Unravel Programming Sessions with THRESHER: Identifying Coherent and Complete Sets of Fine-granular Source Code Changes

Stephanie Platz, Marcel Taeumel, Bastian Steinert,

Robert Hirschfeld, and Hidehiko Masuhara

Development teams benefit from version control systems, which manage shared access to code repositories and persist entire project histories for analysis or recovery. Such systems will be efficient if developers commit coherent and complete change sets. These best practices, however, are difficult to follow because multiple activities often interleave without notice and existing tools impede unraveling changes before committing them. We propose an interactive, graphical tool, called Thresher, that employs adaptable scripts to support developers to group and commit changes—especially for fine-granular change tracking where numerous changes are logged even in short programming sessions. We implemented our tool in Squeak/Smalltalk and derived a foundation of scripts from five refactoring sessions. We evaluated those scripts' precision and recall, which indicate a reduced manual effort because developers can focus on project-specific adjustments. Having such an interactive approach, they can easily intervene to accurately reconstruct activities and thus follow best practices.

1 Introduction

Software developers benefit from version control systems (VCSs), which manage collaborative work and persist whole project histories. In systems such as Git[†1] and Mercurial,[†2] developers can *commit* changed source code with descriptive messages into distinct branches to separate features or bug fixes. When committing, branching, or merging, new versions of the project arise and a particular development trace emerges in the VCS. Such historical information can then be used to ease code comprehension [31] or to reason about software evolution patterns [3].

Software evolution induces very high costs in software development [2] and hence developers are advised to follow best practices to maximize the benefits of VCSs. One important practice is *continuous integration* [7]: commit and test code on a regular basis. When committing, only store *small, coherent, complete* change sets with a brief yet descriptive message to improve comprehensibility [29]. A well-organized project history can also improve code review processes [1] and recommender systems [4]. —Robbes et al. [22] observed that developers frequently commit once per working day on average. Further research ([14][16][22]), however, shows that developers tend to commit *incoherent* change sets. Buse and Weimar [5] also found that only two-thirds of commits get assigned with an descriptive message.

We argue that one challenge for developers is that multiple activities interleave [18]—often without being noticed even in short sessions as illustrated in Figure 1. Manual refactorings, for example, typically include changes to affected code that get accidentally postponed because developers sim-

---

完全かつ一貫した細粒度コード変更集合の検出によりプログラミングセッションを解きほぐす THRESHER

Stephanie Platz, Marcel Taeumel, Bastian Steinert, and Robert Hirschfeld, Hasso Plattner Institute, University of Potsdam, Germany.

増原英彦, 東京工業大学 数理・計算科学専攻, Department of Mathematical and Computing Sciences, Tokyo Institute of Technology.

†1 `http://git-scm.com`

†2 `http://mercurial.selenic.com`

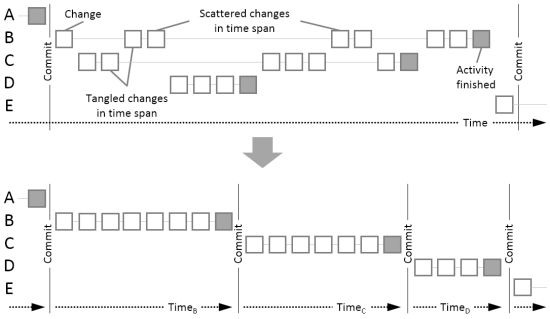

Fig. 1 Multiple activities (A to E) often interleave without notice (top) and best practices suggest to untangle and assemble changes before committing them (bottom). This is often challenging with existing tools.

ply forgot. Existing tool support typically poses high effort on following best practices, especially for fine-granular change models that track numerous changes even in short sessions [27]. Developers tend to explain all tangled and scattered changes from one session in a *single* commit message—if any. Consequently, understanding project histories will be impeded. Based on these observations, we address the following research question:

> How can we support developers to identify coherent and complete sets of fine-granular changes when committing code to a repository?

We define a change set as *coherent* if it contains only changes that origin from a single activity; it is *complete* if it contains all changes from that (completed) activity. We focus on fine-granular change models where every modification to the code base is tracked, even two consecutive changes that rescind each other. We think that semi-automatic tool support should favor *coherent sets over complete ones*, that is, avoid false-positives but accept false-negatives. Then, developers do only have to merge change sets to make them complete. In fact, we believe that there cannot be one fully automatic approach but the developer has to be part of the process to benefit from tacit project knowledge.

We propose a scriptable, interactive, graphical tool that supports developers to (1) identify consecutive changes that indicate continuity of an activity and (2) collect scattered runs of changes that belong to the same activity. Based on a lab study [27] including 5 programming sessions with up to 250 fine-granular changes each, we derived a foundation of scripts that can be combined to automatically reveal activity-specific change sets. When evaluated, a script assigns changes into *groups* and adds *recommendations*†3 if changes are already part of distinct groups. After this scriptable analysis, developers can explore all proposed groups, adjust them manually, or resolve recommendations. We implemented the graphical tool Thresher in Squeak/Smalltalk,†4 which visualizes intermediate results concisely and reduces the effort for manual adjustments. The underlying rules of identifying coherent and complete sets of changes can easily be accessed and modified.

In this paper, we make the following contributions:

- A design for an interactive, scriptable tool we call Thresher, which supports developers to assemble coherent and complete change sets for fine-granular source code changes
- A description of an exemplary scripting interface, which supports the Squeak/Smalltalk change model, to emphasize the simplicity of extending Thresher for domain-specific needs
- A brief description of scripts we extracted from a lab study to support Smalltalk-based refactoring sessions—including an evaluation in terms of precision (almost 100%) and recall (33% to 57%) of the proposed groups, which illustrates coherence over completeness

The next Section 2 summarizes related work. Section 3 describes existing challenges in state-of-the-art commit tool support. In Section 4, we present our tool Thresher and give an introduction to its scripting capabilities; we also present a foundation of scripts for developers to get started. We evaluate those scripts in Section 5 and discuss limitations and applicability to other environments. Section 6 sketches hypotheses and next steps for this kind of commit tool support. Finally, we conclude our thoughts in Section 7.

