Explicit Concern-Driven Development with ArchEvol

Eugen C. Nistor and André van der Hoek
Department of Informatics
University of California, Irvine
Irvine, CA, USA
{enistor, andre}@uci.edu

Abstract—Supporting developers in examining and evolving a software system in terms of concerns is considered a critical capability in the face of the scale and complexity of today’s software. A number of existing approaches make an inroad to providing this support, but they fall short in key ways. This paper introduces ArchEvol, a new programming environment that embodies a new kind of approach, one we term explicit concern-driven development. The vision is threefold: (1) a fine-grained concern model maps concerns to code, (2) concerns are visualized at both the code level, to assist in the actual act of making changes, and the architectural level, to assist in gauging levels of scattering and tangling, and (3) automated support assists developers in maintaining the concern mapping over time. Developers, then, continuously examine, structure, and modify the software they produce in terms of concerns. We introduce our approach, discuss how we have realized it in ArchEvol, and present the results of a first set of evaluations that demonstrate its potential.

Keywords—software concerns; development environments; software evolution.

I. INTRODUCTION

The size and complexity of today’s software systems have become of such scale that it is usually impossible for an individual to comprehend a system in its entirety. Instead, developers have to make do with understanding only parts of a system. Clearly, then, being able to locate, understand, and modify just those parts of a system that are relevant to a particular task is a critical capability.

A question arises as to from which perspective these activities (locating, understanding, and modifying relevant code) should take place. Just knowing which parts of a software system (e.g., lines of code, modules, interfaces, configuration files) must be accessed to enact a particular change is insufficient. A growing consensus is emerging that developers must be able to examine and evolve a system in terms of concerns, that is, semantic and common understandings of why certain pieces of code exist and how they are related.

Developers often already have some notion of where concerns are implemented in a system. In order to make changes to a code base, however, they need a much more precise understanding. They need to know which concerns are implemented by which code elements so that the modifications they are about to make can introduce new concerns, modify existing concerns, or delete obsolete concerns—all while respecting the presence of other concerns in possibly the same parts of the code. Moreover, with the cumulative effect of change after change, the entire “map” of where concerns are implemented may change dramatically. Particularly when concerns become increasingly scattered and tangled, this may significantly deteriorate the quality of the system. Identifying concerns in code is therefore not just a one-time exercise in support of a particular change. It is crucial to continuously examine and improve the structure of the system as a whole.

To date, two canonical approaches have emerged that assist developers in examining and evolving code from a concern-based perspective. The first, inspired by the original principle of separation of concerns [10, 26], centers on providing language and tool support to modularize a code base in terms of concerns. Aspect-Oriented Programming is perhaps the best known example of this approach [20], extending traditional modularization techniques with special language constructs for specifying concerns that crosscut module boundaries, and providing tool support for weaving such concerns into a baseline version of the code.

The second canonical approach relies on analysis methods, either fully automated or as guided by developers, to find code that is relevant to a particular concern. One example is FEAT, which, by constructing a concern model through queries over the code, helps developers locate “related code” [27]. Other approaches look for similarities in the structure of code to find crosscutting concerns automatically [5, 6].

Both approaches have their intrinsic drawbacks. The first faces the problem of dominant decomposition [31]. Irrespective of the primary modularization strategy that is selected, there will always be concerns that crosscut the resulting modularization and thus cannot be cleanly separated. The implementation of those concerns becomes scattered throughout the system, leading to difficulties in keeping track of and maintaining them since the language and tool cannot account for them.

The second approach suffers from having to rely on analyses (often direct queries) over the source code to approximate concerns. However, concerns can be nebulous
and not directly related to some set of syntactic relationships in the code. The results of an analysis, then, may not capture the human intent of what the developer meant a concern to be.

In this paper, we propose a third kind of approach, one in which software development is explicitly driven in terms of concerns. Our vision relies on three main elements: (1) a fine-grained concern model in which every concern has an associated mapping to precisely the lines of code in precisely the set of artifacts that implement the concern, (2) visualizations of concerns at both the code level, to assist developers in the actual act of making changes, and the architectural level, to assist in gauging overall levels of scattering and tangling, and (3) automated support in the form of various heuristics through which the concern model is maintained over time. As a result, the concerns at play are always visible to a developer making changes, who can, when coding, determine how the changes being made impact these concerns. Working with code, then, becomes a continuous reflection over concerns, with the concern model serving as an explicit guide in the exploration and evolution of the system at hand.

We have implemented our approach in ArchEvol, an extension to Eclipse that wraps it with a concern-based focus. Using ArchEvol, we performed a three-pronged evaluation to assess the potential of explicit concern-driven development. Because ArchEvol’s overall environment and approach is quite complex, we sought to initially evaluate individual aspects. Particularly, a first study examined if users were able to understand, use, and update the concern model; a second study evaluated the value of the architectural visualizations; and a third study assessed the long-term effectiveness of the heuristics with a simulation of an extended use of ArchEvol. Results from these studies show promise for our approach in terms of its value in informing developers of which concerns govern which parts of the system, guiding them in their changes, and assisting them in effectively maintaining the concern model over time.

