The Emergent Structure of Development Tasks [*]

Gail C. Murphy,[1] Mik Kersten,[1] Martin P. Robillard[2] and Davor Čubranić[3]

[1] Department of Computer Science
University of British Columbia
murphy@cs.ubc.ca, beatmik@acm.org
[2] School of Computer Science
McGill University
martin@cs.mcgill.ca
[3] Department of Computer Science
University of Victoria
cubranic@cs.uvic.ca

Abstract. Integrated development environments have been designed and engineered to display structural information about the source code of large systems. When a development task lines up with the structure of the system, the tools in these environments do a great job of supporting developers in their work. Unfortunately, many development tasks do not have this characteristic. Instead, they involve changes that are scattered across the source code and various other kinds of artifacts, including bug reports and documentation. Today's development environments provide little support for working with scattered pieces of a system, and as a result, are not adequately supporting the ways in which developers work on the system. Fortunately, many development tasks do have a structure. This structure emerges from a developer's actions when changing the system. In this paper, we describe how the structure of many tasks crosscuts system artifacts, and how by capturing that structure, we can make it as easy for developers to work on changes scattered across the system's structure as it is to work on changes that line up with the system's structure.

1 Introduction

The tools that developers use to build a large software system provide an abundance of information about the structure of the system. Integrated development environments (IDEs), for example, include views that describe inheritance hierarchies, that present the results of system-wide searches about callers of methods, and that report misuses of interfaces. These IDEs have made it easier for developers to cope with the complex information structures that comprise a large software system.

However, some of our recent work suggests that the focus on providing extensive structural information may be having two negative effects on development:

– developers may be spending more time looking for relevant information amongst the morass presented than working with it [10], and

---

[*] ©**Springer-Verlag**. Appears in *Proceedings of ECOOP 2005*.

– developers may not always be finding relevant information, resulting in incomplete solutions that lead to faults [4, 19].

We believe that these problems can be addressed by considering how a developer works on the system. More often than not, development tasks require changes that are scattered across system artifacts. For instance, a developer working on a change task might change parts of several classes, may read and edit comments on parts of a bug report, may update parts of a web document, and so on. As a developer navigates to and edits these pieces, a structure of the task emerges.

In this paper, we describe how this structure crosscuts the structures of system artifacts and we explore how the capture and description of task structure can be used to present information and support operations in an IDE in a way that better matches how a developer works. We believe support for task structure can improve the effectiveness of existing tools and can enable support for new operations that can improve a developer's individual and group work.

We begin with a characterization of how tasks crosscut system artifacts, providing data about the prevalence of scattered changes and arguing that the changes have structure that is crosscutting (Sect. 2). We then introduce a working definition of task structure (Sect. 3) and describe what an IDE with task structure might provide to a developer (Sect. 4). We then elaborate on the possibilities, explaining some of our initial efforts in making task structure explicit (Sect. 5), discuss some open questions (Sect. 6), and describe how our ideas relate to earlier efforts (Sect. 7).

2 Tasks Crosscut Artifacts

Building a software system involves many different kinds of tasks. A one-year diary study of 13 developers who were involved in building a large telecommunications system found 13 different kinds of tasks, including estimation, high-level design, code, and customer documentation [15]. In this paper, we focus on *change tasks* that affect the functionality of the system in some way, by fixing bugs, improving performance, or implementing new features. To simplify the discourse in this paper, we use the term *task* to mean change task.

To complete a task, a developer typically has to interact with several kinds of artifacts, including source code, bug descriptions,[4] test cases, and various flavours of documentation. Conceptually, these artifacts form an information space from which an IDE draws information to display to a developer. Since the source code tends to form the majority of the structure, we focus our characterization mainly on it, returning to the more general information space in later sections of the paper.

2.1 Occurrence of Scattered Changes

It has long been a goal of programming language and software engineering research to make it possible to express a system such that most modification tasks require only

---

[4] Bug descriptions, or reports, at least in many open-source projects, are used to track not just faults with the system, but enhancements and other desired changes.

localized changes to a codebase [14]. To achieve this goal, modularity mechanisms have been introduced into the programming languages we use (e.g., classes in object-oriented languages) and various design practices have evolved (e.g., design patterns [6]). Despite these advances, we contend that the completion of many tasks still requires changes that are scattered across a code base.

To illustrate that many changes have this property, Fig. 1 shows the number of files checked-in as part of transactions from two large open-source projects — Eclipse and Mozilla.[5] Following a common heuristic used for open-source projects, a transaction is defined as consisting of file revisions that were checked in by the same author with the same check-in comment close in time [13]. For both of these systems, over 90% of the transactions involve changes to more than one file.

