Using Task Context to Improve Programmer Productivity

Mik Kersten and Gail C. Murphy, FSE 2006

Introduction

Modularity: Traditional Definition

- Modularity enables programmers to develop and evolve complex software systems.

- Modularity in programming languages:
  - Enables separate compilation, making it tractable to modify and test a small part of a system.
  - Enables parallel development, making it tractable to develop large systems in less time

Modularity in IDEs

- IDEs use modularity to present views of a system in support of a programmer's tasks, such as bug fixes and feature additions

- Example:
  - A common way to access code in Eclipse is through the Package Explorer, which shows the modular structure of projects, packages, files, and classes.

- Current IDEs appear to encode two assumptions:
  - A programmer will often be able to find a desired piece of the system by traversing the modular structure
  - 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.

Problems with “Package Explorer” View

- We have observed two problems with these assumptions.

- First, many modifications are not limited to one module.
  - We found that over 90% of the changes committed to the Eclipse and Mozilla over a period of one year involved changes to more than one file
  - We then selected 20 changes from Eclipse and found that 25% of them involved significantly non-local changes.

- Second, even when the actual changes related to a modification are within a single 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

Problems with “Package Explorer” View

- Result: programmers 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.
  - 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.
- This constant need to re-create the context of the task reduces the programmer’s productivity.

Mylyn: Proposed Solution

- 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.

- In this paper, we expand and refine the concept of a DOI function by creating and validating a new model that includes an explicit representation of task and contexts

Example

- To give a flavor for our approach, we provide an overview of what it is like to use our Mylar tool.

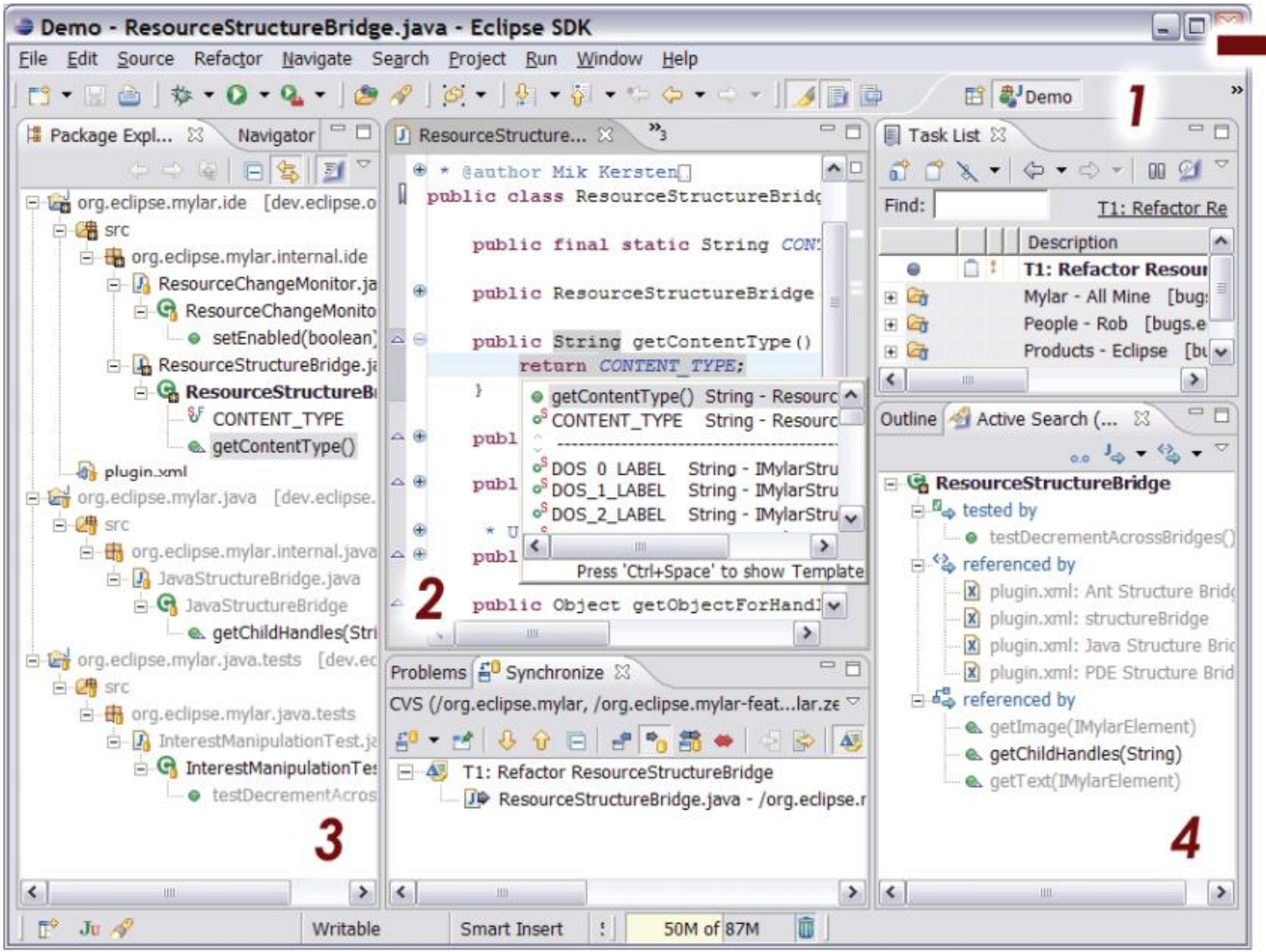
- The scenario involves a Java programmer using Mylar on a code base with over one thousand classes