†3 Note that we use the term "recommendation" unrelated to recommendation systems in software engineering.

†4 The Squeak/Smalltalk programming environment: `www.squeak.org`

2 Related Work

We are not aware of projects that try to proactively involve developers in the process of committing changes. There are, however, many projects that investigate how to automate the identification of coherent change sets—only after commits have been taken place. Some projects derive rules by mining project histories, other projects create rules for changes that have detailed context information. Most of them deal with domain-specific characteristics, which emphasizes the need of adaptability and manual intervention when using in such tools.

When mining repositories, existing changes are often modeled and analyzed at a fixed level of detail. Herzig and Zeller ([13][14]) applied machine learning to adjust project-specific weights or thresholds for data dependencies, lexical distance measures, or test impact couplings. They analyzed coherent change sets that were already assigned to particular bug fixes. Kawrykow and Robillard [16] looked for behavior-preserving thus non-essential changes, which include rename-refactorings. They claim that such changes affect coherence negatively. Kim and Notkin [17] derived logical rules from change sets, which also identify anomalies and hence tangling changes such as "All methods X in classes that implement interface Y got deleted, *except class Z*." This should also help describe commit messages. — We derived a foundation of scripts from 5 programming sessions, which arguably reflect many important characteristics of object-oriented projects in Smalltalk. One can easily extend this approach with mining some Squeak/Smalltalk repositories to identify more corner cases. As scripts represent concise and accessible descriptions of a particular change model, we argue that developers can easily adapt those scripts to accommodate domain or project-specific needs.

When approaches are open to extensions in the change model, identification and classification methods can benefit from any kind of context information such as developer activity tracking. For example, Coman and Sillitti [6] tracked tool interactions and created the notion of *time intervals of intensive access* to help detect activity switches; Zou and Godfrey [35] verified this approach in an industrial setting. Robbes et al. ([23][24]) also track timestamps and use time-coupling to improve recall. Besides that, they track tool window usage, code authors, and annotations to indicate automated refactorings. If actually noticed, the developer can express the beginning of a new activity explicitly. Yoon et al. [33] log all low-level events of the Eclipse code editor in an XML format to be processed by other tools. — Thresher promotes fine-granular change tracking without making assumptions about the level of detail that is tracked. For example in our Squeak/Smalltalk environment, there were logs for tool-controlled refactoring activities, which later simplified script code. Having this, other extensions to the change model could further improve both scripts, results, and hence the overall utility.

Working with numerous fine-granular changes raises challenges for graphical tools from an information visualization perspective. Many approaches apply a timeline metaphor to provide overview and details on demand as pluggable visualizations in programming environments. Examples include Azurite [34] and CodeTimeline [20]. — We argue that common list, table, or tree views are capable of presenting huge amounts of data in a supportive way. Problems will arise primarily if developers have no simple means to configure the views' level of detail. Thresher employs those means by giving developers that required configuration support in terms of adaptable scripts with direct feedback after script modifications.

3 Challenges in Existing Tools

In this section, we highlight important factors that substantiate the need for better tool support when committing source code to a VCS. We cover the nature of interleaving programming activities, state-of-the-art tool support, and specifics of the Squeak/Smalltalk change model—as we built Thresher in that programming environment.

3.1 Interleaving Programming Activities

The more programming activities interleave during a session, the more important it is to untangle them to commit only coherent and complete change sets. The number of actually traceable activities will be influenced by the developer's task

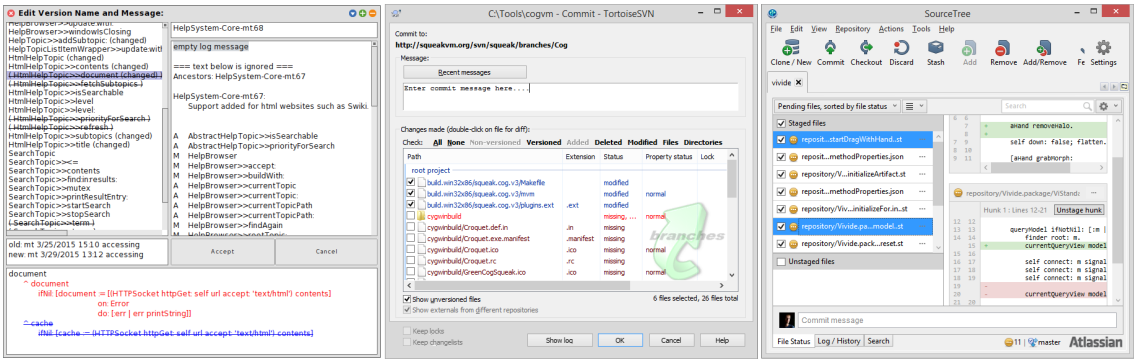

Fig. 2 Usually, commit tools show changes in long, scrollable lists, which developers have to untangle by tedious manual selection for each single commit. LTR: Monticello for Squeak (`http://www.squeak.org`), TortoiseSVN (`http://tortoisesvn.net`), SourceTree (`http://www.sourcetreeapp.com`)

and the level of change tracking editing tools provide. As for the task, a simple bugfix may spawn only one activity but "drive-by corrections" and other activities interleave quite often ([18][21])—those should be documented in separate commits. As for the change tracking, the more editing events get tracked, the higher the chance that a developer's activity drifts get documented for the long term—all the more if there are many modifications that rescind each other during a session.

File-based environments (such as Java + Eclipse + Git) compare the local working copy with the latest version in the VCS to determine changes at the level of text lines, which typically match program statements. We consider such an approach as *coarse-grained* because overwritten changes are not tracked as coherent program entities but only at the level of character ranges. Object-based environments (such as Smalltalk + Squeak + CoExist [26]) log any modification to the code base by means of method and class changes. This allows for rewinding or replaying single decisions and whole programming sessions. Such a change model is more *fine-grained* and persists more details of a project's history.

There is a programming approach called *explorative programming* [8], where developers are encouraged to make any change as they go without having a specific task in mind. They do not know upfront whether they will fix some bugs or add some neat features or clean-up some code. Such a *working mode* frees them from being too specific too early in highly uncertain scenarios like many research projects face. At the end of such a session, developers have numerous changes to untangle. We argue that there is a need for better commit tool support to even promote explorative programming and still ease comprehension of and learning from project histories.