To provide some insight into the relationship between the changes and the structure of the system, we randomly sampled 20 transactions that involved four files from the Eclipse data. Of these transactions, fifteen involved changes in multiple classes located close together (i.e., within the same package). These changes are scattered, but it might be considered that they are contained within some notion of module (i.e., a Java package). However, five transactions included changes across packages, and of these five, two included changes across more than one plug-in (a significant grouping of related functionality in Eclipse). Assuming that a transaction roughly corresponds to a task,[6] a reasonable number of tasks (25% of those sampled) involved changes scattered across non-local parts of the system structure.

2.2 Crosscutting Structure of Changes

Are scattered changes simply the result of a bad system structure or is there some structure to the scattering? To provide some insight into these questions, we consider a typical change in the Eclipse code base. We chose to use an example from Eclipse because it is generally considered to be well-designed and extensible. We follow Eclipse documentation guidelines in the approach we take to implementing the change.

The task of interest involves a change to a hypothetical Eclipse plug-in to support the editing and viewing of an HTML document. This HTML plug-in provides an outline view that displays the structure of an HTML document as a tree, where the headings and paragraphs are nodes in the tree. Imagine that your task is to modify the outline view of the HTML plug-in to add nodes that represent hyperlinks.

To perform this task, you need to update both the HTML document model and the view. Assuming the recommended structure for Eclipse plug-ins, this means changing methods in a `ContentProvider`, a `LabelDecorator` and a `Selection-Listener` class. You also need to add a menu action and update appropriate toolbars which requires modifying another class. In addition, you need to declare the new action and any associated icon in an XML file (i.e., `plugin.xml`). In total, this simple

---

[5] The Eclipse project can be found at eclipse.org and the Mozilla project can be found at mozilla.org. The check-in data for Eclipse comes from 2001/04/28 until 2002/10/01 and the check-in data for Mozilla comes from 1998/03/27 until 2002/05/08. Only data for transactions involving 20 or less files is shown.

[6] This is a reasonable assumption because of the work practices used in developing this open-source system.
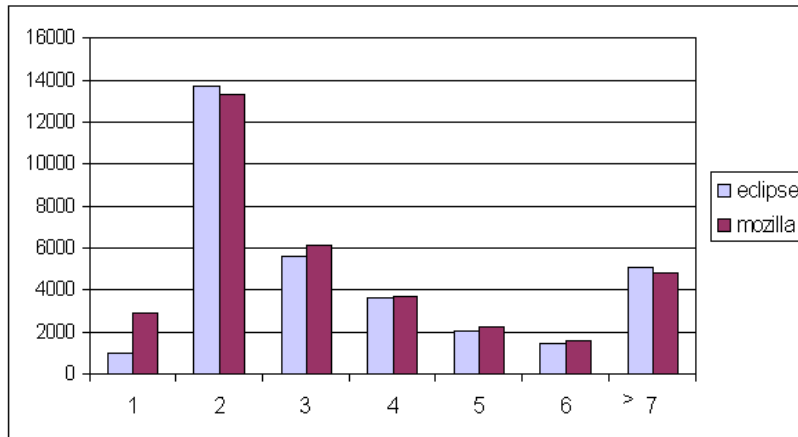
Fig. 1. The number of files (x-axis) involved in check-in transactions (y-axis) for Eclipse and Mozilla.

change task involves modifications scattered across four Java classes, two parts of an XML file, and an icon resource.

Although these changes are scattered, there is *structure* to the change task; the structure happens to *crosscut* multiple parts of multiple artifacts. In simple terms, two structures crosscut each other if neither can fit neatly inside the structure provided by the other [12]. A developer well-versed in Eclipse plug-in development would be able to explain this structure, and much of it is recorded in the documentation about how to extend Eclipse. The structure of the source code has been chosen to make adding a new listener to a view a change that is localized in the structure, whereas adding a brand-new element (as in our task) is a change that crosscuts the structure.

We believe that many of the tasks involving scattered changes are not ad hoc, but that they do have a crosscutting structure. In our work, we have found that this cross-cutting structure emerges from how a developer works with the code base [19, 10]. In the remainder of the paper, we show how this structure, once made explicit, can be used to make IDEs work better for developers.

3 Task Structure

To ground our discussion, we introduce a simple working definition of task structure.

A **task structure** consists of the parts of a software system and relationships between those parts that were changed to complete the task.