- One task on which the programmer is working is an improvement to the code base of the system:
  - Task-1: Refactor ResourceStructureBridge

Task-1: Refactor ResourceStructureBridge

- Activating the task causes Mylar to track the parts of the system artifacts that the programmer accesses while working on this task
- For example, 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)
- 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).

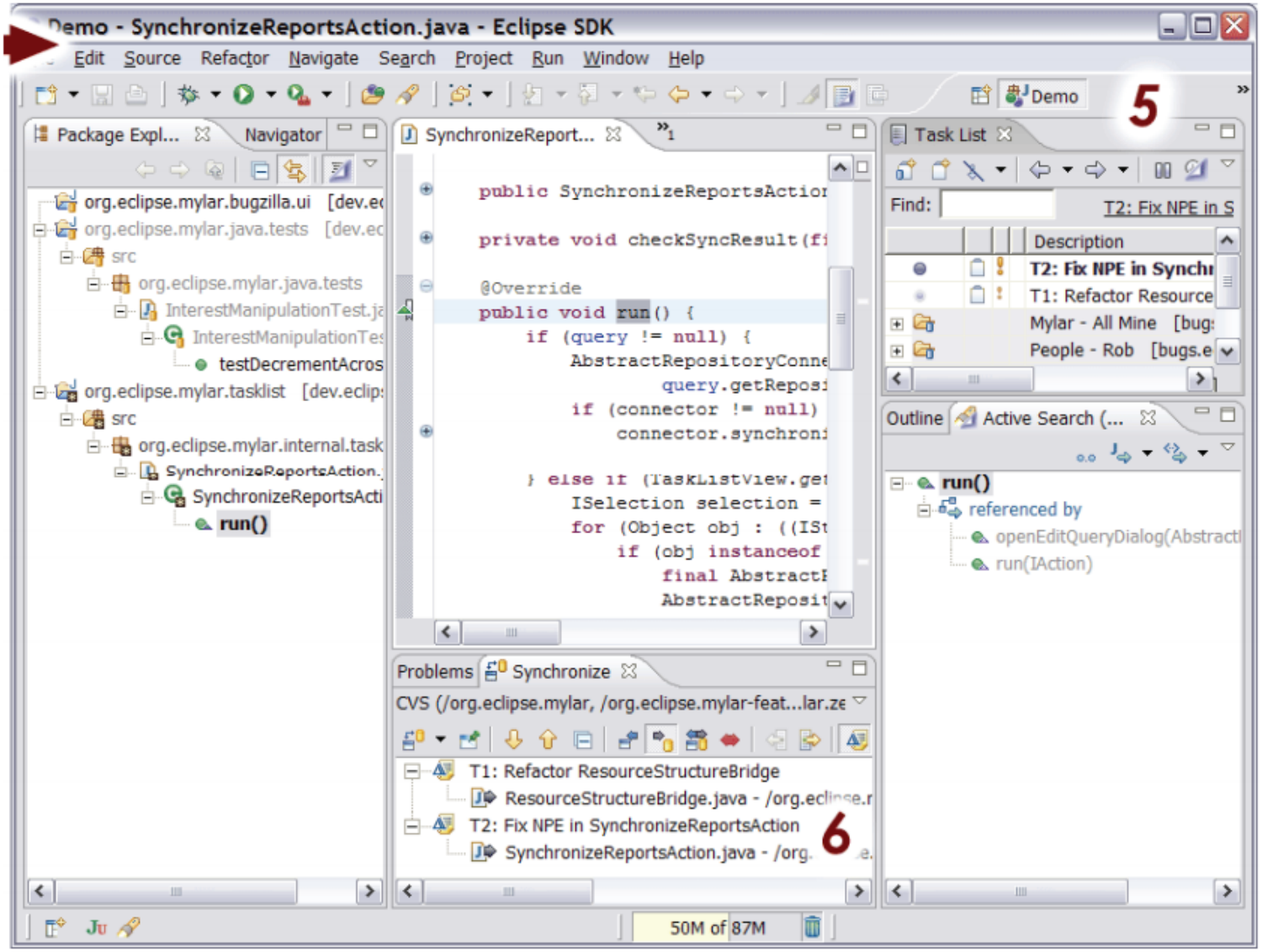
Task-1: Refactor ResourceStructureBridge



