Using Task Context to Improve Programmer Productivity

Mik Kersten and Gail C. Murphy
University of British Columbia
201-2366 Main Mall, Vancouver, BC V6T 1Z4 Canada
{beatmik, murphy} at cs.ubc.ca

ABSTRACT
When working on a large software system, a programmer typically spends an inordinate amount of time sifting through thousands of artifacts to find just the subset of information needed to complete an assigned task. All too often, before completing the task the programmer must switch to working on a different task. These task switches waste time as the programmer must repeatedly find and identify the information relevant to the task-at-hand. In this paper, we present a mechanism that captures, models, and persists the elements and relations relevant to a task. We show how our *task context* model reduces information overload and focuses a programmer's work by filtering and ranking the information presented by the development environment. A task context is created by monitoring a programmer's activity and extracting the structural relationships of program artifacts. Operations on task contexts integrate with development environment features, such as structure display, search, and change management. We have validated our approach with a longitudinal field study of Mylar, our implementation of task context for the Eclipse development environment. We report a statistically significant improvement in the productivity of 16 industry programmers who voluntarily used Mylar for their daily work.

Categories and Subject Descriptors
D.2.6 [**Software Engineering**]: Programming Environments – *integrated environments, programmer workbench.*

General Terms
Algorithms, Experimentation, Human Factors

Keywords
IDE, task management, interaction history, program views, degree-of-interest

1. INTRODUCTION
Modularity enables programmers to develop and evolve complex software systems. Modularity in programming languages, for instance, enables separate compilation [1], making it tractable to modify and test a small part of a system. Design modularity enables parallel development, making it tractable to develop large systems in less time [21]. Development environments use modularity to present views of a system in support of a programmer's tasks, such as bug fixes and feature additions. For example, a common way to access Java[1] code in the Eclipse[2] Integrated Development Environment (IDE) is through a view of the containment hierarchy called the Package Explorer, which shows the modular structure of projects, packages, files, and classes. Eclipse also presents search results in terms of the system's hierarchical structure. Current development environments, including Eclipse, appear to encode two assumptions: a programmer will often be able to find a desired piece of the system by traversing the modular structure, and modifications will often fit within the modular structure so that once the point of interest is identified it will be relatively easy to perform the desired modification.

We have observed two problems with these assumptions. First, many modifications to a system are not limited to one module. For example, we found that over 90% of the changes committed to the Eclipse and Mozilla[3] source repositories over a period of one year involved changes to more than one file [20]. We then selected 20 changes from Eclipse and found that 25% of these transactions involved significantly non-local changes. Second, even when the actual changes related to a modification are within some form of module, say one Java package, a programmer often needs to know how this module works within the system, requiring them to access many other modules and understand their interconnections [24]. The result is that a programmer must spend an inordinate amount of time navigating around the modularity-based views in an IDE to access the information needed to complete a particular task.

If a programmer worked on only one task at a time, the mismatch between the organization of information in the IDE and the programmer's needs might just be annoying. In practice, programmers often work on multiple tasks during a typical work day [9]. A programmer is constantly looking for the information needed to work on a particular task, setting up their workspace, and all too often before completing the task must perform similar steps for another task, only to again redo the same work when returning to the first task. This constant need to re-create the context of the task reduces the programmer's productivity.

We have been investigating how an explicit representation of the information related to a task can alleviate this mismatch and can help improve a programmer's productivity. In an initial exploration, we demonstrated how we can transform data about how a programmer interacts with system artifacts into a degree-of-interest (DOI) weighting for each program element [17]. We showed how to use this weighting to complement the way modern IDEs display modular structure by focusing views and editors on only the relevant modularity instead of displaying the modularity of the system as a

SIGSOFT'06/FSE 14, November 5–11, 2006, Portland, Oregon, USA.

[1] http://java.sun.com/reference verified 01/09/06

[2] http://www.eclipse.org verified 01/09/06

[3] http://www.mozilla.org verified 01/09/06

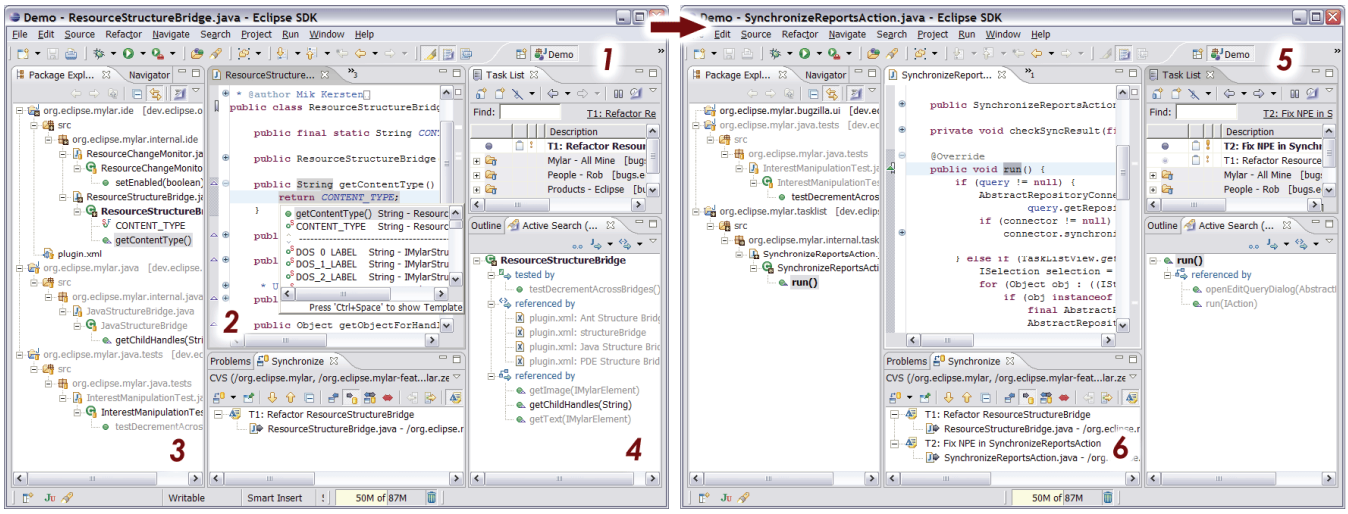

Figure 1: Task context in Mylar while working on Task-1 (left), and shortly after activating Task-2 (right)

whole. In a preliminary study we performed of this approach on six industry programmers, we found that despite promising results the approach had critical shortcomings:

- The model needed to incorporate tasks explicitly because a programmer often works on, and switches between, multiple tasks.
- The model needed to include related artifacts that were not directly accessed by the programmer but that could also be of interest for the task-at-hand.

In this paper, we report on how we overcame these shortcomings by expanding and refining the concept of a DOI function based on programmer interaction, and by creating and validating a new model that includes an explicit representation of task and contexts. A *task context* represents the program elements and relationships relevant to completing a particular task. We have progressed from the simple weighting model used in our earlier approach and in other tools (Section 7) to a more sophisticated weighting based on both direct and indirect interactions that occur with program elements, relations between elements, and the programming tasks themselves. Previous work, including our own, considered only elements and direct interactions (see Section 7).

