RICH TEXT CODE FORMAT FOR ENHANCING COMPREHENSION

Dr. Shikha Sarkar, Sambuddha Majumder, Jayanta Majumder,
Beautiful Mondays Ltd
http://facebook.com/beautifulmondays
shikha.paimajumder@gmail.com

Abstract - The idea of using rich text for programming looks promising. Some new avenues can open up by a programming environment that allows the programmers to use rich text formatting options (e.g. fonts, colors, highlighting, hyperlinks, embedded images, notes, audio, videos etc.) in their program text. Such an environment can be very useful towards enhancing program comprehension, especially in the context of maintaining large long-lived software systems. Computer programs are among the most complex and valuable creations of the human mind. Mankind has already produced billions of lines of code worth trillions of dollars. Countless man-hours of intense cognitive work would be required to maintain them. Any tool that improves maintainability of such assets can be a useful addition.

***Keywords*:** Text editor, Rich text, Annotations

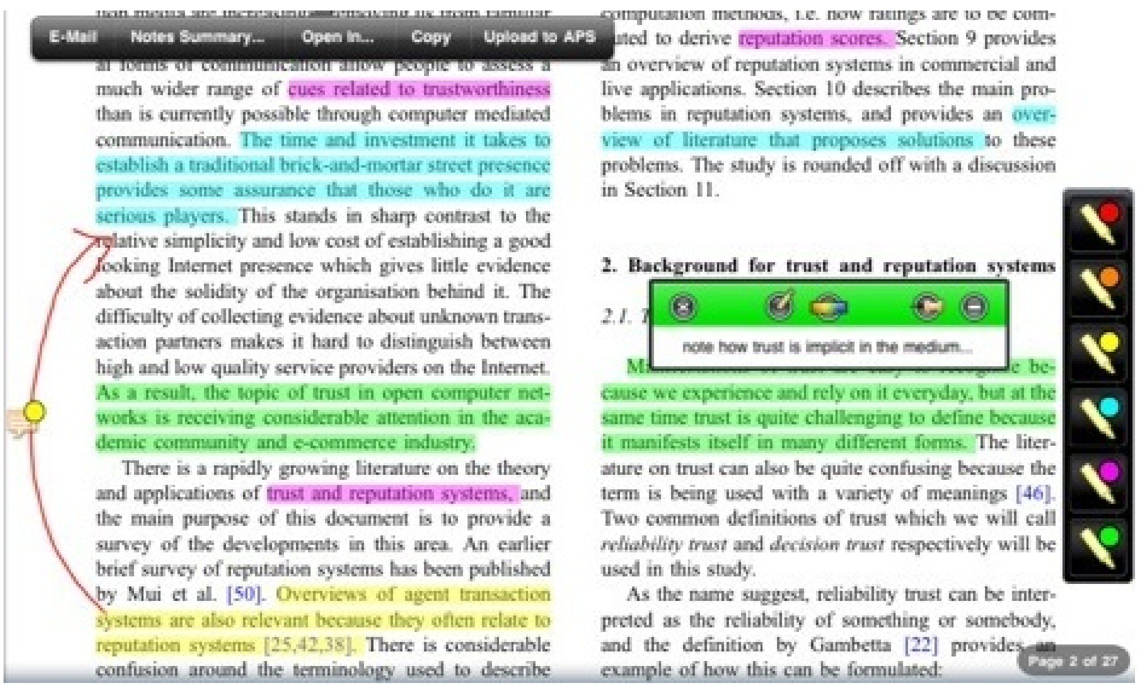
1. Introduction

Rich text is more conducive to human perusal due to the availability of additional visual cues. However, the existing rich-text formats are not suitable for use with programming. This paper presents a new rich text format that is amenable for programming in that it can be easily used in diff/merge and version control processes. A text editor based on this rich text format is also presented. The editor is called ‘*spectral*’. The central idea of this format and editor is a marriage between rich-text and plain text-formats which brings about the best of both worlds. It begets the maintainability merits of plain text (e.g. simplicity of automatic merging, byte-level concatenation, and version management) alongside the visual appeal of rich text (e.g. fonts, color, graphics, etc.). The file format represents rich text in a way that preserves line-to-line correspondence with the plain text content. The *rich text* view would incorporate pictures, notes, audio, video etc. to enrich the programming and program comprehension experience, while the corresponding *plain text* source code is preserved all along with a simple and intuitive mapping between the plain and the rich text.