II. BACKGROUND

The principle of separation of concerns, as introduced by Dijkstra decades ago [10], has emerged as one of the fundamental principles of software engineering. The principle states that one should be able to reason about a concern in isolation of others, such that complex systems can be examined and understood from the perspective of smaller problems.

It is interesting to note that Dijkstra did not provide a definition of a concern, nor has there emerged a uniformly accepted definition since then. A somewhat commonly used description is that a concern can mean “any matter of interest in a software system” [30]. Other descriptions tie their interpretation of what concerns are to the actual modularization that is used to realize them. For instance, some state they are aspects [20], while to others they are views [22]. Concerns that are identified before coding are sometimes called early aspects or themes [3].

This paper, we define a concern as a concept that relates a group of software fragments. A concern often is a semantic, and ideally also common, understanding of why certain pieces of code exist and are related. Our definition goes back to the intent of the principle of separation of concerns, observing that it aims to allow developers to reason about a concern in isolation from others. It adds to that, however, the desire that a concern has longevity. It should not be useful to just one developer at one time, but generally be an artifact of fluid permanence through which groups of developers are guided as they evolve a software system. We also note that our definition intentionally is more precise than the aforementioned “anything in which a developer may be interested” in tying concerns to code fragments and their existence. This adds purpose to the concern, and grounds the concept of a concern more strongly in the software domain.

Our definition of concerns embraces a number of intrinsic properties of concerns. First, it views a concern as existing over some set of artifacts. While in certain cases it may be possible to codify concerns into the target artifacts (for instance as aspects, or even simply as classes), the problem of dominant decomposition prevents a clean treatment of any concerns that crosscut the chosen dominant modularization. In some cases, multiple concerns are overlapping the same lines of code in such a way that it becomes difficult to separate them cleanly [12]. That is not to say that aspect-oriented programming and other such approaches are not valuable; it just states that there exist limitations to the applicability of these approaches and situations in which they will break down.

The second property of concerns that our definition is intended to embrace is that scattering and tangling of concerns is normal. It is to be expected that certain concerns are spread out over multiple artifacts, particularly the concerns that exist at a higher level of abstraction (a system requirement of reliability, for instance). It is similarly to be expected that certain code fragments are part of multiple concerns (a logging component that is created to improve the reliability of a system may also need to address a particular security concern and a scalability concern). The significance of this property of concerns is not to be understated: for any approach to managing concerns to effectively support the software development process, it must integrally deal with the scattering and tangling of concerns.

The issues of scattering and tangling become particularly important as concerns may exist at any level of abstraction, which is the third property of concerns. In reality, hundreds or thousands of concerns might govern a system, from concerns that address a minute portion of the code to

---

2 This definition is an adaptation from the one by Robillard [19], tying concerns more directly to the code.
overarching concerns that relate significant portions of an entire system [25].

A closely related issue is that different concerns may relate groups of software fragments from different points of view, a fourth key property. One concern may relate a set of source files that implement a feature. Another concern may look at the same source files from the perspective of testability and label certain fragments as more troublesome than others. These different perspectives are equally valid in providing useful information to developers.

Finally, we return to the issue of longevity. The fact that concerns have a life span and permanence (as discussed) is the fifth property, but it is the implications of this fact that lead to the sixth property: the group of software fragments that are related through a concern evolves. Code undergoes myriad changes over time, as a result of which new fragments may be added to a concern, existing fragments may be changed, or certain fragments may become obsolete for a particular concern and have to be removed from its grouping. Of course, this even holds for the concerns themselves: as the system evolves, new concerns could arise, existing concerns may change, and old concerns may become obsolete. This fluidity is a key property of concerns, and—indeed—the driver for much ongoing research and development in concern management.

III. APPROACH

The approach introduced in this paper adopts the philosophy of placing concerns front and center in the development process (as in the first canonical approach), but also recognizes that a concern model must be maintained separately from the code so that the full range of benefits of concern management can be achieved (as in the second approach). Our approach, then, relies on a permanent concern model that is kept separately from the code, but that is readily and continuously updated as the code evolves.