In this paper we present a generic description of our task context model (Section 2) and describe operations that can be performed on the model to focus programming tools (Section 3). Our Mylar tool[4] (v0.3 to the current v0.6) implements our approach by extending the Eclipse IDE to make task context a first-class abstraction (Section 4). We validated our new model through a field study in which 16 industry Java programmers used Mylar for multiple weeks of their daily work (Section 5). We describe how the use of Mylar results in a statistically significant increase in a programmer's *edit ratio*: a measure of the amount of editing versus the number of selections performed by the programmer. We conclude the paper by discussing the benefits and limitations of our task context model (Section 6) and comparing our model to earlier efforts (Section 7).

1.1 Example

To give a flavor for our approach, we provide an overview of what it is like to use our Mylar tool. The scenario involves a Java programmer using Mylar on a code base with over one thousand classes. One task on which the programmer is working is an improvement to the code base of the system.

Task-1: Refactor `ResourceStructureBridge`

This task involves identifying, inspecting and changing all of the clients of `ResourceStructureBridge`. In a Mylar-specific view, the programmer has named this task and selected it to indicate that it is the active task (i.e., in the Task List view the task has a solid dot next to it, Figure 1-1). Activating the task causes Mylar to track the parts of the system artifacts—the program elements and relationships—that the programmer accesses while working on this task. From this interaction, Mylar builds a model of the task context.

Mylar uses the explicitly modeled task context as a parameter to a filtering operation that shows the programmer only the information needed to complete the task. For example, even though the code base contains over 1000 classes and numerous other kinds of artifacts, only the artifacts relevant to the current task context are visible in a hierarchical modularity-based view of the system structure (the Package Explorer, Figure 1-3); all other elements are elided from the view. This view also indicates the relevance of elements to the task context by making the most relevant bold. As an example of another operation on a task context, Mylar expands the task context to include structurally related elements of potential interest (the Active Search view, Figure 1-4).

As the programmer is working on Task-1 a new high priority bug is assigned to him that must be attended to immediately.

Task-2: Fix NPE in SynchronizeReportsAction.

Using Mylar's Task List view, the programmer activates the second task (view Figure 1-5), causing the context of the first to be stored and all files in the context to be closed. As the programmer starts working, a context starts building up for the second task. The IDE views are now filtered according to the second task context, although the underlying system information has not changed. To return to the first task, the programmer simply needs to reactivate it, causing the views and editors to return to the state visible on the left of Figure 1.

[4] http://eclipse.org/mylar, verified 01/09/06

2. TASK CONTEXT MODEL

We define a task as "a usually assigned piece of work often to be finished within a certain time" [18]. For a programmer, tasks include bug fixes, feature additions, and code base explorations. Some of these tasks are short-lived, requiring only a few minutes to complete; others are longer-lived, sometimes being worked on each day over the course of weeks or months. We focus on this atomic unit of a task. Higher-level abstractions and organization of tasks, such as hierarchies and sequences of task [2], can be layered on top.

In our approach, a task context is the information—a graph of elements and relationships of program artifacts—that a programmer needs to know to complete that task. Each element and relationship in the model corresponds to a weighting of its relevance to that task. For example, Task-2 includes the Java class of interest, all methods that refer to the class, any tests of the class and the XML elements that refer to it. The elements with highest relevance will be those that the programmer edited and selected most. We form a task context from the interactions that a programmer has with system artifacts and from the structure of those artifacts. In this section, we describe the interaction stream and algorithm that we apply to the stream to create a task context. In Section 3 we sketch operations on and with task contexts that can be used to facilitate a programmer's interaction with system artifacts. After presenting the model and operations, we then describe how we have realized this model in the Eclipse IDE (Section 4).

2.1 Encoding Interaction

We derive a task context from an *interaction history*, which is a sequence of *interaction events* that describe accesses of and operations performed on a software program's artifacts. Each event captures six pieces of information (Table 1).

Table 1: Interaction event data

Time	The time of the event occurrence
Kind	Classification of event (Table 2)
Origin	Identifier describing the UI affordance or tool that caused the creation of the event
Content Type	Identifier describing the kind of element operated upon.
Handle	Identifier for the target element
Delta	State change that occurred with the event

Some interaction events are the result of the programmer's *direct interactions* with program elements. For instance, a programmer may select a particular Java method to view its source, edit it, and then save the file containing it. Each of these actions corresponds to an event of a different kind being appended to the interaction history (Table 2).

Other interaction events are *indirect*, where program elements and relationship are affected without being directly selected or edited by the programmer. For example, when working on Task-1, the programmer refactors the name of the `ResourceStructureBridge` class, causing all of the elements referring to that class to be updated. Each referring element updated through the refactoring results in an indirect *propagation* of the edit being appended to the interaction history. When the programmer directly selects the `getContentType` method (Figure 1-2), each containing parent of that method (its class, source file, package, source folder, and containing project) becomes relevant to the context and a *propagation* of the selection is appended for each parent.

Table 2: Classification of interaction events

event kind	mode	description
selection	direct	Editor and view selections via mouse or keyboard
edit		Textual and graphical edits
command		Operations such as saving, building, preference setting
propagation	indirect	Interaction propagates to structurally related elements
prediction		Capture of potential future interaction events

The model also support *prediction* events, which describe possible future interactions that a tool anticipates the programmer might perform. An example of *prediction* is an event describing that a test may be of interest to the current task because it references a class in the task context. Table 3 sketches the sequence of interaction events that result from the programmer's initial work on Task-1. For simplicity, we use an event number to stand in for the time field of an interaction event.

Table 3: Sample interaction history

event	kind	origin	Target(s)
1	selection	Package Explorer	ResourceStructure Bridge class
2..5	propa-gation	Package Explorer	.java file, package, source folder, project
6	com-mand	Rename refactoring	ResourceStructure Bridge class
7	edit	Java Editor	ResourceStructure Bridge declaration
8..16	propa-gation	Refactoring monitor	4 XML and 5 Java references to `Resource StructureBridge`

2.2 Constructing a Task Context

We derive a task's context by processing an interaction history that describes the activity performed for a task. Each event from the corresponding portion of the interaction history contributes to a graph that represents the task context. If the handle of an event being processed refers to an element not yet represented in the graph, a node for the element is added to the graph. A *selection* event from the interaction history contributes an edge to the graph when the target element of the current *selection* event is structurally related to the target element of the last selection event processed. For example, if a programmer navigates from a method call to its declaration, the interaction history will contain the selection of the caller followed later by a selection of the callee. This results in an edge representing the Java reference relation between the two corresponding element nodes. The graph of task context can contain cycles (e.g., as a result of navigating recursive method calls) and can have multiple edges between nodes (e.g., both reference and inheritance).

