

U Can Touch This: Touchifying an IDE

Benjamin Biegel, Julien Hoffmann, Artur Lipinski, Stephan Diehl
University of Trier, Germany
{biegel,diehl}@uni-trier.de

ABSTRACT

Touch gestures are not only often very intuitive, but their direct manipulation characteristics also help to reduce the cognitive load. Since software development poses complex cognitive demands, our goal is to exploit the advantages of direct manipulation to support professional software engineering processes. In this paper, we demonstrate how touch gestures can be used within a professional integrated development environment. As for that, we have enriched the Eclipse IDE with common and invertible multi-touch gestures which can be used for both controlling the graphical user interface and triggering built-in refactoring tools. The design of our extensions was informed by an early user study revealing problems of using the Eclipse IDE with the default touch support provided by the operating system. By using the emerging prototype during its implementation, we were able to iteratively improve the prototype based on our own experience and gain first insights into the potential of using direct manipulation methods within the IDE. First results suggest that using an additional touch device within the classical desktop setup enables a precise and fast work flow.

Categories and Subject Descriptors

D.2.6 [Software Engineering]: Programming Environments;
H.5.2. [User Interfaces]: Interaction styles

General Terms

Human Factors, Design

Keywords

Touch gestures, refactoring, IDE, radial menus, usability

1. INTRODUCTION

As touch devices are frequently used in our everyday life, common touch gestures are widely known and a basal vocabulary for direct manipulation has evolved, which is shared

across operating systems, applications and touch devices. Developers spend more of their time reading and navigating code than writing it [8]. As for that, the classical desktop setup, consisting of a keyboard and a mouse, holds several inconveniences for developers. As described by Shneiderman and Plaisant [17], the mouse only provides an indirect interaction, thus, the coordination of the eyes, the hand and the elements on the screen ends up in a high cognitive load. In contrast, (multi-)touch gestures offer a direct user interaction without losing the focus on the screen. Hence, it is not surprising that there are first efforts in software engineering research to explore and leverage this direct manipulation technology (cp. Section 5). Nevertheless, touch interactions are only rarely used in software development, especially in IDEs. Modern IDEs are feature-rich, making it more and more difficult for the developer to navigate through the software project, to reorder views, or to use functionality that is hidden in complex menu structures or shortcuts. That is why touch devices could be well-suited for IDEs, especially for reading tasks. Thus, in this paper we present the following two main contributions in order to touchify an IDE.

Contribution 1: Touchifying the GUI of an IDE.

We assume that developers prefer using their well-known programming environment. Thus, despite creating a completely new graphical user interface, which is tailored for touch interactions, we decided to follow a different strategy: We use a common IDE as a starting point and extend it with touch gestures step-by-step. In order to lower the barrier for developers to make use of a touchified IDE, we adopt the following main design goals:

- **Keep the bento box design:** The original user interface should be modified as little as possible. In most IDEs the user interface is split into several views. The challenge is to keep this so called bento box design [3] unaffected while adding new direct manipulation interactions.
- **Maintain existing user interactions:** The developer should still have access to all the functions of the IDE, including the familiar keyboard and mouse interactions. That is why the touch gestures have to be applied as an additional layer without disturbing the existing interaction opportunities.
- **Meet developer's expectations:** By using specific touch gestures, it is very likely that developers will have a clear expectation of the behavior of the user interface. Hence, the challenge is not only to find new intuitive gestures but also to adapt the user interface as expected.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

CHASE '14, June 2 – June 3, 2014, Hyderabad, India

Copyright 2014 ACM 978-1-4503-2860-9/14/06 ...\$15.00.

This first contribution mainly affects elements of the graphical user interface but not the source code editor itself. Hence, with our second contribution we also make first steps in using touch gestures for manipulating source code on the example of refactorings. Refactoring is an essential software development process in order to maintain the structure and the design of a software system [6]. Since refactorings are performed frequently by developers, many IDEs provide tools to apply structural changes (semi-)automatically to a code base without changing its behavior. Nevertheless, most of these refactoring tools are rarely used by developers and most refactorings are performed manually instead [14]. Previous research has shown that the use of refactoring tools comes with two main problems [13]: First, the application and context menus in conventional IDEs are too long and confusing. This turns the search for a specific refactoring into a time-consuming and annoying task. Second, before performing a specific refactoring, developers have to recall the mapped shortcut or, in order to find the corresponding menu entry, its name. Thus, for a novice it is a tedious task to become a heavy user of refactoring tools.

We claim that the use of touch gestures facilitates restructuring code. This task also requires reading, navigating, and in particular, pointing to a source code element to trigger the automated refactoring. Thus, In this paper we also introduce a new approach that adopts context-sensitive direct user interactions for performing refactorings.

Contribution 2: Mapping Common and Invertible Multi-Touch Gestures to Refactoring Tools.

