GUI Change Method according to Roles of Widgets and Change Patterns

Junko SHIROGANE^{†a)}, Member, Hajime IWATA^{††b)}, Kazuhiro FUKAYA^{††c)}, Nonmembers,
and Yoshiaki FUKAZAWA^{††d)}, Member

SUMMARY To develop usable software, it is necessary to develop Graphical User Interfaces (GUIs) in iterative steps, such as evaluating the usability of GUIs and improving GUIs. In improving GUIs, developers are often required to modify both the GUI and the logic code of the software. In our research, to facilitate GUI improvement, we propose a method of automatically searching for code to be modified and suggesting how to modify them. To search for appropriate code to be modified, we define the roles of widgets according to their purpose and the patterns for how to change GUIs. In our method, how to change GUIs is specified, and then the parts of source programs that are required to be modified are searched for. Also, we classify methods for each widget according to their functions. Using this classification, a method of modifying the code that is searched for is suggested.

key words: GUI, widget, usability, source program modification

1. Introduction

The usability of software is important, and human-centered software design has been receiving considerable attention. When considering the usability of software, Graphical User Interfaces (GUIs) are very important, because they directly interact with end-users. This means that the usability of software employed by end-users depends predominantly on the usability of the GUIs of the software. Therefore, it is very important to develop usable GUIs to realize software of high usability.

End-users' experiences and preferences markedly affect the usability of the GUIs employed. Thus, it is difficult to develop sufficiently usable GUIs for end-users at one time. In many cases, to make GUIs more usable, GUIs are developed first, and then the following steps are repeated: both experts in usability and end-users evaluate the usability of the GUIs; the GUIs are improved on the basis of the results of evaluation [1]. Also, there are many cases in which some usability problems are found not during the development of the software but during actual use. In these cases, GUIs are often required to be modified on the basis of these problems. That is, it can be considered that GUIs are often

changed to improve the usability of the software. Moreover, when GUIs are required to be changed, it is necessary to modify the source programs of the software.

We assume that usability in our research means understandability of window flows for a task as well as understandability and operability of each input, selection, and display item in a window, that is, operational ease. For example, there are several input, selection, and display items for executing a task, and they are shown to users as being divided into some groups and on some windows. In this case, there are many strategies for division. As another example, when users input text and select an item, some processes are performed on the basis of the input and selection. In this case, the users' input and selection can be defined as the trigger of the processes. Also, a widget for the trigger of the processes can be prepared; the users' operations to the widget after input and selection can be considered to be defined as a trigger. In addition, there are some types of widgets that can be used as an input, a selection or a display item, and it is important to select type of widgets for the item in terms of usability. This division of items to windows and usage of widgets affect operational ease. To improve usability of GUIs, this division of items to windows and usage of widgets are often required to be changed, and modification of source programs is required in these GUI changes.

The source programs of the software can be classified into the following three types: GUI, logic and connection code. The GUI code deals with GUI processes, which involve widget generation and widget arrangement on windows. The logic code provides the body of the software, such as data processing and file handling. The connection code performs data transfer between the GUI code and logic code. When the improvement in GUI usability involves only layout changes, such as changes in the size and arrangement of widgets in a window, it is not necessary to modify the logic and connection code. There are many tools that can support layout changes, such as Eclipse [2] and NetBeans [3]. However, in many cases, it is necessary to change not only the layout but also the types of widgets and windows on which widgets are placed. In these cases, it is necessary to modify the connection code between GUI and logic code, and even logic code itself (we call these code to be modified "CtbM"). When changing the GUIs of large-scale software, the source programs may include many CtbM, which may be spread widely throughout the programs. In this case, software developers often make mis-

Manuscript received July 2, 2007.

Manuscript revised October 17, 2007.

[†]The author is with Tokyo Woman's Christian University, Tokyo, 167-8585 Japan.

^{††}The authors are with Waseda University, Tokyo, 169-8555 Japan.

a) E-mail: junko@lab.twcu.ac.jp

b) E-mail: hajime.i@fuka.info.waseda.ac.jp

c) E-mail: kazupyon@fuka.info.waseda.ac.jp

d) E-mail: fukazawa@fuka.info.waseda.ac.jp

DOI: 10.1093/ietisy/e91-d.4.907

takes in identifying and modifying the CtbM.

Recently, to develop software efficiently, Integrated Development Environments (IDEs), such as Eclipse and NetBeans, have been widely used. When changing GUIs, it is necessary to suggest semantically required CtbM candidates to support developers; however, IDEs suggest not semantically required candidates but syntactically near-spelling candidates. Thus, GUI changes constitute onerous work for developers and there are few methods for supporting the suggestion of CtbM candidates and instructions on how to modify them for GUI changes.

In our research, to make source program modification for GUI changes easy and correct, we propose a method of searching for CtbM and suggesting instructions on how to modify them. Here, we focus on the Java programming language, and example widgets used in this paper are in the Java Swing package [4]. However, our method does not depend on specific programming languages or GUI packages. Also, in this paper, we call widgets before GUI changes (currently used widgets in the target software) “pre-widgets” and those after GUI changes “post-widgets”.

2. Policy of Code Searching in GUI Change

2.1 Cases of GUI Change

To consider GUI changes, examples of GUIs are shown in Fig. 1, Fig. 2 and Fig. 3.

(a) Sample of three items in one window

(b) Sample of items in multiple windows

Fig. 1 Samples of window allocation.

(a) Sample of processing after pushing button

(b) Sample of processing after only selecting item

Fig. 2 Samples of triggers of process.

Both Figs. 1 (a) and (b) show GUIs where users fill in three fields. However, in Fig. 1 (a), all input fields are in one window, and in Fig. 1 (b), there are three windows with one input field in each window. Thus, there can be many window allocation variations of widgets.

Figures 2 (a) and (b) show GUIs where users select a country from the combo-box, then prefecture names are shown in the list-box below. In Fig. 2 (a), the user selects a country and pushes the “select” button. Then, prefecture names are shown in the list-box. In Fig. 2 (b), the user merely selects a country, then prefecture names are shown in the list-box. Thus, there can be some variations of the trigger of processes.

Figures 3 (a), (b) and (c) show GUIs where users select an item among the given items. A list-box widget is shown in Fig. 3 (a), a combo-box widget is shown in Fig. 3 (b), and some radio-button widgets are shown in Fig. 3 (c). Thus, there can be many variations of widgets used for each item.