We use a task's interaction history to compute a weighting for each element in the task context. The weighting is a real number value representing the element's degree-of-interest (DOI) for the task. This DOI value is based on the frequency of interactions with the element and a measure of the interactions' recency. The frequency is

determined by the number of interaction events that refer to the element as a target. Each event kind has a different scaling factor constant, resulting in different weightings for different kinds of interaction. Recency is defined by a *decay* that is proportional to the position in the event stream of the first interaction with the element; like frequency, recency is also scaled. Algorithm 1 is a naïve but clear representation of how we compute a DOI value for an *element* with an interaction history *events* sequence that contains one or more events with the element as the target. We iterate over a subsequence consisting of just the events involving the element (line 4), increment the *interest* value of the element based on the kind of the current event (line 5) and if the *interest* has not offset the decay, reset the decay to start at the last interaction with the element (lines 7-9). This algorithm ensures that elements which have decayed to a negative interest have their interest become positive when interacted with again.

```
DOI(element, events)
1  elementEvents = WITH-TARGET(element, events)
2  decayStart = elementEvents[0]
3  interest = 0
4  for each event in elementEvents
5     interest += SCALING(KIND(event))
6     currDecay = DECAY(decayStart, event, events)
7     if interest < currDecay then
8        decayStart = event                    // reset decay
9        interest = SCALING(KIND(event))   // reset interest
10 totalDecay = DECAY(decayStart, LAST(events), events)
11 return interest – totalDecay

DECAY(fromEvent, toEvent, eventSeq)
12 decayEvents = SUBSEQ(fromEvent, toEvent, eventSeq)
13 return |decayEvents| * SCALING(KIND-DECAY)
```

Algorithm 1: DOI for Task Context

The SCALING function returns the constant associated with each event kind and with KIND-DECAY. The DECAY function computes the decay to be proportional to the size of the SUBSEQ subsequence from *decayStart* to the most recent event, and includes events not in *elementEvents*. As an example, consider how the interaction history from Table 3 contributes to the weighting of the `ResourceStructureBridge` element, most recently edited at event 7. Assuming SCALING returns 1 for selections, 0 for commands, 2 for edits, and 0.1 for KIND-DECAY, and noting that there were no propagated events with that element, the three iterations through the loop will result in 1+0+2 = 3 for *interest*, and (16-1)(0.1) for *totalDecay*, resulting in a DOI of 1.5. If 30 more interactions happened with another element the DOI value would become -1.5. A subsequent selection would cause the DOI to be reset to 1 – 0.1 = 0.9.

A relation in the graph is composed of source and a target element. The DOI of a relation is computed using the same DOI algorithm, by means of the relation's target element:

DOI-R(*relation*, *events*) = DOI(TARGET(*relation*), *events*)

For example, if a programmer navigates repeatedly between a method call and its declaration, the DOI of that relation will increase from repeated selections of the declaration. If the programmer navigates 'back' and 'forward' between the two several times, the two resulting directed edges with both have the same DOI.

Our construction algorithm for task context takes as input any sequential stream of interaction events whether it is being gathered on-line or was stored off-line and reloaded. At any point in the construction process, each node and edge in the task context's graph can be queried for its DOI value; this value is computed from the interaction history associated with the task context available when the query is made. Task context can thus be built interactively as a programmer works, or recreated by parsing a previously stored interaction history.

2.3 Task Activity Context

To correlate each context to a particular task and to support multiple task contexts, a mechanism for associating interaction events with tasks is needed. We achieve this by capturing a separate stream of interaction events in which the target elements are tasks instead of system artifacts. We use the term task activity context to describe the programmer's interaction with tasks. A task activity context is a meta-context over task contexts.

We process the interaction history for the task activity context the same as we do a task context. Interaction events can be direct or indirect. For example, a programmer can indicate that work should be associated with a particular task by opening a bug report; this action causes a *selection* event on that task. Each task can have a reminder date; setting a new reminder causes an edit event on the task and can cause a future *prediction* event. The act of a programmer switching to another application window causes a *command* event that indicates work on a task has stopped. Since the model and algorithm for a task activity context is identical to that of task contexts, the operations discussed in the next section also apply at this meta-level.

3. TASK CONTEXT OPERATIONS

We can use the information in a task context as input to various operations that help focus the IDE's display of information and automate the retrieval of information specific to completing a programming task. These operations fall under two categories: those which operate on one or more contexts, and those which use a context to operate on the system's artifacts.

3.1 Operating on Task Contexts

Although a task context scopes the amount of information with which a developer works, it can still be too large or contain too many different kinds of elements and relations to assist with particular programming activities, such as unit testing. We use the term *slicing* to refer to an operation that produces a subset of a given task context. Sometimes the opposite is true and a single task context may not contain all of the relevant information needed for an activity, such as a code review. We use the term *composition* to refer to operations that produce a composite task context from individual task contexts. To enable a programmer to tailor a task context manually we also support *manipulation* operations.

3.1.1 Slicing

Task context slicing is an operation that takes as input a task context and outputs all elements and relations of the context that meet a particular constraint. A constraint can test the kinds of interactions associated with elements or relations in the context (e.g., include elements that were edited), DOI values (e.g., include elements and relations with a high DOI), or the underlying information (e.g. include elements that are Java methods). For example, a slice with the constraint to include all interesting files that have interaction events of the kind *edit* can determine which files to include in a source code commit.

3.1.2 Composition

Each interaction history corresponds to a single task. However, some programming activities can require displaying the context of several tasks simultaneously. For example, the programmer might want to create a *composite context* from Task-1 and Task-2 (Section 1.1) to perform a code review of programming activity. The composition operation takes as input one or more task contexts, and combines them to form a single composite context. The operation forms a union of all of the interaction events of both contexts, which produces a composite context where the DOI function includes interaction from each context in computing the value and inclusion of the elements and relation.

3.1.3 Manipulation

Our DOI function provides an approximation of interest, and can produce a value that fails to match the programmer's expectation either by being tuned incorrectly or by failing to monitor a relevant interaction (e.g., one performed outside of the Eclipse). We provide a mechanism for directly manipulating a task context by allowing the programmer to issue command events that result in predictable changes in the model. For example, if an element is interesting but should not be, a "Make Less Interesting" command can issue the interaction events to reduce the interest of that element.

3.2 Operating with Task Contexts

The creation of task contexts allows a programmer to build up an appropriate set of information needed to work on a task. We can use that task context to filter the amount of information presented to a programmer by *projecting* a context onto the system artifacts, and can also use it to *predict* information that may also be relevant to completing the task.

3.2.1 Projection

We can project a task context onto any data structure containing similarly structured elements and relations. A projection operation allows us to use the DOI values from the task context to create a weighted version of the target data structure. This is the operation with which we filter information not relevant to the task in the IDE's views. It can be combined with context slicing if the display mechanism is focused on displaying one kind of element or one kind of relation. For example, our Mylar tool projects a task context onto a hierarchical view of system structure (the Package Explorer, Figure 1-3) to show only elements with a positive interest in the current task context, eliding all uninteresting elements. Similarly, a projection of a task context onto a table can be used to sort elements by interest.

3.2.2 *Prediction*

A significant fraction of the commands executed by programmers are commands to look for related elements to grow a task context [19]. We can use the information in a task context to predict what elements might be relevant to completing the task, but with which the programmer has not yet interacted directly. For example, if the programmer is working on a Java class, and that class is referred by an XML element, that XML element can get a predicted interest if a tool determines that it is likely to be part of the task context at a future time. These predictions come from running automatic searches on the programmer's behalf, and can use context slices as both input and scope for the searches. The output of the prediction operation is a set of interaction events, each of which corresponds to a search result (Section 4.2.1), and each is added to the interaction history as *prediction*. This approach ensures that results are ranked using the DOI function, with less frequent results decaying in interest while more recent and more frequent results yield a higher interest.