3.2 Interactive Commit Tool Support

Figure 2 gives an impression of how existing commit tools make use of graphics and interaction. First and foremost, those tools present changes in scrollable lists with support for manual selection only. "Advanced" filtering covers presets such as *all*, *none*, and *invert selection*. It doesn't matter whether changes are file, method, or line-based: Such interfaces do not support untangling many changes very well. Furthermore, multiple activities have to be considered one after another.

However, we do consider lists or tables as appropriate views because they can visualize chronological order in a compact fashion. What existing tools miss is a way to filter and group changes according to some rules, which may not be expressed with a single button-click but rather simple scripts. Additionally when seeing a complete and coherent change set, developers may be better suited to provide a descriptive commit message.

3.3 The Squeak Change Model

Squeak has a direct notion of changes, change files, and change sets[†5]—but all in a local sense and not shared like common VCSs such as Git. Tracked changes include any source code modification affecting classes, methods, and their categories. In Squeak, change sets represent means to manually organize changes to easily file-out and share them, for example, via mailing lists. There is a tool called *Change Sorter*, which supports moving changes between sets in case the developer forgot to switch before starting her activity. Having this, Squeak provides a local history of everything that happens in the programming environment. However, not all changes are reversible per se but, at the least, they can be replayed.

The Smalltalk community has a VCS called *SqueakSource*[†6] and an in-image tool called *Monticello* as counterpart (left in Figure 2). When committing changes, Monticello only sees the latest versions of classes and methods. Cherry-picking is supported at the level of methods—not lines or ranges of text.

We make use of a recent research project called CoExist [26], which extends Squeak's change model by storing data to revert changes. It also adds the capability to Monticello to store fine-granular changes in SqueakSource such as two consecutive modifications of the same method. In that sense, CoExist elevates the fine-granular but local change tracking of Squeak to the level of Monticello/SqueakSource, which is now comparable to other VCSs.

Having this, we created Thresher for Squeak/Smalltalk but are arguably able to transfer our findings to other programming languages, tools, and environments.

3.4 The Vivide Tool Building Framework

We implemented our tool Thresher with the Vivide[†7] tool building framework [28]. Vivide provides a direct mapping between all graphical parts of the user interface and the internal tool logic. It is implemented in Squeak/Smalltalk and builds on top of the Morphic framework, which supports direct manipulation of all graphical objects. Having this, the developer can easily find responsible data transformation scripts starting from a visual impression and express modifications in the script source code. Due to this simple yet powerful abstraction, the framework can update all running tools consistently. Thus, Vivide is both a programming environment and a tool building framework. Building tools means composing widgets and writing script code.

The underlying concept of Vivide, that is describing tools as data processing pipelines, fits to our idea of configurable and combinable analysis stages for grouping changes. Developers open it after finishing a programming session to assemble recent changes into groups, add descriptive messages, and eventually commit them to the VCS with only a few clicks.

4 Thresher

In this section, we explain structure and use of our tool as well as details about how changes are processed with scripts. The name "thresher" derives from the fact that we want to identify all changes that belong to a certain activity and ignore the rest (for now)—just like the agricultural machine does with *winnowing*.

4.1 Changes, Groups, and Recommendations

The overall goal is to make developers commit coherent and complete change sets with a descriptive message. We classify a change set as *coherent* if it contains only changes that origin from a single activity; it is *complete* if it contains all changes from that (completed) activity. In Thresher, scripts support to detect activity continuations and thus grouping changes coherently. Completeness might often not be achieved on a scripting level because some developer knowledge might be hard to express or even ambiguous. Then, manual grouping via Thresher's graphical interface will take place.

There are multiple scripts that modularize the change analysis process in a sequence of stages. At the first stage, each change has its own group assuming that each change belongs to its own pro-

†5 Except for this section, we do not use the term *change set* in a Squeak-specific way but in a broader sense throughout this paper.

†6 `http://www.squeaksource.com`

†7 `https://github.com/hpi-swa/vivide`

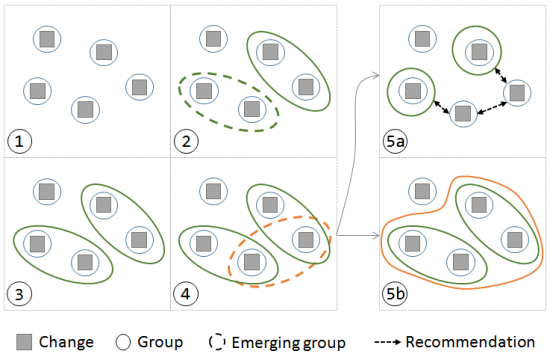

Fig. 3 Thresher works bottom-up: groups of changes will be merged (1-3) according to scriptable rules. There are no partial merges (4) but recommendations (5a) to be resolved manually by the developer.

gramming activity. Intermediate stages either expand those groups by merging them or propose recommendations to be resolved manually. In the last stage, each resulting group is meant to be committed to the VCS. The concept is illustrated in Figure 3.

In script code, programmers can iterate over all changes, check for specific properties, and assign matches to new groups. Those checks depend on the information available; a stage may also add new information to changes by integrating external data sources. For example, results of static or dynamic code analysis can be embedded this way.

4.2 The Graphical Interface

Thresher's graphical user interface is shown in Figure 4. It consists of four main views:

① **Sources View** shows a chronologically ordered list with all recent changes. The order can be adapted.

② **Result View** shows resulting groups and recommendations. Developers can resolve recommendations with button clicks, change groups via drag-and-drop, and add descriptive messages.

③ **Diff View** shows details about all selected groups to support developers to revise their decisions in detail.

④ **Stages View** reveals intermediate results of the involved scripts, supports adapting (add/remove/reorder) them as well as reviewing (or debugging) their effects.

There are several tasks that the developer accomplishes when working with Thresher. Those describe a seamless workflow and should encourage to follow best practices by committing only coherent, complete change sets as well as adding descriptive messages.