We have enriched the Eclipse IDE with intuitive and common multi-touch gestures that trigger built-in refactoring tools. In order to encourage developers to use our proposed gesture-to-refactoring mapping, we adopted the following design goals in addition to the ones previously mentioned:

- **Map multiple refactorings to a single gesture:** The developer should be able to create a natural association from the desired refactorings to the required gestures. Thus, similar refactorings have to be mapped to a single gesture, and further, the selection of the desired refactoring depends on the particular context.
- **Use common touch gestures:** To provide a quick and easy start, only common touch gestures should be used that are mainly known from smartphones or tablets. We assume that most developers are familiar with these gestures.

Overall, in our vision, the developer is mainly using a touch screen and keyboard setup. We claim that the developer is capable of performing most mouse interactions with a touch-screen display or on a tablet computer. Additionally, the touch gestures enable a wider interaction vocabulary for intuitive direct manipulation, which may help reducing the cognitive load during the software development process.

2. STATE-OF-THE-ART INTERACTION

Since one cannot simply assume that all problems that have been identified in general HCI studies, also occur within a more restricted application scenario, we based our guidelines on observations made in a small user study. Thus, we were first of all interested in the touch features that come

with the operating system (here: Microsoft Windows 7) by default and how they work in the Eclipse IDE. The operating system basically turns touch gestures into mouse gestures, e.g., taps are simply turned into mouse clicks.

For a preliminary user study we asked two right-handed graduate students who are experienced developers with eclipse and make heavy use of mobile touch devices. Within a 45-minute think-aloud session they had to solve small programming exercises by using a keyboard, a touch-screen display, and the original implementation of Eclipse IDE. No mouse was used. The results of this study were used to determine the following guidelines (**G**) for better touch support in the Eclipse IDE.

Small Elements. The height of the elements in menus and trees depends on the text or image they contain. Thus, in most cases the height is too small to hit these elements with the finger tip. This issue is known as the *fat finger problem* [19]. In order to manage multiple source code files at once, tabs are used in the editor view to switch between the files. While the size of these tabs is sufficient, the symbols for closing the tabs are too small. The same holds for the symbols for minimizing, maximizing, and restoring the Eclipse window and the icons in the toolbar.

G1: *Some components can be optimized for touch just by enlarging them.*

Reordering and Resizing Views. Sashes are optical separators for adjacent views. The size of these adjacent views can be changed by dragging these sashes. Since a sash only has the width of a few pixels, it is quite hard to hit it with the mouse; by only touching the sash, it is nearly impossible to hit it. Enlarging the sash is not a solution, because it is only a separator and should not waste too much space.

G2: *In order to keep the visual aspects of a sash, for touch interaction, a new visual interface element has to be developed in order to also adapt its resizing functionality.*

Occlusion. The occlusion of content by user's hand is a serious problem[2, 10]. In order to read the occluded content, the user is forced to move the hand aside, which leads to a disruption of a fluent interaction. In particular, when using menus that open in the direction of the dominant hand (e.g. bottom right for right-handed users), the occlusion was noticed in a negative manner by the test persons. Another often occluded component is the status bar, which is typically placed at the bottom of the IDE.

G3: *New visual interface elements have to be developed for the main and context menu.*

G4: *Important read-only content has to be moved to areas with less frequent user interaction.*

Source Code Transformations. The selection of several lines of code is possible in a reasonable way. However, in order to select single words, our test persons had to have a lot of patience. Since source code elements are represented as a sequence of characters and not as hierarchical nested objects, it is necessary to select every single character. The precision of the interaction must be on such a level that a rectangle with a width and height of one single character can be hit. Thus, it is very hard to place the cursor exactly beneath a particular character. Furthermore, when select-

ing a certain source code fragment, it is also very difficult to stop on a target, because the finger occludes some characters. In particular these issues, combined with the use of complex menu structures, make it hard to trigger automated refactoring tools.

G5: *For source code selection and transformation, new interaction techniques have to be developed.*

Ergonomic Problems. Both participants felt uncomfortable moving their right (dominant) hand to the left side, especially, to the upper left corner. This is due to the fact that they had to overstretch their arm by rotating their upper body. In these situations they used their left hand instead, which results in slower and less precise interactions. Thus, one of the participants used the middle finger as a guide at the edge of the screen. As already described in literature [12, ?], interacting with arms in the air is very exhausting and sometimes inaccurate. That is why from time to time both participants laid their elbows on the table.

G6: *Heavily used controls have to be on the bottom and on user’s dominant side.*

3. DESIGN AND IMPLEMENTATION

Based on the results of the above user study, we were able to elaborate concepts with the aim of reducing the occurring problems. Our solutions are threefold: First, concepts that can be realized just by adjusting existing components. Second, concepts that require the introduction of additional components with a novel appearance and interaction design. Third, as an example for interacting directly with the source code (**G5**), a concept that provides a mapping from touch gestures to refactorings. Further concepts for selecting or transforming source code by touch gestures are left for future work.

3.1 Adjusting Existing GUI Elements

In a first step we tried to touchify Eclipse by reconfiguring and modifying existing components.

Adjusting Component Sizes. The size is a major problem for most components to hit them with a finger. Thus, a simple idea is to resize specific components. But in practice it is not that easy. Some components would waste too much space after resizing (e.g. sashes) or could not be displayed completely (e.g. the main menu). Thus, we carefully selected a small set of components for resizing. This includes tree structures (e.g. the package explorer), lists, tables, tabs, icons, and the toolbar.