2. Literature Review

This work belongs in the body of literature on program comprehension. Readers looking for precursor work are advised to consult publications made in the annual International Conference on Program Comprehension over the years. However this work does not seem to have a direct precursor in the academic literature. [Dunsmore1990], [Anderson2001] and [Desmond2006] may be seen broadly as related precedents. [Storey2006] provides a survey of the topic and the available approaches. Despite the many academic projects in this area, the practical programmer’s program comprehension toolkit remains relatively basic – limited to debugger, profiler, and log files. The proposed rich text editor is simple enough to be immediately useful to the practical programmer as it does not involve much of a learning curve.

There are many commercially and freely available document annotation tools. One such tool’s user interface is shown in Fig. 1. These have features similar to the tools presented here, however these only apply to document formats like DOC, PDF, and HTML. Given the success of these tools, the annotation toolset should be very useful to program text.



[illegible]
[illegible] much wider range of cues related to trustworthiness than is currently possible through computer mediated communication. The time and investment it takes to establish a traditional brick-and-mortar street presence provides some assurance that those who do it are serious players. This stands in sharp contrast to the relative simplicity and low cost of establishing a good looking Internet presence which gives little evidence about the solidity of the organisation behind it. The difficulty of collecting evidence about unknown transaction partners makes it hard to distinguish between high and low quality service providers on the Internet. As a result, the topic of trust in open computer networks is receiving considerable attention in the academic community and e-commerce industry.

There is a rapidly growing literature on the theory and applications of trust and reputation systems, and the main purpose of this document is to provide a survey of the developments in this area. An earlier brief survey of reputation systems has been published by Mui et al. [50]. Overviews of agent transaction systems are also relevant because they often relate to reputation systems [25,42,38]. There is considerable confusion around the terminology used to describe

computation methods, i.e. how ratings are to be computed to derive reputation scores. Section 9 provides an overview of reputation systems in commercial and live applications. Section 10 describes the main problems in reputation systems, and provides an overview of literature that proposes solutions to these problems. The study is rounded off with a discussion in Section 11.

2. Background for trust and reputation systems

2.1. [illegible]

[illegible] because we experience and rely on it everyday, but at the same time trust is quite challenging to define because it manifests itself in many different forms. The literature on trust can also be quite confusing because the term is being used with a variety of meanings [46]. Two common definitions of trust which we will call *reliability trust* and *decision trust* respectively will be used in this study.

As the name suggest, reliability trust can be interpreted as the reliability of something or somebody, and the definition by Gambetta [22] provides an example of how this can be formulated:

Fig.1. A document annotation tool

3. The Rich Text Format

The proposed rich text file format is a plain tex sequence of records, each record identified by its *type* followed by its *details*. The record type descriptor tells as to what kind of record is represented by the details that follow - i.e. it tells whether it is a text record, whether it is an image record, an audio/visual record, a span record, and so on.

A text record's type descriptor is the letter **T**, which is followed by the text itself. The type descriptors and the detail blocks are mutually separated by white-space, as are the records. The blocks are enclosed by braces where such delimiting is required due to internal white-space within the block. Alternately spaces in a block can be escaped using a backslash.

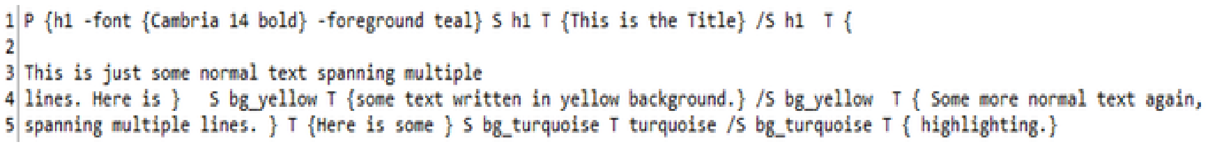
The following rule sets the proposed format apart from the other existing rich text formats:

The only blocks that can have a new-line in them are text details, and every new-line in a text detail block corresponds to a new-line in the underlying plain text.

This requirement, in itself, ensures that there is a line-to-line correspondence between the rich-text and its plain text equivalent. The font and text-colors are described by span records. Spans are regions of text enclosed by records of type *span-start* represented by the record descriptor **S** and *span-end* given by the descriptor **/S**.