We term our approach explicit, concern-driven software development. The key idea is to perform all coding activities in terms of concerns; that is, to make all software changes in terms of specific concerns that the changes address. This requires two primary capabilities. First, the concern model must be made visible to developers. They must know the set of concerns that a system addresses and where each of these concerns is implemented in the software. Second, as the code evolves, the developers must be able to keep the concern model and its mapping onto the code up to date so it always reflects the current version of the software. Realizing these capabilities involves the following design considerations:

- How are concerns structured in a concern model and how is the concern model mapped onto the code? Our choice here is to: (1) implement the concern model as a tree of concerns, and (2) trace each concern to fine-grained, textual fragments of code. We choose a tree so developers can add some form of organization to what would otherwise be a potentially very long list of concerns. A full graph, which is often supported in other approaches [15, 30], could also have been used, but it would have led to the added requirement of managing intricate webs of concern to concern relationships whereas most concerns are simply either independent or a subconcern of another. We choose to map each concern onto a set of textual fragments rather than more semantic code elements (classes or methods) to increase the flexibility of our approach. It becomes possible, for instance, to include parts of a line of code into a concern, such as a method parameter, an inheritance relationship, or some other fine-grained code element. Generally, it may not be necessary to do so and mappings will be on sets of contiguous lines of code, but the option has to exist to support fine levels of abstraction when needed.

- How to inform developers of concerns governing certain parts of the code? Our choice here is to offer two visualizations, each with a different purpose. The first visualization is always present and uses small colored hash marks in front of each line of code to indicate which concerns are addressed where in a source artifact. This helps in providing a detailed view of concerns when editing some particular code, as a developer always knows which other concerns are present and should be taken into account during the modification of that code. The second visualization is interactive and geared towards exploration. By selecting a concern from the concern model, all fragments belonging to that concern are marked in the color of that concern. Both visualizations have been carefully designed to not be intrusive with respect to the task of coding; the concern information is present, but does not cognitively intrude.

- How are concerns visualized at the system level? To show concerns at a higher level of abstraction than code, we use an architectural view based on the main components of the system and their inter-connections. We overlay information about concerns by aggregating the code links from the concern model to the level of the components. Specifically, we calculate two metrics: one that shows how much of a given concern is implemented in a component, and one that shows how much of a given component implements a given concern.

- How are concerns kept in sync with the code, when the code evolves? We offer two complementary, heuristic techniques. The first technique automatically tags code during the process of making changes. A developer selects one or more concerns, then edits their code, and any code that has been modified during the editing process is labeled with the concern(s). The second technique relies on where developers make changes to tag or not tag the code with concerns when editing some parts of a line. We offer two complementary, heuristic techniques. The first technique automatically tags code during the process of making changes. A developer selects one or more concerns, then edits their code, and any code that has been modified during the editing process is labeled with the concern(s). The second technique relies on where developers make changes to tag or not tag the code with concerns when editing some parts of a line. We offer two complementary, heuristic techniques. The first technique automatically tags code during the process of making changes. A developer selects one or more concerns, then edits their code, and any code that has been modified during the editing process is labeled with the concern(s). The second technique relies on where developers make changes to tag or not tag the code with concerns when editing some parts of a line.
keyboard or mouse to highlight a text fragment, and to then assign it to or unassign it from a concern.

Comparing our approach to the six intrinsic properties of concerns discussed in the previous section, we note that: (1) it involves a separate concern model that overlays onto the code, (2) it visualizes scattering and tangling of concerns, (3) it supports developers in using concerns at multiple levels of abstraction, (4) it equally supports multiple points of view, (5) it supports long-term value of the concern model, and (6) it supports the co-evolution of the concern model and the code, so that they stay in sync. The approach, thus, addresses all six of the intrinsic properties of concerns.

At the same time, we recognize that our approach relies on developers interpreting, using, and updating the concern model properly. The value of the concern model stems from being accurate and complete in its mapping onto code. The heuristics offer an important step in this direction, but if this mapping becomes incorrect or incomplete, the effectiveness of using our approach goes down. Note that this does not mean that the approach fails entirely; it just means that the concern model is less helpful than it could be. Moreover, it does not mean that the problem is permanent or will continue to grow out of hand once a mapping has been corrupted. It, in fact, is quite possible that a developer may correct the mapping to once again be in sync with the code, sometimes even through the automatic heuristics and without manual intervention.

IV. IMPLEMENTATION

We have implemented our approach in ArchEvol, a set of extensions to Eclipse [11] that provide features for modeling and mapping concerns to code, visualizing concerns, and maintaining the mappings over time.

A. Concern modeling

The features of ArchEvol build upon an existing infrastructure that maps architectural elements onto code elements [23]. Specifically, each component is implemented in its own Eclipse project, and the architectural description is stored in a separate project.

ArchEvol extends this basic architecture to code mapping, which is based on XML [8], with a new schema for capturing concerns as well as the associated references to source code fragments. A concern is defined by a name, description, and color. A unique identifier is internally used to distinguish concerns. A concern has a list of references to source code fragments, and may also contain other concerns, making the model hierarchical. The reference list points to the textual fragments in the source code files that make up a software system. These fragments may reside in different Eclipse projects, being part of the implementation of different components. Note that these source code files are not necessarily restricted to contain actual Java or other code; the files may equally be configuration files, build files, or any other textual artifact used in a project. Because our concern model is kept separate from the code, this generality is enabled, a small but critical advance over other concern models.