Ergonomic Optimizations. In the default layout of Eclipse the active components (toolbar and perspective switcher) are at the top and the passive components, which display information with less or even without interaction (status line, heap status and progress bar), are at the bottom. However, when using a multi-touch screen, the inverted ordering is preferable. Thus, we swapped the upper and the lower part. An overview of the new order is shown in Figure 1.

This new ordering enables shorter paths for hand movements and allows that the user can place the elbow on the desk. Furthermore, during the interaction, read-only components stay visible at the top and are not occluded by the hand. In the default Java perspective of Eclipse, the editor

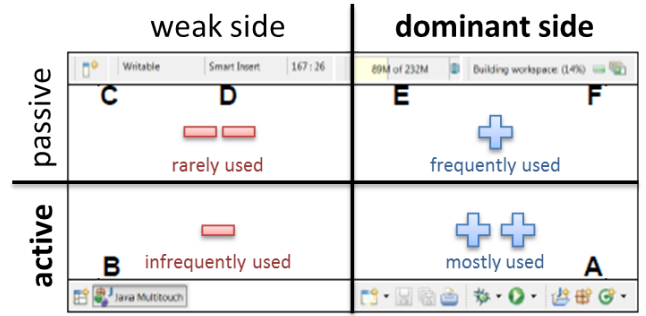


Figure 1: Reordered components: toolbar (A), perspective switcher (B), fast view (C), status line (D), heap status (E), and progress bar (F).

is placed in the center, the package explorer can be found on the left side, the outline view on the right, and the console view is placed under the editor. We reordered the editor and its surrounding views based on the interaction frequency of our test users also shown in Figure 1. Thus, the most frequently used view, the editor, is placed at the user’s dominant side (right half), navigation views (package explorer and outline view) are placed at the bottom left, and the console view can be found in the upper left part. The final ordering can be seen in Figure 3. Finally, we changed the opening behavior of tree elements such that subtrees can be unfold by a single tap instead of a double-tap. This should enable a more precise user interaction.

3.2 Implementing New GUI Elements

Some functionality has to be replaced by new components with completely different interaction techniques and appearance. Our prototype provides the following extensions:

Radial Main Menu. The main menu in the Eclipse IDE has a lot of items, and thus, resizing the menu items would not work in practice. The menu would become too large and parts of it could not be displayed on the screen. Thus, our goal was to find a solution that makes the menu sufficiently large without removing any menu item. To achieve this we use a radial menu structure [4, 16]. As shown in Figure 2(a) the radial layout of the main menu covers a larger area of the screen, and thus, allows to use a larger design for the menu items. In our implementation the position is fixed to the center of the screen and can be triggered by a button on the top. Furthermore, this layout can easily be optimized for right-handed as well as for left-handed users just by changing the starting point and angle. To reduce occlusion effects the lower quarter at the user’s dominant side is omitted. The menu items are arranged clockwise starting at this gap. Submenus are also placed radially in outer circles. To ensure readability none of the menu captions has an angle greater than 90° and all menu items have a uniform size.

Similar to our radial approach are hierarchical marking menus, e.g. flower menus that were introduced by Bailly et al. [1]. They note that an advantage of radial layouts lies in the human declarative memory. Because of the spatial arrangement of the menu items, the user is able to memorize their position just by performing the touch gesture.