The operational feelings of GUIs in Fig. 1, Fig. 2 and Fig. 3 are different, and which GUIs are used affect usability. The determination of which of the GUIs in Fig. 1, Fig. 2 and Fig. 3 are the most usable is based on various factors, including users’ preferences, the actual use environment and the processed data in the software. Therefore, it is difficult to determine appropriate GUIs on the basis of the design once in the software development phase.

Also, to reduce the dependence between the GUI code and the logic code, Model View Controller (MVC) architecture [5] has been proposed and widely used. The model and controller correspond to the logic and connection code; the view corresponds to the GUI code in our research. In MVC architecture, the model is not required to be modified when changing the view. However, the view and controller often strongly depend on each other. That is, the controller is required to be modified when changing the view. In addition, the structure of the controller often becomes complex. Hence, it is not easy to modify the controller when changing the view. That is, in the case of using MVC architecture, it is considered that changing GUIs is not always easy to

(a) Sample of a list-box for selection

(b) Sample of a combo-box for selection

(c) Sample of radio-buttons for selection

Fig. 3 Samples of widget usages.

achieve. Therefore, it is important to support the modification of source programs when changing GUIs.

2.2 Necessity of Supporting GUI Changes

Now, IDEs, such as Eclipse [2] and NetBeans [3], are widely used. When GUIs are changed using these IDEs, IDEs indicate syntactic errors in accordance with the GUI changes and suggest instructions on how to modify them. However, these errors and instructions are on the basis of syntax, not semantics.

For example, in changing GUIs in Fig. 3 (c) to those in Fig. 3 (a), some radio-button widgets are replaced with a list-box widget. Radio-button widgets are used for single selection, and one selection item is allocated to one radio-button widget. The selection is determined by which radio-button widget is selected. That is, software confirms the state of selection of each radio-button individually, and identifies the selected radio-button widget. In the case of the Java Swing package, for this identification, the “isSelected()” method defined in the radio-button widget is used. The example of the source program is shown in the dotted square of Fig. 4 at the left. Thus, software identify the selected radio-button widget by the “if” statement and the “isSelected()” method. After identifying the selection, some processes are performed, such as setting values to variables. When radio-button widgets are replaced with a list-box widget, this “isSelected()” method and processes including “if” statement are required for modification to another method or process.

List-box widgets can be used for selection, and they include all selection items in them. The selection is identified by which item is selected in the list-box widgets. In the case of the Java Swing package, for this identification, the “getSelectedValue()” or “getSelectedIndex()” method defined in list-box widgets is used.

When radio-button widgets are replaced with a list-box widget, the “isSelected()” method and processes must be modified to the “getSelectedValue()” or “getSelectedIndex()” and processes. Therefore, the instructions should be suggested as (a) in Fig. 4. However, when IDEs are used for these GUI changes, IDEs only show the instruction that the “isSelected()” method of radio-button widgets should be modified to the “isSelectionEmpty()” method of list-box widgets as (b) in Fig. 4. The “isSelectionEmpty()” method is used to confirm whether or not there is a selected item in the list-box widget. This purpose of the “isSelectionEmpty()”

method is different from that of the “isSelected()” method, so we consider that the instruction by IDEs is not semantically appropriate.

Thus, in changing GUIs using IDEs, IDEs only suggest syntactically near-spelling candidates as instructions. Therefore, IDEs cannot suggest a group of statements as in the above example or instructions on semantically equal strategies of modification. In our method, these instructions on semantically equal strategies can be suggested.

2.3 Roles of Widgets

There are many types of widgets, and these widgets are used for various purposes, such as text input and item selection. When replacing widgets with other widgets as part of a GUI change, the purposes of GUI operation should not be changed after the replacement. For example, a list-box widget and a combo-box widget similar to those in Figs. 3 (a) and (b) are used for item selection. Thus, even if a list-box is replaced with a combo-box, the purpose of the GUI operation is not changed. Also, a text-field is used for text input. If a text-field is replaced with a list-box, users cannot enter text using the keyboard. This indicates that the purpose of the GUI operations is changed from free text input to item selection. Therefore, the types of post-widgets should be restricted according to the purpose of the usage of the pre-widgets. In our method, we classify the types of widgets according to the purposes of usage in advance. We call these purposes “roles”, and the classified types are listed in a table (Widget Replacement Table (WRT)). We now define the following seven roles.

Display role

A display role widget displays the results of processes in logic code and input contents from users for confirmation. Some examples are label and list-box (in Java Swing, JLabel and JList) widgets.

Selection role

A selection role widget enables users to select items from several given items. Some examples are combo-box and radio-button (in Java Swing, JComboBox and JRadioButton) widgets.

In this role of widgets, software developers define no processes. That is, when users select items, no processes that software developers define are performed. In selecting items, some processes are performed in all widgets. For example of a combo-box widget, a selected item is displayed at the top of entries of the combo-box widget. However, these processes are performed in the background, and software developers are not required to define these processes.

Selection & action role

A selection & action role widget enables users to select items from several given items and then carries out some processes. Some examples are combo-box and radio-button (in Java Swing, JComboBox and JRadioButton) widgets.

Fig. 4 Example of desirable and actual instructions.

In this role of widgets, software developers define some processes. That is, when users select items, some processes that software developers define are performed. As mentioned, in the selection role, there are some background processes for widgets. Selection & role widgets also have such background processes. However, some processes, not background processes, are added to widgets of this role. For example, this role of widgets is often used for address input using combo-box widgets. In this case, two combo-box widgets are arranged in one window: one for selecting a state/prefecture and the other for selecting a city. After users select a state/prefecture name in the first combo-box widget, the name of a city in the selected state/prefecture is entered to the combo-box for the second combo-box widget. This process for entering the names of cities into a combo-box widget is required to be defined by software developers.

Input role

An input role widget is used to input text. An example is a text-field (in Java Swing, `JTextField` and `JTextArea`) widget.

In this role of widgets, software developers define no processes. That is, when users enter text, no processes defined by software developers are performed. In entering text, some processes are performed in all widgets. For example of a text-field widget, entered text is displayed on the text-field widget. However, these processes are performed in the background, and software developers are not required to define these processes.

Input & action role

An input & action role widget enables users to input text and then it carries out some processes (in many cases, some processes are performed after users have input text and hit the Enter key on the keyboard). An example is a text-field (in Java Swing, `JTextField` and `JTextArea`) widget.