At first, the freshly opened Thresher window analyzes recent changes with the existing scripts to form groups and add recommendations. The source view ① lists all changes chronologically with a distinct label. The result view ② shows all resulting groups, which contain these changes and maybe several ungrouped[†8] ones. At the top of that view, *discarded changes* are listed such as recognized debugging code. All groups get monogram labels for easier recognition. At this point, the diff view ③ is not used because no group is selected.

After that, the developer reviews all group's contents to verify coherence and completeness. Selecting a group in the result view ② will reveal more details in the diff view and highlight the contents, which are automatically selected in the source view. Any group can be marked as *ready-to-commit* by clicking the checkbox near the group name. This will separate that group visually by moving it to the top as shown in Figure 4 for the group "bug fix: less than key."

Group C contains 6 changes and 4 recommendations, which are highlighted in blue and orange. To see the content of single changes, the developer can hover over one of them in the source view. To discard the changes, the developer selects them and drag-drops them onto the "Discarded Changes" group.

Recommendations can either be single changes or whole groups as proposed by previous stages. When hovering over a recommendation, the reason is shown in a tooltip (Figure 4, bottom right). The developer can accept recommendations by clicking the tick-icon next to it; the cross-icon rejects it.

After resolving all recommendations, the developer can merge groups if she discovers a more appropriate intent. This happens by simply drag-and-drop groups on top of each other. It is also possi-

†8 With "ungrouped" or "single changes" we mean single-change groups.

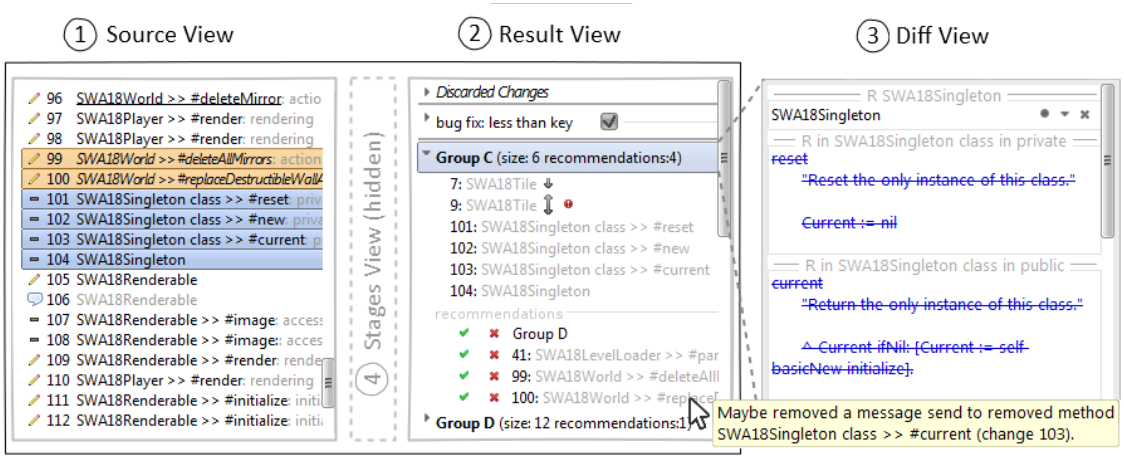

Fig. 4 The main views of Thresher: ① local, uncommitted changes, ② computed change sets with recommendations, and ③ detailed source code diffs.

ble to move selected changes (or groups) between groups.

Finally after reviewing a group, the developer can propose a descriptive name, which is displayed as the group's name to help identify complete ones. When she decides to commit, name and description are used as *commit message*. Besides manual naming, a short description of structural changes is appended automatically such as "added class `SWA18World`" and "modified method `Player >> render:`."

4.3 Scripting and Stages View

Developers can accommodate domain or project-specific needs by adding, removing, or reordering stages. Each stage has a current script, which developers can access, modify, and re-execute to update the results. Once visible, the *stages view* (Figure 5 resp. ④ in Figure 4) shows intermediate grouping results. Scripts access the change model and create groups with the help of a `grouper` like this:

```
[:grouper | |newGroup|
  newGroup := CustomGroup new.
  grouper allChanges do: [:change | "See below."].
  grouper addGroup: newGroup.
].
```

In our implementation, the scripting language is Smalltalk and scripts are just blocks evaluated with the `grouper` instance automatically. Each script is like an anonymous function that will be called with a grouper as argument. That function has no return value but can trigger side effects via the grouper's interface, that is, creating and adding groups of changes.

If there are already existing groups, developers can inspect their contents and consider them. At the beginning, there are as many groups as there are changes. For each match in a script, that change's current group is fetched and the nested structure of groups (Figure 3) is extended. The following snippet extends the example above:

```
"..." [:change |
  change property = 'SomeFilter' ifTrue: [
    group add: (
      (grouper groupOf: change) -> 'Reason␣for␣
      addition.'
)]]. "..."
```

To write effective scripts, developers need to know about possible change properties and the scripting language to write scripts. Here, filters have the conciseness and expressiveness of the Smalltalk language and are only limited by the environment's loaded libraries. In particular, external data sources may also be queried and their answers used in such expressions. For example, one might think of a library that supports connecting email accounts with source code and hence reveal further information about the original author of the changed piece of code.

Thresher's scripting capabilities can be summarized like this: **Scripts** are high-level descriptions

of change characteristics and create or merge appropriate *groups* while using a *grouper* for navigation. Different **group** classes represent distinct change characteristics with an optional *pivot change* for identification. Initially, each change is in a single-change group. Arbitrary changes can be put into a custom group. Groups are unit of reuse in *scripts*. A **grouper** supports navigating changes in the current group hierarchy and manages top-level *groups* across all *scripts*. It is not meant to be extended. **Recommendations** are created automatically if changes are associated with two distinct groups.