4. ADDING TASK CONTEXT TO THE IDE

To support investigations into the effect that an explicit task context has on programmer productivity, we needed a high-fidelity integration of task context with an IDE. Mylar (v0.3 and later) implements the task context model for the Eclipse IDE.[5] In this section, we describe relevant UI features of Eclipse that have been extended or altered with task contexts and present an overview of key decisions and implementation details in the implementation of the tool.

4.1 Integration

The Mylar IDE integration allows programmers to work with task context in almost every commonly used [19] part of the Eclipse IDE: it provides DOI-based element decoration in all structure views that display Java, XML, and files; DOI-based filtering in all applicable tree and list views (Package Explorer, Document Outline, Navigator, Search, Members and Types); DOI-based ranking of elements and relationships in tables views (Content Assist and Problems view); and DOI-based folding in the editor. In addition, Mylar adds facilities specific to task context, including an Active Search view that shows relations and elements of predicted interest, an Active Test Suite that creates and runs all unit tests in the task context, and an Active Hierarchy view that shows the inheritance context of the task.

4.1.1 Supporting Task Management

IDEs provide facilities for working with files and with the structure of those files. To integrate tasks and contexts with the way that programmers work in the IDE, we added similar facilities for working with tasks, including a view for managing tasks (Task List, Figure 1-1), mechanisms for sharing and synchronizing tasks (Bugzilla[6] task/issue tracker integration) and mechanisms for working with a personalized view of shared tasks. The task management tool support also includes a radio-button style toggle that allows the programmer to easily indicate the task on which he is currently working, and browser style back/forward lists to facilitate multi-tasking. These facilities make it easy to work with tasks within Eclipse, where Mylar monitors activity (Figure 1-1).

4.1.2 Focusing the IDE on Task Context

To focus the UI of the IDE on the elements relevant to completing the programming task, the task context can be projected onto any structure view or editor in the IDE. This process involves DOI-based *filtering* and *ranking*. The example in Section 1.1 describes filtering for views. For editors the filtering process is similar, but instead of elements being hidden they are folded to hide their contents and only elements in the task context are unfolded (Figure 1-2).

Ranking can be done for views that order elements, and involves sorting each element in the view on its DOI value. For example, Eclipse's content assist provides a ranking of suggested completions in the editor based on Java heuristics. Mylar projects DOI values onto that ranking, adds a separator, and puts the elements contained in the task context on top of the content assist list (Figure 1-2). A similar mechanism works for table views such as the Problems list, in which

[5] Earlier versions did not support task context. This section describes v0.3: http://eclipse.org/mylar/doc/new.php verified 01/09/06

[6] http://bugzilla.org verified 01/09/06

compiler warnings are sorted by their DOI relevance to the active task context.

Decoration is a form of ranking that uses visual cues, such as text style and background highlight color. Schemes for decoration can display either a continuous or discrete range of DOI values. Highlight decoration of element backgrounds based on DOI is not on by default, but a user can set a different highlighter for each context, and specify if the highlighter color gradients should be discrete or continuous. Discrete interest decoration of element fonts is always on when a task is active, and supports the following interest thresholds (visible in Figure 1-3 and Figure 1-4): very interesting elements called 'landmarks' that appear bold and black, directly interesting elements that appear black, indirectly interesting elements that appear gray and uninteresting elements that are filtered and that do not appear at all. We support this with two *interest thresholds*: one for landmark DOI values, and one for 'interesting' values. Threshold tuning is discussed in Section 6.2.1.

In addition to focusing the UI of the IDE, Mylar also uses task contexts to focus existing operations in the IDE. For example, it extends Eclipse's search mechanism to provide the option of including only the active task context in textual and other searches. Mylar also maintains a test suite that slices the active task context to include all subtypes of `junit.framework.TestCase`, enabling the programmer to run only the tests relevant to the task.

4.1.3 Predicting Interest and Active Search

The Active Search view surfaces the relations and predicted interest elements in the task context (Figure 1-4). The input for Active Search is a slice of the active task context for elements with a DOI value over the landmark threshold. The scopes that Active Search uses are slices of the context model; in the UI we refer to the different kinds of scopes as *degrees-of-separation*. This term is indicative of the 'distance' from the highest interest elements, where distance is defined both by DOI level and by containment relations. For example, Active Search includes the following degrees-of-separation for defining search scopes: landmarks, interesting elements, interesting files, interesting project and dependencies of interesting projects.

As a context grows, lowering the degree of separation is akin to tightening the search scope to decrease the number of results. To focus results the Active Search view uses the same ranking and decoration mechanism as the other views.

4.2 Implementation Details

To test Mylar in an industry setting, the tool needed to scale up to handle large systems with many kinds of program artifacts. We achieved a suitable level of integration to support work on real programming tasks through an architecture that bridges to the standard Eclipse development tools. We achieved suitable performance through mechanisms for storing and collapsing interaction histories.

4.2.1 Bridge Architecture

While our task model is defined in terms of a generic set of interaction events, the actual events need to be issued by a mechanism that understands both domain structure (e.g., Java) and the UI of the tool for working with that structure (e.g., the Eclipse UI for Java development). We call this mechanism a *bridge* from the context model to the domain structure. Each bridge handles a single content type. We have created bridges for Java, two XML dialects (Ant and Eclipse plug-in descriptors), generic files, and tasks.

A *structure bridge* is responsible for mapping elements and relations in the task context to and from the domain structure of a particular content type. This involves mapping context elements to domain model elements and resolving relations between elements. Structure bridges must also update the identity of elements in the model if those elements move within the domain structure as a result of refactoring. For example, the Java structure bridge integrates with Eclipse's Java model (derived from a Java AST), and is able to map between the handle identifiers of Java elements in the task context and the objects corresponding to those elements in the IDE. It resolves the relations between Java elements including references, inheritance, and read/write access of fields. When elements are moved or refactored and their handle identifier changes, it notifies the context model so that the identity of those elements can be preserved and the previous interaction with that element maintained.

A *UI bridge* is responsible for monitoring interaction with the parts of the IDE that it understands, such as Eclipse's Java tools in the case of the Java UI bridge. It maps the programmer's interaction with the UI to the interaction history schema of selections, edits, and commands. These can include keystrokes in the Java editor, refactoring commands, and element selections. Each UI bridge also specifies which views and editors participate in the interest projection.

Figure 2 shows the dependency structure of a bridge implemented as an Eclipse plug-in. The `context` plug-in provides the structure and UI bridge extension points that the `java` plug-in uses to map and display the concrete Java elements that the programmer is working on. Bridges for other languages may exist instead of alongside the bridges for Java. Bridges can also be composed. For example, the `java` structure bridge extends the resource structure bridge (not shown in Figure 2), which is responsible for understanding file and directory structure.