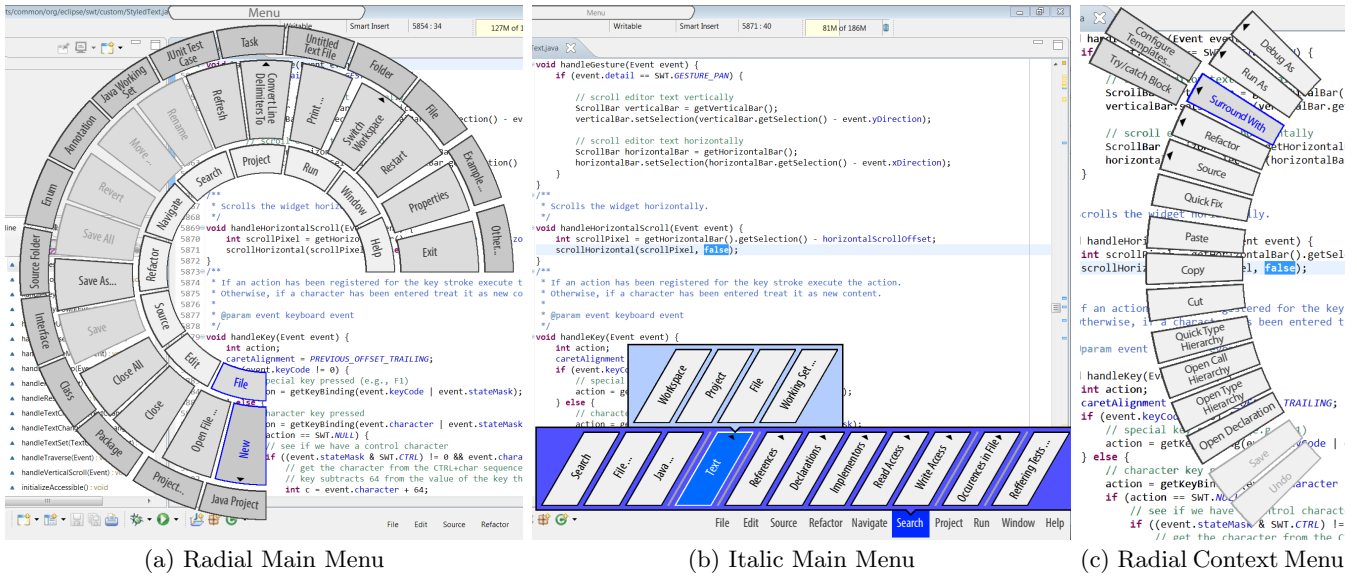


Figure 2: Three touch-optimized menus.

Italic Main Menu. A disadvantage of the radial main menu is that the user has to tap at a button to display the menu. In the default implementation the top-level menu entries are always listed side by side and can be immediately triggered by the user. An alternative solution to the radial main menu that addresses the above concerns is a linear menu with oblique captions—called *italic main menu*. The implementation of this menu can be found in Figure 2(b). The goal of this menu is a seamless integration into the IDE similar to the default Eclipse menu. In contrast to the default Eclipse menu, the entries in the italic main menu are arranged from left to right. The concept exploits that in general the screen has a greater width than height. The italic main menu is easy to reach because it is placed at the bottom of the screen. The levels of the submenus are arranged from bottom to top, hence, they are not occluded by user’s hand.

Radial Context Menu. In general, it is no problem that the main menu gets the full attention of the user and may overlay large parts of the screen. In contrary, by using a context menu users have to keep their focus to the related context. By designing the context menu, we would like to get the advantages of the human declarative memory and, at the same time, we have to consider the related context. We also decided to use a radial layout, but as shown in Figure 2(c) only a quarter of a circle. At the screen borders the context menu will be rotated. Thus, at any place it can be displayed within the borders of the screen, also at its corners. The radial layout helps the user to keep the focus because the menu is placed around it. In other words, the focus or context also represents the center of the circle.

Sash Overlays. A sash has a clear visual functionality, thus, changing the appearance is not a solution at all. In order to keep the appearance we introduce an additional, overlaid component—called *sash overlay*. It represents visual handle for manipulating the position of an underlying sash and the size of its adjacent views respectively. Figure 3

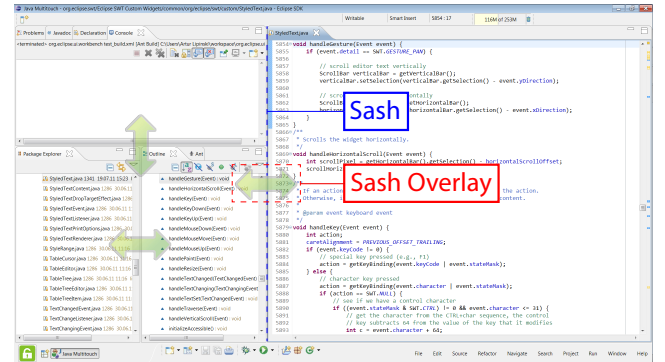


Figure 3: Sash overlays (shown as double arrows) for manipulating the position of the underlying sashes.









shows such sash overlays that enable to resize the editor and its surrounding views. A sash overlay is large enough for touch interaction and can be triggered in two ways: First, by clicking on a lock symbol in the bottom left corner, or second, by just swiping over the related sash.


3.3 Mapping Touch Gestures to Refactorings


We selected a vocabulary of multi-touch gestures which is as small as possible. In the following we list their redefined semantics in respect to the refactoring context:


Pinch open: Usually this gesture is used to zoom into a document. In contrast, we assign the metaphor of *extending the scope* of selected source code entities. For example, after selecting a bunch of statements, this gesture triggers the Extract Method refactoring. Hence, the scope of these statements is extended from within a single method to at least all methods in the enclosing class.


Table 1: Interaction concept for triggering refactoring tools with common touch gestures.

Gesture	Context	Refactoring	Inverse Refactoring	Context	Gesture
	statements	Extract Method	Inline Method	method call/declar.	
	expression	Extract Local Variable	Inline Local Variable	local variable use	
	methods, fields, nested classes	Extract Class	<i>undo</i>	—	
	—	<i>undo</i>	Introduce Parameter Object	parameters	
	local variable declaration	Convert Local Var. to Field	<i>undo</i>	—	
	anonymous class declaration	Conv. Anon. Class to Nested	<i>undo</i>	—	
	class declaration	Move Type to New File	<i>undo</i>	—	
	identifier	Rename	Rename	identifier	
	modifier	Decrease Visibility	Increase Visibility	modifier	
	method parameter	Reorder Method Parameter	Reorder Method Parameter	method parameter	
	method declaration	Reorder Method	Reorder Method	method declaration	
	field declaration	Reorder Field	Reorder Field	field declaration	

 **Pinch close:** As one might expect, the inverse counterpart of the pinch open gesture *narrows the scope* of a selected entity. In respect to the example above, this gesture triggers the Inline Method refactoring.