In this role of widgets, software developers define some processes. That is, in many cases, after users enter some text and hit the Enter key on the keyboard, some processes that software developers define are performed. As mentioned in the input role, there are some background processes for widgets. Input & role widgets also have such background processes. However, some processes, not background processes, are added to widgets of this role. For example, this role of widgets is often used for authentication. In this case, two text-field widgets are arranged in one window: one for entering a login ID and the other for entering the corresponding password. After users enter a login ID and password and hit the Enter key on a keyboard, the processes for authentication are performed. These processes for authentication are required to be defined by software developers for widgets.

Action role

An action role widget prompts user events, such as pushing a button or hitting the Enter key. Some examples are button and menu-item (in Java Swing, `JButton` and `JMenu-`

Item) widgets.

Container role

A container role widget is used as a mat on which widgets are placed. Some examples are panel and frame (in Java Swing, `JPanel` and `JFrame`) widgets.

In our present method, there are the above 7 roles of widgets, however, there may be more roles. For example, the selection & action role is considered to be a combination of the selection role and the action role. The input & action role is similar. We believe that there are other combinations of roles. Therefore, we will continue to survey other roles.

Also, the WRT[†] is created by us and built into our system. We consider that software developers will be able to modify the WRT in the future.

2.4 Change Patterns

When changing widgets, there are several cases to consider, such as changing the window on which widgets are placed (for example, changing the GUI in Fig. 1 (a) to that in Fig. 1 (b)), changing the trigger of processes (for example, changing the GUI in Fig. 2 (a) to that in Fig. 2 (b)), and replacing the current widgets with other types of widgets (for example, changing the GUI in Fig. 3 (a) to that in Fig. 3 (b)). Also, as in the example of changing the GUI in Fig. 3 (a) to that in Fig. 3 (c), there are cases in which one widget is replaced with several widgets. There are also the cases in which several widgets are replaced with one widget, and several widgets are replaced with several widgets.

To support GUI changes appropriately, we adopt the term "change patterns" to indicate the available patterns by which GUIs can be changed on the basis of the windows on which widgets are placed, the number of widgets and the operation of GUIs. Change patterns make the change in GUIs explicit. Therefore, when change patterns are selected, our system can ask software developers for the factors that should be considered in changing GUIs explicitly, such as how many windows one window is to be divided into and which widgets should be replaced with which other widgets. Thus, our system can search for CtbM and suggest instructions appropriately. The following 12 change patterns are currently used.

Window division pattern

A window is divided into several windows. For example, the GUI in Fig. 1 (a) is changed to that in Fig. 1 (b).

Window integration pattern

Some windows are integrated into one window. For example, the GUI in Fig. 1 (b) is changed to that in Fig. 1 (a).

Window allocation change pattern

A widget or a set of widgets is moved from one window

[†]The present WRT is on
<http://www.twcu.ac.jp/~junko/WREATH/WRT.xml>

to another window.

Window deletion pattern

A window is deleted.

Action inclusion pattern

A widget (a set of widgets) without an action role and a widget with an action role are replaced with one widget (a set of widgets) with an action role. That is, an input role widget and an action role widget can be replaced with an input & action role widget, or a selection role widget and an action role widget can be replaced with a selection & action role widget. For example, the GUI in Fig. 2 (a) is changed to that in Fig. 2 (b) in this change pattern.

Action separation pattern

A widget (a set of widgets) with an action role is replaced with a widget (a set of widgets) without an action role and an action role widget. That is, an input & action role widget is replaced with an input role widget and an action role widget, or a selection & action role widget is replaced with a selection role widget and an action role widget. For example, the GUI in Fig. 2 (b) is changed to that in Fig. 2 (a) in this change pattern.

One-to-one pattern

A widget is replaced with another type of widget, for example, changing the GUI in Fig. 3 (a) to that in Fig. 3 (b).

One-to-many pattern

A widget is replaced with a set of widgets of another type (all post-widgets are of the same type), for example, changing the GUI in Fig. 3 (a) to that in Fig. 3 (c).

Many-to-one pattern

A set of widgets is replaced with another type of widget (all pre-widgets are of the same type), for example, changing the GUI in Fig. 3 (c) to that in Fig. 3 (a).

Many-to-many pattern

A set of widgets is replaced with a set of widgets of another type (all pre-widgets are of the same type, and all post-widgets are of the same type), for example, a set of check-box widgets is replaced with a set of toggle-button widgets.

Widget addition pattern

A widget or a set of widgets is added. For example, changing the GUI in Fig. 5 (a) to that in Fig. 5 (b). Figures 5 (a) and (b) show GUIs where users select an item using combo-box widgets. In Fig. 5 (a), instruction to users is shown in a combo-box widget where users select an item. In Fig. 5 (b), instruction to users is shown using a label widget.

Widget deletion pattern

A widget or a set of widgets is deleted. For example, changing the GUI in Fig. 5 (b) to that in Fig. 5 (a).

Fig. 5 Samples of instruction to users.

2.5 Widget Replacement Table

Each widget has several methods for various tasks, such as those for obtaining a value for the widget and for setting the size of the widget. Some of these methods are used for the same purpose in all widgets. Most of these methods are inherited from a common superclass of widgets. Therefore, such methods are used for the same purpose in all widgets. On the other hand, some methods are specific to each individual widget. Furthermore, widgets with the same role may have different methods for the same purpose.

For example, all widgets have the `getSize()` and `setName(String)` methods. Widgets inherit these methods from the common superclass of widgets. On the other hand, the `getSelectedValue()` method is a specific method in the `JList` widget, and the `addItem(Object)` method is a specific method in the `JComboBox` widget. The `getSelectedValue()` method of the `JList` widget is used for obtaining a selected value of the widget. The `addItem()` method of the `JComboBox` widget is used for adding an item to the widget. Both the `JList` and `JComboBox` widgets have the selection role. Widgets with the selection role also have methods for obtaining a selected item. Both the `getSelectedValue()` method for the `JList` widgets and the `getSelectedItem()` method for the `JComboBox` widgets are used for obtaining a selected item. Thus, there are methods that have different names, but are used for the same purpose.

To appropriately suggest both `CtbM` and instructions for replacing widgets, it is necessary to identify the required functions for each role and to classify the methods into these functions according to the purpose of the method. For example, the function of obtaining a selected item is required for selection role widgets. The `JComboBox` widgets have the selection role, and the `getSelectedItem()` method of the `JComboBox` widgets is used for this function, so this method is classified into this function. This classification is listed in the WRT.

