Interfaces for Programmers

Seminar on Post Desktop User Interfaces

Philipp Siebenkotten **Tobias Lietke**

RWTH Aachen University, Germany
{philipp.siebenkotten, tobias.lietke}@rwth-aachen.de

ABSTRACT

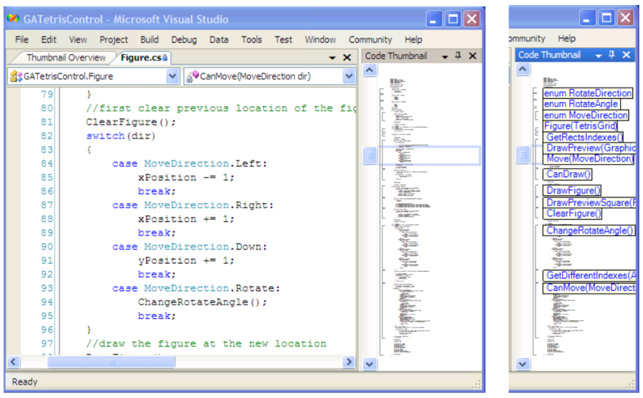
Common interfaces for programmers (integrated development environments, IDEs) provide a flat file structure and basic text editing facilities. Yet, programmers tend not to exclusively think in these structures, so in this paper we highlight different approaches to ease the activity of programming. Supporting the developers in creating a mental model of their work is the key factor we considered. We identified the key areas of navigation, learning and social aspects. While flat structures — more appropriate for compilers — are prevalent in established IDEs, we found semantic relations of elements in these areas to better supplement the developers in the gist of their programming activity.

INTRODUCTION

This paper intends to summarize recent publications on programmer interfaces development, so we will start out by summarizing facts about modern application development and how programmers approach it. This provides an appropriate structure to arrange and classify the different papers.

Smith [8] describes staff distributions over different project phases for projects of several sizes (12, 27 and 115 person months). According to these numbers, about 70% of the total effort is spent doing the concrete implementation. An interface for programmers should therefore especially support the developers during implementation, in order to have the best possible impact on their overall performance. Furthermore the numbers suggest the team size nearly triples from the initial inception phase to the main construction phase, where the features are implemented. If an architecture is designed in an early phase, this results in a need for efficient communication or other means of knowledge transfer. Even in a more agile approach, with fast iterations over such phases, one could argue that the same basic setting is present: A few developers design the overall system, whereas a larger number is involved with the actual implementation. This makes knowledge transfer an important aspect, which a good interface should support. Especially when new developers enter a team, eg. for bug fixing, the knowledge previous built up, could be harnessed in order to support them.

Another factor to consider is the way in which developers work with the source code. Mayrhauser and Vans have summarized research on software comprehension models [6]. Their important conclusion is that developers (especially experts) tend to build a mental, structured model and when working on the code, they frequently transfer between an abstract and a detailed view on the project. We will present research which implements alternatives to classical development environments, which might enhance the programmer's ability to think in such structures.



```
}
//first clear previous location of the fi
ClearFigure();
switch(dir)
{
    case MoveDirection.Left:
        xPosition -= 1;
        break;
    case MoveDirection.Right:
        xPosition += 1;
        break;
    case MoveDirection.Down:
        yPosition += 1;
        break;
    case MoveDirection.Rotate:
        ChangeRotateAngle();
        break;
}
//draw the figure at the new location
```

Figure 1. Code Thumbnails — Visual Studio extended by the thumbnail (left) and showing navigation aids during mouse-over (right). [3]

NAVIGATION

Software developers spend 35% of their time navigating the source-code [5]. This is not only the case in projects with a big amount of code, but already when project is of moderate size. Thus inefficient navigation causes task to take too much time and even experienced programmers spend a lot of time navigating when working on a small code base. There are several reasons for this navigation-problem. First the navigation in common IDEs is based on memorizing symbol names [3]. For every navigation step one needs to know the name of a package, file, class or method. So the developer has to remember many names and this can overburden him even in small projects.

Code Thumbnails

One idea to avoid this navigational problem is proposed by Robert DeLine and colleagues from Microsoft Research [3]. They introduce the tool Code Thumbnails, an add-on for the Microsoft Visual Studio IDE. This add-on tries to use the spatial memory for navigation instead of relying on a cognitive task. To achieve this, Code Thumbnails extends Mi-

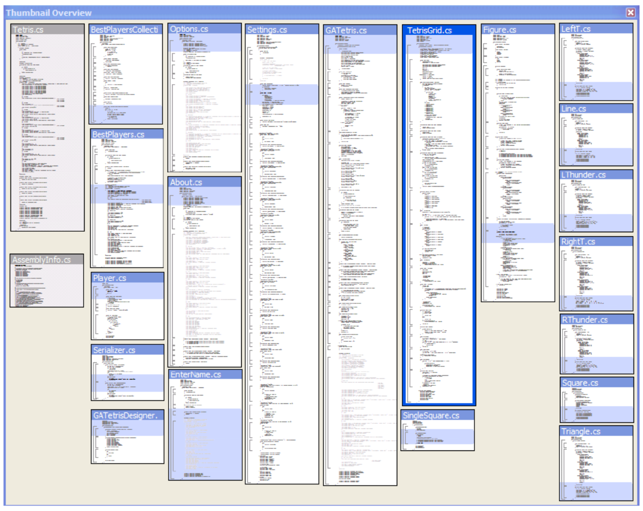


Figure 2. Code Thumbnails Desktop — Showing all project files in an overview [3]

crosoft Visual Studio with two new features in order to reduce navigation time. The underlying approach is to use the text shape to make navigation a perceptual task and as a result forming a spatial memory map:

Code Thumbnails Scrollbar

The first feature is the Code Thumbnails Scrollbar (CT Scrollbar or CTS) (figure 1), which is used for intra-file navigation. It extends the common scrollbar with a new window besides the scrollbar, which visualizes the code the user can scroll through. The window shows a small overview of the text in the scrollable file with a font size of a maximum of 2.5 points, so that it is not possible to read it, with the goal that not the text itself is perceived, but the structure as a shape. Then the programmer can use the CTS as something like a map of the file. For orientation the current shown code is marked in the CTS-window with a box around it. To highlight the code hierarchy the second- and third-level nodes are marked with brackets. So there is a visual feedback of the whole file when scrolling through. To speed up the scrolling time the user can also switch directly to the part of the file he wants to see, by just clicking on the corresponding part in the CTS. While the mouse is inside the CTS, the view of the CTS is augmented with labels naming of the second- and third-level navigation targets. Therefore the user has an overview of the file hierarchy and can jump directly to the desired code fragment.

Code Thumbnails Desktop

The second feature is the Code Thumbnails Desktop (CT Desktop or CTD, figure 2). It provides an overview of every source file in the current project, so the developer can use it for inter-file navigation. Therefore the CTD shows a thumbnail image as in CTS but of every file in the project, arranged as an workspace filling desktop. To highlight the currently used code, every part of the code visible in an editor window is highlighted by a blue background. The text of every thumbnail has the same font size, so the user is able to differ the length of a file by the height of the thumbnail. Every closed file is shaded gray. The currently active file is marked by a thicker border then the other thumbnails. To navigate with the CTD the user can do several things: By moving the mouse cursor over the representation of a file, the labels of the code hierarchy are shown as in CTS. If the user clicks on the title area of a thumbnail, this file becomes active. To jump directly to a desired part of one file, the developer can also click directly on the corresponding part in the thumbnail. A closed document can be opened and activated by double-clicking the grayed thumbnail.