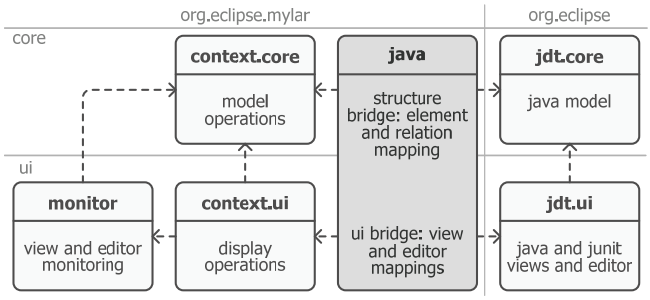

Figure 2: Mylar plug-in architecture and Java bridge

4.2.2 Mapping to Interaction History

Bridges do not issue interaction events directly. For example, when a propagated event needs to be issued for the parent of a selected element, the bridge corresponding to the element is asked for the identifier of \the parent, and the interaction history facility issues the event. The same is true for predicted interest, where the relations may be all Java references to a particular method. In this case resolving the relation involves performing a Java search, returning the results, which are then appended to the interaction history by the interaction event facility.

To support voluntary use of the tool for daily work on real systems, we needed to ensure that the interaction event architecture scaled to large systems without excessive memory or performance overhead. Task context grows with the amount of interaction, not with the system size, ensuring scalability for large systems. Since the number of propagation events can vary with system size, the number of propagation steps is bounded by an exponential drop off that limits

the number of events issued from each interaction. The search scopes used for prediction are proportional to the size of the context. .

4.2.3 Interaction History Storage

A storage mechanism is required to enable the recall of past contexts. Since the interaction history particular to each task encapsulates all of the information needed to derive the task context, the task context model is not persisted. Instead, when a task is re-activated, the corresponding interaction history stream, stored as an XML file, is re-processed. We also store a single XML file of the same form for the task activity context. In memory we maintain the meta-context and a single composite context that allows any number of task contexts to be loaded concurrently. As the programmer works interaction events are appended to the corresponding XML file. If more than one context is active the events are distributed evenly among the files (Section 6.1.1). The approach of storing only the interaction insulates the storage mechanism from the implementation, algorithm, and processing method.

Early benchmarks indicated interaction history file sizes of 1-10MB for a full workday of interaction. However, given our field study data we estimated that programmers working full time generate roughly 1MB of interaction history information per month. The difference comes from the large redundancy in interaction histories, that results from repeated interaction with the same elements. To address this, Mylar's persistence support can *collapse* the interaction history for any context. Collapse can be lossless if it uses run-length encoding, or lossy if it produces aggregate events for all interactions of one kind with a single element. On average the latter reduces file sizes by 10x. Our remaining redundancy comes from storing interaction histories as XML text files, and text compression yields another 10x file size reduction. Since UI actions such as opening editors are the bottleneck on task activation, we only compress contexts when interaction histories are transferred over the network.

5. VALIDATION: FIELD STUDY

Previously, we reported on a preliminary user study we conducted in which six industry programmers used an earlier version of our tool. Mylar v0.1 used a primitive DOI weighting across all of a programmer's work [17]. We learned that programmers need separate contexts for the different tasks on which they work and that a simple weighting of the frequency of element selection is not sufficient. Although this earlier study suggested our basic approach had potential, we learned it was not ready for daily use in a production environment, and lacked specific evidence that it improved programmer productivity.

To answer the question of whether an explicit task context improves programmer productivity, we conducted a longitudinal field study. We chose a field study because the time-constrained tasks performed on medium-sized systems possible in a laboratory setting are not representative of the real long-term tasks performed on large systems in industry.

5.1 Participants

The target subjects for our study were industry Java programmers who used the Eclipse IDE. To solicit participation we presented a prototype of the Mylar tool at an industry conference (EclipseCon, March 2005) and advertised the study on a web page. Early access to Mylar was only possible by signing up for our study through a web form. 99 individuals signed up for the study over the 8 months between the announcement and the conclusion of the study. The majority of these individuals were industry programmers, about half of them worked in organizations with more than 50 people and most identified their industry sector as software manufacturing. A detailed breakdown of the demographics of the individuals is available in our report on how Java programmers use features of the Eclipse IDE [19].

5.2 Method and Study Framework

We designed the field study to measure the effects of our tool within-subjects. A participant joining our study was asked to install a subset of the tool, called the Mylar Monitor, whose role is to transparently capture and store a programmer's interaction history without adding anything to the Eclipse user interface. The monitor was extended with a module that would periodically prompt the participant to upload their interaction history as to a server at UBC, along with exception logs and feedback. To ensure anonymity each participant was assigned a unique identifier. To ensure privacy of the system information any part of the interaction history referring to the elements of what the participant worked on, such as Java type names, was obfuscated using a one-way hash function. We refer to this period of a participant's involvement in the study as their baseline period.

After the participant reached a certain threshold of interaction, which we chose to be 1000 edit events[7] and no less than two weeks of activity, the participant was prompted as to whether they wanted to install the Mylar task context and task focused UI features. Installing Mylar moved a participant into the treatment phase of the study. As before, the monitor periodically prompted the participant to upload their interaction history to a server at UBC. A participant was also notified when there were updates available for Mylar, including both feature additions and bug fixes. We ran the study for four months, July 6th to October 28, 2005 using Mylar v0.3. The task context model, scaling factors, and UI thresholds were frozen for the duration of the study.

5.3 Subject Acceptance

To study whether and how Mylar affects programmer productivity, we needed to be able to compare activity during a participant's baseline period with their treatment period. For instance, if a participant was mostly coding during the baseline period and mostly testing during the treatment period, the two interaction histories would not be comparable. Since we were interested in comparing activity as a participant worked on multiple tasks, we also needed to ensure that both periods were long enough to encompass typical tasks.

Based on these goals, we defined criteria for a participant to be included in our analysis. The first was to ensure an appropriate amount of programming by setting the thresholds on edit events, as was done in determining when to move a participant from the baseline to the treatment period. The second was to ensure that the effects of learning to use Mylar did not overly bias the usage data. To meet these criteria, our threshold of acceptance of a participant for analysis was 3000 edit events, a tripling of the baseline to treatment threshold. We refer to a participant who was accepted for analysis as a subject. We standardized on the number of events rather than the time spent programming in order to account for variations in the rate at which different programmers work.

[7] 1000 edit events corresponded to approximately 1-3 weeks of full-time programming based on trials of individuals in our lab.

Of the 99 initial participants, 16 met the criteria to be considered subjects. This 1 in 6 ratio is indicative of the challenge we had in recruiting subjects: industry developers have little time to try out new tools unless they perceive an immediate and concrete benefit. The minimum 2 week delay in getting the Mylar UI was one contribution to the drop-off, as was the need to use Mylar continuously in daily work that resulted from the 3000 edit event acceptance criteria.

Feedback from the cross section of participants indicated that those who did not meet the criteria did not program as much during this period, did not use Bugzilla which is the only issue tracker Mylar 0.3 integrated with, or stopped using the tool after they encountered a bug or incompatibility with another Eclipse plug-in they were using.