In some cases, a method of a widget corresponds to a function. In other cases, several methods of a widget and several statements correspond to one function. For example, in the `JComboBox` widgets, the `getSelectedItem()` method corresponds to the function of obtaining a selected item. The `JRadioButton` widgets also have the selection role. To accomplish the function of obtaining a selected item, the `JRadioButton` widget that has been selected must be first identified, and then the text or the name of the `JRadioButton` widget must be obtained, or some processes must

Table 1 Example of widget replacement table (selection role).

Widget	Getting a selected item	Getting widget size
JList	getSelectedValue()	getSize() (in JComponent)
JComboBox	getSelectedItem()	getSize() (in JComponent)
JRadioButton	if(<widget variable>.isSelected()) getText() (getName() or some processes) ;	getSize() (in JComponent)

be performed. That is, the function of obtaining a selected item in the JRadioButton widget corresponds to an if statement and the isSelected(), getText(), or getName() method or some processes. These correspondences are also given in the WRT. Part of the WRT is shown in Table 1.

3. Overview of System Usage

Figure 6 shows the overview of system usage in our method. To render GUIs more usable, their usability is evaluated. For usability evaluations, end-users and software developers (particularly experts of usability) use GUIs of software in trials and evaluate them.

There are various methods of usability evaluations performed in the early development phase and the late development phase. For example, heuristic evaluation wherein experts of usability evaluate GUIs on the basis of their experience is often performed [1]. In end-user evaluations, end-users use GUIs of software. Then, the execution time of a task is recorded, questionnaires surveys of end-users are carried out, and comments from end-users are recorded and analyzed [1].

On the basis of the results of usability evaluations by these various methods in various phases development, software developers consider the policies of GUI changes. Then, they use our system, the Widget REplacement Analysis and Trace Helper (WREATH), to carry out the GUI changes.

4. System Architecture

Figure 7 shows the architecture of WREATH. WREATH consists of the following four parts: widget extraction, policy determination, CtbM search, and CtbM suggestion subsystems.

In this architecture, all subsystems have been implemented and integrated in WREATH. The WRT has also been created by us and built into WREATH. Therefore, when developers specify the source programs of the target software, WREATH automatically analyzes the source programs. Also, when developers specify policies of GUI changes, WREATH automatically searches for the appropriate CtbM and suggests instructions on how to modify them.

4.1 Widget Extraction Subsystem

WREATH translates the source programs of the target software into XML format [6]. JavaML is a markup language for Java source programs. To translate source programs into JavaML, the JavaML converter [6] is used. This is a tool for adding modules for translating source programs into

Fig. 6 Overview of system usage. **Fig. 7** System architecture.

JavaML to Jikes [7], which is a compiler for Java.

After translation, WREATH reads the generated XML format of source programs and extracts the types (class names) of widgets, the variable names of widgets and windows on which widgets are placed.

In the present WREATH, when source programs are specified by developers, WREATH automatically invokes the JavaML converter, and the XML format of source programs is generated. Developers can specify the source programs using GUIs. Then, WREATH automatically extracts the above information on widgets.

4.2 Policy Determination Subsystem

After usability evaluation, how GUIs should be changed is

determined. Then, software developers use WREATH on the basis of the policies of GUI changes as described in Sect. 3. Part of the flow of specifying how to change GUIs is shown in Fig. 8.

First, after extracting the types of widgets, the variable names of widgets and windows on which widgets are placed, WREATH shows change patterns, as in the window in Fig. 8 (1). Developers then select an appropriate pattern in this window. Next, WREATH asks for details on how to make the change on the basis of the selected change pattern, as in the windows in Fig. 8 (2)-a-1, (2)-a-2, (2)-b, and (2)-c, and the details are specified. When specifying details, windows differ according to the selected change patterns.

For example, when the window division pattern is selected, WREATH shows the candidate windows and asks which window is to be divided and into how many windows the target window is to be divided (the window in Fig. 8 (2)-a-1). Developers specify the target window and the number of divided windows, then WREATH asks how the widgets of the target window are to be allocated into the divided windows (the window in Fig. 8 (2)-a-2).

When the action inclusion pattern is selected, WREATH first asks for the identification of the pre-widgets not including action roles and of action role widgets, then developers select each of these from the candidates by clicking the “Select” button (Fig. 8 (2)-b) in the “pre-widgets” area. Then, WREATH shows the available candidate post-widgets with the same role including the action role, in the combo-box widget in the “post-widget” area in the window in (2)-b, and one of them is selected. In the combo-box widget in the “post-widget” area, when selection role widgets are selected as the pre-widgets, selection & action role wid-

gets are shown. When input role widgets are selected as the pre-widgets, input & action role widgets are shown.

If the one-to-many pattern is selected, WREATH first asks for the identification of the pre-widgets, and developers select the pre-widgets from the candidates by clicking the “Select” button (Fig. 8 (2)-c) in the “Pre-widgets” area. Then, WREATH shows the available candidate post-widgets with the same role as the selected pre-widget in the combo-box widget in the “Post-widget(s)” area in the window in (2)-c, and one of them is selected.

In other change patterns, developers can specify how to change GUIs similarly. Thus, in the present WREATH, these processes for showing change patterns, asking for details on how to make the change, and showing the available candidate post-widgets are realized as tools. That is, developers can specify these on GUIs.

4.3 CtbM Search Subsystem

WREATH searches for candidate CtbM on the basis of the change pattern and the details on how to identify the change in the policy determination subsystem. In this subsystem, candidate CtbM are searched for by the following three steps: search target class identification, search target method identification, and candidate CtbM identification. In the present WREATH, all the above processes can be automatically performed. Also, these steps are performed by analyzing source programs statically in WREATH. In object oriented programming, invoked methods are often determined dynamically in executing, so it is impossible to analyze method (constructor) invocation completely with the static analysis. However, if the analysis is performed using

Fig. 8 Flow of specifying how to change GUIs.

the dynamic information, such as execution history, there often are parts of source programs that are not executed. Thus, there are problems for the coverage of CtbM that WREATH shows. Therefore, we adopt the static analysis in WREATH.