 **Rotate left/right:** Since the rotate gesture is generally used to rotate elements, it does not make much sense in an IDE. Hence, we redefine this gesture to *reorder adjacent entities*. For example, with the rotate left gesture a selected method parameter can be moved continuously to the beginning of a parameter list.

 **Flick left/right:** Inspired by the gesture to strike-out text that has to be replaced, we use the flick gesture to *change the identity* of entities. For example, for a selected identifier this gesture triggers the rename refactoring. If an entity has an enumerative character, we exploit the directional nature of the flick gesture to switch between the predecessor (flick left) and the successor (flick right) of that entity. For example, in order to change the visibility from public to private, the user can simply use flick left.

 **Pan:** Since refactoring tasks involves a lot of reading, we use the pan gesture not to trigger refactorings but to easily scroll the source code.

In order to generate a consistent mapping which takes conceptional similarities of refactorings into account, we converted the three mapping rules (**MR**) of Murphy-Hill et al. [13] from their directional use to multi-touch gestures:

MR1: The nature of a gesture has to be reflected in the structural nature of the refactoring.

MR2: Every gesture should have an inverse counterpart that triggers the inverse refactoring.

MR3: Conceptual similar refactorings have to be linked to a unique gesture.

The mapping derived from these rules is presented in Table 1. As can be seen, some refactorings have no inverse counterpart. Thus, after executing a non-invertible refactoring, the inverse gesture is assigned to the *undo command* instead, until the user changes the context.

As proof of concept, we implemented the proposed multi-touch gestures, excluding those only triggering non-invertible refactorings, namely: Rename, Extract/ Inline Method, Extract/ Inline Local Variable, Increase/Decrease Visibility, Reorder Method Parameter, Reorder Method.

Since the text selection with touch gestures is still inconvenient, we decided only to make minimal use of it. For most gestures it suffices to tap on the entity which should be refactored. Only refactorings which involve multiple entities require a proper selection. After the selection, gestures can be executed anywhere on the screen to prevent the hand from occluding the context of the source code. Then refactorings are performed immediately without interrupting the work flow by configuration dialogs or pop-ups. For every gesture Figure 4 gives representative examples demonstrating how a refactoring and its inverse counterpart can be performed in our prototype implementation.

4. EVALUATION

In order to gain first insights into the suitability and applicability of the proposed concepts, we performed two independent think-aloud user studies. The focus of the first study was on the modified GUI, whereas in the second study we have investigated the gesture-to-refactoring mapping.

4.1 Study 1: Adapted and New GUI Elements

For the first user study, we recruited 8 right-handed developers (5 master/diploma students and 3 academic researchers) to participate. First, they had to fill in a questionnaire to assess their prior knowledge and skills. One half of the participants stated to have “advanced” and the other half stated to have “good” programming skills. Especially 3 declared to have “advanced”, 4 “good” and 1 “minor” experience with the Eclipse IDE. Finally, concerning the experience in using several touch devices we received the following answers: (a) Smartphones: 3 “advanced”, 4 “good” and 1 “minor” experience; (b) Tablets: 3 “advanced”, 1 “good”, 3 “minor” and 1 “no” experience; (c) Touch monitor: 7 “minor” and 1 “no” experience.

In summary, the participants were quite familiar with smartphones and tablets but they are unversed in using a touch monitor.

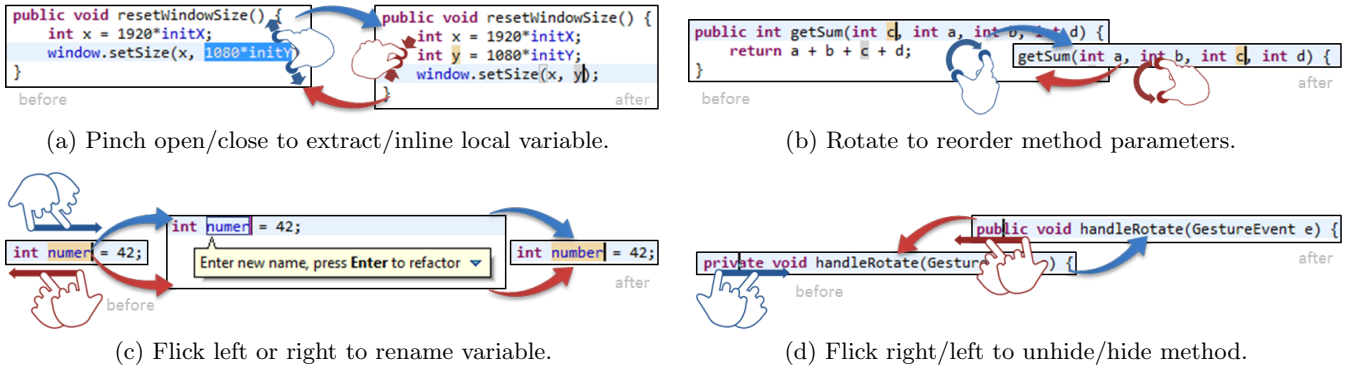


Figure 4: Examples of implemented gestures to perform refactorings (blue) and their inversions (red).