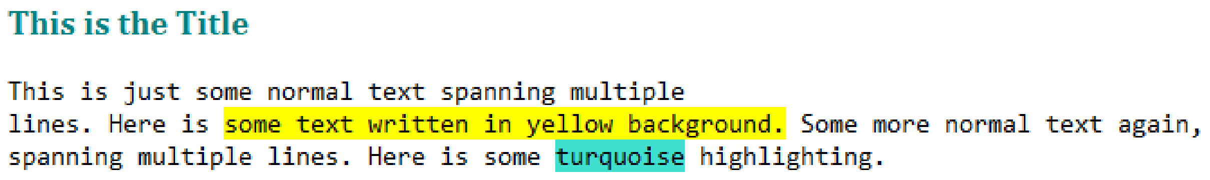
The detail block for a span record is a name, whose properties may be described in a property **(P)** record. There are record types for images, multimedia files, and so on. New record types can be introduced without getting in the way of existing record types, because all we need to take off on the new type's meaning is a unique type descriptor symbol.

Fig. 2 presents the content of a sample rich text file, which when rendered by spectral would look like what is shown in Fig. 3.



```
P {h1 -font {Cambria 14 bold} -foreground teal} S h1 T {This is the Title} /S h1  T {

This is just some normal text spanning multiple
lines. Here is }   S bg_yellow T {some text written in yellow background.} /S bg_yellow  T { Some more normal text again,
spanning multiple lines. } T {Here is some } S bg_turquoise T turquoise /S bg_turquoise T { highlighting.}
```

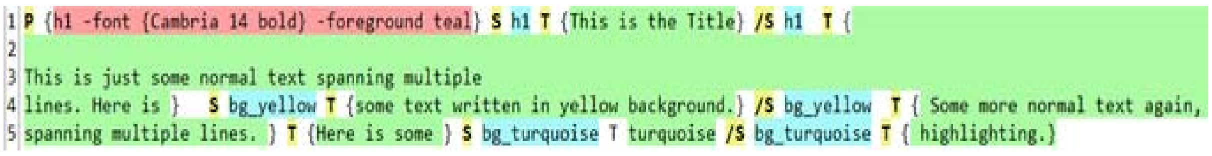
Fig.2. Sample of Spectral rich text file



This is the Title

This is just some normal text spanning multiple
lines. Here is some text written in yellow background. Some more normal text again,
spanning multiple lines. Here is some turquoise highlighting.

Fig.3. Visual rendering of the rich text given in Fig. 1



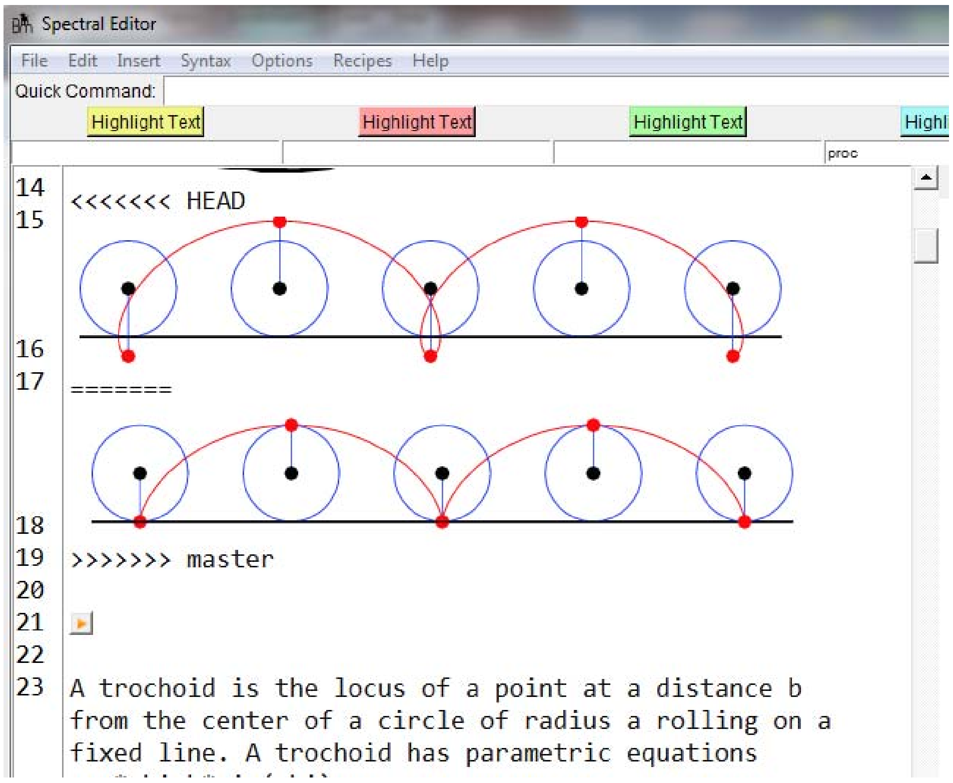
```
P {h1 -font {Cambria 14 bold} -foreground teal} S h1 T {This is the Title} /S h1  T {

This is just some normal text spanning multiple
lines. Here is }   S bg_yellow T {some text written in yellow background.} /S bg_yellow  T { Some more normal text again,
spanning multiple lines. } T {Here is some } S bg_turquoise T turquoise /S bg_turquoise T { highlighting.}
```