Each reference stores the position of the fragment as a start and end location inside a file. A snapshot of the text contained by the fragment is kept in the model for visualization and performance purposes.

High-level information about concerns is calculated as two aggregate metrics on the concern model: concern extent in a component (CEC) and component relevance for a...
concern (CRC). CEC indicates the importance of a concern to a component. It is calculated by dividing the number of lines of code in a component that relate to the concern by the total number of lines of code in that component. The higher the number, the more this concern pervades the component. CRC indicates the importance of a component to a concern. It is calculated by dividing the number of lines of code in a component addressing a concern by the total number of lines of code implementing that concern across components. The higher the ratio, the more the component is responsible for the implementation of the concern.

When calculated for each of the concerns and each of the components, CEC and CRC help identify scattering and tangling. Multiple high CEC values within a single component likely indicates undesired tangling, and multiple high CRC values across components for the same concern likely indicates undesired scattering.

B. Visualization

The user interface of ArchEvol provides different visualizations. The concern model is standardly visualized in a tree view, as shown in Fig. 1 as (1) on the far right side. The tree view is similar in organization to the package explorer in Eclipse, and offers a view of the concern model that can be expanded down to each of the fragments residing in the code. This view is typically used to examine the concern model and navigate to and explore the source code based on concerns.

A second view, labeled (2) in the figure, was created for visualization purposes. This view splits the content of the tree view. The top of the view is a compact version of the concern model, with each concern showing only a color textbox and no additional details. It provides users with an overview of the concerns and their colors, something that the tree view on the right cannot effectively do as soon as some concerns are expanded. The bottom of the view shows the fragments that constitute the concern that is selected in the top view. From here, a developer can navigate to each of the fragments. From each of the two concern model views, individual fragments belonging to a concern can be highlighted in the source code by double-clicking a fragment reference, in which case the fragment is temporarily selected in the editor. When a more global, permanent marking is needed, selecting the checkbox next to a concern in the compact view will use the Eclipse annotation mechanism to highlight all of the fragments visually in the source code by changing the background color of the fragments to correspond to the color of the concern.

Naturally, some concerns overlap, with the same lines of code belonging different concerns. A limitation of the existing annotation mechanisms in Eclipse editors is that it does not support the visualization of overlapping annotations (a single line of code can only have one background color). To overcome this limitation, we created the Concern Overview view inside Eclipse, which is shown in Fig. 1 (3) to the immediate left of the source code editor (4). For each line of code, and for each concern in the concern model, this view shows a box colored in the same color as the concern if one of its source code fragments includes this line. The presence of multiple boxes (two, for instance, in the middle part of the code shown in Fig. 1, one green and one purple) indicates multiple concerns overlapping. The view is synchronized and scrolls along with the source code editor.

The architectural view, shown in Fig. 2, presents the CEC and CRC metrics as small, color-coded boxes inside the

Figure 2. ArchEvol’s architectural visualization.
components. Typically, either CEC or CRC metrics are shown across the entire architecture, easily toggled via a switch. For each concern for which a metric has a value greater than zero for a metric, a box is drawn as a progress bar filled up with the value of the metric as a percentage. The Concern Visualization View controls which concerns are selected in the architectural view, so the developer can make pairwise and groupwise comparisons of concerns and how they scatter and tangle over the components.

We intend the architectural visualization to be used in exploring where one or more concerns are implemented in the system. Selecting one concern and the CEC metric shows where in the system is implemented, and how much of each component is related to the concern. For example, in Fig. 2, the buffer concern is implemented in four of the five visible components. In three of these, most of the code is related to it. In one (options), only a small fraction of the code is related to the concern. Switching to visualizing CRC metrics inside each component would enable analogous conclusions to be drawn, just now from the perspective of the concerns instead of components.

C. Concern maintenance

The features described thus far help a developer understand code and the concerns that govern it, but they also need to be able to evolve the concern mapping with the code changes that they make. Two default heuristic behaviors and a manual technique are available.

The first heuristic, the proximity heuristic, is fully automatic, updating the fragments when text is inserted or deleted from a file. ArchEvol listens to change events stemming from Eclipse editors and, when text is modified inside a file, all fragments that point to locations in the file below the change need to be updated. In addition, if the modified text touches a fragment, then this fragment needs to be modified to either include the new text or trim out the deleted text. Adding text immediately next to a fragment will cause this fragment to expand and include this text. At the end, modified fragments are compacted, by merging overlapping fragments and removing empty ones.

The second heuristic is implemented by the ArchEvol concern recorder. This recorder operates by a developer choosing one or more concerns that are made active. Any change made to the code becomes a part of those concerns, with care taken to resolve overlap between this method and the default behavior. This is analogous in operation to how changes in CM systems are tagged with the modification request number of bug number as a result of developers choosing a task to work on. Similar to that situation, care must be taken by the developers to be diligent to switch the concerns when they switch what they are working on.