To present a more elaborate example, the following script analyzes changes for method additions/removals and combines them with all other method changes that update corresponding message sends:

```
[:grouper |
  grouper allChanges
    select: [:change | change
    isMethodAddOrRemove]
    thenDo: [:pivot | | group |
      group := MethodAddOrRemoveGroup new.
      group add: (
        (grouper groupOf: pivot) -> pivot name).
      grouper methodChanges do: [:change |
        "1) Check for removed methods."
        pivot isRemoval &
        (change isSendRemoved: pivot) ifTrue: [
          group add: (
            (grouper groupOf: change)
              -> ('Send removed:', pivot
              selector)].
        "2) Check for added methods."
        pivot isAddition &
        (change isSendAdded: pivot) ifTrue: [
          group add: (
            (grouper groupOf: change)
              -> ('Send added:', pivot
              selector)]].
      "Add new group to hierarchy."
      grouper addGroup: group]].
```

Here, *associations*, which attach a textual description as *reason* to each change, are used to fill groups with changes. Developers do not have to provide reasons but debugging unexpected results can benefit from such contextual explanations.

With only 19 lines of Smalltalk code, the developer can automate such an untangling of changes. If existing changes are ungrouped, the grouper will just create new groups. If they were already assigned to groups in preceding stages, Thresher will handle conflict resolution and may add recommendations because changes can only have a single group. Developers do not have to deal with conflict handling when writing scripts.

Scripts can also modify existing groups that were created by previous stages. By default, scripts do not modify previous decisions but *add, merge, or recommend* new changes from emerging groups. For example, two groups of the same kind such as `MethodAddOrRemoveGroup` are usually merged and not added as sub-groups. Eventually, top-level groups are important for the developer because those groups are meant to be committed.

4.4 A Foundation of Scripts

We implemented a set of complementary scripts that aim for detecting most characteristics as extracted from several programming sessions [27]. In most cases, they use structural/code change information but they may also consider chronological affinity. Having this, they encode the Smalltalk language characteristics besides Squeak's change model and hence serve as a valid baseline for other projects in the Squeak/Smalltalk environment. Due to space constraints we cannot print all scripts' details here but will only summarize their intents:

1. **Consecutive grouping** All consecutive changes on the same source code artifact[†9] are grouped.
2. **Refactoring grouping** All changes that were triggered by tool-driven refactorings are grouped. Depends in respective hints being present in the change data.
3. **Renaming grouping** Detects a manual or tool-driven renaming of a source code artifact and groups it with all consecutive changes of the same kind. Affects also renamed variables.
4. **Organizing grouping** All consecutive changes that reorganize software artifacts are grouped. This includes class categories and method protocols.
5. **Added/removed-method grouping** The addition or removal of a method is grouped with all changes that update the respective

†9 Source code artifacts can be packages, classes, or methods.

message sends in the source code. Applies static analysis to locate all sends.

6. **Added/removed-variable grouping** The addition or removal of an instance/class variable is grouped with all changes that update the respective references in the source code.
7. **Added/removed-class grouping** The addition or removal of a class is grouped with all changes that concern this class' methods, variables, and references.

The last three stages add recommendations:

8. **Close-to-artifact recommendation** Single changes are *recommended* to groups whose changes affect the same artifact. Considers chronological affinity.
9. **Close-to-polymorphism recommendation** Single changes are *recommended* to groups that concern method additions/removals but are ambiguous due to polymorphism. Considers chronological affinity.
10. **Pretty-print recommendation** Single changes of cosmetic nature are *recommended* to groups whose changes affect the same artifact if chronological affinity is high. If there is no such group, those changes are merged into a new group. Such changes include modification of whitespace or comments, renaming of temporal variables, or cascading message sends.

Having this, the scripts do not detect activity *switches* but rather describe *continuations*. From one stage to another, scripts merge existing groups of changes into bigger ones. Therefore, the order of scripts can influence the results. For now, Thresher should stop processing if further merging might decrease coherence of groups. Splitting up groups in scripts and hence describe activity switches would be a valuable addition to our concept and is considered future work.

5 Evaluation of Scripts

In this section, we evaluate the accuracy of the basic scripts in Thresher, which we derived from a programming study [27]. We then estimate the manual effort that remains to complete the mapping of all changes to their activities—by interacting with the tool's graphical interface.

We have not yet evaluated to overall utility of Thresher. *To some extent, we rely on the positive effects of the* Vivide *programming environment, in which* Thresher *runs. This, however, remains subject to further research.*

5.1 Analyzing Programming Sessions

Our proposed scripts are based on sample data, which Steinert et al. collected during a lab study [27] with 22 developers that had the task to improve source code of a game in 2-hour sessions. The number of changes range from 40 to 250 per session. We manually mapped those changes to activities, which included debugging, refactoring, and code clean-up. Having this, we derived several scripts that should automate this mapping process in terms of *coherent groups of consecutive changes* and which can be combined to *assemble scattered change runs*. Reconsider Figure 1 for the distinction of tangled and scattered changes.

In particular, we chose 5 out of the 22 sessions as representatives because their changes contained many refactorings or frequently interrupted activities with much tangling and many revisions; some of them contained at least one larger refactoring. We reviewed each fine-grained change *manually* and added it to coherent, complete sets to get kind of a *gold standard* for our analysis. Some developers created their own commits with Monticello, which we only partially took into account because at that time, it forced them to commit all changes at once. We did not interview the developers but were familiar with the domain of the game and its source code. Eventually, we identified recurring characteristics that indicate strong relationships or obvious activity switches as summarized in Table 1. We used the 6^{th} data set to "simulate" a programming session whose contents were not explicitly used to derive those scripts.

5.2 Method

We quantify the accuracy of Thresher's basic scripts as well as the manual effort that is necessary to get an ideal result.

