWidth;
    Camera camera = scene.Camera;
    for (int x = 0; x < screenWidth; x++)
    {
        Color color = TraceRay(new Ray(camera.Pos, GetPoint(x, y, camera)), scene, 0);
        rgb[x + stride] = color.ToInt32();
    }
}
```
Locals for RenderRow

Name	Value
this	{Microsoft.ParallelComput
scene	{Microsoft.ParallelComput
rgb	{int[160000]}
y	200
stride	80000
camera	{Microsoft.ParallelComput
color	{Microsoft.ParallelComput
x	0

Microsoft.ParallelComputingPlatform.ParallelExtensions.Samples ▸ RayTracer ▸ RenderRow
```
private void RenderRow(Scene scene, Int32[] rgb, int y)
{
    int stride = y * screenWidth;
    Camera camera = scene.Camera;
    for (int x = 0; x < screenWidth; x++)
    {
        Color color = TraceRay(new Ray(camera.Pos, GetPoint(x, y, camera)), scene, 0);
        rgb[x + stride] = color.ToInt32();
    }
}
```
Locals for RenderRow

Name	Value
this	{Microsoft.ParallelComput
scene	{Microsoft.ParallelComput
rgb	{int[160000]}
y	300
stride	120000
camera	{Microsoft.ParallelComput
x	0
color	{Microsoft.ParallelComput

Microsoft.ParallelComputingPlatform.ParallelExtensions.Samples ▸ RayTracer ▸ RenderRow
```
private void RenderRow(Scene scene, Int32[] rgb, int y)
{
    int stride = y * screenWidth;
    Camera camera = scene.Camera;
    for (int x = 0; x < screenWidth; x++)
    {
        Color color = TraceRay(new Ray(camera.Pos, GetPoint(x, y, camera)), scene, 0);
        rgb[x + stride] = color.ToInt32();
    }
}
```
Locals for RenderRow

Name	Value
this	{Microsoft.ParallelComput
scene	{Microsoft.ParallelComput
rgb	{int[160000]}
y	100
stride	40000
camera	{Microsoft.ParallelComput
x	0
color	{Microsoft.ParallelComput

ions.Samples ▸ RayTrace
```
ouble y, Camera camera)
camera.Forward,
erX(x), camera.Right),

   Vector.Times(RecenterY(y), camera.Up))));
}
```

Figure 1. The user stepping through a parallel ray tracing program using Debugger Canvas. All four threads have their own colored borders. The currently executing method has a prominent yellow border. Each code bubble has its own Locals pop-up (gray title bar), allowing state comparison.

- Are there particular tasks or programming situations where practicing programmers would prefer the new design?

In this paper we report on our experience deploying Debugger Canvas. Our contributions include:

- novel features beyond the previous Code Bubbles and Code Canvas papers, including support for debugging concurrent programs and debugging from execution traces;
- the design rationale behind our decisions when merging user experiences from Code Bubbles and Code Canvas;
- an evaluation of the performance overhead of the approach; and
- user feedback on our design from practicing programmers, including both adopters and non-adopters.

II. Debugger Canvas's User Experience

Much of Debugger Canvas's user experience has been documented in the previous papers on Code Bubbles and Code Canvas. Here, we describe a few of the differences and enhancements.

To marry the new and existing user experiences, we implemented the canvas as a tool window that docks side by side with other tabbed documents. The user can create an arbitrary number of canvases, using a "New Canvas" menu item. Whenever the debugger is launched, Debugger Canvas opens any resulting bubbles on the most recently created canvas. This allows the user retain old debugger sessions. Unlike Code Bubbles, the canvases are not persisted documents, but are temporary tool windows, whose contents disappear when the user quits Visual Studio. The canvas contents, however, can be saved and emailed as an XPS document.[*]

We also added two new features beyond previous papers, to provide better support for two difficult debugging situations: debugging based on execution traces and debugging concurrent programs.

A. *Debugging Based on Execution Traces*

Visual Studio 2010 provides a tracing facility, called Intellitrace, which records two kinds of execution events: (1) relatively infrequent, domain-specific events, including user interface events, web server requests, and debugger operations like breakpoints and tracepoints[†]; and (2) all method entries, normal exits, and exceptional exits. An execution trace is displayed as a large, demand-loaded treeview. Debugger Canvas enhances the Intellitrace experience by allowing a user to drag and drop any method in the treeview onto a canvas, which causes the subtree rooted at the dragged method to open as bubbles on the canvas, as shown in Figure 2. To support large traces, we open bub-

[*] Microsoft XPS is a document format similar to Adobe PDF.

[†] A tracepoint is a user-created print statement, inserted in the program through binary instrumentation rather than source editing.