To improve search in the code, the search results of any standard search tool in Microsoft Visual Studio are highlighted in the CTS and the CTD.

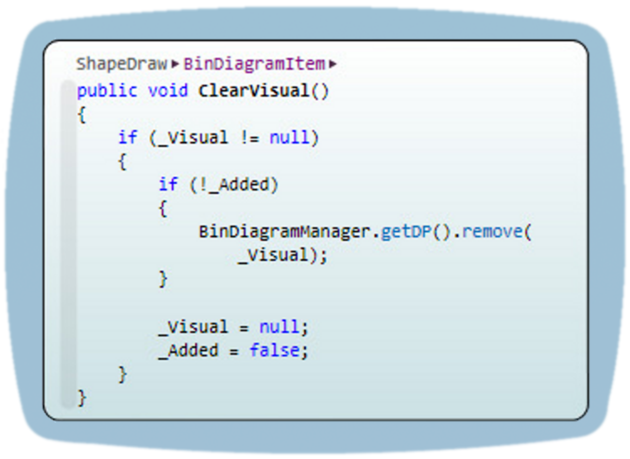
Evaluation

The Microsoft Research team evaluated Code Thumbnails formative in a user study with eleven participants. They get 75 minutes time to solve three programming task in a C# project of an Tetris game. Afterwards they performed a quiz about the spatial memory. Therefore they first had to click on a blank screen in CTD size to localize methods and classes of the Tetris project. Then in a similar second task all thumbnails gets revealed without file names. To test the CTS the participants first has to localize methods names in a blank screen in CTS size and then in same screen, but with revealed code thumbnails without any text labels. Because of a technical problem only five participants got evaluated. The participants used Code Thumbnails in 40% to 91% of there navigation activities and they gave a positive feedback about the tool. The spatial memory quiz was solved significantly faster in the second and fourth task. This fact means that the participants were beginning to create a map of the code structure so that the approach of spatial memory seems to work.

Code Bubbles

While Code Thumbnails tries to uses the spatial memory instead of the cognitive memory, but leave the IDE's basic functionality as before, Bragdon et al [1, 2] go one step further and try to rethink the whole structure of the IDE. Instead of the common file-based window-design as used in every common IDE they introduce the bubbles metaphor as a new design concept (figure 3). The reason for this new design approach is that file-based views have many disadvantages. First, files show parts of the source code depending on the necessity and not based on the semantics of the source code. And every window in a file-based view is large. There are many buttons, bars and other items around every window and therefore much display-space is wasted by white colour background without information, because source code does not fit in a rectangle in a smart way. And of cause a window is also limited by the screen size of the monitor.

So Code Bubbles is introduced as a new design choice. A prototype of this IDE has been implemented as an alternate GUI for the Eclipse IDE, which uses bubbles instead of windows and methods instead of the whole file as foundation.



```
ShapeDraw ▸ BinDiagramItem ▸
public void ClearVisual()
{
    if (_Visual != null)
    {
        if (!_Added)
        {
            BinDiagramManager.getDP().remove(
                _Visual);
        }

        _Visual = null;
        _Added = false;
    }
}
```

Figure 3. Code Bubbles — a single code bubble [2]

The bubble-metaphor
A bubble represents one method in the code, so there will be many bubbles on the screen, therefore one goal of the bubble-design was, that it should be easy to handle. For a tiny design every bubble has a maximum size of 55 characters by 40 lines. If the code does not fit to this restriction three steps are done: First the bubble gets minimized with a syntax aware algorithm. Then basic code blocks like while-blocks or if-blocks get elided. The user can then expand the code block manually when needed. If the code still does not fit, as a final minimizing measure, a scrollbar is displayed.

Another approach to realize a tiny design is the minimization of decorations. There is only a thin border to resize a bubble. No title bar is displayed, instead there is only a breadcrumb bar, which the developer can use to scroll trough the class file to search for a method. More buttons or bars are not implemented (figure 3). Manipulating the bubbles is realized by mouse actions: The developer can move a bubble by right click and drag, a middle click closes a bubble, with a left click he can manipulate the text caret and the right click without drag opens a context menu, which displays additional options. To recover from unintentional clicks, an undo-button is displayed on the screen, whenever a bubble is closed.

Multiple bubbles
As mentioned before, of course there are multiple bubbles in one working set, which is displayed on the screen. So some default design choices were made to make different views look similar and to make the view easy to handle. The first thing is the avoidance of overlapping. Bubbles are non-overlapping. If the user tries to place a bubble in that way, that it overlaps other bubbles, a spacer algorithm pushes the other bubbles away, optimized to avoid too much movement, which would cause spatial adjacency (which would not be good, as seen in CT).

When the user opens a new bubble, the spacer algorithm pushes the other bubbles away and the new bubble is initially highlighted orange and then faded to the normal background colour. This supports the user in focusing on the new bubble.

Arrows are drawn to see function calls and their corresponding method bubbles (figure 4.H). Whenever the user moves the mouse cursor over a bubble, the bubbles connections are highlighted.

To make operations on several bubbles easier, bubbles are grouped into bubble groups. Whenever two bubbles are close to each other, they are joined into one group. To show the group membership of a bubble, each bubble has a semi-transparently coloured background shadow around it and each group is drawn in a unique colour. It is possible to name a group and many operations can be performed on a group in one step. (figure 4.E)