Conceptually, consider forming a graph based on information found in all of the artifacts comprising the system. In this graph, the nodes are structural parts of the artifacts and the edges are relationships between those parts. The structure of a task consists of a collection of subgraphs from this graph. Each node in the graph includes information

about the artifact in which it appears, the name of the part, and the type of the part: each kind of artifact will have its own types of parts. For example, the types of parts found in a Java class include method definitions, field definitions, and inner class definitions. As another example, the types of parts found in a bug report include dates when the report is opened or closed, and text fields with discussions about the report. Each edge in the graph includes information about the artifact in which it appears (if any) and the type of the relationship. Some relationships will be defined explicitly in a project's artifacts, whereas others may be inferred by tools. For example, a call between two methods in a Java program appears explicitly in the source code, whereas a relationship that indicates a file revision helped solve a particular bug may be inferred by a tool [3].

As a concrete example, we return to the task of adding a new element to an existing view in an Eclipse plug-in. Figure 2 shows a portion of the graph of parts and relationships from artifacts comprising the system.[7] Even though only a fraction of the structure of a small number of artifacts is included in the graph, the amount of information is overwhelming. However, only small parts of the graph relate to the task; the highlighted nodes and edges in Fig. 2 form the structure of the task.

Fig. 2. A graph showing parts of artifacts and relationships between the parts comprising a simple system. The highlighted portions represent the structure of one task performed on the system.

Our definition of task structure is based on completed tasks. An advantage of this definition is that the task structure can be determined with certainty if the time points at

---

[7] The graph was generated using prefuse [9].

which the task started and finished are known. However, we also want to make use of task structure as a task is being performed. We use *task context* as a means of approximating task structure and as a means of describing the subgraphs of the information space of interest when performing a task.

> A **task context** consists of parts and relationships of artifacts relevant to a developer as they work on the task.

This definition of task context relies on the concept of *relevance* of parts of a system to a task. Relevance can be defined in a number of ways, all of which include some element of cognitive work on the part of a developer [26]. A simple way to determine relevance is for a developer to manually mark the parts and relationships as relevant as they are exploring code [20]. Automatic determinations of relevance are also possible. For example, we are investigating two approaches in which relevance is based on the interaction of the developer with the information in the environment, such as which program elements are selected. In one approach, the interaction information is used to build a model of the degree to which a developer is interested in different parts of the system [10]. This degree-of-interest model is then used to predict interest in other elements and in related project artifacts. In the second approach, relevance is determined by analyzing the interaction information according to the frequency of visits to a program element, the order of visits, the navigation mechanism used to find an element (e.g., browsing, cross-reference search, etc.), and an analysis of the structural dependencies between elements visited [21].

In the rest of this paper, we assume that task structure and task context information is available and focus on providing some examples of how it might be used to improve a developer's work environment.

4   Improving a Developer's Work with Task Structure

Imagine that you are a developer working with an IDE that includes support for capturing, saving and operating on task contexts and task structures. In this section, we describe what it might be like to use this IDE to work on a development task. As we indicate in the scenario, several features we describe have been built or proposed as part of earlier efforts. Task structure enables these operations to be more focused and to provide more semantic information, without any significant input from the developer.

The system on which you are working allows a user to draw points and lines in a window, and to change their colour.[8] The system also has a mode, which when set through a radio button, supports the undo of actions taken by the developer through the user interface.

Your current task involves adding support to enforce the use of a predetermined colour scheme in a drawing. A radio button is to be provided to turn the colour scheme enforcement on and off. When the enforcement is on, the colours of points and lines in the drawing are to be modified to meet the colour scheme and any subsequent request to change a colour will be mapped to the colour scheme.

---

[8] This example is based on a simple figure editor used to teach the AspectJ language [11].

You start the task by navigating through some of the code attempting to find relevant parts. As you navigate, the IDE is building up your task context based on your selections and edits. After some navigation you determine that you need to add some code in the `ButtonsPanel` class to add in the necessary radio button. At this point, your task context includes information about several methods you have visited and the constructor in `ButtonsPanel`. As you add in the call to add a new radio button, a green bug icon appears in the left gutter of the editor (Fig. 3).[9] This icon appears because a tool in the IDE, running in the background, has determined that there is a completed change task whose task structure is similar to your task context.