5.4 Results

To analyze subjects' interaction histories we created a reporting framework that allowed us to 'play back' interaction to reproduce usage patterns and gather statistics. We used this framework to analyze the effect of Mylar on what programmers did and how they did it. The tuning of the scaling factors and thresholds for the study is discussed in Section 6.2.1.

5.4.1 Quantitative Analysis: Edit Ratio

Our focus for the study was to measure the effect of Mylar on programmer productivity. We approximate productivity by comparing the amount of code editing that programmers do with the amount of browsing, navigating, and searching. To capture this behavior, we define the *edit ratio* [17], which is the relative amount of edit vs. selection events in any interaction history (i.e., *edits*/*selections*). Edit ratio treats interaction with any kind of artifacts consistently, whether the artifact type is source code, binary libraries, or other kinds of files.

Table 4: Field study data and percentage improvement (bold)

	edit ratio			filtered selections			activity	
id	**base.**	**treat.**	**delta**	**explorer**	**outline**	**probs.**	**hours**	**tasks**
3	2.9	7.8	**172.0**	25%	7%	0%	91.3	61
8	10.1	26.4	**161.4**	16%	0%	0%	71.3	30
6	14.1	36.0	**155.8**	0%	0%	41%	64.7	23
7	2.6	5.4	**111.3**	18%	5%	3%	44.4	54
12	2.7	5.4	**102.3**	3%	0%	0%	24.4	5
15	1.7	3.3	**91.7**	0%	0%	0%	25.3	3
16	8.8	13.0	**47.7**	30%	14%	0%	35.1	7
10	5.8	8.2	**42.1**	32%	22%	40%	11.3	6
2	11.3	14.8	**30.8**	8%	1%	0%	27.5	11
9	6.7	8.7	**30.7**	27%	0%	0%	43.4	12
13	6.8	7.4	**9.7**	14%	3%	0%	48.5	4
5	4.1	4.3	**5.4**	2%	3%	0%	6.5	12
11	2.2	2.2	**3.5**	6%	0%	6%	12.4	7
1	7.7	6.9	**-10.2**	25%	5%	0%	62.5	52
14	15.9	13.5	**-14.7**	0%	0%	0%	66.2	9
4	11.0	8.1	**-26.6**	0%	0%	0%	17.1	1

Table 4 shows the edit ratios for each of the subject's baseline and treatment periods and highlights percentage change in the ratio. To determine whether there was statistical significance in the changes of edit ratios we normalized the edit ratios across individuals by taking the log of each, and performed a paired t-test. The result is statistically significant with $p = 0.003$, indicating that the use of our Mylar tool improves edit ratio. Given that our choice of acceptance criteria for a participant to be considered a subject in the study was somewhat arbitrary, we also wanted to verify if there was stability in this result for different acceptance criteria. We thus analyzed the edit ratios of programmers with both lower and higher thresholds of baseline and treatment edit event cut-offs. Statistical significance of the t-test ($p < 0.05$) holds until we include numerous individuals whose usage data indicates that they did not use Mylar beyond an initial experimentation, and until the threshold is turned up to the point where only six subjects remain.

5.4.2 Qualitative Analysis

Our main hypothesis is that Mylar improves programmer productivity by modeling the appropriate information to complete a task. The edit ratio analysis provided in the previous section provides evidence that for at least one measure, Mylar improves programmer productivity. In this section, we further analyze the content of the task contexts created by the programmers to determine whether or not the contexts were capturing the appropriate information. We consider the following questions: How accurately did the model capture the context of programmers' tasks? Did the programmers create and use multiple tasks that they returned to? How much and in which views was filtering used?

Accuracy

Across the 16 subjects, we observed three notable trends in the selection of elements: 84.17% of the selections events were of elements in the model with a positive DOI (i.e., the elements were visible in a filtered view); 5.32% of the selections were of elements that had only a propagated or predicted interest (i.e., not previously selected or edited, but visible in either a filtered view, Active Search, or Active Hierarchy); and 2.06% of the selections were of elements with a negative DOI (i.e., the elements that decayed out of visibility in a filtered view).

The first observation is indicative of the trend that programmers work on only a subset of the system artifacts, and provides evidence to confirm that a task context does capture the majority of the elements often used when working on the task. The number of propagated and predicted element selections is slightly lower than expected, in part due to our decision to not allow subjects to install the Active Search view until they had used Mylar for half of the treatment period's threshold (1500 interaction events). We delayed the introduction of this view to avoid an overly steep initial learning curve. Once it was introduced, Active Search was used repeatedly by only five users. Qualitative feedback indicated several reasons for a lack of use including a confusing UI, performance bugs, lack of screen real-estate, and the search reporting too many matches. Although the ability to automatically show related elements was promising, these problems need to be addressed before such a facility is integrated enough for daily use.

The number of selections of elements with a negative DOI indicates that the decay scaling factor may have been tuned too high. In contrast, data about the use of the "Make Less Interesting" action indicates that at other times too many elements were being shown, since two subjects frequently used this action (225 times for user 3, 210 for user 7, none for all others). This tension between data indicating that in some cases too much was shown, while in other cases too little was shown, highlights the difficulty of providing a fixed set of scaling factors for all tasks and all users (Section 6.2.1).

Task Activity

We designed our study around measuring the effects of task contexts, and unfortunately did not include sufficiently rich monitoring of the task activity meta-context to determine when the subjects recalled a specific previously worked-on task. However, we do know how often subjects switched tasks (Table 4). Although Mylar is designed around facilitating work with multiple tasks, it can be used with one active, often long-running task (i.e., subject 4, whose usage data indicates he or she worked on with the same task active across eight Eclipse

sessions). We are encouraged by the fact that most subjects switched tasks multiple times during a work day (on average 2.3 tasks switches per active hour). Those with the largest improvement in edit ratio used tasks most heavily. Time active is an indication of how long the subject worked with a task active, approximated by issuing a time out event when no interaction events had been observed for 3 minutes.

View Filtering Usage

Whenever a task was active in the treatment period, a task context was being formed and the UI of the IDE would show which elements were interesting through decoration (Section 4.1.2). To inform and guide the effectiveness of UI mechanisms by which we project the interest model onto the IDE, we also analyzed usage trends related to the view filtering and predicted interest facilities. The percentages of selections made with the view in filtered mode are visible in Table 4 (for Package Explorer, Outline, and Problems views). Unfiltered selections result from either no task being active, or the task being active but the view in unfiltered mode. The regular use of the Package Explorer, the most used Eclipse view [19], by half of the subjects is encouraging. When a view is in unfiltered mode, many more selections are required to find the same information than when filtered due to the need to expand and collapse tree nodes. This causes the percentage of selections in filtered mode appear lower than a programmer might actually perceive.

5.5 Threats

One threat to the accuracy of the study results is that the subjects are not representative of typical industry programmers. The incentive to participate in the study was gaining access to a preview release of Mylar, and as such this selection process was likely biased to early adopters of new programming technologies. Our study results must be viewed in terms of this potential weakness. Another threat is that we had no control over the tasks performed by subjects between baseline and treatment periods so their activity may have varied widely. This threat is addressed in part by the large amount of both baseline and treatment interaction we had for each subject, and consistency that we observed in interaction behavior between baseline and treatment periods (e.g., command and selection usage patterns). If programmers had worked on a single task across the baseline and treatment periods, changes in the edit ratio across the lifecycle of a single task could have been a problem. However, we have evidence of frequent task switching. Finally, bugs in interaction history creation, parsing, and analysis could skew results. Our bootstrapping, testing, ongoing use of the Mylar Monitor framework by ourselves and others is continuing to harden it against such errors.