Complementing the heuristics is ArchEvol’s manual approach. By selecting some lines of code and assigning them to a concern, or unassigning them from a concern, developers can update the concern model themselves. This allows them to correct any mistakes made by the heuristics, as well as integrate legacy code into our concern-driven approach.

V. Evaluation

Ideally, our approach is evaluated with a long-term study of multiple developers using ArchEvol to develop and evolve a software system. Such a study is costly and requires complete buy-in into the tool, however. As a first assessment, we therefore performed a set of three smaller, focused evaluations that each assess a different aspect of ArchEvol yet together cover all of its functionality. Note that our objective with each of the three studies below was not to obtain statistically significant results, but simply to provide us with initial indications regarding the viability of our approach. The below necessarily is a brief summary of our evaluation set up and results. Full details are available in a dissertation [25].

A. Study 1: jEdit

The first study looks at the use of the concern model and its code-level visualizations by developers. Besides testing the core functionality and ideas of ArchEvol in and of themselves, the study was designed to also provide initial comparative results with respect to two other settings: use of the basic Eclipse development environment (as a baseline) and use of FEAT [27] (a tool that also provides a concern model, albeit in a different format and with a somewhat different intent).

Setup: We recorded and analyzed the activities of eight participants who each made three changes to jEdit [18], an open source document editor with a relatively sizeable set of features. The choice of jEdit was influenced by its use in similar studies [9, 28]. Eight participants, all graduate students in the Department of Informatics at the University of California, Irvine who had not seen the code for jEdit before, were assigned to three groups, balanced by experience level: four participants used ArchEvol, two FEAT, and two Eclipse.

Each participant was given three incremental change tasks. The first task was similar to the task used in the previous studies, and required the participants to enhance the autosave feature in jEdit with an option to turn it on or off. The second task required participants to change the existing backup feature by removing the functionality through which users could choose a prefix and suffix for the name of the file for the backup, and replacing it with a fixed prefix. The last task required participants to modify the system such that backups are performed on every autosave instead of every save.

This set of changes was chosen because it exercises a variety of scenarios in terms of how to use the concern model and how to keep it up to date. The first task represents the addition of functionality, the second task removal of code, and the third task a mix of the two. In each case, the concern model has to be updated as well.
For each change to be made, a participant was given a brief description, the name of an initial class from which to begin their exploration (to equalize across the three tools), and, if applicable, a pointer to where in the user interface the change had to be made. The study had a total length of two and a half hours, with one and a half hours allocated to the three change tasks.

Participants using ArchEvol or FEAT were given a brief tutorial of the features of the tool using a small sample application different from jEdit. For the study proper, the concern model was prepared beforehand with thirteen concerns, of which two were the ones the participants were asked to actively use in this study. For ArchEvol, source code references for the fragments belonging to the concerns were provided. For FEAT, standard queries were provided. For straight Eclipse, a separate document listing the concerns was provided.

Observations: We used screen recordings to analyze how developers performed their tasks, marking points in time when developers switched context among different types of activities. Two activities are relevant for our analysis here: navigation (searching for a particular piece of code) and inspection (interpreting a piece of code). We expect the use of the concern model to impact navigation time, because it can be used to quickly find relevant pieces of code, and inspection time, because the concern visualizations highlight which concerns govern which lines of code, relieving developers from manually deriving such information.

It is crucial to not just take these times at face value, as individual differences among developers can heavily skew the results. We therefore relate the time spent to the performance of each participant, a score that measured how many of the small individual code changes that together make up each task are actually performed. We gave 1 point for each such small change that was performed as expected, 0.5 points for each change that was partially correct but did not quite fit in with the existing code, and 0 points for incomplete or incorrect changes.

We performed a variety of analyses, the most meaningful of which is presented in Fig. 3. We grouped the time spent on navigation and inspection for all participants and for all tasks, and plotted them against the performance. With the exception of one outlier, a participant who was unable to complete any of the tasks, we note that participants with comparable performance spent less time on navigating and inspecting the source code when using ArchEvol.

We also examined whether developers were able to keep the concern model in sync with each code change they made. The total number of updates that were required to the mapping was thirty-four. We differentiated: (1) when participants made updates by hand, (2) when ArchEvol made updates through its heuristics, (3) updates that were not made but should have been made, (4) updates that were not made since the subject did not complete the change, (5) updates the tool could not support (only with FEAT, which cannot support concerns involving files other than source code). Participants using Eclipse were asked to manually keep a log of changes, based upon which we categorized the updates. Fig. 4 presents the results of this categorization, with the five categories captured as line segments from left to right. We see that: (1) participants using ArchEvol ended up with mappings that were as complete or more complete than participants using FEAT or Eclipse (as determined by comparing the cumulative of the first two segments), (2) participants using ArchEvol performed less manual mapping of concerns (as determined by comparing the first segment), and (3) the automated techniques captured a significant portion of the updates that were necessary (as determined by examining the length of the second segment). Taking into account tasks that participants did not complete by the time limit, thus ignoring mappings that were not necessary yet, participants using ArchEvol correctly mapped from 60% to 75% percent of all fragments they should have mapped up to that point in their task.