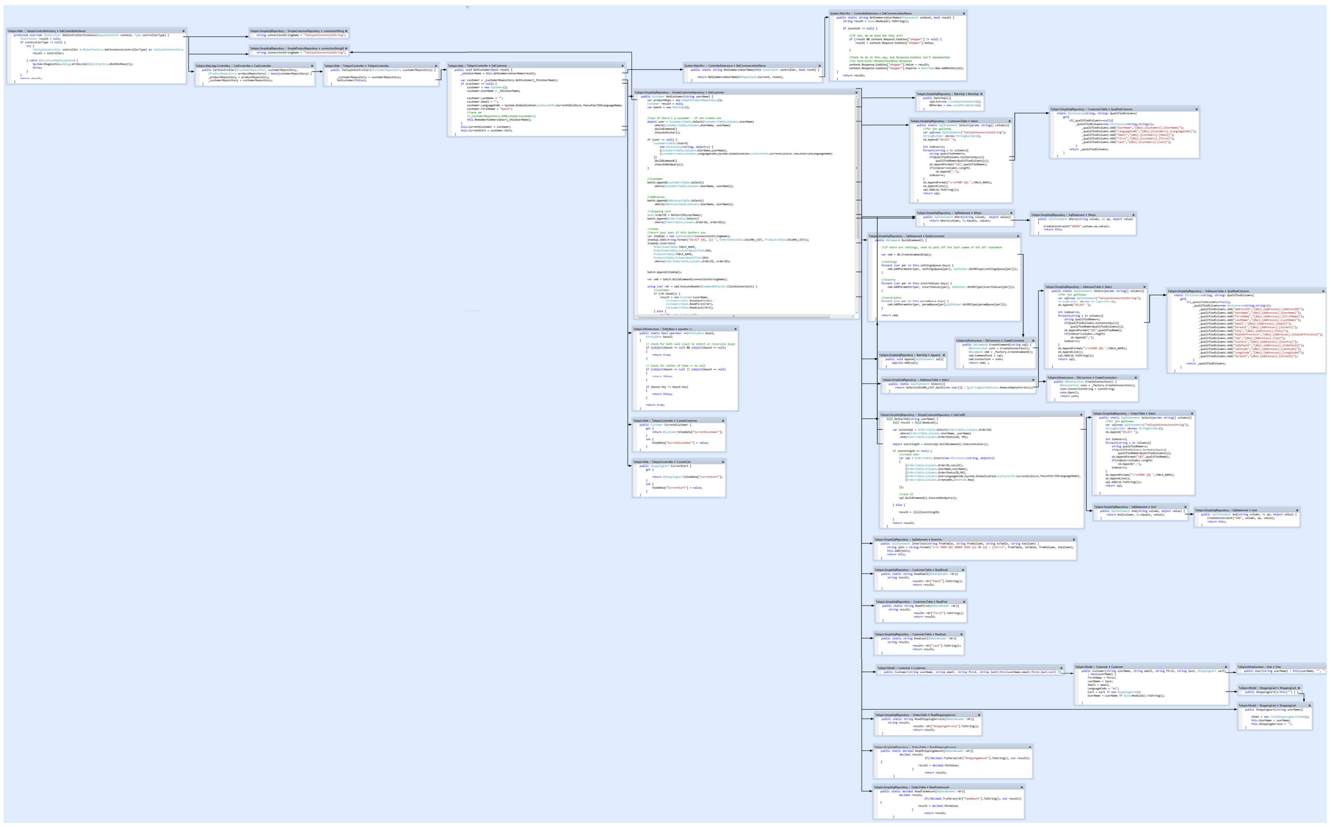


Figure 2. The user viewing a portion of an execution trace for a web site POST request, shown at 25% zoom.

bles asynchronously, which allows the user to work with existing bubbles as new bubbles continue to open.

This feature is useful for exploring unfamiliar code, based on execution behavior of interest. For example, say that a new member of a shopping web site team has been assigned a bug that happens when a shopper adds an item to the cart. The programmer knows the relevant behavior, but not the responsible code. To the find the code, the user turns on tracing, reproduces the behavior and browses a short list of web server requests. The user then selects the request that adds an item to the cart, switches to Calls View, and drags the resulting call tree into the canvas. This causes Debugger Canvas to populate the canvas with all the methods that were executed during that web server request. The user can then use Intellitrace to do "time travel debugging," stepping backwards and forward through the execution in the bubbles, using tooltips to inspect the values of parameters.

B. Debugging Concurrent Programs

Breakpoint debuggers are often difficult to use with concurrent programs, particularly when multiple threads are simultaneously executing the same method's code. In this situation, with a typical debugger, the highlighter bar indicating the currently executing line of code will seem to jump around at random as thread switches occur, and the tool windows only allow one thread's state to be viewed at a time.

In contrast, Debugger Canvas lays out different threads in different horizontal bands, using a different colored bubble border for each thread, as shown in Figure 1. When different threads are executing the same method's code, the user can see a different copy of the method as a different bubble for each thread. Each copy has its own pop-up with local variables and its own highlighter bar for that thread's program counter. In addition to each thread's thin colored border, there is a thick yellow border that indicates the currently executing bubble. When a thread switch occurs, the yellow border switches to the new thread's currently executing bubble and that bubble's highlighter bar advances as the user steps through that thread's code. In short, every thread's own program counter and local state can be viewed simultaneously in the same display.