You decide to click on the bug icon. A popup window appears that describes some information about the bug (Fig 3). You read the description of the bug and you realize that it is similar to the task on which you are working.[10] Since the related bug has been resolved, it has an associated task structure. You expand this task structure and the tree view of the structure shows you which parts overlap with your task context (the highlighted nodes in Fig. 4). You notice the `HistoryUpdating` aspect listed in the task structure that supports the undo functionality. You have not considered whether you will use an aspect to complete your task. However, you look at the code for the aspect and realize that it implements similar functionality to what is needed for your task. After considering the options, you decide to use an aspect-oriented approach and you create a `ColourControl` aspect based on the `HistoryUpdating` aspect. Guided by the previous task context, you also add an image for the new action to the system.
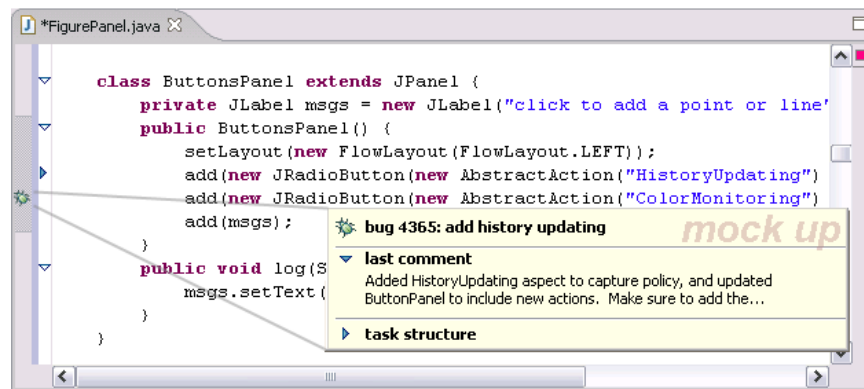

Fig. 3. Based on the task context, a green bug icon appears indicating another bug report may be relevant to the task being performed.

Before you check-in the code for your completed task, you want to ensure your changes will not conflict with concurrent changes being made by other members of

---

[9] The user interfaces described are mock-ups of how the described functionality might be provided.

[10] This type of functionality is similar to our Hipikat tool [3]. We sketch the differences between the Hipikat approach and using task structure for this purpose in Sect. 5.
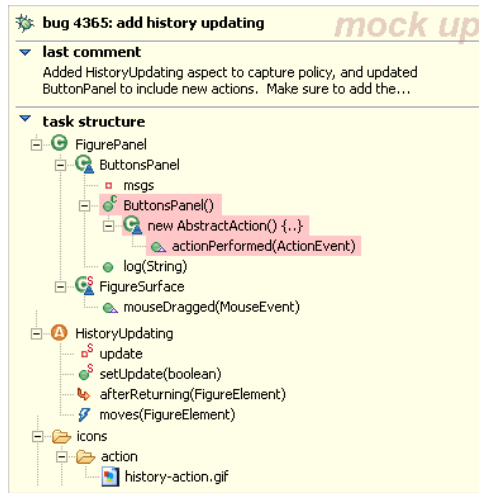
Fig. 4. The task structure of the completed task with highlights indicating overlap with the current task context.

your team. As you check-in your code using the facilities of the IDE, you select an option to compare your task structure with any task contexts that your team members have made available (by selecting an option in the IDE). The IDE tool supporting this comparison looks for overlap between your task structure and your team members task contexts and if it finds overlap, it considers any effect, using static analyses, each task has had on the overlapping parts.

Figure 5 shows the results of the comparison for the task you are about to complete. It shows that part of your local task structure includes a call to a `setColor` method (left side of Fig. 5). It also shows that one of your team member's task context's has modified the `HistoryUpdating` aspect to add advice that narrows `setColor` [17]. Narrowing advice may result in the `setColor` method not being called under some circumstances. Given this information, you can contact your colleague to determine how to resolve the conflicts between your changes.[11]

As noted, several of the features that task structure makes possible in this hypothetical scenario have been proposed previously. In comparison to these existing approaches, task structure provides three benefits over existing approaches:

1. it can be determined with minimal effort from the developer as it emerges from how the developer works on the system,
2. it provides a conceptual framework and model that can be built into an IDE to make task-related tools easier to build, and
3. it provides information that may be used to focus views in the IDE, allowing a greater density of relevant information to be displayed.

---

[11] This type of fine-grained conflict determination has similarities to soft locking in Coven [2].
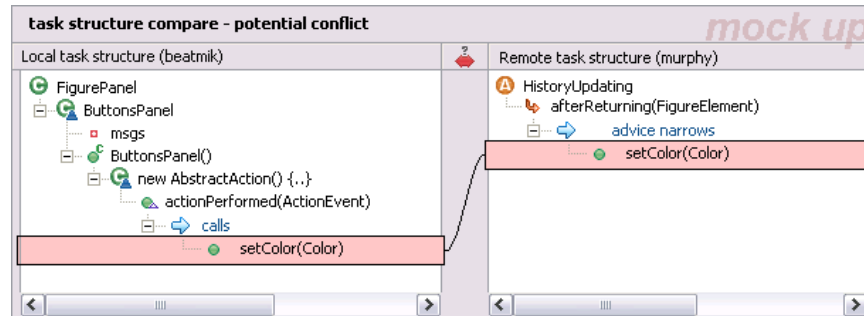
Fig. 5. A comparison of your task structure with a team member's task context identifies a possible conflict. The conflict is determined by statically analyzing the effect of each task context and comparing the results.