For the accuracy, we compare the expected set of related changes with the calculated one. Let the expected groups be $A_E = \{3, 5, 6\}$ and $B_E = \{1, 2, 4\}$ where digits indicate a change's running number. Let the calculated groups be $A_C = \{3, 5\}$, $B_C = \{6\}$, and $C_C = \{1, 2, 4\}$. The expected set of rela-

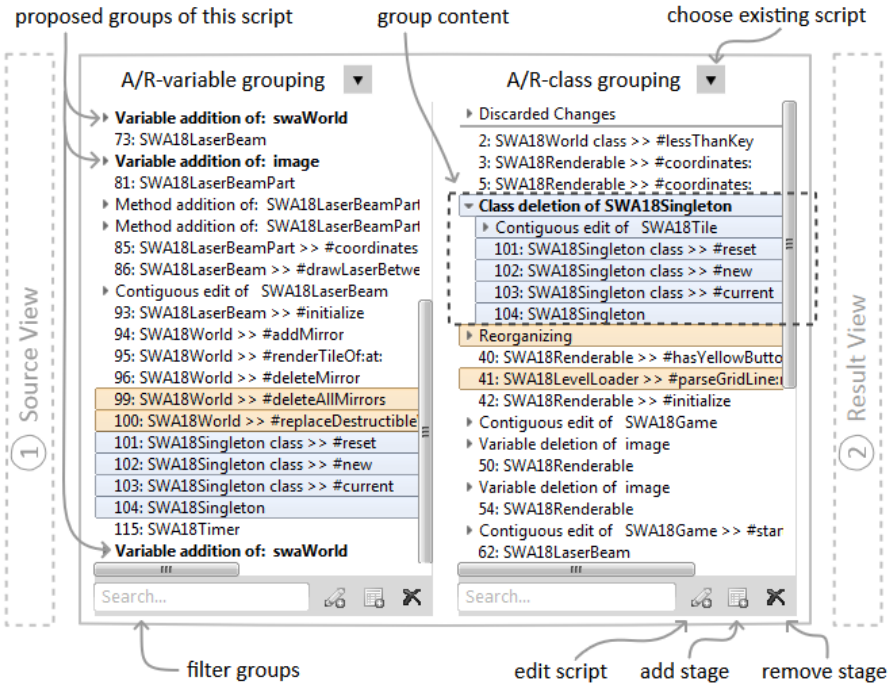

Fig. 5 The stages view in Thresher supports exploring intermediate results and adapting the scripts.

Characteristics	Data sets
Programming breaks between different refactorings (activity switch)	I II IV
Changed statements for debugging output or breakpoints (to be ignored)	I V
Consecutive changes in the same artifact	I II III IV V
Consecutive renamings of artifacts including updates of their uses in the code	IV
Consecutive reorganization of methods' protocols	I V VI
Changes of methods in different classes with shared protocol	I III V VI
Similar patterns in identifiers of artifacts	II III IV VI
Added/removed a method and update their uses	I II III IV V VI
Added/removed an instance/class variable and update their uses	I IV V VI
Added a class and initial methods/variables as well as initial uses	I VI
Removed a class and update their uses	I II III V VI

Table 1 Characteristics that rendered data sets interesting for deriving a foundation of scripts in Thresher.

tions would be

$$\mathbb{E} = \{\{1,2\},\{1,4\},\{2,4\},\{3,5\},\{3,6\},\{5,6\}\}$$

and the calculated one would be

$$\mathbb{C} = \{\{1,2\},\{1,4\},\{2,4\},\{3,5\}\}$$

by expanding the groups into pair sets. The *precision* is now defined as $\mathbb{P} = |\mathbb{E} \cap \mathbb{C}| \,/\, |\mathbb{C}|$ and *recall* as $\mathbb{R} = |\mathbb{E} \cap \mathbb{C}| \,/\, |\mathbb{E}|$. Both describe the accuracy of our scripts' results, respectively quality and quantity, as known from the domain of information retrieval [30]. In this example, precision is 1.0 (100%) and recall is 0.67 (67%), which indicates that *all* calculated groups are coherent but manual reviewing and merging is required for completion.

For the manual effort, we think of two extremes: developers either are directly able to assign changes to activities after reviewing them once (*min* =

$2 \cdot n_{changes}$), or they have to look through all changes for *each* activity again ($max = n_{activities} \cdot n_{changes}$). When using THRESHER, this effort is reduced because it depends on the number of proposed groups instead of single changes. The developer clicks to accept/reject single recommendations or all at once. Drag-and-drop interactions can merge groups or single changes.

5.3 Results

We got a precision of 96% to 100% for our five data sets as summarized in Table 2. Having this, we met our goal of avoiding false-positives in groups by favoring coherence over completeness in the scripts.

The upper part in the table shows information about the session size by means of *number of changes* and *number of expected groups*. The middle part shows the accuracy as described above. It includes the number of calculated groups that *exactly match* the expected commits and the number of recommendations that should help reduce manual effort. The lower part approximates the manual effort including the minimal amount of click/drag-and-drop *actions* in the user interface.

The recall indicates that 33% to 57% of the relations, which express activity continuation, were detected by our scripts. For the data set V, we simply adapted the script to increase recall; we expect developers to do the same when facing many small groups. Regarding the number of groups compared to the expected commits, an expected commit is roughly spread over two to three groups. Since some commits were completely detected (see "# Exact Matches"), other commits are spread over more than three groups.

We explicitly counted the number of manual actions; this number can be approximated as two times the number of expected commits because this corresponds to the distribution of commits over proposed groups. For some data sets this is even less, which means that the given recommendations help reduce manual effort. Notice that the developer could also accept or reject *all* recommendations for a particular group with a single click.

Programming session VI was analyzed to give a first impression of the quality and reusability of THRESHER's base scripts. We extracted patterns from sessions I to V, which had to be adapted to consider several corner cases in session V. A precision of 100% and a recall of 25% for session VI, however, indicate a satisfying result without further adaptation needed. The developer does only have to assemble the proposed groups with the help of recommendations and manual drag-and-drop interactions.

5.4 Discussion and Threats to Validity

After the manual analysis of all changes, we confirmed that THRESHER's scripts cannot run fully automated but developers have to intervene. On the one hand, several decisions rely on the developer's insights in the particular domain and project, which were *not materialized* in the change data. On the other hand, our scripts may not correctly reflect the characteristics found. Although we cross-checked several characteristics among us, there was not always consensus. For example, there were several method extractions across the code base and one of us argued to combine them into a *refactoring group* but another one would rather consider more *domain-specific information* and split them apart. We found similar conclusions in [6][10][25]. THRESHER is well suited to interactively help developers to decide which particular characteristics to pursue and when to merge proposed groups.