While this does not reach the ideal of 100% correct mappings after the three changes, it is a very encouraging number for a new approach to supporting concerns in software development. Participant ArchEvol2 is particularly of note, as they used ArchEvol’s heuristics the most effectively and consequently had a large number of
TABLE I. ARCHSTUDIO STUDY RESULTS.

<table>
<thead>
<tr>
<th>Concerns</th>
<th>Metric</th>
<th>Components</th>
<th>Dev. 1 Errors</th>
<th>Dev. 2 Errors</th>
</tr>
</thead>
<tbody>
<tr>
<td>change set</td>
<td>CEC</td>
<td>11</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>change set event</td>
<td>CEC</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>detach</td>
<td>CEC</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>explicit ADT</td>
<td>CEC</td>
<td>3</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>change set</td>
<td>CRC</td>
<td>11</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>change set event</td>
<td>CRC</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>detach</td>
<td>CRC</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>explicit ADT</td>
<td>CRC</td>
<td>3</td>
<td>6</td>
<td>1</td>
</tr>
<tr>
<td>UI, change sets</td>
<td>-</td>
<td>3</td>
<td>4</td>
<td>-</td>
</tr>
<tr>
<td>change set event</td>
<td>change set sync</td>
<td>-</td>
<td>3</td>
<td>2</td>
</tr>
<tr>
<td>change set sync, relationships, detach</td>
<td>-</td>
<td>0</td>
<td>0</td>
<td>-</td>
</tr>
<tr>
<td>detach, explicit ADT</td>
<td>-</td>
<td>2</td>
<td>2</td>
<td>-</td>
</tr>
</tbody>
</table>

mappings detected automatically. ArchEvol2 principally used the recording feature, that is, before every change they chose the concerns they were working on. The other ArchEvol participants solely relied on ArchEvol’s default behavior concerning contiguous fragments, and manually adjusting these mappings as necessary. As such, we would anticipate improved performance with further training into how to effectively use ArchEvol’s heuristics.

B. Study 2: ArchStudio

The second study examined the value of ArchEvol’s architectural features. In particular, we wanted to examine the role of our two metrics and how they do or do not augment the mental model that developers have about the software systems that they are developing.

Setup: We used, as our baseline, a system that already used an architectural model in its development. In this way, we could isolate the effects of our added visualization from the use of an architectural model itself. The system that we chose to use is ArchStudio [1], a software architecture development environment. ArchStudio is used to create and maintain architectural models, but also has an architectural model of itself that its developers actively use.

Two participants were used in this study, both graduate students within the Department of Informatics at the University of California, Irvine, and both working together in developing a novel, crosscutting feature for ArchStudio. They particularly were including support for “change sets” [16] as part of their day-to-day work, a non-trivial, extensive, and multi-week modification.

Independent of the participants, we manually populated a concern model with a set of concerns relevant to the change sets feature. Five concerns were selected, based on the most relevant keywords from the names of packages and classes: “change set sync”, “change set event”, “detach”, “relationships”, and “explicit ADT”. These concerns were implemented in 11 out of the total 46 components in ArchStudio.

The study had three main phases. The participants were asked, first, to give their own description of each of the five concerns that were used in the system. This step helped to validate that the participants understood the meaning and role of the concerns we chose. The second step asked the participants to examine the architectural view without the concern visualization, and to try to give their best estimate as to where concerns are implemented in the system. Then, in the third step, they were asked to use the architectural visualization with the CRC and CEC metrics and discuss the values they found in contrast with their previous estimates.

In step 2, particularly, we asked the participants to name the components that participated in the implementation of a concern, and give an estimate for the value of our two metrics (CRC and CEC) as one of “very little”, “some” and “most or all”. The questions included both evaluating single concerns and determining where combinations of two or three concerns overlapped in the components.

Observations: The results of our study show three important facts about the concern model and the usefulness of the architectural component of ArchEvol. First, it gave us an encouraging result about the concern model. Both participants were able to recognize the selected concerns, by their names, as relevant concepts for the project. They also gave consistent, albeit slightly different definitions for each of them. A concern model can reconcile these different points of view, by creating a common vocabulary of concerns with precise definitions that can be shared by all developers working on the same project.

The second observation comes from comparing the ability of the developers to evaluate each concern and the associated metrics. Table I shows how many errors the two developers made in evaluating where and how concerns are implemented. For each developer, their first column shows how many components they identified erroneously (they thought the concern was implemented in a component but it was not, or failed to name a component in which it was) and the second column shows, in the case when they identified a component properly, how many metric values they evaluated wrongly (the rough estimation given was not within the realm of what they found using ArchEvol).