III. Design Rationale

Although there is a lot of overlap between the Code Bubbles and Code Canvas designs, there are several areas in which they differ. Combining the designs caused us to revisit the rationale and clarify some of the trade-offs. Here we discuss some of the major design issues that any design in this space will face.

A. Historical Debugging and Non-Historical Debugging

Code Bubbles employed a *historical* model of debugging, in which each new function that the user stepped into opened a new corresponding bubble on the screen – even if that function was already visible. In addition, hitting a new breakpoint would also always open a new bubble. This design has the advantage of simplicity and predictability for the user, but it has the disadvantage of opening a large number of bubbles onscreen if the user debugs over an extended period of time. It also means that if a user debugs the same code multiple times, they will see a new trace each time, which is advantageous if the user wishes to compare two sessions, but is a disadvantage if the user simply wants to step through the same path again to see the same values.

With Debugger Canvas, we explored an alternative, *non-historical* design in which bubbles are always reused. In addition, by default we also reuse the same bubbles for a new run of the program, unless the user explicitly opens a new canvas. First and foremost, this removes the notion of historical debugging from the design, as a given configuration of bubbles no longer maps to a single possible execution/debugging path, and indeed, in practice a given layout of bubbles in this design is quite ambiguous. It is notable that this limits the ability for a user to compare two debug sessions, compare values or execution in a single debug session, or share a given debug session with another user. It is also notable that debugging recursive functions with non-historical debugging is essentially identical to debugging them with a traditional file-based editor. Despite these limitations, the advantage of this approach is that less screen space is used, and users who wish to reflexively step through the same execution path may work from a single set of bubbles that is reused. This can provide a lightweight experience for users who need to debug a simple problem repeatedly, as they make changes to the code or environment.

In a product implementation, both options could be made available to the user, perhaps via an easily accessible dropdown at the top of the canvas, so that the user can use historical debugging for tasks that require extensive logging or comparison, and non-historical debugging for tasks that involve short debugging sessions, or minimal comparisons. Indeed, in a later version of Debugger Canvas we added an option to enable historical debugging (turn off non-historical debugging).

B. Tabs and Channels

Code Bubbles also proposed a channels metaphor, in which each debug session is visible in its own horizontal space that can grow as needed, and also be panned as needed as well, facilitating comparison between debug sessions.

In Debugger Canvas we experimented with a simpler design that removes channels completely; by default the same canvas is reused (see above), and the user may create a new canvas that opens in a separate IDE tab manually. This has the disadvantage that the user can no longer rely on being able to always perform a comparison; in Code Bubbles the user can perform a comparison at any time. Conversely, the advantage of this design is that it is simpler for novice users to learn, as it does not introduce a new window management concept, and power users can still drag tabs onto separate monitors if they choose.

C. Call Stack Bubble

Code Bubbles employed a call stack bubble in the debugger to show the parent methods above the current bubble in the call stack bubble. In Debugger Canvas we explore an alternative design in which no call stack is immediately visible in the canvas; rather, a tool window is visible off to the side (outside the canvas) indicating the current execution stack, and the navigation buttons appear above the bubble containing the instruction pointer. These buttons allow the user to incrementally open additional bubbles from the stack. This design was based on the observation that many users wanted to go "up one" in the stack, rather than to a specific method in the call hierarchy. While the Code Bubbles call stack bubble also allows the user to accomplish this, it takes up more space.

D. Integrating Code Bubbles with Traditional File Workflow

The Code Bubbles prototype employed a hybrid design, in which the user was presented with two separate main windows: the Eclipse IDE workbench window for legacy scenarios, and the Code Bubbles IDE window. Commands were added to allow the user to go back and forth between the two, but fundamentally, they remained separate.

In Debugger Canvas, we instead open the canvas which hosts bubbles, as another tab within the main IDE window. This is advantageous in that it is easier for the user to switch between bubbles and files, and the user need only manage a single main window, a single set of toolbars, menus and keyboard shortcuts. However, this incurs several disadvantages: all of the users' tool windows are still visible by default, consuming valuable screen real estate needed to benefit from the concurrent visibility of bubbles, while adding visual noise; in addition, prevailing "legacy" modes of user interaction are now expected to work, making it more challenging to introduce new, bubble-centric modes of interaction, such as gestures. We considered auto-hiding tool windows while a Debugger Canvas tab was active, but we felt this would be too disruptive. In addition, the Debugger Canvas interactions are designed to be consistent with the rest of Visual Studio, where possible, for familiarity; however, this has also limited our ability to incorporate novel gestures.