(1) Search target class identification

CtbM are not always contained in only the class in which the window to be changed or pre-widgets are declared (We call this class a “window/widget-declared class”). Therefore, it is necessary to identify target classes that are required to be modified in other classes. In our method, target classes to be searched for are identified using an “Instance Creation/Call Graph (ICCG)”. This graph describes the relationship between created and creating instances of classes.

An example of the ICCG is shown in Fig. 9. This example is a part of the ICCG of a video rental system. In this graph, a node represents a class, a solid arrow extends from a class that creates an instance to a class in which the instance is created, and a dashed arrow extends from a class that calls an instance to another class that the instance is called. For example, an instance of class RentVideo is created in class VideoRental, and an instance of class ApplicationData is called from class RentVideo in Fig. 9.

To generate an ICCG, WREATH reads the source programs of the target software in XML format and analyzes a class. In this analysis, the creating and calling instances in the class are extracted, and the classes of the instances are identified. All classes in the target software are analyzed, and an ICCG is generated by connecting classes of instances and classes that create and call these instances.

In WREATH, the ICCG of the target software is generated, and target classes to be searched for include the following. In this example, class RentVideo in Fig. 9 is regarded as the window/widget-declared class.

- Window/widget-declared class (class RentVideo in Fig. 9)
- Predecessor classes of widget-declared class on ICCG (i.e., classes that create or call instances of

Fig. 9 Example of instance creation/call graph.

window/widget-declared class; class VideoRental in Fig. 9)

- Successor classes of window/widget-declared class on ICCG (i.e., classes of instances created or called from window/widget-declared class; classes ProcessRentVideo and ApplicationData in Fig. 9)

(2) Search target method identification

After generating an ICCG, WREATH extracts methods that are defined in the target classes identified in step (1) and generates a call graph of these extracted methods. A call graph describes the relationship between the called and calling methods. A node represents a method, and an arrow extends from a method that calls another method to the called method. An example of the call graph is shown in Fig. 10. This example is part of a call graph based on Fig. 9.

This call graph is used to identify the candidate methods in which code are required to be modified in step (3). The target methods to be searched are the following.

1. Methods in the window/widget-declared class (methods RentVideo and actionPerformed in the RentVideo class in Fig. 10)
2. Predecessor methods of 1. on the call graph. (e.g., methods VideoRental and actionPerformed in Fig. 10)
3. Successor methods of 1. on the call graph (e.g., method getCastInfo in class ApplicationData and method rentProcess in class RentVideoProcess in Fig. 9)

(3) Candidate CtbM identification

WREATH generates a program dependence graph (PDG) [8] and searches for candidate CtbM using the PDG. In PDGs, a node represents a statement, and edges represent data dependences. A PDG describes the data and control dependences between statements in the program. In our method, PDGs are generated for each method and constructor. Each method and constructor are described procedurally, so PDGs in our method can be generated as well as procedural languages. PDGs are generated by the following

Fig. 10 Example of call graph.

procedure.

1. Extract statements of defining and referring variables
2. Analyze data dependence relationships
3. Create PDGs

The examples of generated PDGs are shown in Fig. 11. These PDGs at the right are generated along with radio-button widgets “action” and “history” during their replacement with a combo-box widget. The number in each node corresponds to the line number of the source program at the left.

After generating PDGs, WREATH searches for candidate CtbM by tracing the data dependence in a PDG. The algorithm of searching for candidate CtbM is as follows.

1. Identify a node of the declaration of pre-widgets (widgets in the window to be changed or widgets to be replaced with another type of widget) in the window/widget-declared class.
2. Trace edges of data dependence from the node identified in step 1, and identify nodes.
3. If a traced node is a statement that calls for a method or a constructor, perform the following:
 - a. identify the method or constructor using the call graph generated in (2) (Search target method identification),
 - b. if the parameters of the method or constructor are the pre-widgets, trace edges of PDGs of the method or constructor based on the parameters, that is, perform this algorithm from step 2,
 - c. if the return values of the method or constructor are the pre-widgets, trace edges of PDGs of the method or constructor based on the return values, that is, perform this algorithm from step 2.
4. Repeat steps 2 and 3 using the nodes identified in step 2 as the nodes in step 1.

Fig. 11 Example of generated PDGs.

In our method, PDGs are generated for each method and constructor. When variables of pre-widgets are defined as fields of classes, WREATH searches for nodes related to the pre-widgets using all PDGs of all methods of the classes using the above algorithm. When variables of pre-widgets are defined as local variables of a method, WREATH searches for nodes using the PDGs of the method using the above algorithm. In this procedure, statements of the identified nodes are candidate CtbM.

For example, in Fig. 11, nodes “10 (action)” and “10 (history)” are identified in step 1 of this algorithm. In step 2, nodes “44”-“49”, “150”-“153”, and “204” are identified. According to the source program at the left in Fig. 11, the node “204” indicates the method (constructor) invocation. In this method (constructor), the parameter is the target for searching, so the invoked method (constructor) is identified using the call graph in Fig. 10, a PDG of the method (constructor), i.e., “ConfirmVideoRegistration” constructor is generated on the basis of the parameter, and candidate CtbM are also searched for.

4.4 CtbM Suggestion Subsystem

It may not be necessary to modify all code identified by the CtbM search subsystem. Therefore, WREATH extracts the required CtbM from among the searched code and suggests instructions on how to modify the code. These procedures consist of the following two steps: (1) extraction of required CtbM and (2) suggestion of instructions on modification. In the present WREATH, the required CtbM can be automatically extracted and instructions can also be automatically suggested. These suggestion of instructions are shown using GUIs.

(1) Extraction of appropriate CtbM

The required CtbM include the following: variable declaration, variable reference, the method names of widgets, return type declaration in methods, and the type declaration of parameters in methods. In some cases, method names are required to be modified, whereas in other cases, they are not. These are identified on the basis of the pre-widgets and post-widgets.

For example, when a JList widget is replaced with a JComboBox widget, they both have the getSize() method for obtaining the widget size. The JList and JComboBox widgets inherit this getSize() method from the JComponent class, which is the common superclass of the JList and JComboBox classes. That is, this method is not specific to the JList or JComboBox widgets. Thus, the methods inherited from the same class need not be replaced, because the parameters, the return type, and the purpose of usage are the same. However, the JList widgets have the getSelectedValue() method for obtaining a selected item, which is defined in the JList class. This method is specific to the JList widgets. Therefore, the getSelectedValue() method of the JList widget is required to be replaced with a corresponding method of JComboBox. Thus, the methods defined in each

class of widgets are required to be replaced with another method or process. In the WRT, the classifications of these methods are also given. Therefore, WREATH extracts the appropriate CtbM from the searched code using the WRT.