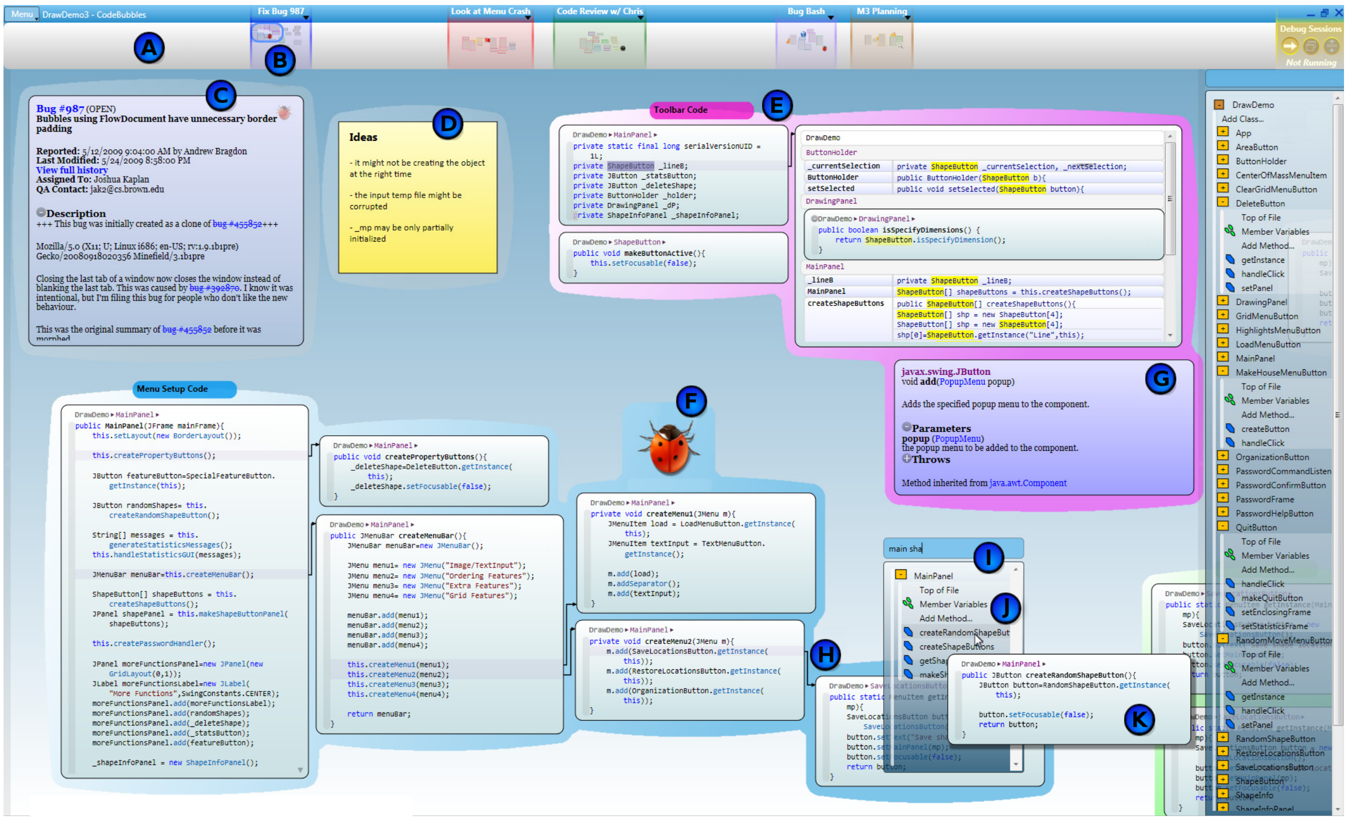
Virtual screen space
To make many bubbles in one workspace possible, the tool provides a virtual screen, which is very large though not infinite. It is 1.5 times the height and 20 times the width of the actual display. For a better overview of the virtual screen, a zoom function is implemented. If the user presses F9, the view switches between a 50% reduced view and the default view.

Another feature for overview is the workspace bar (figure 4.A). It gives the user an overview of the working sets. The currently used workspace is marked (figure 4.B). Different working sets can be labeled for an easy use. This gives the user the possibility to switch quickly between several tasks and therefore supports interruptions in the workflow.

Working with the IDE
To search for a bubble the developer can use a pop-up search box (figure 4.I), which is opened by right clicking on the background. This search box has a list of all packages and classes in the project (figure 4.J) and users can search for methods. If the user hovers over the method name, a preview of the bubble is shown (figure 4.K). Pressing Enter or clicking then creates a corresponding bubble.

Because a developer does not only read and write code, some special bubbles are implemented in the IDE. Note bubbles (figure 4.D) give the user the possibility to add his notes directly to the workspace. Web bubbles gives the user the access to a web browser in a bubble, Javadoc bubbles (figure 4.C) enable him to look for something in the documentation. Flag bubbles (figure 4.F) can be used to mark bubbles with an icon and an optional label. This might be useful to highlight bugs, todo items or anything else. Bug bubbles(figure 4.G) show the user bugs from a database.

Debugging with bubbles
One important feature of the IDE is the bubble based debugging support. The bubbles are useful for debugging, because they are well qualified to show the program context over time and not only at a single point. So whenever the program stops, a new debugging area is displayed and a code bubble with the corresponding code is opened together with a call



Bug #987 (OPEN)
Bubbles using FlowDocument have unnecessary border padding

Reported: 5/12/2009 9:04:00 AM by Andrew Bragdon
Last Modified: 5/24/2009 8:58:00 PM
View full history
Assigned To: Joshua Kaplan
QA Contact: jak2@cs.brown.edu

Description
+++ This bug was initially created as a clone of ~~bug #455852~~+++

Mozilla/5.0 (X11; U; Linux i686; en-US; rv:1.9.1b1pre) Gecko/20080918020356 Minefield/3.1b1pre

Closing the last tab of a window now closes the window instead of blanking the last tab. This was caused by ~~bug #392870~~. I know it was intentional, but I'm filing this bug for people who don't like the new behaviour.

This was the original summary of ~~bug #455852~~ before it was morphed

Ideas

- it might not be creating the object at the right time

- the input temp file might be corrupted

- _mp may be only partially initialized

```
DrawDemo▸MainPanel▸
private static final long serialVersionUID =
    1L;
private ShapeButton _lineB;
private JButton _statsButton;
private JButton _deleteShape;
private ButtonHolder _holder;
private DrawingPanel _dP;
private ShapeInfoPanel _shapeInfoPanel;
```

```
DrawDemo▸ShapeButton▸
public void makeButtonActive(){
    this.setFocusable(false);
}
```

```
DrawDemo
ButtonHolder
_currentSelection   private ShapeButton _currentSelection, _nextSelection;
ButtonHolder        public ButtonHolder(ShapeButton b){
setSelected         public void setSelected(ShapeButton button){
DrawingPanel
  DrawDemo▸DrawingPanel▸
  public boolean isSpecifyDimensions() {
      return ShapeButton.isSpecifyDimension();
  }
MainPanel
_lineB              private ShapeButton _lineB;
MainPanel           ShapeButton[] shapeButtons = this.createShapeButtons();
createShapeButtons  public ShapeButton[] createShapeButtons(){
                    ShapeButton[] shp = new ShapeButton[4];
                    ShapeButton[] shp = new ShapeButton[4];
                    shp[0]=ShapeButton.getInstance("Line",this);
```

javax.swing.JButton
void **add**(PopupMenu popup)

Adds the specified popup menu to the component.