IV. Implementation and Performance

The Debugger Canvas team consisted of three full-time and one part-time developers, two part-time testers, one part-time user experience designer and one full-time program manager. The first release was developed using an Agile methodology over 12 two-week sprints. After the first release in June 2011, we gathered user feedback, described in the next section. Based on this feedback, two developers then spent another 4 two-week sprints preparing a second release, scheduled for November 2011.

Debugger Canvas is implemented as a Visual Studio extension reusing as much of Visual Studio's existing functionality as possible. In addition to using the existing code editor and debugger, we also reuse the code analysis functionality that is used by the Architecture Explorer in Visual Studio Ultimate. This functionality transforms the different languages (such as C# or VB) into a generalized graph-based code model so that tools like Architecture Explorer and Debugger Canvas can work with any supported language without having to know the specifics of that language.

The user interface and canvas use a model-view-view model (MVVM) architecture with the Windows Presentation Foundation (WPF). This allowed us to quickly iterate on UI design without having to write any graphics or composition code. It also integrated well with Visual Studio 2010's new code editor which uses WPF along with sophisticated virtualization techniques. Panning the canvas remains responsive (even over Remote Desktop) when there are upwards of 100 code fragments on the canvas (which is normally far too many to fit on a standard display).

The biggest decrease in performance relative to standard Visual Studio Ultimate without Debugger Canvas installed is the time it takes to start a debugging session. Debugger Canvas needs to load the graph-based code model provided by Architecture Explorer, which results in a scan of the entire solution when starting to debug. On one test this doubled the average start time from 1.5 seconds to 3 seconds. Note that this measurement is the time after compilation has completed, so the extra delay is less noticeable compared to the much slower compilation time. We tested performance in two conditions: with Debugger Canvas both installed and active (showing code fragments on the canvas when debugging); and when Debugger Canvas was installed but not used during debugging. The relative slowdown of three operations is shown in Figure 3. (As described in the next section, many users complained about performance after the initial release. The figures below are from our upcoming second release, with performance greatly improved.)

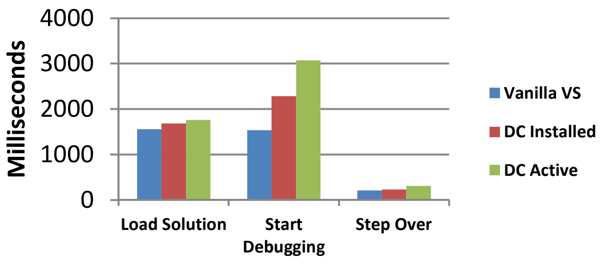


Figure 3. Relative slowdown of three operations.

Stepping through code on the canvas generally takes about 100 ms longer per step regardless of the number of code fragments on the canvas. However, if there are duplicate copies of the same code fragment being debugged (e.g. when debugging a highly recursive method with the "*Reuse Bubbles when Content is the Same*" option turned off) then the performance degrades linearly in relation to the number of duplicate code fragments being shown. On a single core 2.6 GHz Virtual PC this resulted in the debugger taking over 1 second for each 'Step Over' after reaching 70 recursive calls to the same function as seen in Figure 4.

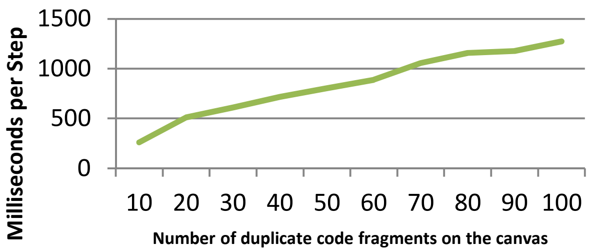


Figure 4. Slowdown in debugger stepping, as the number of bubbles per method increases.

This slowdown only applies when stepping through duplicated code bubbles. When stepping through an unrelated code fragment then performance returns to normal regardless of how many unrelated duplicate fragments are on the canvas. This is because the slowdown is due to the existing Visual Studio debugger attempting to update the margin glyphs and active statement highlighting for every one of the duplicated code fragments even though only one of them is active at any given point. The existing Visual Studio debugger was not written with this scenario in mind. In fact, the most difficult hurdle during implementation was that most of Visual Studio's existing functionality expects that there is only one code file per tab, whereas Debugger Canvas hosts multiple code fragments within a single tab via the canvas. This ended up breaking several of the existing tool windows in Visual Studio (e.g. Solution Explorer, Class View, and Find Results). Separate work was needed for each individual tool window to fix bugs in Visual Studio or add workarounds so that they would work properly when multiple code fragments are hosted on the canvas.

V. User Feedback

We gathered feedback from professional programmers both before and after the public release. Before release, we did usability testing in the lab. After release, we tracked adoption using both download counts and data from the Microsoft Customer Experience Improvement Program, which provides anonymous product usage data. Finally, to understand the adoption trends, be conducted a survey with 99 respondents and 11 semi-structured interviews, among both Debugger Canvas adopters and non-adopters.