5 Making Use of Task Structure

The scenario described in the last section illustrates how explicit support of task structure in an IDE can benefit a developer. In this section, we elaborate on these points and describe more possibilities. Through this section, we use the term task structure to simplify the discourse as it should be clear when the use of task context would be more precise.

5.1 Improving IDE tools

An ideal IDE would present the information a developer needs, when it is needed, and with a minimum of interaction from the developer. Such an IDE would reduce the amount of time a developer spends trying to find relevant information. We outline four ways that an IDE with support for task structure could help move towards this goal.

Reducing Overload in Views and Visualizations. IDEs present system structure mostly in lists and tree views, with some graphical visualizations [24]. When used on large systems, these existing presentation mechanisms tend to overload the developer with information, making it difficult to find the information of interest. For example the Package Explorer, a commonly used tree view in Eclipse which shows the decomposition of Java source into packages, files, classes, and other structural elements, often contains tens of thousands of nodes when used on a moderately-sized system (e.g., see the left-hand side of Fig. 6; notice that the tree structure is not visible). A task's structure can be used to determine what information should be made more conspicuous to a developer. For instance, the task structure can be highlighted [22]. Or, the task structure can be used to filter the view so as to show only task-relevant information as is the case in our Mylar prototype (see the right-hand side of Figure 6; the bolded parts are the elements that are the most important to the task) [10]. Either way, the views can make it clear to the developer the elements important to the task.
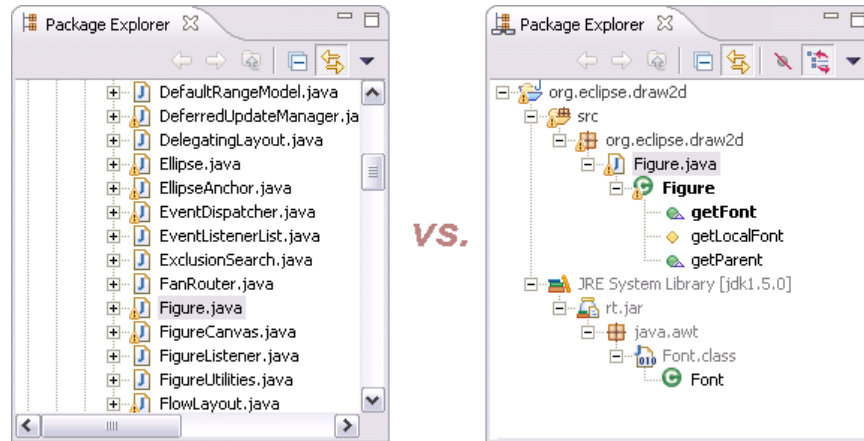
Fig. 6. A view of the containment hierarchy of a system without Mylar active (left-hand side) and with Mylar active (right-hand side). In the Mylar view, the focus provided by task context enables the relevant information to fit on the screen without a scrollbar, and enables the structural relationships to be visible.

Scoping Queries. Task structure can be used to scope the execution of code queries performed by a developer. A default setting, for instance, may be to query code only within one or two relationships in the overall graph of structural information (Sect. 3) from the code on which the query is invoked. Scoping queries with task structure could have two potential benefits: queries may execute more quickly for large systems, and the information returned may be more relevant, reducing the time needed for a developer to wade through search results.

Performing Queries Automatically. In addition to scoping queries, parts in the task structure can be used to seed queries that run automatically. We call these active queries and they could be used to seed active views [10]. For example, the active search view in the Mylar prototype eagerly finds and displays all Java, XML, and bug reports related to parts in the task structure [10]. This kind of view provides a developer with the structural information they need when it is needed. The result could be a reduction in the number of interruptions a developer must typically make to think about and formulate a query, as the most relevant queries are formulated and executed automatically. Active queries also do away with the need to wait on query execution, since queries are executed automatically in the background.

The concept may also be helpful in the implementation of the IDE. The Eclipse IDE, for instance, requires the Abstract Syntax Tree (AST) for a class to be in memory in order to support features such as semantic highlighting.[12] However, operations that span multiple files, such as the rename method refactoring, are time consuming and

---

[12] Semantic highlighting refers to the ability to highlight code according to properties such as whether the code is an abstract method invocation, a reference to a local variable, etc.

require the developer to wait until all related ASTs are loaded into memory. The task structure could be used to define the slices of ASTs that should be kept in memory in order to make common refactorings instantaneous (i.e., as quick for changes across files as they are for changes within the file).

Supporting Task Management. IDEs provide little to no support for managing the tasks a developer is performing. The best support may be an ability to read and manage bug reports within the IDE. Task structure can improve this situation. For example, as part of our Mylar project, we have prototyped support for enabling developers to associate task structures with specific tasks and to switch between them. Mylar can then filter the views of the system according to the selected task. The task structure can also be attached to a bug report, enabling a developer to re-start if they return to the bug at a later time. The task structure, in effect, is a form of externalization of the developer's memory of the task.