Parameters
popup (PopupMenu)
the popup menu to be added to the component.
Throws

Method inherited from java.awt.Component

```
DrawDemo▸MainPanel▸
public MainPanel(JFrame mainFrame){
    this.setLayout(new BorderLayout());

    this.createPropertyButtons();

    JButton featureButton=SpecialFeatureButton.
        getInstance(this);

    JButton randomShapes= this.
        createRandomShapeButton();

    String[] messages = this.
        generateStatisticsMessages();
    this.handleStatisticsGUI(messages);

    JMenuBar menuBar=this.createMenuBar();

    ShapeButton[] shapeButtons = this.
        createShapeButtons();
    JPanel shapePanel = this.makeShapeButtonPanel(
        shapeButtons);

    this.createPasswordHandler();

    JPanel moreFunctionsPanel=new JPanel(new
        GridLayout(0,1));
    JLabel moreFunctionsLabel=new JLabel(
        "More Functions",SwingConstants.CENTER);
    moreFunctionsPanel.add(moreFunctionsLabel);
    moreFunctionsPanel.add(randomShapes);
    moreFunctionsPanel.add(_deleteShape);
    moreFunctionsPanel.add(_statsButton);
    moreFunctionsPanel.add(featureButton);

    _shapeInfoPanel = new ShapeInfoPanel();
```

```
DrawDemo▸MainPanel▸
public void createPropertyButtons(){
    _deleteShape=DeleteButton.getInstance(
        this);
    _deleteShape.setFocusable(false);
}
```

```
DrawDemo▸MainPanel▸
public JMenuBar createMenuBar(){
    JMenuBar menuBar=new JMenuBar();

    JMenu menu1= new JMenu("Image/TextInput");
    JMenu menu2= new JMenu("Ordering Features");
    JMenu menu3= new JMenu("Extra Features");
    JMenu menu4= new JMenu("Grid Features");

    menuBar.add(menu1);
    menuBar.add(menu2);
    menuBar.add(menu3);
    menuBar.add(menu4);

    this.createMenu1(menu1);
    this.createMenu2(menu2);
    this.createMenu3(menu3);
    this.createMenu4(menu4);

    return menuBar;
}
```

```
DrawDemo▸MainPanel▸
private void createMenu1(JMenu m){
    JMenuItem load = LoadMenuButton.getInstance(
        this);
    JMenuItem textInput = TextMenuButton.
        getInstance();

    m.add(load);
    m.addSeparator();
    m.add(textInput);
}
```

```
DrawDemo▸MainPanel▸
private void createMenu2(JMenu m){
    m.add(SaveLocationsButton.getInstance(
        this));
    m.add(RestoreLocationsButton.getInstance(
        this));
    m.add(OrganizationButton.getInstance(
        this));
}
```

```
DrawDemo▸MainPanel▸
public JButton createRandomShapeButton(){
    JButton button=RandomShapeButton.getInstance(
        this);

    button.setFocusable(false);
    return button;
}
```

Figure 4. Code Bubbles — IDE User Interface [2]

stack bubble. The user can now work in this working set to edit the code. For exceptions a Javadoc bubble detailing the thrown exception is displayed.

A debugged instance is stored in a so called channel. It is possible to save a session and reload it. To compare the current channel with other channels, a channel can be displayed below the current channel.

Evaluation
Bragdon et. al evaluated Code Bubbles quantitatively and qualitatively.

For the quantitative analysis they compare Code Bubbles with the common IDE Eclipse to find out how many functions one can see simultaneously and how many UI operations one must use to create concurrently visible working sets for both IDEs. Therefore they looked at three open source Java applications and compared the results for the worst case, the random case and the typical case. In every case Code Bubbles was able to show more functions simultaneously. The average increase was between 28.55% and 83.33%. In the random and typical case Code Bubbles reduces the UI operations by an average of 54.64% and 47.07%. Only in the worst case Eclipse performed better in 3 of 6 trails.

In a qualitative evaluation 23 professional developers got six tasks to solve for an unknown Java project with 2658 lines of code in the Code Bubbles IDE to solve in about 1.5 hours. In the tasks the participants had to compare methods, understand code and code hierarchy, get interrupted to work on other working sets and later resume the interrupted part. Also, debugging tasks were part of the evaluation. The results were very positive. The tool gets a rating of 4.33 ± 0.26 on a 5-point Likert scale (5.0 = ”very convenient”). The developers liked, that the IDE is working set-based and that they could use the large virtual screen space, also the Debugging features were rated very high. To edit code the developers state, they could use Code Bubbles for most tasks, but there might be tasks, in which they want to use the file structure and bring the class to the front. Nevertheless Code Bubbles seems to be a very promising project we would like to hear more about in the future.

LEARNING

Relo

Relo [7] is a tool developed by Sinhal et al, which helps the developer to get an overview of a project. With Relo the developer can explore the static structure of the code in a visualization similar to UML diagrams. This is very helpful, if a developer gets the task to edit the code of a project, for example to add new features, even if the developer never worked with this project before. Then Relo gives him the possibility to easily get a good visualization of the needed parts in code. Therefore the tool gives an overview of the code part the user chooses by showing an inheritance tree as well as an call-hierarchy tree in one view.

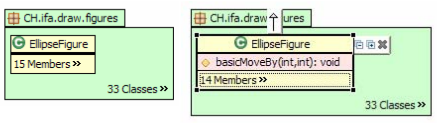
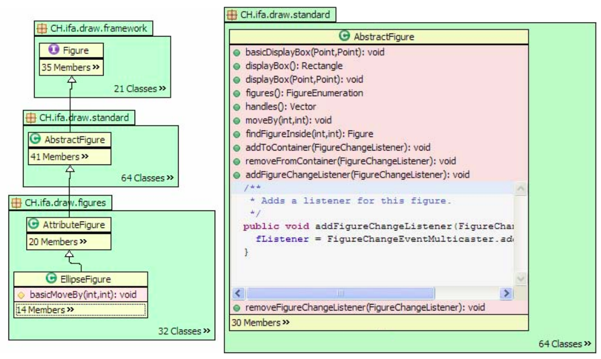


Figure 5. Relo — a starting class in Relo (left) and a class with an added method (right) [7]


CH.ifa.draw.framework
Figure
35 Members »
21 Classes »
CH.ifa.draw.standard
AbstractFigure
41 Members »
64 Classes »
CH.ifa.draw.figures
AttributeFigure
20 Members »
EllipseFigure
basicMoveBy(int,int): void
14 Members »
32 Classes »
CH.ifa.draw.standard
AbstractFigure
basicDisplayBox(Point,Point): void
displayBox(): Rectangle
displayBox(Point,Point): void
figures(): FigureEnumeration
handles(): Vector
moveBy(int,int): void
findFigureInside(int,int): Figure
addToContainer(FigureChangeListener): void
removeFromContainer(FigureChangeListener): void
addFigureChangeListener(FigureChangeListener): void
removeFigureChangeListener(FigureChangeListener): void
30 Members »
64 Classes »

```
/**
 * Adds a listener for this figure.
 */
public void addFigureChangeListener(FigureChan
  fListener = FigureChangeEventMulticaster.ad
}
```

Figure 6. Relo — showing class inheritance (left) and a view with open methods, code editing is supported (right) [7]

To achieve this the developer has to choose a relevant class as a starting point and add it to the view (figure 5). From this first class he can traverse the inheritance tree and add other classes to the view (figure 6). In every class, it is possible to show the methods of this class and if he finds a relevant method, he can mark it, so it stays in the view. It is also possible to edit the method directly in the diagram (figure 6). Then for important methods the developer can expand them, which then shows the call-hierarchy to this method (figure 7).

Relo is implemented as an Eclipse plugin. So a user can work in his projects with Eclipse as usual, but also has the possibility to switch to the Relo view, which is automatically updated, when the developer works with the code inside other Eclipse views.

To operate with Relo there are so called navigation aids. These are context-sensitive buttons, which are shown beside the currently selected element of code. There are only navigation aids for the most common relationships.