A. *Usability Testing during Development*

Fourteen weeks before the first release, we used the Rapid Iterative Testing and Evaluation method [4] to improve the usability of the implementation. Briefly, we asked 10 participants to use Debugger Canvas to complete three tasks in one-hour sessions. The goal of each session was to see where the user struggled with the user experience and to gather feedback. After each user session, we identified any remaining critical usability problems and fixed them before the next session. While this method introduces too much variability between users to take controlled measures, the method is an efficient way to improve the tool and reduce overall participant frustration.

The most important usability problem we fixed in this process was our design decision to create a new canvas automatically for each debugging session. Our RITE users consistently debugged in many, short sessions (often focused on a single method) and therefore found the resulting canvases to be "clutter." We updated the design so that debugging sessions all take place in the same canvas, unless the user explicitly creates a new one.

B. *Download and Usage Data*

We measured number of downloads for the tool, as well as number of users per day and per month. In the adoption numbers we were mostly looking for trends. We expect a non-useful tool to have bad word of mouth, leading to downloads going down sharply after the initial launch, while conversely, a useful tool should have a long tail after the initial spike, leading to a significant number of downloads beyond the first 2-3 weeks. The download curve is shown in Figure 5.

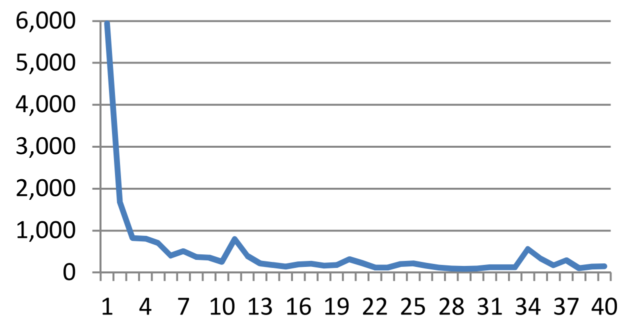


Figure 5: Number of unique downloads per week, after the initial release on 13 June 2011.

Download trends show a strong spike the first week, as would be expected, then settles into a mostly flat pattern. Downloads from week 3 and out represent 45% of the total. Given our initial criteria, this represents a positive result. It seems like Debugger Canvas may have enough usefulness that a relatively steady stream of users get pointed our way, despite no marketing activities from us after the initial launch, up until week 32 when we announced the second release.

The Microsoft Customer Experience Improvement Program (CEIP) provides the ability for customers to upload product usage data with complete anonymity. To participate in this program, users opt in to share their data with Microsoft, meaning that such data represents a self-selected sample of all users. (The Visual Studio team estimates that roughly 15% of their customers participate.) This data is then collated into counts of users who performed this action per day and per month.

To receive data, a team must instrument the operations in its product. For Debugger Canvas, we instrumented the operation of stepping with the debugger inside a code bubble, as well as our menu commands. Table 1 shows the frequency of Debugger Canvas's operations, relative to stepping inside a code bubble (our most frequent instrumented operation).

Table 1. Relative use of Debugger Canvas's commands.

Command	**Use relative to stepping**
Step into bubble	1.0
Create New Canvas	.11
Start Debugging Without Debugger Canvas	.07
Show Video Tutorial	.03
Start Debugging With Debugger Canvas	.03
Save As XPS	.01
Send Feedback	.01
Send As XPS Attachment	.01

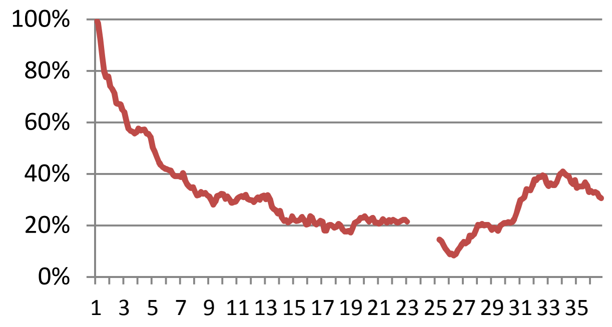


Figure 6. Users per day who step into a code bubble at least once, as a percentage of usage on the first day. (The gap is due to missing data.)

Figure 6 shows the number of users per day (in the CEIP sample) who step into a code bubble, starting in week 10 after release. (The gap is due to a problem with data collection in December 2011.) The trend of users per day is mostly flat and then picks up after the second release in week 32. When seen in the context of the trickle of new downloads in Figure 5, the overall curve in Figure 6 suggests that many users dropped out after initial use, but a large fraction continue to use it steadily.

The relatively stable usage over time suggests that some users are finding Debugger Canvas useful. Ideally, we would like the usage data to trend upwards to match the download curve. The steady download numbers mean an increase in *potential* usage but it is clear from the usage numbers that this is not translating directly into *actual* usage. This indicates that some users are either abandoning the tool or using it less frequently over time. Our next question is therefore what reasons users had to adopt or abandon the tool.