5.2   Improving Collaboration

Over the lifetime of a system many developers work on many tasks. We believe that communicating this structure to other developers as they work, and storing it as the system evolves, can provide collaborative tools with an effective representation of group memory.

Forming and Accessing a Group Memory. It is not uncommon when working on a software development project to come across a problem that is reminiscent of a past problem with the system that has since been solved (Sect. 4). In earlier work, we demonstrated the benefits of processing the artifacts comprising a software system to form a group memory that may then be searched for relevant information as a developer is performing a task [3, 4]. One benefit is that developers may be more aware of subtle, but relevant, information. For example, in an experiment we conducted, newcomers to a project took into account additional information presented from the group memory and finished an assigned task more completely than experts who did not have access to the group memory [4]. Our previous work treated task structure implicitly, forming links automatically between parts of related artifacts. Explicitly stored task structures enable more focused comparisons between a past system and a current system and allow new operations across the group memory, such as an analysis tool that could identify all of the third party APIs involved in commonly reported defects.

Sharing Task Structure. Task structure encapsulates a developer's knowledge about the system. As discussed above, developers may want to store this knowledge in order to access it at a later time. In addition, they may want to share it with others. For example, a developer delegating a task could include the task structure in order to help the team member pick up the task where it was left off. Sharing of task structure could also be done in real-time in order to make developers aware of the activities of their team members. For example, in an open-source project where team members are distributed across time zones, knowing the parts of the system that have been worked on

by others can encourage dialog and prevent merge problems. In comparison to existing approaches to providing such awareness [23], task structure can enable a deeper comparison, seeding automated handling by tools or discussions between involved developers with more information.

5.3 Improving the IDE platform

In addition to improving the developer's experience, task structure may help solve issues related to a number of tools provided by an IDE, and may help simplify the development of tools.

Capturing and Recommending Workflow. In this paper, we have focused on information overload that developers face when working on the content of large systems. These developers also face information overload in the user interfaces of IDEs. Enterprise-application development tools, such as IBM's Rational Software Architect, offer sophisticated support for development across the lifecycle, which results in dozens of views and editors, and hundreds of user interface actions. It can be difficult for developers to know what features exist, let alone try to find them. Adaptive interfaces [7] and Eclipse's capabilities[13] address this in a general way, based on aggregate information about how features are used. We see potential for making the user interface more aware of the task being performed by capturing the task structure of developers who use these tools effectively, and then mining this information for task-specific interaction patterns of the user interface. Mined patterns may suggest ways to focus the user interface on only those tools needed for the completion of a particular task [22].

Simplifying Tool Development. IDE platforms such as Eclipse make it easy to build new tools that expose system structure. For example, a new view that shows all methods overriding the currently-selected method is easy to add. In our experience, it is harder to add tools that depend on some notion of task, and each tool must develop its own ad hoc model of task. While it is possible to layer task information on the models provided by the IDE through an index over existing elements and relationships, we see potential for task information to be more central. For example, it would be beneficial to be able to tag an element as being part of some named task, and to then be able to trigger an action based on when an attempt is made to synchronize that element with the repository. Task structure information could also be used to arbitrate user interface issues; for example, a tool might use a task's structure to determine which of several competing annotations are most applicable to show in the gutter of an editor.

6 Open Questions

Our working definition of task structure is simple and extensional. These characteristics make it easy to describe the possibilities of task structure and do not unduly constrain

---

[13] A capability in Eclipse is a feature set that can be enabled or disabled by a user. Capabilities are pre-defined and configured when the IDE is shipped.

what a task is or how developers work on tasks. It is an open question as to whether this definition is too simple. It may be that tools built on this definition require information about why artifacts were changed, or the order in which they were changed, to provide meaningful information to a developer. It may also be necessary to include in the definition notions of what constitutes a task, whether a task is worked on in one time period or across various blocks of time, amongst others. These questions will need both empirical and formal investigation.

Regardless of the programming language and software engineering technologies used, we believe many change tasks have an emergent crosscutting structure because it is impossible to simultaneously modularize a system for all kinds of changes that may occur. This statement deserves investigation, such as a characterization of task structure for changes performed on systems intended for a variety of domains, written in a variety of languages, of different ages, and so on. It is also an open question as to whether, at this point, more benefit to the developer might result from better support for explicit task structure than new means of expressing sophisticated modularity.