(2) Suggestion on modification

WREATH suggests instructions on how to modify the appropriate CtbM identified in step (1) using the WRT. The classifications of methods and widgets and the available change pattern for each widget are given in the WRT. Thus, WREATH can suggest corresponding instructions on how to modify each CtbM using the WRT. Examples of suggested instructions are shown in Fig. 12 and Fig. 13. In Fig. 12, the GUI in Fig. 1 (a) is changed to that in Fig. 1 (b). In Fig. 13, the GUI in Fig. 3 (a) is changed to that in Fig. 3 (c). Also, Fig. 12 (a) and Fig. 13 (a) show the valid instructions suggested by WREATH, and Fig. 12 (b) and Fig. 13 (b) show the real images of the suggested instructions by WREATH (these real images show instructions that WREATH really suggested, therefore, they include some invalid instructions). In Fig. 12 (b) and Fig. 13 (b), the left sides of the windows show the target source programs, and the right sides

of the windows show the instructions. In these windows, instructions on the right sides are suggested by the number of lines, and the corresponding lines of the suggested instructions are highlighted on the left sides.

Instructions suggested by WREATH are following five types:

(a) Move instruction

This type of instruction indicates that it is necessary to move the CtbM identified in step (1), and WREATH gives suggestions on where to move them. This instruction is often suggested for the window division, window allocation change and window integration patterns. For example, instructions with the <move> label in Fig. 12 are of this type.

(b) Replace instruction

This type of instruction indicates that it is necessary to replace the CtbM identified in step (1), and WREATH gives suggestions on how to change them. This instruction is often suggested for the action inclusion, action separation, one-to-one, one-to-many, many-to-one and many-to-many patterns. For example, instructions with the <replace> label in Fig. 13

Fig. 12 Example of instructions for dividing a window.

Fig. 13 Example of instructions for replacing a widget.

are of this type.

(c) Add instruction

This type of instruction indicates that it is necessary to add some code, and WREATH gives suggestions on code to be added and the location at which they should be added. This instruction is often suggested for the window division, one-to-one, one-to-many, many-to-one, many-to-many and widget addition patterns. For example, instructions with the <add> label in Fig. 12 and Fig. 13 are of this type.

(d) Delete instruction

This type of instruction indicates that it is necessary to delete the CtbM identified in step (1). This instruction is often suggested for the window deletion and widget deletion patterns.

(e) Change-necessity instruction

This type of instruction indicates that it is necessary to modify the CtbM identified in step (1), but WREATH cannot give a suggestion on how to modify them. This instruction is suggested when post-widgets do not have a method corresponding to the pre-widgets' method or when parameters and their number are different between the pre-widgets' method and the post-widgets' method. For example, instructions with the <indicate> label in Fig. 13 are of this type. In this example, the JRadioButton widget has no method corresponding to the "setLayoutOrientation(int)" of JList widgets for setting the display layout of items. In such cases, WREATH only indicates the necessity for modification.

5. Evaluation

We evaluated WREATH using four sample applications, namely, the video rental system, JFtp, JMail, and jabbaIRC. The video rental system was developed by us, and JFtp, JMail, and jabbaIRC are open source software tools from SourceForge.net [9]. We change GUIs in three types of change patterns: one-to-many, action inclusion and window division patterns.

5.1 Evaluation of Search for CtbM

We evaluated the search for appropriate CtbM using Eclipse [2] and WREATH. We consider that Eclipse can support GUI changes, because when the syntax of source programs is incorrect, Eclipse gives the candidates for modification. In Eclipse, we opened the source program files of the applications, changed the declaration of widgets, and counted the number of statements that Eclipse identified as having errors. In WREATH, we counted the number of statements that WREATH identified with CtbM.

The results are shown in Table 2. In this evaluation, we changed GUIs of each sample application by the window division pattern, action inclusion pattern, and one-to-many pattern. The values in each column are the sum of the number of CtbM of all GUI changes for sample ap-

Table 2 Evaluation of correctness of search.

	Video	JFtp	JMail	jabbaIRC
WREATH(a)	105	49	113	56
WREATH(b)	63	46	78	50
WREATH(c)	42	3	35	6
WREATH(d)	1	24	17	11
Eclipse	32	30	51	42
Manual	64	70	95	61

plications. "WREATH(a)" indicates the number of CtbM found; "WREATH(b)", the number of correctly shown CtbM; "WREATH(c)", the number of incorrectly shown code; and "WREATH(d)", the number of necessary CtbM not shown. "Eclipse" indicates the number of CtbM in changing GUIs using Eclipse. "Manual" indicates the number of CtbM in changing GUIs manually. In this evaluation, we consider the correct CtbM to be "Manual" in Table 2 and compare it with "WREATH" and "Eclipse". In this table, the value of "WREATH(a)" is the sum of "WREATH(b)" and "WREATH(c)". The value of "Manual" is the sum of "WREATH(b)" and "WREATH(d)". Also, all CtbM shown by Eclipse are included in "Manual".

According to the results, WREATH did not show some necessary CtbM in "WREATH(d)". One reason for this is that WREATH could not show container role widgets that are no longer required for replacing widgets. We consider that these unnecessary container role widgets should be deleted. WREATH will be able to suggest instructions for deleting these unnecessary container role widgets by improving the algorithms. The strategy of improving the algorithms is as follows: for unnecessary container role widgets resulting from GUI changes, WREATH assesses whether there are widgets on each container role widget after GUI changes. If no widgets are on a container role widget, WREATH shows the statements for the container role widget as CtbM, such as declaration and initialization of the container role widgets.

Another reason for not showing necessary CtbM is that WREATH cannot show statements that should be modified but are not relevant to the GUI code; for example, if a radio-button is selected, a value is set to a variable. In this case, it is also necessary to improve the algorithms, and we are now considering the strategies of improvement.

However, in this table, if the difference between "WREATH(b)" and "Manual" is small, our method for finding CtbM is considered to be effective. Also, if "WREATH(b)" is larger than "Eclipse", our method for finding CtbM is considered to be more effective than Eclipse. Hence, in this evaluation, it is important to compare "WREATH(b)" and "Manual", and also to compare "WREATH(b)" and "Eclipse". The comparison is shown in Fig. 14. This figure is a graph of "WREATH(b)", "Eclipse", and "Manual".