Fig.4. A highlighted version of Fig. 1

Fig. 4 shows the text content of Fig. 2 with some color annotations for ease of the following description. The yellow highlighted boldfaced characters in Fig. 4 are the record type descriptors. The first one among them is a **P** (i.e. property) descriptor, followed by its details - i.e. the red-highlighted part - that describes the properties of a span called "**h1**" (this span is subsequently used for the title text). The details say that the font for the "**h1**" spans would be boldfaced Cambria, of size 14, and teal in color. The first text block (i.e. one preceded by the type descriptor **T** contains the text "`This is the Title`". It is surrounded by the span blocks "**S h1**" and "**/S h1**" meaning that the font and colour properties of the "**h1**" span applies to that text. There are some pre-defined span tags (e.g. **bg_yellow** and **bg_turquoise** in this example) named after the background color that applies to their spanned text.

A user of the editor does not need to deal with the above details since the editor presents a what-you-see-is-what-you-get (WYSWYG) interface. One can simply open a text file (let's say a source code file named "`main.cpp`") on this editor, and add value to it using color, fonts, graphics, audio, etc. Each time the file is saved, the editor will save the plain-text changes to the original file ("`main.cpp`") and the rich text to a file named by appending "`.hlt`" to the original file's name, which in this case will be "`main.cpp.hlt`". If one subsequently changes the original file using some other editor, the "`.hlt`" file will go out of sync with the original text but the next time it is opened on spectral, the change will be merged into the rich text. The merged regions of the text will be highlighted with a special background and a helpful merge report will be displayed in a pop-up window. The rich-text merge report would include all the pre-existing content that got modified or deleted due to the merge, and if required these may be copied and pasted back to the editor window.

If the "`.hlt`" files are stored in a source control system (e.g. svn, git, perforce, etc.), and merge conflict markers are introduced by a merge from concurrent changes to the same file, the markers will not corrupt the file. Instead the conflict markers will get superposed into the rich-text, within a text block. This is a slight deviation from the file format described above. The handling of conflict markers may be seen as an additional pre-processing that scans the files for conflict markers and convert them into text records that comply with the format described earlier. If the conflict block happens to have some graphics in it, the graphics will also be displayed as shown in Fig. 5.



```
<<<<<<< HEAD

=======

>>>>>>> master


A trochoid is the locus of a point at a distance b
from the center of a circle of radius a rolling on a
fixed line. A trochoid has parametric equations
```

Fig.5. Merge conflict markers displayed on Spectral

Concatenation of two such rich text files would not corrupt the file, and result in the naturally expected concatenation - one that preserves the rich text markup and graphics while concatenating the textual content.