7 Related Work

7.1 Tasks and Desktop Applications

Explicit capture and manipulation of task information has been studied in the domain of desktop applications (e.g., document processors and email clients). Of these, the project most similar to some of our efforts is TaskTracer [5], which is intended to help knowledge workers deal effectively with interruptions, and which seeks to help knowledge workers reuse information about tasks completed in the past. TaskTracer monitors a worker's interaction with desktop application resources, such as mail messages and web documents, attempting to build up a grouping of resources related to a particular task. The worker has to name the task being worked upon when they start the task. Although some of our goals overlap, we differ fundamentally from TaskTracer in our intention to maintain fine-grained structural information across artifacts; TaskTracer works only at the level of resources or files. We believe the collection of fine-grained information provides several benefits. For instance, we can support detailed comparisons about how a current and a past task compare. As a second example, we can trigger the recall of potentially useful information for a task based on the current task context.

7.2 Tasks and Development Environments

In the context of development environments, the term task has largely been used from a tool builder's point of view. For example, the Gandalf project recognized the variety of tasks that needed to be supported by the software development process and created a suite of tools to support the generation of an environment particular to a project [8]. The researchers recognized the need to deal with such issues as expertise of the developer, but focused on the problems that were more important at that time, such as handling the syntax and semantics of the languages being used to develop a system.

More recently, IDEs have introduced user-defined scopes as a way of approximating a concept of task similar to the way that we use the term in this paper. For example, in

Eclipse, a developer can define a *working set* which is a set of resources related through the system's containment hierarchy over which queries may be executed and saved. Working sets are more coarse-grained than task structure and a developer must evolve working sets as they change tasks, as opposed to our concept of task structure which evolves from a developer's work.

7.3 Manipulating Program Fragments

The size and complexity of software systems has led to many approaches for extracting and operating on fragments of a system. We describe four approaches that are similar to our idea of task structure as a collection of system fragments. Tarr and colleagues introduced the idea of multi-dimensional software decomposition to support fragments that correspond to concerns [25]. Task structure is similar in cutting across artifacts, but a task need not have the same conceptual coherence that one expects from a concern. In multi-dimensional software decomposition the main operation supported on a fragment is the integration of code into a system, whereas we have considered how task structure supports development-oriented operations, such as work conflicts between team members. Our earlier work on concern graphs is also related to modelling and manipulating fragments of a system's structure that relate to concerns [20]. A main operation supported on a concern graph is the detection of inconsistencies between a version of a system in which a concern graph is defined and an evolved version of the system [18]. The support of inconsistency detection is possible because a concern graph captures more intentional information than task structure. As a third example, virtual source files were proposed to allow a programmer to define an organization for parts of the system appropriate for a task [1]. A virtual source file is defined intentionally based on queries and can be used for such operations as determining conflicting changes between team members (similar to our description in Sect. 4). In contrast to virtual source files, task structure emerges from how a developer works on the system as opposed to requiring the developer to state their intention on a structure of parts of artifacts relevant to a task. Finally, Quitslund's MView source code editor supports the juxtaposition of code elements selected by a query in a single view [16]. Although a fragment in his system is restricted to the results from a set of queries over source code, it shares a similarity with task structure in enabling a development-oriented operation over the fragments, namely enabling editing of the code in a single window. Dealing with fragments in a task-oriented manner may enable better integration of the various fragment ideas into the work environments of developers.

8 Summary

Tools are supposed to make us work more effectively. IDEs have served this purpose for developers in recent years. However, as systems grow more complex, the effectiveness of these development environments is breaking down because they do not adequately support tasks that involve changes to multiple artifacts. In this paper, we have described how many of these tasks do have a structure; the structure emerges from the way in which a developer works with the system. This emergent task structure can be identified

and used by an IDE to focus existing views and enable new operations. This support matches the way a developer works, allowing them to modify a system without being overwhelmed by its complexity.