We found that even though the developers were close in identifying where concerns are implemented in the majority of cases, they had difficulty in evaluating the correct value of the metric. For example, they answered “very little” before looking at ArchEvol, and “most or all” when examining the architectural visualization with ArchEvol. Even more, never were the two participants in agreement in their initial answer, while after using ArchEvol they were much more so. This shows that the architectural view, in particular, can be very
useful in clarifying many situations where the developers do not agree or are unsure about a concern.

The third observation is that the ability of developers to determine where concerns are implemented deteriorates with the number of concerns being investigated in relation to each other. In Table I, these results are in the last four rows. While the answers for a single concern were mostly in agreement with ArchEvol, the answers for the questions asking where two or three concerns are implemented were considerably more difficult to estimate. Both subjects identified correctly, on average, only half the components that should have been included in their answers.

Clearly, then, having the architectural visualization with the associated metrics is helpful, and provides developers with a much more precise understanding of the actual state of the code base. Both participants confirmed this in subsequent conversation, and were surprised that they—despite being experts on the code—forgot or otherwise misunderstood parts of the code.

C. Study 3: ArgoUML

The third study is an offline study, with the purpose of determining whether the concern model can be used for a longer period of time than just a few changes. The study did not use any participants, instead simulating the use of ArchEvol by replaying historical changes as captured in the CM archive of ArgoUML [2].

Setup: We simulated the changes made to ArgoUML’s code base over a few years, as if a developer would have edited them in an Eclipse environment with ArchEvol. The concern model was primed with four concerns, manually identified in one of the first revisions available from the code repository (release 0.12, which corresponds to revision number 4608). We interpreted each change from the versioning repository by transforming the diff file format into a series of character deletions and insertions on individual lines of code. Using the Eclipse editor API, these insertions and deletions were replayed on the corresponding files.

ArchEvol intercepted these changes and modified the concern model accordingly using its proximity heuristic. At major releases, which occurred usually after several hundred intermediate versions, we manually inspected the concern model and performed the necessary updates to fix any missing concern-to-code links and remove any superfluous such links. Seven major manual synchronizations were performed, with the final revision number of ArgoUML imported being revision number 12046.

Observations: The main objective of our ArgoUML experiment was to determine how much effort developers would be required to expend on maintaining a concern model over time. As an indicator for this effort, we measured the number of fragments, at each major revision, that we had to manually fix to make the concern model accurate again. Fig. 5 shows the results.

It can be observed that, in the majority of cases, only about 10% of the fragments needed to be modified, implying that the proximity heuristic performs well in automatically updating concern-to-code links. Clearly, many of the changes in the ArgoUML repository pertain to existing fragments, either updating code within a fragment or placing new code directly next to it. This is an interesting find in and of itself, and shows that our choice to incorporate the proximity heuristic is crucially important in complementing the concern recorder where developers have to explicitly choose a concern.

In examining where the proximity heuristic breaks down, a common case was the insertion of new code that belonged to a concern but was placed in an entirely different part of the source code, expanding the scattering of the concern. This frequently happened in the early phases of incorporating new features, when their presence across the code base rapidly expanded. Clearly, in these cases it would be easy for the developer to use the concern recorder instead to make
sure that the new code became linked to the appropriate concern.

D. Threats to validity

The collective results provide the beginnings of evidence concerning the potential of ArchEvol. Developers can usefully and effectively use the concern model, they are able to maintain it, the architectural view and metrics augment their knowledge, and the heuristics are able to automate much of their task of keeping the concern model up-to-date.

That said, there are several limitations to our studies that restrict our ability to draw general conclusions. First, we had a limited number of participants and our results are (intentionally, at this stage of the research) not statistically significant. It was more important to us to first observe participants in using the basic functionality of ArchEvol, study its potential as an approach different from the two canonical existing approaches, and understand detailed behaviors in ArchEvol’s use, rather than push ArchEvol’s limits to the test. The lessons learned here clearly have influence on our next steps in developing ArchEvol (see Section 7), and can be used to design more in-depth future studies.

Other threats to validity are that we used graduate students, who may not be reflect programmers in industry, that jEdit might not be representative of all software, or that the changes we asked the participants to make do not reflect real-world kinds of changes. We mitigated these as much as possible by modeling our user studies after other, comparable experiments [9] and by choosing modification tasks that span a representative spectrum of change activities (new feature, removal of old feature, integration of one feature with another).

The fact that we only included a limited number of concerns in our studies is another threat to validity. A more comprehensive evaluation will take into consideration the ability of participants to cope with a large number of overlapping concerns. Clearly, this is an issue of future work.

VI. RELATED WORK

The problem of integrating concerns into the software development process has been, in general, treated from two different perspectives. One set of approaches uses concerns as the basis for modularizing a system into individual modules that each encapsulate a concern, changing the modularity mechanism of the programming language and updating tool support for the programming language to facilitate doing so.