4.1.1 Experimental Design

For each person we performed a 45-minute think-aloud session (with video record) in which the participants had to solve small usability exercises by only using a keyboard and a touch monitor instead of a mouse. The experiment contained the following parts: (1a) Testing graphical elements of the original Eclipse implementation and (1b) of our adapted prototype, (2) solving a small programming exercise by using preferred GUI features, and (3) a concluding interview. For each original, adapted, and novel graphical element the participants were asked to rate four properties: (i) the *behavior* is as expected, (ii) the *appearance* is clear and well-arranged, (iii) the *handling* is easy, and (iv) it *supports the work flow*. The rating itself was done by Likert scales consisting of 6 items (“strongly disagree”, “disagree”, “tend to disagree”, “tend to agree”, “agree”, “strongly agree”).

4.1.2 Results

By using the original Eclipse IDE, the participants agreed very well to the observations we gained from the preliminary user study (Section 2). Hence, they implicitly confirmed our proposed guidelines, from which we derived the ideas for the extensions. All participants agreed that our extensions enhanced the handling of a touch monitor by solving the tasks of this study even it was unusual to some extent. Nevertheless, not every extension has enhanced touch handling. In contrast, sometimes a problem was replaced by another.

The participants preferred neither the radial nor the italic menu layout. The first impression of the radial menu was somewhat unusual. Nevertheless, some participants mentioned that the compact layout centered in the middle of the screen was clearly structured and readable. After a while, they were able to guess the spacial position of several menu items. In contrast, the participants were more familiar with the italic menu because it is similar to the list layout of the original main menu. By considering readability the opinions were very different. Some participants had problems to read the italic font, while others do not.

On the whole, occlusion effects were not recognized with one exception—touching very small elements. Since we have resized most graphical elements, in our study this fact only occurred by controlling the original sashes because the accuracy of the touch input had to be with pixel precision. Thus, by resizing graphical elements the touch handling could be enhanced but comes with an undesired side effect. Some participants criticized that the larger elements negatively

influence the layout, or in other words, the adapted layout looked strange to some extent. Hence, such modifications had to be worked out by professional UI designers. Nevertheless, most participants agreed that the new rearranged layout enhances controlling Eclipse by touch gestures, even if it was unusual.

At this point it is worth to underline that we only have adapted an existing Eclipse instance. Problems that come with decisions of the Eclipse developers, like overloaded menus, or wasteful and inefficient graphical elements were not part of this work. There is no doubt that such aspects has to be considered by designing a touchified IDE.

In respect to the experimental setup, the general opinion among the participants was that most of our touch extensions were beneficial to control the GUI by touch gestures. Nevertheless, in respect to a non-experimental scenario, their opinions were more controversial. Three participants could not imagine that a touchified GUI of an IDE could support the work flow of a professional developer. Three others argued exactly the opposite, whereas the remaining two were unsure. Some participants stated that touch gestures are well-suited for reading and navigation tasks, thus, they could facilitate to perform code reviews or presentations. Furthermore, developing apps for touch devices could be very natural and intuitive by using a touchified IDE. Altogether, we assume that some inconveniences occurred because of some concepts that were completely new for the participants. 45 minutes are not enough to become familiar with these novel concepts. Thus, in order to achieve more significant results, a more extensive user study has to be performed over a longer period of time.

4.2 Study 2: Mapping from Invertible Touch Gestures to Automated Refactorings

For the second user study, we recruited another group of 8 right-handed developers (2 bachelor students, 3 master/diploma students, 3 academic researchers) to participate. They also had to fill in a questionnaire to assess their prior knowledge and skills first. 6 participants stated to have “advanced” and 2 stated to have “good” programming skills. 3 had “little” experience with touch devices, whereas the others indicated “good” and “advanced” experience. They had prior experience with using smartphones (6), tablets (6), and touch monitors (2). Regarding gestures they are familiar with, they mentioned pinch open/close (8), flick (8), and rotate (1). Nobody mentioned uncommon gestures such

as *five-finger rotate*. 3 have “good”, 4 “little” and 1 no experiences with refactorings. 2 make “often”, 5 “little” and 1 “never” use of refactoring tools during programming sessions. Half of the participants even declared to have exclusive refactoring sessions without adding any new functionality.

4.2.1 Experimental Design

We performed three experiments for each person: (1) common desktop setup with only keyboard and mouse interactions; (2) desktop touch setup by only using a touch monitor; (3) mobile touch setup by only using tablet interactions.

For the 30-minute thinking-aloud experiment the participants were given some source code including a description of the refactoring tasks they had to perform. These task descriptions were put as a comment above the code explaining what to do without using parts of the refactoring name (e.g. for renaming a variable the description was: “change the identifier of the following variable reasonably”). This prevented the participants from guessing the name of the required refactoring by the words of the task. While they were performing the task, notes about their behavior and comments they made were taken. This was repeated 3 times (once for every setup). We concluded the experiment with an interview. The participants were asked about their experience with and without using multi-touch gestures.