Context Changing

- 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.
- 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

Task-2: Fix NPE in SynchronizeReportsAction



Proposed Solution

Task Context Model

- We define a task as “a usually assigned piece of work often to be finished within a certain time”.

- 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

Task Context Model

- A task context is the information 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.
- 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

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

- 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).

Interaction Events

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

Indirect Interactions

- Other interaction events are indirect:
  - program elements and relationship are affected without being directly selected or edited by the programmer.

- 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.

Indirect Interactions

- The model also support prediction events, which describe possible future interactions that a tool anticipates the programmer might perform.

- Example:
  - An event describing that a test may be of interest to the current task because it references a class in the task context

Computing Degree-Of-Interest (DOI)

- 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.
- The 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 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.

Computing Degree-Of-Interest (DOI)

- Algorithm that computes a DOI value for an element with an interaction history events sequence that contains one or more events with the element as the target.

```
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

Sample Interaction History

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 `ResourceStructureBridge`

DOI Calculation 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, the three iterations through the loop will result in:
  - 1+0+2 = 3 for interest,
  - (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.
  - (1+0+2) –(46-1)*0.1 = -1.5

Tool Support

Mylyn Tool

- The Mylyn IDE integration allows programmers to work with task context in almost every commonly used part of the Eclipse IDE.

- Mylyn 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 folding in the editor.

- 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

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

Active Views

- 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,
  - An Active Hierarchy view that shows the inheritance context of the task.

Evaluation

Field Study

- 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.

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
- The majority of these individuals were industry programmers.

Method: Baseline

- A participant joining our study was asked to install a subset of the tool, called the Mylar Monitor:
  - Captures and stores 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

- We refer to this period of a participant's involvement in the study as their baseline period.

Method: Treatment Phase

- After the participant reached a certain threshold of interaction, which we chose to be 1000 edit events 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

- 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

Method: Treatment Phase

- 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.

Method: Treatment Phase

- 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

- 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.

Method: Treatment Phase

- Participants indicated 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
  - Stopped using the tool after they encountered a bug or incompatibility with another Eclipse plug-in they were using.

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, which is the relative amount of edit vs. selection events in any interaction history (i.e., edits/selections)

Quantitative Analysis: Edit Ratio

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

Conclusion: use of our Mylar tool improves edit ratio

Qualitative Analysis

- 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);
  - 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)

Qualitative Analysis

- Unfortunately we did not include sufficiently rich monitoring to determine when the subjects recalled a specific previously worked-on task.
- However, we do know how often subjects switched tasks.
- 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

View Filtering Usage

- 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, by half of the subjects is encouraging.

Conclusion

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.

Conclusion

- 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.