C. User Survey

To better understand customer adoption and non-adoption, we performed a survey of 341 users who had downloaded the tool internally. We got 99 responses, of which 72 reported having used the tool; of these, 53 were still using it and 19 had abandoned it. When asked to list reasons to abandon the tool, users reported the reasons in Table 2.

Table 2: Reasons why 19 surveyed users stopped using Debugger Canvas. Each respondent may report multiple reasons.

Reason to abandon	#Respondents	Type
Editing not discovered	4	Usability
Bugs	4	Bugs
Performance	4	Bugs
Doesn't support my platform	3	Other
Screen too small	2	Useful
Data tips bug	2	Bugs
Wants features	2	Utility
Concept didn't work for me	2	Utility
Want to resize bubbles	1	Utility
Instruction pointer update bugs	1	Bugs
Navigation not discovered	1	Usability
On demand not discovered	1	Usability
Sum	27	

Of these 12 factors that users reported, 5 were related to bugs, performance or platform support, seen in a total of 14 responses. Another 3 issues, in 6 responses, were that various features were not discovered (i.e. the user complained about Debugger Canvas lacking a feature that it in fact has). The remaining 4 issues in 7 responses can be interpreted as related to utility.

D. User Interviews

To understand the survey results in more depth, we conducted 11 half-hour, semi-structured interviews with users of Debugger Canvas, during which we probed for specific situations in their own work when Debugger Canvas was helpful and not helpful.

1) When Is Debugger Canvas Useful?

Overall, 9 out of 11 users mentioned situations when they found Debugger Canvas to be particularly useful. A few themes were mentioned often: deep or complicated call paths (6 respondents mentioned this); dynamically linked code (4); large code bases (3); and unfamiliar code bases (2). We discuss each of these in turn. (In the quotes below, we use numbers as pseudonyms for the interviewed users.)

The most common situation where users found the canvas to be useful was when debugging long and complicated call paths. 6 out of 11 reported this.

> *I often have to debug several layers on our side from the UI, via middle tier to the data layer. It often gets confusing to go into the deeper layer. This is where the canvas helps, you hit a breakpoint here and can see the stack trace as you step through the layers. This helps us debug things much faster.* (10)

When call trees get larger, it gets increasingly hard to remember relevant information about the code that has already executed. With Debugger canvas, referencing this code is easier since only relevant code is shown and the code is laid out to reflect the call paths rather than file organization.

> *Visualizing the flow of control through the debugger is a huge asset. Method calls and their state is visualized and trivially navigable. The left-to-right, top-to-bottom spanning of the tree makes intuitive sense.* (3)

Large and unknown codebases pose similar problems for users, and not unexpectedly, 5 out of 11 users mentioned this as an area where Debugger Canvas was useful.

> *I was working on a large project for only a week. There was a huge ramp up, of course, and Debugger Canvas was invaluable for stepping into the code to see what was going on.* (5)

> *With a really large code base that you are not familiar with it is really handy. It helps wrap your head around other people's code. That kind of visualization really helps to follow code as it crosses different classes and projects. Go-to-definition and using Reflector** *is just too cumbersome to navigate through all that code.* (6)

A third useful area is debugging code with significant use of dynamic linking. 4 out of 11 users mentioned this as an area where they found the canvas useful.

> *A lot of calls go through factories, very generic. There's also lots of circular calls and bad naming. You are never sure what are you debugging. Debugger Canvas shows*

* .NET Reflector is reverse-compiler for .NET bytecode.

you the call stack, you can see the picture where you were before. It helps me decide which branches are important for the analysis, and which aren't. (4)

We use dependency injection a lot. For someone working with a lot of dependency injection, this is very handy. (8)

Factories: You have to use the debugger! What type does it return? Oh this type! (11)

The canvas also seems to provide benefits beyond helping individual users understand the code that they are debugging. Two out of 11 users found that using the canvas to communicate with team members was a major advantage.

Sometimes it is very difficult to show how you fixed the bug. With the canvas you are able to show how you made the change, how you got it, what modules were impacted. (1)

I get the bugs that my team members cannot figure out. Typically I like to debug together with them. I usually go to their machine first. If it is 20 minutes in, I go back to my machine. Now I can also use Debugger Canvas and send them the image of what I found. That does help the communication. (10)

2) *When Is Debugger Canvas Not Useful?*

Given that the advantages of the canvas seem to lie in debugging complex, unknown often dynamic paths, it is not surprising that users find it less useful when those factors are not present.

For a "normal" project it isn't worth the hassle with performance. (5)

I don't always want to get into the canvas. When I'm debugging something small: for example - Did the parameter get here? Then it doesn't warrant opening up the canvas.(10)

Familiarity with how code is organized in the file, also sometimes weighs against using the Canvas.