4.2.2 Experiment with Keyboard and Mouse

In total only 2 participants were able to recall the shortcut for Rename. Shortcuts for other refactorings were unknown. Hence, in order to solve the other refactoring tasks all participants had to use the overfilled menu. Searching the applicable menu entry took all of them over 5 and up to 15 seconds. 5 participants said that they would prefer to manually perform refactorings instead of spending most of their time searching. The naming was not always self-explanatory. For example, most participants had no idea how to change the visibility of a method automatically, since this functionality is covered by the Change Method Signature refactoring. In some cases after selecting an entity, the desired refactoring was not listed in the context menu. 3 participants mentioned that the configuration dialogs interrupted their work flow and that they were superfluous because they always used the default configuration without using the preview functionality.

4.2.3 Experiment with Multi-Touch Gestures

First, the novel multi-touch gestures were introduced by short examples, so that the participants still had to explore parts of the functionality by themselves. The participants started scrolling through the text by using the pan-gesture straight-away, although they were not told that this feature is supported. Although most of them were familiar with touch devices, however, in the beginning they had difficulties to integrate the gestures into their programming work flow. After a few trials the participants were able to use the gestures effectively, thus, after reading the refactoring task they solved it usually under 5 seconds. Occlusion effects were not reported.

The most challenging part seemed to be making use of the rotate gesture. Some participants took 3 attempts to learn how to turn their arms or place their fingers correctly. The main issue was the text selection required by extract refactorings. This made some participants use both the mouse

for a precise selection and the touch monitor to perform the refactoring. They said that this combination felt very naturally.

Since the tablet had a much better touch recognition than the touch monitor in our experiment, performing the required touch gestures was a bit easier for all participants. Without a mouse, the participants were again confronted with the inconveniences of selecting text. Because of the built-in virtual keyboard, renaming identifiers was no problem for anybody. Some participants mentioned that they would like to use the tablet for code review and pure refactoring sessions. Nevertheless, in a real programming environment they would prefer the classical desktop setup with an additional multi-touch monitor.

5. RELATED WORK

An increasing number of application areas already benefit from the advantages of direct user interactions. However, in the area of software engineering, touch interactions are only rarely used. In the following we discuss related work investigating both the use of multi-touch gestures during programming tasks and performing refactorings with gestures.

Previous work has shown that the use of direct manipulations enables interesting opportunities for different software engineering domains. Edwards and Barnette [5] use tablet PCs with a pen in a laboratory programming course without adapting the software to this novel input device. While the students liked the mobility of their novel programming devices, they felt uncomfortable in using a pen for writing, testing, and compiling programs. After a while, they brought their own keyboards to the course. By turning smartphones into tangible digital CRC cards, Lutz et al. [11] introduced a novel approach for modeling software and its requirements. This application scenario demonstrates how touch devices can be used for co-located collaboration within the field of requirements engineering. Tillmann et al. [18] proposed *TouchDevelop*—a new programming language and environment for developing scripts, which can be created and executed on mobile phones. It targets mainly hobbyists for personalizing their phone. Hesenius et al. [7] presented a similar solution which takes advantage of tablets’ larger screen size to use more advanced features like a data and call stack view, a dictionary with vocabulary filter, and a separate view for navigation tasks. In contrast to those approaches we do not present a completely new GUI but introduce a concept for adapting an existing IDE step-by-step.

Murphy-Hill et al. [13] introduced a novel context-sensitive mapping from directional gestures to refactorings, which was represented by marking menus. The results of their user study indicate that developers are able to easily recall this mapping and to initiate a refactoring tool even without knowing its name. Since we are moving this idea forward to multi-touch gestures, their approach fundamental forms the basis for our gesture-to-refactoring mapping. However, in our work we only create an additional interaction layer and refrain from using additional visualizations. Raab et al. [15] propose a new set of touch gestures for performing refactorings on a tablet. To this end, they asked 16 participants to invent and to perform suitable gestures to trigger specific selection, editing, and refactoring operations. Our work differs in three main aspects: First, we use a narrow enclosed set of commonly known multi-touch gestures. Second, performing a gesture directly triggers a refactoring. In

other words, we never use multiple composed gestures to trigger a single refactoring. Third, in contrast to Raab et al. we introduce a first prototype which implements the proposed touch gestures and which was used in a user study to evaluate these suggested gestures. Since the Eclipse IDE only supports a few refactorings which can be triggered via drag-and-drop gestures (e.g. *Move Class*), Lee et al. [9] contributed a more extensive set of drag-and-drop gestures, initiating both single and composed refactorings. In contrast to our work, their approach is optimized for using a mouse and not a touch device.

6. CONCLUSIONS

To the best of our knowledge, this work presents the first approach to touchify a widely used IDE, i.e. to make most of its functionality available through touch gestures. In a user study we revealed problems of using the Eclipse IDE with the default touch support provided by the operating system. Subsequently we introduced new interaction techniques addressing the design goals mentioned in the introduction. As a first step we made touch optimizations by adjusting and reconfiguring the components of the existing user interface. This includes resizing and reordering of components by ergonomic aspects as well as adding touch support in existing components that are expected by the test users. Some elements of the IDE were not suitable for touch usage, thus, we introduced new elements including a radial main and context menu, an italic main menu, and overlaid handles for sashes. Finally, we provided a novel mapping from multi-touch gestures to refactoring tools.