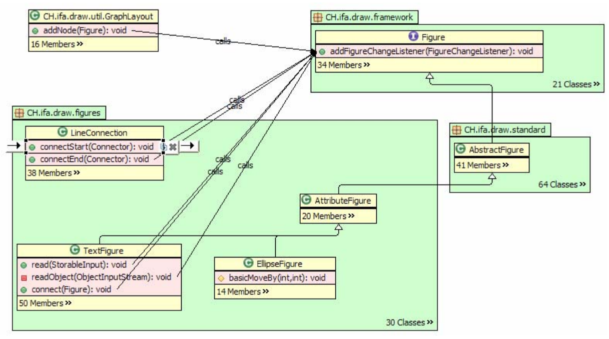


Figure 7. Relo — a view with call-relationships and inheritance relationships [7]

It is possible to add annotations to the view. Relationships between items can be labelled, it is possible to group items, and comment writing is supported as well. Relo supports some automatic services, but gives the developer the possibility to control the view by himself. For example Relo automatically draws relationships between items in the view, or the containing class is drawn, if there are multiple items with the same parent. Also an autobrowse feature is available. If multiple artifacts are selected, autobrowse searches for user items which are related to at least two of the selected artifacts and add them to the view. But if the developer chooses to do so, he can edit the view by himself and therefore is always able to remove items by clicking the corresponding navigation aids.

As Relo does not want to overwhelm the developer, the default is to show as little information as possible. For example a class or a method initially only shows its name, but can be expanded by clicking the corresponding navigation aid.

In every expansion level a maximum of ten new elements should be shown. To manage this problem, Relo automatically groups the new elements, for example based on access or name. Also there are some default view constraints. Method calls are drawn horizontally, inheritance edges vertically and other layout choices are realized in a way the developer would expect them.

Relo has been evaluated by nine developers with an average of 6.75 years experience with Java. They get a short introduction in Relo and then had to solve three programming tasks in a project of over 150.000 lines of code. First there was a warm-up task during which the participants get help with Relo. Then they have to fix a bug in the code, and the last task was a feature addition. The developers used Relo in an average of 55% of their task time. Most participants didn't feel overwhelmed by Relo, considered the tool useful and would like to use it for their own projects. But they also mentioned some disadvantages of Relo, for example the missing integrated search function and some bugs in the tool.

Nevertheless Relo seems to be a good approach to give the developer the possibility to visualize projects with a large code base, to learn about the semantic structure of the source code.

ParseWeb

Thummalapenta and Xie have worked with another approach and developed a tool called ParseWeb [9]. ParseWeb gives the developer a nice possibility to reuse existing libraries and frameworks, which are available on the web. So the developer doesn't have to write code again, which has already been written before. ParseWeb can solve problems where a source object has to be transformed to a destination object. It shows the user often used method-invocation sequences (MIS). ParseWeb is implemented as an Eclipse plugin and supports searching for code written in the Java language.

The approach is splitted into five components. There is the code search engine, the code downloader, the code analyzer, the sequence postprocessor and the query splitter. Google Code Search is used as code search engine, but it would not be difficult to implement an alternative search engine instead. The code downloader submits the queries from the developer, which has to be in the form *Source* $\rightarrow$ *Destination*, to the code search engine and downloads the corresponding code and stores it locally.

The code analyzer analyzes the locally stored code and tries to find different MISs and cluster similar MISs to present a solution to the query. Therefore it first builds an abstract syntax tree for the available code samples. Based on the abstract syntax tree a directed acyclic graph is created to handle the control-flow information and method inlining. Tree nodes represent single statements and the edges the possible control flow. With some heuristic methods additional type information is gathered. Then MISs are generated from the directed acyclic graph, which are using the shortest possible path from the source class type to the destination type.

Then the sequence postprocessor clusters MISs by using another similar heuristic. Afterwards the sequence postprocessor builds a ranking based on the importance. As last step, if at least one useful solution has been determined, the query splitter shows this solution. Otherwise the calculated MISs are splitted into sub-queries, which are processed in a divide-and-conquer fashion.

Hipikat

It is a common task for developers to explore an unknown codebase. This is the case if new people enter an existing team, as well as when a third party library has to be used in a project. In an onsite development team where there is knowledge already available to some developers, this task can be supported by mentoring. But today, there is a lot of code developed as open source projects, where such a process is uncommon. In the navigation section we introduced research, which attempts to optimize this process by providing an optimized interface.

Especially in open source projects there is a lot of scattered documentation available, eg. commit logs in a version control system (VCS), bug databases, newsgroup posts or email lists and information on websites in various forms usually exists. These secondary products of the development effort are called *artifacts*. In this section we will introduce Hipikat, an extension to the Eclipse IDE developed by Čubranić and Murphy [10], which uses such artifacts to model a group memory of the project. This can be utilized by developers to query for pieces of information related to a task at hand. Hipikat was developed as a research prototype tailored to the Eclipse project in 2003, i.e. to support the development of Eclipse itself. Also, the website[1] was last updated that year. As a result, it interfaces with the tools of the Eclipse developers' community at that time, which used Concurrent Versions System (CVS[2]) as VCS and newsgroup postings, which might not be the state-of-the-art tools in current development. But this can be neglected, since the main contribution is the tool's concept which should be easily adaptable to a modern VCS or eg. forum posts.

Server architecture

Hipikat is implemented as a client-server architecture, where the server is responsible for maintaing a database of available artifacts. This is done by three modules, which are respectively retrieving new artifacts from the mentioned sources, updating the internally modelled representation from these artifacts and responding to user queries by determining the appropriate pieces of information and providing links to them.

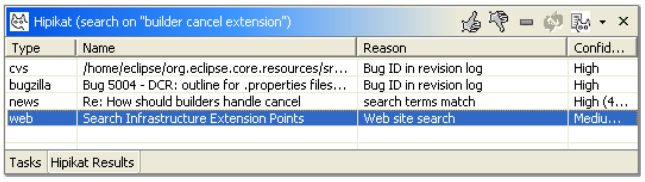
The update module uses CVS and the Network News Transfer Protocol (NNTP) to fetch source code commit logs and news posts, while Bugzilla and project websites are retrieved using a webcrawler. The identification module has submodules to update the group memory based on the newly captured artifacts. CVS commits are matched by common patterns (eg. "fix for bug 1234") to corresponding bugs, recorded as an *implements* links. When a commit and a change of a bug status to "fixed" occur in a small time window, an *implements* link from the commit to the bug is created as well. If multiple commits with an identical log comment, author and within the same time window are detected, the commits are linked together in a similar fashion. A simple matcher is applied to news postings, mapping the news threads together in the group memory. All artifacts in the group memory are also indexed for text similarity: per-file indexing can be done as soon as the artifact is captured by the system, global indexing can be performed after all new artifacts have been retrieved.

The selection module takes queries from the developer and processes them for text similarity like another artifact as described in the identification module. The similarity of artifacts is then used to create a cluster of similar artifacts, capped at 15 items. Submodules corresponding to the different submodules of the identification process provide further artifacts related to the query. The results are provided with *reasons* why they are recommended as well as a measure for *confidence*. If artifacts are relevant via multiple paths in the group memory, this is accounted for in the corresponding *confidence*.

Client design

The client is integrated in the Eclipse IDE, supplementing the search dialog with a new tab containing a simple text input field to query the group memory. Furthermore an artifact can be used as query source, by selecting "Query Hipikat" via the context menu in the outline view, package explorer or CVS repository view. A query results itself can be used to perform a new search. The search results are displayed in a newly introduced view as depicted in figure 8. In addition to a name summarizing the results content, columns for *reason* and *confidence* are available as well as a type. The type column enables developers to sort the list in order to quickly assess results from a specific type of source.