We derived the scripts and tool requirements from only 5 programming sessions. Although we found an arguably representative spectrum of change characteristics for the Squeak/Smalltalk programming environment, other sessions may contain more special cases. As different kinds of activities influence the number and structure of changes, we should analyze more sessions that do not mainly deal with refactorings and bug fixing but also with feature implementations like classified by Hattori et al. [11]. Those could reveal further characteristics [23][32] such as more additions and less modifications of methods.

The concept of recommendations accounts for scripts' interdependence; it helps developers adjust the proposed groups before committing them. However, misleading recommendations may negatively affect the coherence of a change set. THRESHER can miss to recommend the actually correct group; we have no means to prevent that.

Our scripts detect activity continuations rather than switches and thus related changes rather than

Data set	I	II	III	IV	V	V*	VI
Number of Changes	121	57	39	74	157	157	211
Number of Expected Groups	8	5	6	7	4	4	4
Accuracy							
Number of Calculated Groups	20	10	7	12	58	34	40
Number of Exact Matches	1	3	4	1	1	1	0
Number of Recommendations	20	4	2	21	32	31	22
Precision (%)	96	100	100	100	100	100	100
Recall (%)	35	43	57	35	1	33	25
Effort							
Minimum Interactions	242	114	78	148	314	422	
with THRESHER	40	20	14	24	116	48	80
Maximum Interactions	968	285	234	518	628	-	844
with THRESHER	160	50	42	84	232	96	160
Ratio (%)	17	18	18	16	37	15	19
Number of Manual Merge Actions	16	6	4	14	-	16	31

Table 2 Evaluation results comparing manual analysis (expected groups) with the scripts of THRESHER (calculated groups). [* Scripts adapted for V due to low recall]

checking for "unrelatedness"; which is undecidable [13]. This may be contrary to how developers would manually select coherent change sets from a chronologically ordered list of recent changes. Additionally, our scripts do not cope with *successive activities* but would merge them into a single one. For example, the developer may rename methods as a refactoring (A) and then use those to implement a new feature (B succeeding A).

We reduced the implementation and computation effort of the scripts by prepending a *static analysis* of the methods' source code to add useful information such as changed references, variables, and message sends. When implementing such a tool in a different environment, the benefits of such caching depends on the existing change model.

Our scripts only detect low-level relations between changes. Future work should include more abstract characteristics such as design patterns [9] as previously investigated by [15][19]. Dynamic analysis could also reveal interesting relations by, for example, collecting concrete type information [12] to be used in scripts.

Applicability to other object-oriented languages is straightforward because the concept of classes and methods is reflected in many of our scripts. Languages with fundamentally different concepts, such as logic programming, may reveal quite different characteristics. The scripting language may also differ from the one used in the programming environment, which was not the case in Squeak/Smalltalk.

6 Hypotheses and Future Work

Based on our experience with implementing THRESHER in Squeak/Smalltalk and providing a foundation of scripts based on the lab study [27], we derive the following hypotheses:

- Given a foundation of scripts that considers a particular programming language and its change model, developers will rarely have to adapt those scripts except for project-specific needs.
- If developers are knowledgeable about the scripting capabilities in such a commit tool, they are more likely to improve their personal workflow by taking advantage of that.
- If developers adapt such scripts for project-specific needs, they will make fewer mistakes and save time during the commit process.

Besides running experiments to test those hypotheses, we are eager to improve the general idea of how fine-granular changes might be combined into coherent and complete groups. In addition to describing on the *continuity of activities* between changes, we think that it is important to also split up bigger groups to indicate *activity switches* and thus separate commits. Finally, expressing rules of *dependency relationships* in scripts is valuable so

that the correct commit order of change sets can be ensured, too.

Furthermore, we want to try out visualizations based on structures other than lists, tables, or trees. For example, recommendations across group boundaries might be better displayed with a graph similar to Figure 3. Finally, we have to generalize our scripting approach to other programming systems and their change models. File-based models, for example, may reveal different traits than object-based ones.

For all these improvements, we see a need for data from programming sessions that go beyond simple refactoring tasks such as adding features or fixing bugs. Only then, we can establish a useful scripting interface to support programmers to accommodate project-specific requirements.

7 Conclusion

We presented Thresher, a scriptable, interactive, graphical tool that developers can use after programming sessions to identify coherent and complete sets of fine-granular changes to be committed to a VCS. A foundation of scripts automatically detects related changes that indicate activity continuations; developers then manually assemble those scattered groups into complete activity descriptions. We argue that it is very important to involve the actual developer into this process because there is much tacit knowledge about the project and its domain that cannot be manifested in code or change structures.

We showed that our foundation of scripts propose groups with a precision of almost 100% indicating that developers can rely on their coherence and focus on adjusting their completeness with respect to an activity. We argue that such tool support promotes the usage of fine-granular changes, which is beneficial for explorative programming strategies—omitting explicit checkpoints—and detailed tracking of project histories.

Acknowledgments

We wish to thank Jens Lincke, Lena Herscheid, Lysann Schlegel, Mark Rooney, Marko Röder, and Philipp Tessenow for fruitful discussions and valuable feedback. We gratefully acknowledge the financial support of HPI's Research School[†10] and the Hasso Plattner Design Thinking Research Program.[†11]

References

[1] M. Barnett, C. Bird, J. Brunet, and S. K. Lahiri. Helping developers help themselves: Automatic decomposition of code review changesets. *Proceeding of the 37rd International Conference on Software Engineering (ICSE)*, ACM/IEEE, 2015.

[2] B. W. Boehm. The high cost of software. *Practical Strategies for Developing Large Software Systems*, pages 3–15, 1975.

[3] S. A. Bohner and R. S. Arnold. *Software Change Impact Analysis*. John Wiley & Sons, 1996.

[4] M. Borg and P. Runeson. Changes, evolution, and bugs: Recommendation systems for issue management. In *Recommendation systems in software engineering*, pages 477–509. Springer, 2014.

[5] R. P. L. Buse and W. R. Weimer. Automatically documenting program changes. In *Proceedings of the 25th International Conference on Automated Software Engineering (ASE)*, pages 33–42. IEEE/ACM, 2010.