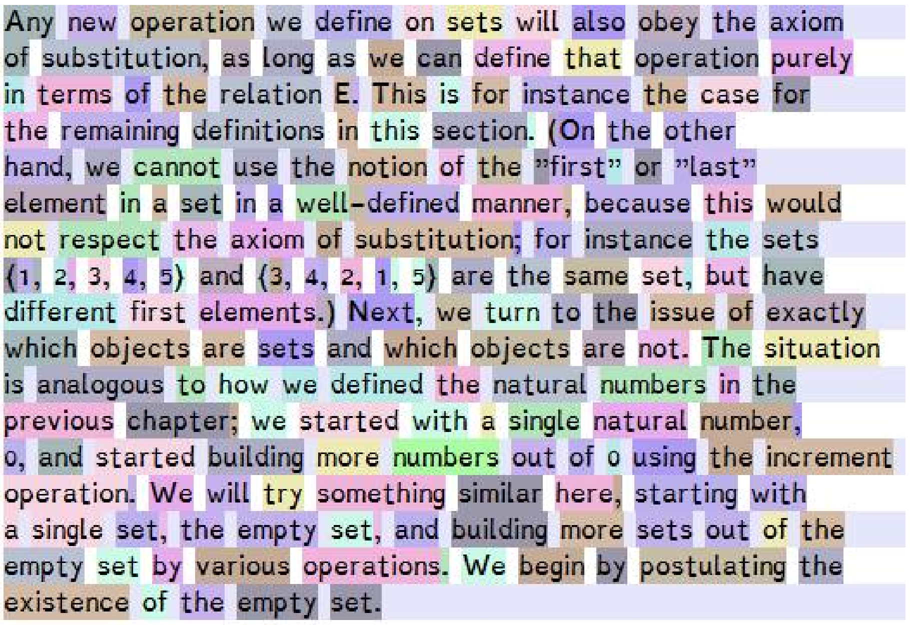
It is also possible to make the **hlt** format the primary code format in a project where Makefile rules are used to extract the source code files from **hlt** files before further compilation. The **hlt** format is so simple that a mere five line script in a common scripting language can serve as the plain text extraction tool.

4. Applying Text Annotations

The spectral format supports images, audio, video, notes, hyperlinks, and references to other files, which in turn can help produce value-added code that is cross referenced with other bits of code, data, and multimedia. There are many possible creative uses of these features. For example, one might want to use spectral to record audio clips during a code walk-through or review meeting. The audio so recorded would be attached with the current cursor/caret position when the record button is clicked, thereby keeping it in the context of the code that was being discussed.

Notes i.e. textual annotations that don't result in change to the underlying source code or plain text files may be similarly attached with code locations. One use of notes is to store dead code. Sometimes people comment out dead code just in case they need to refer to them later. This is arguably a bad practice because it takes up screen real-estate rendering the active (alive) code harder to read. So deleting the dead-code serves to de-clutter the screen real-estate (and after all the source control system would have the old versions). However, if it actually becomes necessary to consult dead code, the existing source control tools don't really offer a good way of fishing out deleted code. Therefore it seems to be a good solution to tuck away dead-code into notes while adding a special keyword (say "dead-code") to notes that contain dead code. That way it would be easier to search for them later.

Some anecdotal evidence has been noted that for people with dyslexia, highlighting words with different colors can enhance the ease of reading. Fig. 6 shows such a block of text with colorfully highlighted words rendered in spectral.



Any new operation we define on sets will also obey the axiom
of substitution, as long as we can define that operation purely
in terms of the relation E. This is for instance the case for
the remaining definitions in this section. (On the other
hand, we cannot use the notion of the ”first” or ”last”
element in a set in a well-defined manner, because this would
not respect the axiom of substitution; for instance the sets
{1, 2, 3, 4, 5} and {3, 4, 2, 1, 5} are the same set, but have
different first elements.) Next, we turn to the issue of exactly
which objects are sets and which objects are not. The situation
is analogous to how we defined the natural numbers in the
previous chapter; we started with a single natural number,
0, and started building more numbers out of 0 using the increment
operation. We will try something similar here, starting with
a single set, the empty set, and building more sets out of the
empty set by various operations. We begin by postulating the
existence of the empty set.

Fig.6. Text with randomly highlighted words in dyslexie font in spectral

The mechanism or statistical significance of the above observation is not well understood. For code comprehension, if all occurrences of chosen set of symbols can be highlighted with a color for each symbol, it becomes easier to spot occurrences without having to explicitly “read” the symbols. Spectral allows such randomized coloring of symbols with simple mouse-clicks.

Fig. 7 shows screenshots of typical code-comprehension sessions on spectral in which selected symbols are highlighted with various colors.