Aspect-Oriented Programming (AOP) [20] is the canonical example. A crosscutting concern can be encapsulated in an aspect module, the code of which is weaved with the implementation of the “regular concerns” that constitute the base code. A similar approach, Multi-Dimensional Separation of Concerns (MDSOC) [31] abandons the distinction between regular classes and aspects in favor of structuring a program entirely in terms of concerns implemented in their own hyperslices. The program is then composed out of a series of hyperslices by weaving their specifications together. In both AOP and MDSOC, exactly how concerns are assigned to aspects or hyperslices is left to the programmer. In other approaches, such as Theme [3], a concern is already identified early in the life cycle and carried through to the programming phase.

With respect to these kinds of approaches, we note that the purpose behind ArchEvol’s concern model is quite different in focusing on understanding where concerns are implemented while still acknowledging that the same lines of code, as well as those surrounding them, can belong to multiple concerns. Concern-based programming approaches would require a developer to either encapsulate one of the concerns, in which case the other will need to be scattered over the base code and the aspect, or create a new aspect that captures the common code, in which case both concerns will be scattered over two aspects. We treat program modularity separately from concern organization, leaving developers the freedom to organize the code as they see fit, while still being able to find relevant concerns in code and reason about a system in terms of concerns.

The second type of approach supports identification of concerns in code. These approaches try to analyze the code and detect and group code elements that belong to a concern, either (semi-)automatically or manually. Program slicing [32] was one of the first methods to isolate parts of the source code related to a particular behavior automatically, which helps programmers understand a complex system. Software reconnaissance [33] is a technique to detect concerns in code by analyzing traces generated from executing a set of test cases. More recently, the analysis of traces generated by regression tests was used as a method to determine feature coupling [13]. All of these then equated to concerns. This certainly is useful in approaches rely on “cuts” through the program that are identifying certain kinds of concerns, but it must be recognized that not all concerns are of the nature of being identifiable in such a manner. Moreover, it is often difficult to reduce the full set of executed code to the particular concern of interest.

Some approaches try to find concerns in existing code so that they can be refactored to aspects. These approaches locate crosscutting concerns by examining properties of the code, such as fan-in analysis [21], source code clone detection [6], or CM repository change sets [5]. Other methods focus less directly on recovering aspects, instead recovering more general concerns by performing static analysis of source code [4], examining the comments associated with tasks [34], or using a representation of the source code based on natural language processing [29]. These methods, however, all rely on the programmers following disciplined naming conventions.

To compensate for errors made by automated approaches, several approaches use human guidance. Generally, they store the results of the human-guided analysis in the concern model, either extensionally (a copy of
the output) or intentionally (a copy of the query). Representative examples include FEAT [27], Intentional Views [22], JQuery [17], AspectBrowser [14], and Spotlight [7] all of which are intended to assist developers in the maintenance task, with the concern model serving as the starting point for navigating and examining a system at hand. ArchEvol relates to these approaches, but distinguishes itself by using both manual and automated heuristics to evolve the concern model.

From an IDE perspective, ArchStudio [1], FEAT [27], CME [15], and CIDE [19] are Eclipse-based environments that include features partially similar to ArchEvol’s. ArchStudio includes an architectural editor and viewer, but does not address the visualization of concerns over the architectural model. FEAT and CME include a concern model that helps developers in exploring the code, but use a representation of fragments that is less fine-grained than ArchEvol. CIDE was created to support development of software product lines (SPLs), and includes code visualizations that highlight feature implementations in a similar manner to an early version of ArchEvol’s [24]. It differs in that it uses the AST representation of the code to assign code fragments features. ArchEvol is unique in its focus on not only creating a concern model, but on integrating visualizations in code and architecture and supporting their evolution.

VII. CONCLUSIONS

We have introduced a novel approach, explicit concern-based development, in support of developers understanding and evolving a software system in terms of concerns. Through a fine-grained concern model that is kept up-to-date through heuristics enacted in the development environment, the approach integrally addresses key aspects of the nature of concerns, including their scattering, tangling, and evolution. A key differentiator of our approach is that it deals with multiple concerns at the same time, precisely illustrating where each concern is located in the code, summarizing the scattering and tangling in an architectural view, and in the process providing developers with a comprehensive, concern-oriented view of the software system at hand.

Initial evaluations show that developers effectively are able to use the concern model, are able to maintain it particularly when all of the heuristics are employed, and are able to augment their own understanding of the system with the architectural view and metrics.

Our future work addresses several directions. First, we wish to explore additional heuristics, to further improve the percentage of mappings that is automatically maintained. An interesting direction is to integrate the analysis employed in FEAT or other approaches; they are complementary to our existing heuristics. Second, we wish to expand the architectural view, particularly to provide a historical view of how scattering and tangling have improved or worsened over time. Finally, we plan to engage in more comprehensive evaluations of ArchEvol and our approach at large.

REFERENCES