Code that you've worked on for hours, and you know that this function is right above this, then it messes with your mind when it is in the bubbles. (2)

Sometimes what I'd like to do is to "peek ahead" because I know that the next call is right below it in the module. (1)

However, this may be at least partially a matter of habit which will be less important over time.

I just expanded it to two monitors, and it seems a lot more appealing. It may have a lot to do with the layout, and the real estate that you need to have to cross the threshold, you give up familiarity, but it looks great! (2)

I kinda sometimes switch back and forth between the two of them. I was more used to see everything, but I'm starting to get more used to seeing the bubbles. (1)

I would probably switch between regular and this if it was part of VS, but eventually just use this. I'm more used to the regular debugger. (7)

As far as I'm concerned this is my new debugger! (9)

On the other hand, there may be tasks that Debugger Canvas doesn't support today that force users to go to files. One such task was to inspect fields of a class, which don't show up in the method bubbles.

Sometimes I need to see field definitions. (2)

I stop using it when I need to see definition of classes. I'm aware of the Go-to-definition feature, but I use ReSharper and lots of tools to navigate, so I find it easier to go back to the file in those cases (4)

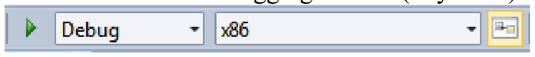
Sometimes the fix that I need to do involves code that is not in the bubbles, but is in the same files, so I'd like to be able to get to the rest of the file easily. (10)

I hit a breakpoint check the value of a private field. That's when seeing the rest of the file comes in handy. (10)

E. *Updates Based on User Feedback*

Based on this user feedback, we spent an additional four two-week sprints preparing an update implementation, scheduled to be released in November 2011. Besides fixing bugs and making the performance improvements mentioned earlier, we added several small features to improve usability.

First, users reported that Debugger Canvas is most useful when call paths are complicated, lengthy or involve dynamically loaded code. Unfortunately, a user often does not know in advance when she will find herself in this situation. In our first release, the user chooses between using Debugger Canvas and the standard debugger at the start of the debugging session. In the next release, we have added a button to the standard debugging toolbar (in yellow)



to allow the user to switch between Debugger Canvas and the standard debugger at any time during debugging. That way, the user can use the familiar debugging experience until finding herself in a debugging situation when the canvas would be useful.

Second, users had a difficult time discovering navigation features, particularly the use of the call stack window. Symptomatic of this is the "lonely bubble" problem: the user sets a breakpoint, launches the debugger and then sees a single bubble on the canvas with the method containing the breakpoint. In this situation, there are no visual hints about how to get further context. In the next release, we added small buttons that appear above the bubble that represents the active call frame:

The buttons open methods above and below the currently active frame on the call stack, namely (from left to right): all frames below; the previous frame below; the next frame above; all frames above. In short, these buttons are designed to make call stack navigation more accessible and visible.

Finally, the support for concurrent programming mentioned before is an addition for the second release.

VI. Related Work

The problems programmers face during development tasks have been documented using both observation and logging [5] [6] [7] [8]. These studies show that programmers frequently navigate among program definitions, with locality patterns that cluster the definitions into working sets. The frequency of navigation can cause disorientation, making re-finding of relevant code less efficient. This pattern gets worse as programmers are interrupted [7].

One approach to making navigation more efficient is to track the programmer's navigation steps and to use the programmer's current location to recommend related places in the code [9] [10] [11]. While these recommendation systems provide useful shortcuts, this does change the underlying hypertext model of program navigation; the programmer still has to build up a mental model of the code's contents as she navigates. In contrast, Code Bubbles, Code Canvas (through its filtered canvases), and Debugger Canvas all present a programmer's working set of program definitions (as well as related artifacts) side by side in the same presentation, easing the programmer's need to remember the relevant code.

Spatial representations of code have a long history, going back at least to the Self programming environment [12]. In the Self tradition, one use of space is to represent the entire software system. The challenge is to deal with the complexity of large system. One strategy is to use the intuitive appeal of a cartographic map metaphor, of which Software Cartography [13] and Software Cities [14] are recent examples. Another strategy is to use semantic zoom, in the style of Shrimp views [15]. Code Canvas uses both of these strategies. Another use of space is to represent the programmer's working set, the approach of both Code Bubbles and Debugger Canvas. While this avoids the scale problems needed to represent the entire system, Code Bubbles and Debugger Canvas nonetheless present more than a screen's worth of content and therefore provide panning and geometric zoom for content management. A recent whiteboard diagramming study shows that programmers are flexible and informal in spatial representations of code [16], which provides the motivation for the flexible layout and annotations that Code Bubbles, Code Canvas, and Debugger Canvas all support. Both Self and the recent Gaucho environment [17] allow code to represented not just as fragments in space, but also in non-textual ways. Debugger Canvas sticks to the familiar textual code presentation to help encourage adoption.