An objective and generic measure of industry programmers' productivity is difficult as it depends on how a developer works (i.e., their process), what they work on (i.e., their domain) and how quality is measured in that domain. While a definitive measure of productivity is elusive, edit ratio provides us with a measure of effort spent writing code vs. effort spent looking for the information needed to write code. Since programmers chose to use the tool voluntarily, their choice to continue using it is also a positive indicator that the edit ratio metric approximates programmer productivity.

6. DISCUSSION

Mylar is now used daily by thousands of programmers[8]. The usage data and large volume of ongoing user feedback[9] since the study have pointed out the following shortcomings. The usage has also uncovered surprises and misconceptions we had about the features that programmers need to work with task context.

6.1 Shortcomings

6.1.1 Related Tasks

Our model currently treats a task as an independent atomic unit. In practice, tasks are often related. Consider a programmer working on fixing a bug. The programmer creates and activates a task for the bug. As work progresses on the bug, the programmer identifies and begins work on a related bug before the first bug can be resolved. With our current model, the programmer has two choices: deactivate the first task and be forced to recreate the context when starting on the second bug, or do both tasks under the context of the first. Both choices are problematic, and while the latter is easier it is also more costly as the programmer cannot return to the context for just the second bug. Addressing this problem requires extending the model to support schemas for tasks (e.g., subtasks, sequences), and allowing the programmer to work on all or on the component tasks grouped by a parent context.

6.1.2 Task Context Lifecycle

Our model is oblivious to the lifecycle of a task. We use the same scaling factors and apply the same algorithms for operations whether a task is near its start and has a sparse context, or near its completion with a rich context. Making the model sensitive to a task's lifecycle could further improve accuracy. For example, at the beginning of a task it may be beneficial to have a slower rate of decay, and suggestions for related structure could come from a broader degree of separation when the task context is small. Near the end of a task, the core set of information in the context has stabilized and the context contains more information. The size of the task context could be used to adapt the DOI function, scaling factors and degrees of separation, helping tailor the contents of the model to the task's lifecycle.

6.1.3 Forgetting Decay

All of the user study release features used the same projection of the task context without modifying the DOI algorithm listed in Section 2.2. However, this turned out to be insufficient for slicing operations pertaining to source revisions. To support a programmer committing only the changes for a particular task to the source code repository, the post-study Mylar 0.4 release provided Active Change Sets, which include all of the modified files in the task context (Figure 1-6). This allows the programmer to perform file synchronizations, updates and commits per-task. In order to ensure that modified files do not to disappear from the context, this slice only tests the events for each element without including decay, and as a result needs to compensate for the decay factor. A better parameterization of decay will provide additional flexibility needed by such slices.

6.2 Surprises

6.2.1 Scaling Factors

A concern we had prior to starting the field study was that poorly tuned scaling factors could prevent the context model from capturing the information programmers needed, and that scaling factors might need to be personalized for different tasks types, programmers, and display resolutions. We decided not to expose a mechanism for a

[8] 3150 average monthly installs recorded in first 6 months of 2006

[9] 536 hundred bug and enhancement reports filed for 0.3.0-0.4.10

programmer to change the scaling factors because we believed that the problem of information overload was so severe for large system development that an approximate tuning would suffice. We chose an order of magnitude value for each setting (scaling factors: 1 for selections, 0.1, 0.01 for decay; interest thresholds: 0 for interesting, 10 for landmark interest) and used it in our daily programming with Mylar. This resulted in slight variations within those orders of magnitude being set for release versions of Mylar (v0.3 and later).

Although we expected to change the scaling factors and thresholds based on feedback from the study participants, the values continue to work and remain unchanged (up to the current v0.6 release). As a result, we believe that a substantial improvement may require a more sophisticated tuning approach that adapts to properties such as the task's lifecycle, the type of task and programming domains, and the user's profile. Further study is necessary to determine how varying and adapting scaling factors affects the accuracy and precision of the context model.

6.2.2 Multiple Active Tasks

Our preliminary study data indicated that programmers needed support for working on multiple tasks concurrently. We interpreted this input as programmers needing to have multiple tasks active, and implemented support for this in Mylar (v0.2) by distributing interaction events among all active tasks. Feedback from the Mylar user community has indicated that our interpretation was wrong. Although our existing user base needs support for easily switching between tasks, they do not need support for working with tasks in parallel. The latter capability was removed in Mylar (v0.4), and instead we have focused on making task switching and recall easier.

7. RELATED WORK

The idea of using a DOI function to control which parts of a large set of structured data should be displayed to the user originated with Focus+Context and fisheye views [8], and was applied to tree views by Card [3]. Our motivation is similar. However, our DOI function differs as it is not a measure of proximity to a point of focus, but a measure of the frequency and recency of activity on specific elements and relations within the interaction history. In this section, we focus our comparison of related efforts to approaches for managing concerns that span module boundaries, approaches for monitoring programmer productivity, and task-centric information management.

7.1 Concern Management

Programmers have long used various forms of query tools, from grep to program databases [26], to locate code relevant to a task. More recently, several efforts have focused on the capture and persistence of descriptions of *concerns*, non-localized pieces of source required to perform a task, based on modularity properties, annotations in the code, or an external specification. For example, a Concern Graph [23] represents the key structure of code contributing to a concern, whereas JQuery [15] and CAT [10] refer to concerns in terms of queries across the code. In each of these cases, the burden to define the relevant structure is on the programmer. In contrast, Mylar captures the program elements relevant to a task (or concern) implicitly, reducing the cost and effort of using the approach.

7.2 Monitoring Programmer Activity

Many IDE tools can show the programmer structural context for the currently selected element, starting perhaps with Interlisp's Masterscope [26]. Providing a richer context than the currently selected element involves monitoring the user's interaction. In the document editing domain, the Edit and Read Wear tool was one of the first to do this by highlighting editing and selection patterns across a set of documents [13]. Just-in-time information retrieval agents provide a mechanism for searching information related to a user's context [22]. Hilbert has described a framework for collecting interaction history data by monitoring application events [12]. Mylar v0.1 [17] and subsequently Wear-Based Filtering expanded this to the programming domain by using interaction frequency to highlight the elements of interest in the IDE UI [5]. Team Tracks builds on this by using interaction with program elements to drive a recommender that can suggest to other members of the team which program elements may be of interest [4]. Context can also be inferred from analyzing navigation paths through a concern graph [24]. Recent Focus+Context UML visualization, in which "A class is displayed at a particular level of detail using a degree of interest (DOI) function based on the frequency of access to a particular class and its distance from the current object in focus", capture a notion of interaction [14].

In contrast, we make tasks a first class part of both interaction and context, support direct and indirect interaction for both artifacts and tasks themselves, and enable novel operations such as propagation, prediction, slicing, and projection. Our task context model captures an interaction-based DOI for both elements and relations, and can be mapped to any domain structure.