[1]http://www.cs.ubc.ca/labs/spl/projects/hipikat
[2]http://savannah.nongnu.org/projects/cvs


Hipikat (search on "builder cancel extension")

Type	Name	Reason	Confid...
cvs	/home/eclipse/org.eclipse.core.resources/sr...	Bug ID in revision log	High
bugzilla	Bug 5004 - DCR: outline for .properties files...	Bug ID in revision log	High
news	Re: How should builders handle cancel	search terms match	High (4...
web	Search Infrastructure Extension Points	Web site search	Mediu...

Figure 8. Hipikat search result view [10]

The results are either opened in an integrated editor (CVS files, Bugzilla entries) or with an external application (news posts, website results). Additionally, CVS entries can be opened using Eclipse's revision comparing view, in order to allow the developer to quickly review modifications introduced by a related commit. The list can be reorganized by the user, although this interaction is not persisted in any way. Čubranić mentioned this might be used in later versions to further refine the group memory.

Evaluation

Two case studies were done, the first one on a student assignment with 12 pairs of students. 5 pairs did not provide the researchers with their final reports. Most of the other participants reported that Hipikat recommended useful artifacts, mainly similar bug entries, which provide links to version changes in the VCS. This helped the subjects identify relevant classes as well as how similar changes were implemented. One pair of students reported, that Hipikat provided a useful artifact pointing to a specific class, but only after they worked out the change on their own, did they realize it's relevance. In their feedback another pair of students mentioned that Hipikat provided the initial links to relevant files, from where they explored the source code in a bottom-up approach.

In the second study the first author reimplemented an already fixed bug in the Eclipse bugzilla database. In that case Hipikat provided useful links to artifacts related to half the task at hand. For the remaining part, the linked artifacts were not applicable, as the hook by which the already implemented code is activated, had to be different. The author then investigated this issue without the help of Hipikat and concluded that the means available in Eclipse were sufficient. Unfortunately a comparison between the author's bugfix and the original solution implemented without Hipikat was not presented.

SOCIAL ASPECTS IN PROGRAMMER INTERFACES

Syde

When software is developed by a team, further requirements to an integrated development environment emerge. Current IDEs therefore integrate version control software, e.g. Subversion[3] or CVS. But this only serves as a partial solution to the collaboration problem, as version control system work on a first-come first-served basis, i.e. whoever commits his changes first updates the current revision. If conflicts occur, they have to be resolved by the developer who wants to commit his edited source. Furthermore the current revision has to be actively checked out, in order to see conflicts with the current working revision. In effect this means, the problem becomes only apparent, after it already occurred.

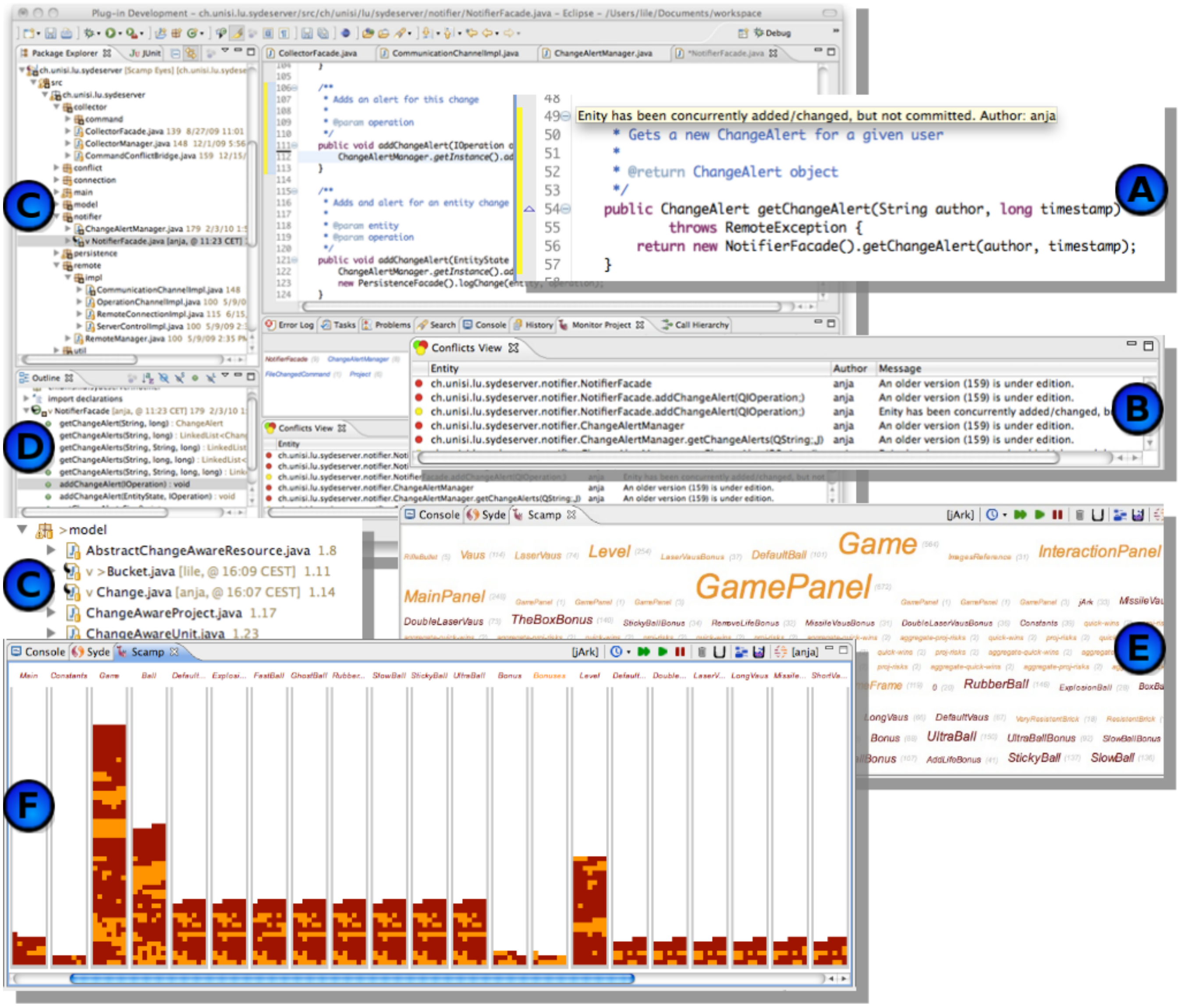
Syde, a tool for **sy**nchronous **de**velopment, is an extension to the eclipse IDE developed by Hattori and Lanza [4] which tackles this problem by improving the developers' awareness of each other. It is implemented as a client-server architecture, with a centralized server collecting changes and checking them for conflicts. The client side consists of extensible plugins which capture code deltas, provide decorations and additional views. As code is written, the deltas are transfered to the server and checked in real-time for possible conflicts with other developer's changes. This enables merge conflict detection as early as possible, so programmers can communicate and adjust early on which approach is to be taken.