A few recent papers have also explored hosting the development environment across multiple devices. Code Space [19] uses the bubbles paradigm in a meeting room setting, where the bubbles can be moved among shared screens and mobile devices. Similarly, CodePad [18] spreads the development experience across several devices, to support an individual programmer's work, particularly multitasking.

VII. Conclusions

Returning to the questions that originally motivated the implementation and release of Debugger Canvas, our experience allows us to draw some initial conclusions.

First, the canvas design can be implemented to scale up to the customers' code bases. The initial release had some performance problems, which have since been identified and fixed. Even with the performance problems in place, many users were using Debugger Canvas on a daily basis. Indeed, in interviews, several users mentioned that large code bases are where Debugger Canvas is particularly helpful.

Second, our experience shows that the canvas idea is best embodied as a mode within the existing user experience. While some users found value in the new user experience, others did not. Even users who were enthusiastic about the idea mentioned that there are situations where the overhead of switch representations from tabbed documents to the canvas is not worth it. One example is rapid-iteration debugging on a single method.

Finally, there are situations where some users strongly preferred the canvas design. Our interview participants reported finding the canvas useful with long or complex code paths, with large code bases with many layers, with unfamiliar code bases, and when the code involved dynamically linked code, factories, or other indirect forms of control flow.

References

[1] A. Bragdon, S. P. Reiss, R. C. Zeleznik, S. Karumuri, W. Cheung, J. Kaplan et al., "Code Bubbles: Rethinking the user interface paradigm of integrated development environments," in *International Conference on Software Engineering*, 2010.

[2] A. Bragdon and R. C. Zeleznik, "Code bubbles: a working set-based interface for code understanding and maintenance," in *Computer Human Interaction - CHI*, 2010.

[3] R. DeLine and K. Rowan, "Code Canvas: Zooming towards better

development environments," in *International Conference on Software Engineering - ICSE*, 2010.

[4] M. C. Medlock, D. Wixon, M. Terrano, R. Romero, and B. Fulton, "Using the RITE Method to Improve Products: a Definition and a Case Study," in *Usability Professionals Association*, 2002.

[5] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, "An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks," *IEEE Transactions on Software Engineering*, vol. 32, no. 12, 2006.

[6] G. C. Murphy, M. Kersten, and L. Findlater, "How Are Java Software Developers Using the Eclipse IDE?," *IEEE Software*, vol. 23, no. 4, 2006.

[7] C. Parnin and S. Rugaber, "Resumption strategies for interrupted programming tasks," in *International Workshop on Program Comprehension - IWPC*, 2009.

[8] M. P. Robillard, W. Coelho, and G. C. Murphy, "How Effective Developers Investigate Source Code: An Exploratory Study," *IEEE Transactions on Software Engineering* , vol. 30, no. 12, 2004.

[9] R. DeLine, M. Czerwinski, and G. Robertson, "Easing Program Comprehension by Sharing Navigation Data," in *IEEE Symposium on Visual Languages/Human-Centric Computing Languages - VL/HCC*, 2005.

[10] M. Kersten and G. C. Murphy, "Mylar: a degree-of-interest model for IDEs," in *International conference on Aspect-oriented software development (AOSD)*, 2005.

[11] J. Singer, R. Elves, and M.-A. D. Storey, "NavTracks: Supporting Navigation in Software Maintenance," in *International Conference on Software Maintenance - ICSM*, 2005.

[12] D. Ungar and R. B. Smith, "Self: The power of simplicity," in *OOPSLA*, 1987.

[13] A. Kuhn, D. Erni, and O. Nierstrasz, "Embedding Spatial Software Visualization in the IDE: an Exploratory Study," in *Software Visualization - SOFTVIS*, 2010.

[14] R. Wettel and M. Lanza, "Visualizing Software Systems as Cities," in *Visualizing Software for Understanding and Analysis - VISSOFT*, 2007.

[15] M.-A. D. Storey, C. Best, J. Michaud, D. Rayside, M. Litoiu, and M. A. Musen, "SHriMP views: An interactive environment for information visualization and navigation," in *Computer Human Interaction - CHI*, 2002.

[16] M. Cherubini, G. Venolia, R. DeLine, and A. J. Ko, "Let's go to the whiteboard: How and why software developers use drawings," in *Computer Human Interaction - CHI*, 2007.

[17] F. Olivero, M. Lanza, M. D'ambros, and R. Robbes, "Enabling Program Comprehension through a Visual Object-focused Development Environment," in *IEEE Symposium on Visual Languages and Human-Centered Computing - VL/HCC*, 2011.

[18] A. Bragdon, R. DeLine, K. Hinckley, and M. R. Morris, "Code Space: Combining Touch, Devices, and Skeletal Tracking to Support Developer Meetings," in *ACM International Conference on Interactive Tabletops and Surfaces*, 2011.

[19] C. Parnin, C. Görg, and S. Rugaber, "CodePad: interactive spaces for maintaining concentration in programming environments," in *Software Visualization - SOFTVIS*, 2010.