7.3 Task-centric information management

Some of the foundations on managing the context of documents in a task-centric way come from the Placeless and Presto projects from Xerox PARC [6]. However, these systems required people to manually categorize their files rather than building up context implicitly. The most directly related task management system is UMEA, which monitors user activities in a desktop environment to create "project spaces" [16]. The Task Tracer system is similar, categorizing each event with the Microsoft Windows and Office system according to a task, and using this to build up a profile for the task [7]. This profile captures the number of interactions with a resource, at the granularity of files and URLs. In contrast, Mylar supports structured data and provides a task context model and DOI ranking. More recent work on Task Tracer has demonstrated the ability to automatically infer task switches, and to use Bayesian learning to predict the user's context [25]. Although the task context model may provide a useful input to such approaches, this and other machine learning mechanisms to inferring context [11] are different from our approach of making context a direct and predictable translation of user behavior and structural relations.

8. CONCLUSION

Development environments have provided programmers with the compiler's view of the system: displaying the current file being edited and compiled, providing browsing views of the entire containment hierarchy, and allowing navigation of the type hierarchy. While these approaches are sufficient for small systems with good modularity, they are not sufficient for the moderate and large systems on which many programmers work. Although the complexity of systems continues to increase, the ability of programmers to handle complexity does not. To address this mismatch we provide a model of task context that can be layered over the existing structure models in the IDE and alongside with integrated task management facilities. We have tested the model on industry programmers, finding both quantitative and qualitative evidence that the use of task context can make programmers more productive.

9. ACKNOWLEDGEMENTS

This work was supported by IBM CAS and NSERC. We thank Christopher Dutchyn, Leah Findlater and Thomas Fritz for their reviews, the study subjects for their participation, and the Mylar users for their ongoing input.

10. REFERENCES

[1] Backus, J.W. Automatic programming: properties and performance of FORTRAN systems I and II. *Proceedings of the Symposium on the Mechanisation of Thought Processes*, The National Physical Laboratory, 1958.

[2] Bellotti, V., Dalal, B., Good, N., Bobrow, D. G., Ducheneaut, N. What a to-do: studies of task management towards the design of a personal task list manager. *Proceedings of the Conference on Human Factors in Computing Systems*. p. 735-742. 2004.

[3] Card, S. K. and D. Nation. Degree-of-Interest Trees: A Component of an Attention-Reactive User Interface. *Advanced Visual Interfaces Conference*, 2002.

[4] DeLine, R., Czerwinski, C. and Robertson, G. Easing program comprehension by sharing navigation data. *Proceedings of IEEE Symposium on Visual Languages & Human-Centered Computing*. p. 241-248, 2005.

[5] DeLine, R. Khella, A. Czerwinski, M. Robertson, G. Visualization frameworks and empirical evaluation: Towards understanding programs through wear-based filtering. *Proceedings of the 2004 ACM Symposium on Software Visualization*. p. 183-192, 2005.

[6] Dourish, P., Edwards, W. K., LaMarca, A., Salisbury, M. Using properties for uniform interaction in the Presto document system. Proceedings of the 12th *Annual ACM Symposium on User Interface Software and Technology*. p. 55-64. 1999.

[7] Dragunov, A.N., Dietterich, T.G., Johnsrude, K., McLaughlin, M., Li, L., Herlocker, J.L.. TaskTracer: A Desktop Environment to Support Multi-tasking Knowledge Workers. *International Conference on Intelligent User Interfaces*. p. 75-82, 2005.

[8] Furnas, G.W. Generalized fisheye views. *Proceedings of the Conference on Human Factors in Computing Systems*. p.16-23, 1986.

[9] Gonzales, V.M., Mark, G. Constant, constant, multi-tasking craziness: managing multiple working spheres. *Proceedings of the Conference on Human Factors in Computing Systems*. p. 113-120, 2004.

[10] Harrison, W., Ossher, H., Tarr, P., Kruskal, V. and Tip, F. CAT: A Toolkit for Assembling Concerns. *Research Report RC22686*, IBM, Yorktown Heights, NY, Dec. 2002.

[11] Hijikata, Y. User modeling II: Implicit user profiling for on demand relevance feedback. *Proceedings of the 9th international conference on Intelligent User Interfaces*. p. 198-205, 2004.

[12] Hilbert, D. M., Redmiles, D.F. Separating the wheat from the chaff in Internet-mediated user feedback expectation-driven event monitoring. *ACM SIGGROUP Bulletin*. p. 35-40,1999.

[13] Hill, W. C., Hollan, J. D., Wroblewski, D., and McCandless, T. Edit wear and read wear. *Proceedings of the Conference on Human Factors and Computing Systems,* p. 2-9. 1992.

[14] Jacobs, T. Musial, B. Debugging and finding faults: Interactive visual debugging with UML. *Proceedings of the 2003 ACM Symposium on Software Visualization*. p. 115-122, 2003.

[15] Janzen, D. and de Volder, K. Programming With Crosscutting Effective Views, *Proceedings of the European Conference on Object-Oriented Programming*. p. 195-218, 2004.

[16] Kaptelinin, V. Integrating tools and tasks: UMEA: translating interaction histories into project contexts. *Proceedings of the Conference on Human Factors in Computing System*. p. 353-360, 2003.

[17] Kersten, M., Murphy, G. C., Mylar: a degree-of-interest model for IDEs. *Proceedings of the 4th international conference on Aspect-Oriented Software Development*. p. 159-168, 2005.

[18] *Merriam-Webster's collegiate dictionary (11th ed.)*, Springfield, MA: Merriam-Webster. 2003.

[19] Murphy, G. C., Kersten, M., Findlater, L., How are Java Software Developers using the Eclipse IDE? *IEEE Software*. Vol. 23, No. 5. 2006.

[20] Murphy, G., Kersten, M., Robillard, M. and Cubranic, D. The Emergent Structure of Development Tasks. *Proceedings of the European Conference on Object-Oriented Programming*. p. 33-48, 2005.

[21] Parnas D. L., On the Criteria to be Used in Decomposing Systems into Modules, *Communications of the ACM*, Vol. 15, No. 12, 1972.

[22] Rhodes, B. and Maes, P. Just-in-time information retrieval agents. *IBM Systems Journal special issue on the MIT Media Laboratory*, 39(3-4):685-704, 2000.

[23] Robillard, M. P., and Murphy, G.C.. Concern Graphs: Finding and Describing Concerns Using Structural Program Dependencies. *IEEE 24th International Conference on Software Engineering*. p. 406-416, 2002.

[24] Robillard, M.P., Automatic Generation of Suggestions for Program Investigation. *Proceedings of the Joint European Software Engineering Conference and ACM Symposium on the Foundations of Software Engineering*, p. 11-20, 2005.

[25] Shen, J., Li, L., Dietterich, T.G., Herlocker, J.L., A hybrid learning system for recognizing user tasks from desk activities and email messages. *International Conference on Intelligent User Interfaces*. p. 86-92, 2006.

[26] Teitelman, W. and Masinter, L. The Interlisp programming environment. *IEEE Computer*, vol. 14, 25-34, 1981.