According to this graph, WREATH showed more CtbM than Eclipse. Also, WREATH showed more than 65% of CtbM in "Manual" in all applications. Therefore, WREATH can identify CtbM more correctly than Eclipse, and can find many correct CtbM compared with the manual

Fig. 14 Comparison of "WREATH(b)", "Eclipse" and "Manual".

Table 3 Results of measuring performances.

	Video	JFtp	JMail	jabbaIRC
Time(JavaML) (ms)	9140	44886	25578	21713
Time(CtbM) (ms)	719-1328	719-2641	609-1531	375-1188
ICCG	32	124	32	66
Call graph	74	1168	312	588
PDG	217	68	176	95
Instruction	5	5	5	5

method.

5.2 Evaluation of Processing Time

To evaluate the performances of our method, we measured the time taken to identify CtbM and suggest instructions, and counted the nodes of generated ICCGs, call graphs, and PDGs. The results are shown in Table 3.

In this table, Time(JavaML) indicates the time taken to translate source programs into JavaML and extract the types and variable names of widgets and windows on which widgets are placed. Time(CtbM) indicates the time for processes of finding CtbM and suggesting instructions after developers specify how to change GUIs. For Time(CtbM), the ranges between the shortest time and the longest time are shown. Also, Time(JavaML) and Time(CtbM) are shown as milliseconds, and they were measured on a computer with an AMD Athlon(tm) 64 processor of 3200+ and 512MB memory. The operating system of the computer was Microsoft Windows XP. ICCG indicates the number of nodes of generated ICCGs. Call graph indicates the number of nodes of generated call graphs. PDG indicates the number of nodes of generated PDGs. The numbers of PDGs in this table are equal to the number of candidate CtbM before identifying appropriate CtbM. Instruction indicates the number of types of suggested instructions. Also, in this evaluation, no errors occurred in WREATH.

In this table, the processes for Time(JavaML) are independent of how GUIs are changed. That is, Time(JavaML) is affected by conditions of computers in measuring, but, is unaffected by how GUIs are changed. Therefore, Time(JavaML) indicates the time for changing GUI once. The processes for Time(CtbM) differ according to how GUIs are changed. Also, to confirm the performance of our method, the time taken to change GUI once is considered to

Table 4 Evaluation of correctness of suggestions.

	Video	JFtp	JMail	jabbaIRC
(a)	63	46	78	50
(b)	61	38	77	49
(c)	61	36	67	46

be very important. Therefore, in Time(CtbM), the shortest and the longest times taken to change GUI once are shown together. ICCGs and call graphs are generated by static analysis of source programs, so their node numbers are always the same in each GUI change, and the number of nodes for each GUI change is shown. The numbers of PDGs differ according to how GUIs are changed. However, the total number of candidate CtbM for GUI changes in each application is considered to be more important than the number of candidate CtbM for each individual GUI change, because the number of candidate CtbM depends on various conditions, such as how GUIs are changed and the processed described in applications, and the effectiveness of our method cannot be determined from only the number of candidate CtbM themselves.

According to Table 3, Time(JavaML) and Time(CtbM) are considered to be practical, i.e., not fast, but not slow. Also, on comparing the numbers of PDG (candidate CtbM) with the numbers of found CtbM ("WREATH(a)" in Table 2), the numbers of found CtbM were much smaller than the numbers of candidate CtbM. This means that many code are found by several of the existing methods for searching for CtbM using PDGs, including many code that need not be modified. That is, WREATH can exclude such code that need not be modified, and shows only code that have high probabilities of requiring modification.

On the basis of to the above results, our method is considered to be effective and practical for searching for CtbM.

5.3 Evaluation of Instructions Suggesting How to Modify Code

We evaluate the correctness of the suggested instructions for how to modify the CtbM. The results are shown in Table 4, where (a) indicates the number of CtbM for which WREATH should suggest instructions (same as "WREATH(b)" in Table 2), (b), the number of CtbM for which WREATH actually suggests instructions, and (c), the number of CtbM for which WREATH suggests correct instructions in (b). For example, in JFtp, instructions of 46 CtbM should have been suggested, whereas WREATH suggested instructions of 38 CtbM. Among these instructions of 38 CtbM suggested by WREATH, instructions of 36 were correct instructions. Also, in this table, correct instructions mean the move, replace, add, and delete instructions described in Sect.4.4. That is, the numbers of CtbM with change-necessity instructions are included in (b), however, not in (c).

In some cases, WREATH cannot suggest instructions for modification, such as cases in which widgets are generated by particular methods or when specific methods of pre-

widgets, for which post-widgets do not have corresponding methods, are used. In the former case, WREATH may be modified to be able to suggest a certain level of instructions by improving the algorithms for searching CtbM and suggesting instructions. In the latter case, it is necessary to improve the WRT.

However, according to Table 4, WREATH suggests correct instructions for how to modify code at a rate of 78%-97%. These percentages are calculated by dividing (c) in Table 4 into (a). (a) indicates the number of CtbM for which WREATH should suggest instructions, and (c) the number of CtbM for which WREATH correctly suggests instructions. For the difference between the rates, two reasons can be considered: the appropriate methods that are not given in the WRT are used, and selection items are registered to combo-box widgets and list-box widgets using an array. That is, in some applications, many methods that are not given in the WRT are used, whereas they are merely used in other applications. Also, in some applications, selection items are registered to combo-box widgets and list-box widgets using arrays, and not in other applications.

An example of methods that are not given in the WRT is the `setEditable(boolean)` of `JComboBox` widgets. In replacing a `JComboBox` widget with a `JList` widget, because this `setEditable(boolean)` was not given in the WRT, only the change-necessity instruction is given. In such cases, the number of correct instructions suggested by WREATH might be increased by improving the WRT. However, corresponding methods or processes for the replaced widget do not necessarily exist for all methods defined in a widget. It is necessary to consider instructions for the case of the non existence of corresponding methods or processes.

Also, when selection items are registered to combo-box widgets and list-box widgets using arrays, change-necessity instructions are suggested. When these widgets are replaced with radio-button widgets, change-necessity instructions are suggested for assigning items to arrays. In this case, each item in the arrays should be shown as text on each radio-button widget. However, the suggested instructions in this case are not appropriate.