[6] I. D. Coman and A. Sillitti. Automated identification of tasks in development sessions. In *Proceedings of the 16th IEEE International Conference on Program Comprehension (ICPC)*, pages 212–217. IEEE, 2008.

[7] P. M. Duvall, S. Matyas, and A. Glover. *Continuous integration: Improving software quality and reducing risk*. Pearson Education, 2007.

[8] R. P. Gabriel. I Throw Itching Powder at Tulips. In *Proceedings of Onward! Essays*. ACM, 2014 (to be published).

[9] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design patterns: Elements of reusable object-oriented software*. Pearson Education, 1994.

[10] N. Gold and A. Mohan. A framework for understanding conceptual changes in evolving source code. In *Proceedings of the 19th International Conference on Software Maintenance (ICSM)*, pages 431–439. IEEE, 2003.

[11] L. P. Hattori and M. Lanza. On the nature of commits. *Proceedings of the 23rd International Conference on Automated Software Engineering Workshops*, pages 63–71, 2008.

[12] M. Haupt, M. Perscheid, and R. Hirschfeld. Type harvesting: A practical approach to obtaining typing information in dynamic programming languages. In *Proceedings of the 2011 Symposium on Applied Computing (SAC)*, pages 1282–1289. ACM, 2011.

†10 `www.hpi.uni-potsdam.de/research_school`

†11 `www.hpi.de/en/research/design-thinking-research-program`

[13] K. Herzig and A. Zeller. Untangling changes. *Unpublished manuscript, September*, 2011. `http://www.st.cs.uni-saarland.de/publications/files/herzig-tmp-2011.pdf`, accessed: July 1, 2014.

[14] K. Herzig and A. Zeller. The impact of tangled code changes. In *Proceedings of the 10th International Workshop on Mining Software Repositories (MSR)*, pages 121–130. IEEE, 2013.

[15] D. Heuzeroth, T. Holl, G. Hogstrom, and W. Lowe. Automatic design pattern detection. In *Proceedings of the 11th International Workshop on Program Comprehension (IWPC)*, pages 94–103. IEEE, 2003.

[16] D. Kawrykow and M. P. Robillard. Non-essential changes in version histories. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE)*, page 351, ACM/IEEE, 2011.

[17] M. Kim and D. Notkin. Discovering and representing systematic code changes. In *Proceedings of the 31st International Conference on Software Engineering (ICSE)*, pages 309–319. IEEE, 2009.

[18] A. J. Ko, R. DeLine, and G. Venolia. Information Needs in Collocated Software Development Teams. In *Proceedings of the 29th International Conference on Software Engineering (ICSE)*, pages 344–353. ACM/IEEE, 2007.

[19] C. Kramer and L. Prechelt. Design recovery by automated search for structural design patterns in object-oriented software. In *Proceedings of the 3rd Working Conference on Reverse Engineering (WCRE)*, pages 208–215. IEEE, 1996.

[20] A. Kuhn and M. Stocker. CodeTimeline: Storytelling with Versioning Data. In *International Conference on Software Engineering (ICSE)*, pages 1333–1336. IEEE/ACM, 2012.

[21] A. Meyer, T. Fritz, G. C. Murphy, and T. Zimmermann. Software DevelopersâĂŹ Perceptions of Productivity. *SIGSOFT FSE, ACM*, 2014.

[22] R. Robbes and M. Lanza. A change-based approach to software evolution. *Electronic Notes in Theoretical Computer Science*, 166:93–109, 2007.

[23] R. Robbes and M. Lanza. Characterizing and understanding development sessions. In *Proceedings of the 15th International Conference on Program Comprehension (ICPC)*. IEEE, 2007.

[24] R. Robbes, D. Pollet, and M. Lanza. Logical coupling based on fine-grained change information. In *Proceedings of the 15th Working Conference on Reverse Engineering (WCRE)*, pages 42–46. IEEE, 2008.

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

[26] B. Steinert, D. Cassou, and R. Hirschfeld. CoExist: Overcoming aversion to change - Preserving immediate access to source code and run-time information of previous development states. In *Proceedings of the 8th Symposium on Dynamic languages (DLS)*, pages 107–118. ACM, 2012.

[27] B. Steinert and R. Hirschfeld. How to compare performance in program design activities: Towards an empirical evaluation of CoExist. In *Design Thinking Research: Understanding Innovation*, pages 219–238. Springer, 2014.

[28] M. Taeumel, M. Perscheid, B. Steinert, J. Lincke, and R. Hirschfeld. Interleaving of Modification and Use in Data-driven Tool Development. In *Proceedings of the ACM Symposium for New Ideas, New Paradigms, and Reflections on Everything to do with Programming and Software (Onward!)*, pages 185–200. ACM, 2014.

[29] Y. Tao, Y. Dang, T. Xie, D. Zhang, and S. Kim. How do software engineers understand code changes?: An exploratory study in industry. In *Proceedings of the ACM SIGSOFT 20th International Symposium on the Foundations of Software Engineering*, page 51. ACM, 2012.

[30] C. J. van Rijsbergen. *Information retrieval (second edition)*. Butterworths, 1979.

[31] A. von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer (IEEE)*, 28(8):44–55, 1995.

[32] B. J. Williams and J. C. Carver. Characterizing software architecture changes: A systematic review. *Information and Software Technology*, 52(1):31–51, 2010.

[33] Y. S. Yoon and B. A. Myers. Capturing and Analyzing Low-Level Events from the Code Editor. In *International Workshop on Evaluation and Usability of Programming Languages and Tools (PLATEAU)*, pages 25–30, 2011.

[34] Y. S. Yoon, B. A. Myers, and S. Koo. Visualization of Fine-grained Code Change History. In *Proceedings of the 2013 IEEE Symposium on Visual Languages and Human-centric Computing (VL/HCC)*, pages 119–126. IEEE, 2013.

[35] L. Zou and M. W. Godfrey. An industrial case study of Coman's automated task detection algorithm: What worked, what didn't, and why. In *Proceedings of the 28th International Conference on Software Maintenance (ICSM)*, pages 6–14. IEEE, 2012.