As new code is written, Syde models the changes closely to the underlying structure. This is achieved by modelling the change deltas based on the abstract syntax tree (AST) representation of the source code, which is more fine-grained and enables conflicts to be detected in a very precise manner. A file-based revision control system does not provide a model for common changes which might occur, like refactoring a function out of a class. If one developer is changing this function before the refactoring takes place, a second developer doing the refactoring might not check the state of the original location of the function (or it might not even be committed yet), as he's now working on his refactored version. Whichever developer tries to commit after the the other has done so, would have to merge the conflict. But a regular revision control software doesn't even map the changes to one merging context. With Syde, these conflicts would be perceived as they occur and maybe an conflict could even be avoided if one developer can wait for the other's changes to be finished first. But even if this is not possible, the respective fragments can be isolated and reviewed together, easing the task of merging.

If conflicts between different working copies are detected, they are annotated in the source code editor (figure 9.A) and appear in a dedicated conflict view (a detailed list view, figure 9.B). In the conflict view entries are marked yellow, if both changes are not commited to the VCS yet and if an edited version conflicts with the VCS version, it is marked in red. Additionally, all files which differ from the VCS version are marked in the project explorer and outline with an overlay icon and an arrow (pointing upward if the developer himself has changed the code and pointing downwards if another developer changed the file; figure 9.C and 9.D). In the latter case, the developers name and a timestamp of the the last modification is added.

Furthermore, the history of the AST's elements can later be reviewed, even if changes like the mentioned refactoring occurred. This again is superior to a file-based approach, as the system can extract the relevant changes (eg. to a function), showing a complete history which might span multiple files if it was moved. Commits unrelated to a function, but updating it's class source code file, can be filtered out.

[3]http://subversion.apache.org/



Enity has been concurrently added/changed, but not committed. Author: anja

```
     * Gets a new ChangeAlert for a given user
     *
     * @return ChangeAlert object
     */
    public ChangeAlert getChangeAlert(String author, long timestamp)
            throws RemoteException {
        return new NotifierFacade().getChangeAlert(author, timestamp);
    }
```

Conflicts View

	Entity	Author	Message
●	ch.unisi.lu.sydeserver.notifier.NotifierFacade	anja	An older version (159) is under edition.
●	ch.unisi.lu.sydeserver.notifier.NotifierFacade.addChangeAlert(QIOperation;)	anja	An older version (159) is under edition.
●	ch.unisi.lu.sydeserver.notifier.NotifierFacade.addChangeAlert(QIOperation;)	anja	Enity has been concurrently added/changed, b
●	ch.unisi.lu.sydeserver.notifier.ChangeAlertManager	anja	An older version (159) is under edition.
●	ch.unisi.lu.sydeserver.notifier.ChangeAlertManager.getChangeAlerts(QString;J)	anja	An older version (159) is under edition.

```
>model
  AbstractChangeAwareResource.java 1.8
  > Bucket.java [lile, @ 16:09 CEST] 1.11
  Change.java [anja, @ 16:07 CEST] 1.14
  ChangeAwareProject.java 1.17
  ChangeAwareUnit.java 1.23
```

Figure 9. Syde — A) Source code annotation B) Conflicts View C) Project Explorer decorations D) Outline decorations E) WordClouds View F) Bucket View [4]

In contrast, with a regular VCS the developer has to know or find the relevant files in the history, can only look at the complete history of that file and depending on the precision of the comments this might be a tedious or error-prone task.

Syde's approach could also work with a distributed version control system (like GIT[4]), since the basic workflow of committing revisions is the same. This would yield the same results, as a distributed VCS does not provide real-time changes. Syde seems better suited for development than concurrent editing as available in Google Docs[5] or Microsoft Word 2010[6]. Concurrent editing would often result in temporary versions which are not compilable, therefore inhibiting the individual programmer in his workflow.

To improve team awareness, Syde introduces two new views which enable developers to get an overview of the project history. The WordCloud (figure 9.E) view shows a list of modified classes, ordered with recently modified classes first and color-coded to show the last committer. The font size is increased based on the number of changes to a class. This provides a quick overview of past development, visualizing the focus classes received and who has been recently working on which parts of the software. To provide a more detailed view on who contributed to a class, a bucket view is available (figure 9.F). It shows one column (bucket) for each class, which is filled with small squares for each change, with color indicating which developer was responsible for the change. The Bucket is filled from bottom to top in chronological order, thereby enabling the users to quickly deter-

[4]http://git-scm.com/
[5]http://docs.google.com/
[6]http://office.microsoft.com/de-de/word/

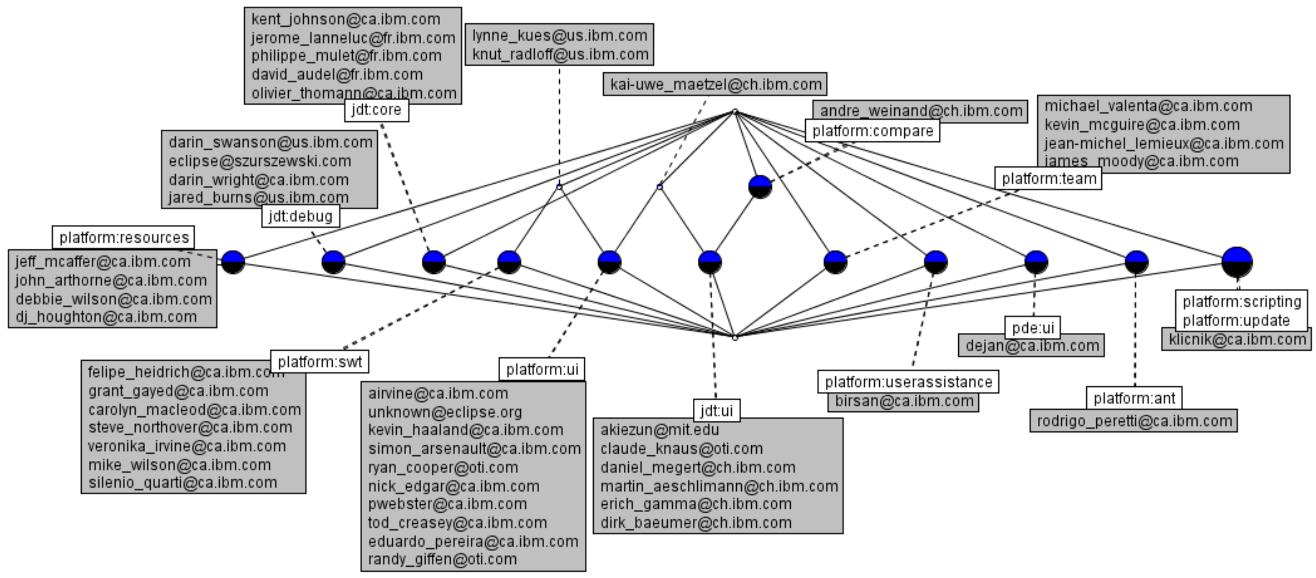


Figure 10. Social hierarchy of eclipse v1.0 bug assignees [11]

mine who took part in development of which classes, when he was involved and how much each respective developer contributed. The idea of these additional views seem valid, although it might be better suited to integrate the presented information on a method level. Code Bubbles could be a way to do this, since it is more tailored towards showing related code in one context.