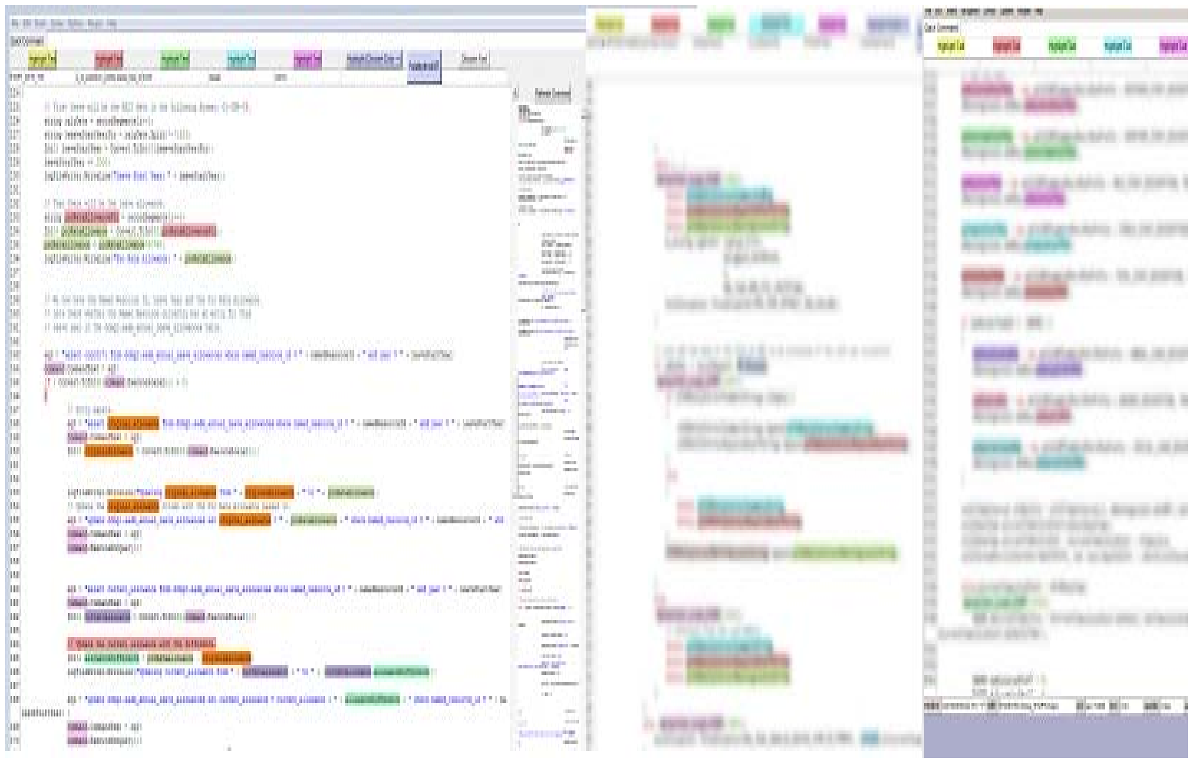


Fig.7. Screenshots of typical spectral editor sessions

5. Conclusion

The cognitive ease that colorful highlighting facilitates arises from broadening the bandwidth of the communication channel connecting the code text and our cerebral cortex. Spectral only adds to the bandwidth of perusal and doesn't take anything away. Especially, while reading code littered with weakly relatable and overly long variable names and other types of symbols (e.g. `CLinkSpanRampJunctionNodeManagerProxy`, or `IAbstractFlowSheetConnnectionTerminatorVisitor` etc.) it certainly helps to think non-verbally in terms of *this turquoise thingy* or *that yellow word* at least until the meanings and purposes become clearer.

Since all the advantages of plain text is retained in the proposed rich text format - due to the line-to-line correspondence with plain text, as described in section 3, there is a strong possibility that rich text programming can go main-stream.

References

[1] Alastair Dunsmore (1998): Comprehension and visualization of object-oriented code for inspections. Empirical Foundations of Computer Science (EFoCS), University of Strathclyde, Glasgow.

[2] Paul Anderson and Tim Teitelbaum (2001): Soft ware inspection using code-surfer In Proceedings of the first Workshop on Inspection in Software Engineering.

[3] Margaret-Anne Storey (2006): Theories, tools and research methods in program comprehension: Past, present and future. Software Quality Journal, 14(3):187–208

[4] Desmond, M. and Storey, M.A., (2006): Fluid source code views for just in-time comprehension. In Proceedings of the SPLAT conference, 2006