9 Acknowledgement

References

1. M. Chu-Carroll and J. Wright. Supporting distributed collaboration through multidimensional software configuration management. In *SCM*, volume 2649 of *LNCS*, pages 40–53. Springer, 2001.
2. M. C. Chu-Carroll and S. Sprenkle. Coven: Brewing better collaboration through software configuration management. In *SIGSOFT '00/FSE-8: Proc. of the 8th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering*, pages 88–97. ACM Press, 2000.
3. D. Čubranić and G. C. Murphy. Hipikat: Recommending pertinent software development artifacts. In *ICSE '03: Proc. of the 25th Int'l Conf. on Software Engineering*, pages 408–418. IEEE Computer Society, 2003.
4. D. Čubranić, G. C. Murphy, J. Singer, and K. S. Booth. Learning from project history: a case study for software development. In *CSCW '04: Proc. of the 2004 ACM Conf. on Computer Supported Cooperative Work*, pages 82–91. ACM Press, 2004.
5. A. N. Dragunov, T. G. Dietterich, K. Johnsrude, M. McLaughlin, L. Li, and J. L. Herlocker. TaskTracer: A desktop environment to support multi-tasking knowledge workers. In *IUI '05: Proc. of the 10th Int'l Conf. on Intelligent User Interfaces*, pages 75–82. ACM Press, 2005.
6. E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.
7. S. Greenberg and I. H. Witten. Adaptive personalized interfaces – a question of viability. *Behaviour and Information Technology - BIT*, 4:31–45, 1985.
8. A. N. Habermann and D. Notkin. Gandalf: software development environments. *IEEE Trans. Software Engineering*, 12(12):1117–1127, 1986.
9. J. Heer, S. K. Card, and J. A. Landay. prefuse: a toolkit for interactive information visualization. In *CHI '05: Proc. of the SIGCHI Conf. on Human Factors in Computing Systems*, pages 421–430. ACM Press, 2005.
10. M. Kersten and G. C. Murphy. Mylar: a degree-of-interest model for IDEs. In *AOSD '05: Proc. of the 4th Int'l Conf. on Aspect-oriented Software Development*, pages 159–168, 2005.
11. G. Kiczales, E. Hilsdale, J. Hugunin, M. Kersten, J. Palm, and W. G. Griswold. An overview of AspectJ. In *ECOOP '01: Proc. of the 15th European Conf. on Object-Oriented Programming*, pages 327–353. Springer, 2001.
12. H. Masuhara and G. Kiczales. Modular crosscutting in aspect-oriented mechanisms. In *ECOOP '03: Proc. of the 17th European Conf. on Object-Oriented Programming*, pages 2–28. Springer, 2003.

13. A. Mockus, R. T. Fielding, and J. Herbsleb. Two case studies of open source software development: Apache and Mozilla. *ACM Trans. Software Engineering Methodology*, 11(3):309–346, 2002.

14. D. L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12):1053–1058, 1972.

15. D. E. Perry, N. Staudenmayer, and L. G. Votta. People, organizations, and process improvement. *IEEE Software*, 11(4):36–45, 1994.

16. P. J. Quitslund. Beyond files: programming with multiple source views. In *Eclipse '03: Proc. of the 2003 OOPSLA Workshop on Eclipse Technology eXchange*, pages 6–9. ACM Press, 2003.

17. M. Rinard, A. Salcianu, and S. Bugrara. A classification system and analysis for aspect-oriented programs. In *SIGSOFT '04/FSE-12: Proc. of the 12th ACM SIGSOFT Int'l Symp. on Foundations of Software Engineering*, pages 147–158. ACM Press, 2004.

18. M. P. Robillard. *Representing Concerns in Source Code*. PhD thesis, University of British Columbia, 2003.

19. M. P. Robillard, W. Coelho, and G. C. Murphy. How effective developers investigate source code: An exploratory study. *IEEE Trans. Software Engineering*, 30(12):889–903, 2004.

20. M. P. Robillard and G. C. Murphy. Concern graphs: Finding and describing concerns using structural program dependencies. In *ICSE '02: Proc. of the 24th Int'l Conf. on Software Engineering*, pages 406–416. ACM Press, 2002.

21. M. P. Robillard and G. C. Murphy. Automatically inferring concern code from program investigation activities. In *ASE '03: Proc. of the 18th Int'l Conf. on Automated Software Engineering*, pages 225–234. IEEE Computer Society Press, 2003.

22. M. P. Robillard and G. C. Murphy. Program navigation analysis to support task-aware software development environments. In *Proc. of the ICSE Workshop on Directions in Software Engineering Environments*, pages 83–88. IEE, 2004.

23. A. Sarma, Z. Noroozi, and A. van der Hoek. Palantír: Raising awareness among configuration management workspaces. In *ICSE '03: Proc. of the 25th Int'l Conf. on Software Engineering*, pages 444–454. IEEE Computer Society, 2003.

24. M.-A. D. Storey, D. Čubranić, and D. M. German. On the use of visualization to support awareness of human activities in software development: A survey and a framework. In *SoftVis '05: Proc. of the 2005 ACM Symp. on Software Visualization*, pages 193–202. ACM Press, 2005.

25. P. Tarr, H. Ossher, W. Harrison, and S. Sutton Jr. N degrees of separation: Multi-dimensional separation of concerns. In *ICSE '99: Proc. of the 21st Int'l Conf. on Software Engineering*, pages 107–119. IEEE Computer Society Press, 1999.

26. D. Woods, E. Patterson, and E. Roth. Can we ever escape from data overload? A cognitive system diagnosis. *Cognition, Technology & Work*, 4(1):22–36, 2002.