Visualizing Social Hierarchies

In the last section, Syde and its different views of the project history were introduced. Yet those views are based on classes, so if a project becomes bigger and the more developers are involved the harder it gets to still extract critical knowledge from these views. Usually when a software system grows, it is split into different modules and different teams maintain different parts of the code base. Wermelinger et al. proposed using formal concept analysis to extract social hierarchies from the project history [11]. Based on this modelling, it is possible to identify lead developers with the best system overview, people having worked on certain parts of the system (eg. java packages), overview these developments over time and determine which parts run the risk to become legacy code.

Formal concept analysis is based on lattice theory, which we will introduce by example: Pairs of source code artifacts and developers who worked on them form the primary *concepts* $\langle o, a \rangle$. $o \subseteq O$, the set of all source code artifacts. $a \subseteq A$, the set of all developers. The lattice defines an ordering, $\langle o, a \rangle \leq \langle o', a' \rangle$ if $o \subseteq o'$. This ordering implies a hierarchy, since a larger set of source code artifacts implies, that developers who worked on all parts, also worked on each subset of the code artifacts: $a' \subseteq a$.

To work with this structure, the smallest *concepts* should be defined in an architectural context, since a finer grained model would not work with the lattice. Assuming a developer worked on a module, it is unlikely that he committed source code to each class. Yet if the model is applied on that level, the lattice hierarchy would require just that if he is to be included in a *concept* of the whole module. Wermelinger and colleagues extracted this data from the Eclipse Bugzilla database and mapped the Eclipse components to the bug assignees. Furthermore they filtered out developers who were not involved in a enough bugs, to reduce the noise in the data. Based on this data a visualization was created using Concept Explorer (ConExp[7]), which allows browsing a concept lattice structure interactively. Instead of utilizing a bug database as data source, using VCS information or e-mail provides similar information. In the case at hand, the approach was to use the bug database instead, in order to capture people who might not have commit rights but contributed via Bugzilla.

An example visualization for Eclipse release 1.0 is depicted in figure 10. The relative flatness of the hierarchy can be attributed to the fact that the eclipse project is well structured, since bugs can be resolved by teams specialized on the respective components. A few people resolve bugs in different components, this kind of information becomes valuable, if these people are leaving the project. The structure shows other developers who already have a subset of experience with parts of the project, as they could discuss who takes such an architecture integrating role. Tools like Syde could be complementary used to determine parts of the software with which the replacement developer(s) should acquaint themselves.

If the changes of the structure are examined, those parts of the system which became legacy code (because nobody

[7]http://conexp.sf.net/

maintains them anymore) can be identified and critical decisions for further development can be improved. Today, the prerequisite data is already available, so these decision can be made at reasonable times in the development cycle and not when problems occur. Furthermore, when projects are big, key people to drive decisions can be extracted from the model. For example if a new component has to be developed, which interacts with two other components: People knowing the existing components, as well as developers having a good overall system understanding, can be identified enabling an efficient communication between the most qualified developers.

SUMMARY AND OUTLOOK

The main contribution an interface for programmers should accomplish, is to ease the programmer's work in his day to day workflow processes. We decided to leave domain specific approaches out of the scope of our work, in order to concentrate on solutions to more general problems. Some approaches could straight-forward complement each other: Code Bubbles could use the concepts implemented in Hipikat, since it provides a good semantic workflow, yet all external artifacts seem to be only accessible by context menus and lists. Syde could also provide some functionality to Code Bubbles, since both annotations and the additional views could be added.

Navigating the source code is an important part and in addition to Code Thumbnails and Code Bubbles, which are strong contributions in this concern, Relo, Syde as well as the paper on social hierarchy visualization add valuable approaches. Code Thumbnails is a valid idea, since the state-of-the-art navigation available today obviously is not optimal. But although using spatial maps to navigate might be successfully applied in some cases, the approach is based on hiding information from the developer. Forming a map in perceptual way could as well utilize a meaningful representation. Relo is a nice example of an approach to generate such a structure. Code Bubbles implements an overview which exemplifies utilizing clusters of task related code fragments, yet could vastly profit from a more generalized depiction when zooming out to further improve navigation efficiency.

For learning purposes, we found Relo, Parseweb and Hipikat to serve as good examples of how to improve this process. Yet Syde and Code Bubbles would nicely complement in this regard: Syde's approach to capture smaller code changes with time-stamps would be an interesting addition, since it could be used as additional crowd-sourced input to refine the algorithmic approaches of the former tools. Code Bubbles, as an in-depth, efficient way to visualize the semantic structure of object oriented code in general, would surely ease these learning tasks as well.

In our opinion, the papers on social aspects are important, because as long as machine learning does not reach the point, that it can understand what a specific code is supposed to accomplish, crowd sourcing provides a possibility to augment relatively simple algorithms with intricate knowledge. Code Bubbles shows how valuable it is to utilize semantic knowledge about the code structure. Yet to go a step further and generalize the structure to provide eg. an architectural overview, computer science has as of today no means available to accomplish this automatically.

It remains to see if an approach like Code Bubbles can take off in a day-to-day development process. With spatially distributed teams ever more common, especially in open source projects, time will show if (distributed) version control and the established communication is sufficient or if advanced concepts as summarized by our paper become the new state-of-the-art.

REFERENCES

1. A. Bragdon, S.P. Reiss, R. Zeleznik et al. Code bubbles: a working set-based interface for code understanding and maintenance. CHI '10, pages 2503–2512, 2010.
2. A. Bragdon, S.P. Reiss, R. Zeleznik et al. Code bubbles: rethinking the user interface paradigm of integrated development environments. ICSE '10, pages 455–464, 2010.
3. R. DeLine, M. Czerwinski, B. Meyers, et al. Code Thumbnails: Using Spatial Memory to Navigate Source Code. VLHCC, pages 11–18, 2006.
4. L. Hattori and M. Lanza. Syde: a tool for collaborative software development. ICSE '10, pages 235–238, 2010.
5. A.J. Ko, H.H. Aung, and B.A. Myers. Eliciting Design Requirements for Maintenance-Oriented IDEs: A Detailed Study of Corrective and Perfective Maintenance Tasks. In *ICSE '05*, pages 126–135, 2005.
6. A.v. Mayrhauser and M.A. Vans. Program Comprehension During Software Maintenance and Evolution. *Computer*, 28(8):44–55, 1995.
7. V. Sinha, , D. Karger, and R. Miller. Relo: helping users manage context during interactive exploratory visualization of large codebases. eclipse '05, pages 21–25, 2005.
8. J. Smith. A Comparison of the IBM Rational Unified Process and eXtreme Programming. *IBM/Rational*, 2004.
9. S. Thummalapenta and T. Xie. Parseweb: a programmer assistant for reusing open source code on the web. ASE '07, pages 204–213, 2007.
10. D. Čubranić and G.C. Murphy. Hipikat: recommending pertinent software development artifacts. ICSE '03, pages 408–418, 2003.
11. M. Wermelinger, Y. Yu, and M. Strohmaier. Using formal concept analysis to construct and visualise hierarchies of socio-technical relations. ICSE '09, pages 327 –330, 2009.