Our user studies suggest that using a touch device in combination with keyboard and mouse improves the work flow of professional software development, especially by controlling the GUI and simplifying refactoring tasks. By using the emerging prototype during its implementation we were able to use our own experience to optimize the conceptual design. We think that especially feature-rich and complex bento box designs, such as the Eclipse IDE, can benefit from the potential of using direct manipulation methods. Although, there was a mouse available, in several situations we preferred to use touch gestures instead. For example we liked to use the radial menu because we noticed the advantages of the spatial arrangement of the menu items (declarative memory). Furthermore, most navigation tasks were performed by direct manipulations because they felt more intuitive and comfortable.

7. REFERENCES

- [1] G. Bailly, E. Lecolinet, and L. Nigay. Flower menus: a new type of marking menu with large menu breadth, within groups and efficient expert mode memorization. In *AVI*, pages 15–22. ACM Press, 2008.
- [2] P. Brandl, J. Leitner, T. Seifried, M. Haller, B. Doray, and P. To. Occlusion-aware menu design for digital tabletops. In D. R. O. Jr., R. B. Arthur, K. Hinckley, M. R. Morris, S. E. Hudson, and S. Greenberg, editors, *CHI Extended Abstracts*, pages 3223–3228. ACM, 2009.
- [3] R. DeLine and K. Rowan. Code canvas: zooming towards better development environments. In *ICSE* (2), pages 207–210. ACM, 2010.
- [4] G. M. Draper, Y. Livnat, and R. F. Riesenfeld. A survey of radial methods for information visualization. *IEEE Trans. Vis. Comput. Graph.*, 15(5):759–776, 2009.
- [5] S. H. Edwards and N. D. Barnette. Experiences using tablet pcs in a programming laboratory. In *SIGITE Conference*, pages 160–164. ACM, 2004.
- [6] M. Fowler. *Refactoring: improving the design of existing code*. Addison-Wesley Professional, 1999.
- [7] M. Hesenius, C. D. O. Medina, and D. Herzberg. Touching factor: Software development on tablets. In T. Gschwind, F. D. Paoli, V. Gruhn, and M. Book, editors, *Software Composition*, volume 7306 of *Lecture Notes in Computer Science*, pages 148–161. Springer, 2012.
- [8] A. J. Ko, H. H. Aung, and B. A. Myers. Eliciting design requirements for maintenance-oriented IDEs: a detailed study of corrective and perfective maintenance tasks. In G.-C. Roman, W. G. Griswold, and B. Nuseibeh, editors, *ICSE*, pages 126–135. ACM, 2005.
- [9] Y. Y. Lee, N. Chen, and R. E. Johnson. Drag-and-drop refactoring: intuitive and efficient program transformation. In D. Notkin, B. H. C. Cheng, and K. Pohl, editors, *ICSE*, pages 23–32. IEEE / ACM, 2013.
- [10] D. Leithinger and M. Haller. Improving menu interaction for cluttered tabletop setups with user-drawn path menus. In *Tabletop*, pages 121–128. IEEE Computer Society, 2007.
- [11] R. Lutz, S. Schäfer, and S. Diehl. Using mobile devices for collaborative requirements engineering. In *ASE*, pages 298–301. ACM, 2012.
- [12] S. Meyer, O. Cohen, and E. Nilsen. Device comparisons for goal-directed drawing tasks. In C. Plaisant, editor, *CHI Conference Companion*, pages 251–252. ACM, 1994.
- [13] E. R. Murphy-Hill, M. Ayazifar, and A. P. Black. Restructuring software with gestures. In G. Costagliola, A. J. Ko, A. Cypher, J. Nichols, C. Scaffidi, C. Kelleher, and B. A. Myers, editors, *VL/HCC*, pages 165–172. IEEE, 2011.
- [14] E. R. Murphy-Hill, C. Parnin, and A. P. Black. How we refactor, and how we know it. In *ICSE*, pages 287–297. IEEE, 2009.
- [15] F. Raab, C. Wolff, and F. Echtler. Refactorpad: editing source code on touchscreens. In *EICS*, pages 223–228. ACM, 2013.
- [16] K. Samp and S. Decker. Supporting menu design with radial layouts. In G. Santucci, editor, *AVI*, pages 155–162. ACM Press, 2010.
- [17] B. Shneiderman and C. Plaisant. *Designing the User Interface - Strategies for Effective Human-Computer Interaction* (5. ed.). Addison-Wesley, 2010.
- [18] N. Tillmann, M. Moskal, J. de Halleux, and M. Fähndrich. Touchdevelop: programming cloud-connected mobile devices via touchscreen. In *Onward!*, pages 49–60. ACM, 2011.
- [19] D. Wigdor and D. Wixon. *Brave NUI world: designing natural user interfaces for touch and gesture*. Elsevier, 2011.