However, the percentages of 78%-97% indicate the rate of CtbM, for which instructions are correctly suggested among those for which instructions should be suggested. Therefore, WREATH is considered to be effective for modifying code.

6. Related Work

Some methods for supporting source program changes and impact analysis have been proposed. These include methods for the use of change history, call-graph-based and model-based impact analyses.

Usage of change history

Ying et al. proposed a method of identifying relevant source programs when modifying programs [10]. In their method, a set of files that previously were simultaneously

changed are recorded. When source programs are changed, the files that should be modified are shown in the history of the changed source programs. This method is effective when source programs that do not change impact sets are required to be modified. Zimmermann et al. also proposed a method of identifying relevant source programs when modifying programs [11]. In their method, programmers' histories of source program changes are saved. When a programmer changes a function, the system shows other functions that other programmers changed among with the former function. In these methods, CtbM can be shown as statements as in our method. In addition to this, concrete instructions on how to modify the source programs are suggested in our method.

Impact analysis based on method call

Law and Rothermel proposed a method of impact analysis using call graphs [12]. In their method, a call graph is dynamically constructed by tracing method calls. Impact sets are predicted by analyzing the histories of method calls. In this call graph, information on returns of methods is included. That is, impact sets can be predicted more precisely by this method than by other call-graph-based impact analyses. Badri et al. also proposed a method of impact analysis [13]. In their method, controls, such as if and loop statements, are added to call graphs (control call graphs), and impact analysis is performed on the basis of the control call graphs. These methods can identify many CtbM, however, the CtbM found may not always require modification. In our method, only the required CtbM are identified.

Class-based change

Abdi et al. proposed a method of impact analysis, in which a model that represents the relationships between classes is constructed [14]. The relationships between classes include references to variables of another class, the aggregation of the objects of another class, the inheritance of the methods and variables of another class, and the invocation of the methods of another class. Once a component is identified as requiring change, a set of classes is identified by tracing the links of the relationships of the classes. Tsantalis et al. also proposed a method of evaluating the probability that each class of an object-oriented system is affected [15]. In their method, the probability is evaluated by analyzing the mechanisms of change, such as the addition of a new method in an interface, a change in the signature of an existing method, a change in the signature of a constructor, and a change in any class or package name. When a new functionality is added or when an existing functionality is modified, the probability of change of each class is evaluated. In the above methods, CtbM are found at the class level. In our method, CtbM can be found at the statement level.

Model-based impact analysis

Hewitt and Rilling proposed a method of impact analysis at the specification level [16]. In their method, scenarios

that represent interactions between users and software are described by Use Case Maps (UCM) [17], and change impact analyses are performed by tracing UCM paths. Scenarios represent interactions between users and software. That is, this method is considered to be suitable for GUI changes. This method is at the specification level. On the other hand, our method is a source-program-level analysis.

7. Conclusion

In this paper, we proposed a method of searching for CtbM in a source program and suggesting instructions on how to modify them for GUI changes, and also showed an evaluation of our method. According to the evaluations, WREATH can show more required CtbM and modification instructions than other methods. Therefore, our method is effective for changing GUIs.

Future work includes the following issues.

- Ability to select a combination of change patterns
- Detection of other change patterns, if any
- Support for modifying the indirect relevant code of widgets related to GUI changes

References

- [1] J. Nielsen, Usability Engineering, Morgan Kaufmann Pub., 1994.
- [2] Eclipse, <http://www.eclipse.org/>
- [3] NetBeans, <http://www.netbeans.org/>
- [4] Java Swing, <http://www.java.sun.com/>
- [5] J.W. Cooper, Java Design Patterns: A Tutorial, Addison-Wesley Pub. Co. (Sd), 2000.
- [6] G.J. Badros, "JavaML: A markup language for java source code," Proc. 9th International World Wide Web Conference, 2000.
- [7] Jikes, <http://jikes.sourceforge.net/>
- [8] S. Horwitz and T. Reps, "The use of program dependence graphs in software engineering," Proc. 14th International Conference on Software Engineering, 1992.
- [9] SourceForge.net, <http://sourceforge.net/>
- [10] A.T.T. Ying, G.C. Murphy, R. Ng, and M.C. Chu-Carrol, "Predicting source code changes by mining change history," IEEE Trans. Softw. Eng., vol.30, no.9, pp.574-586, 2004.
- [11] T. Zimmermann, P. Weibgerber, S. Dichl, and A. Zeller, "Mining version histories to guide software changes," IEEE Trans. Softw. Eng., vol.31, no.6, pp.429-445, 2005.
- [12] J. Law and G. Rothermel, "Whole program path-based dynamic impact analysis," Proc. 25th International Conference on Software Engineering, 2003.
- [13] L. Badri, M. Badri, and D. St-Yves, "Supporting predictive change impact analysis: A control call graph based technique," Proc. 12th Asia-Pacific Software Engineering Conference, 2005.
- [14] M. K. Abdi, H. Lounis, and H. Sahraoui, "Analyzing change impact in object-oriented systems," Proc. 32nd EUROMICRO Conference on Software Engineering and Advanced Applications (EUROMICRO-SEAA'06), 2006.
- [15] N. Tsantalis, A. Chatzigeorgiou, and G. Stephanides, "Predicting the probability of change in object-oriented systems," IEEE Trans. Softw. Eng., vol.31, no.7, pp.601-614, 2005.
- [16] J. Hewitt and J. Rilling, "A light-weight proactive software change impact analysis using use case maps," Proc. 2005 IEEE International Workshop on Software Evolvability (Software-Evolvability '05), 2005.
- [17] R.J.A. Buhr, "Use case maps as architectural entities for complex systems," IEEE Trans. Softw. Eng., vol.24, no.12, pp.1131-1155, 1998.

IEEE and ACM.

Junko Shirogane received the B.E., M.E. and D.E. degrees in information and computer science from Waseda University, Tokyo, Japan, in 1997, 1999 and 2002, respectively. She joined Media Network Center of Waseda University as a Research Assistant in 2000 and Department of Communication of Tokyo Woman's Christian University as a Assistant Professor in 2003. Her research interest includes support tools for development of software with graphical user interface. She is a member of IPSJ, JSSST, HIS,

Hajime Iwata received the B.E. and M.E. degrees in information and computer science from Waseda University, Tokyo, Japan, in 2002 and 2004, respectively. He joined Department of Computer Science of Waseda University as a Doctor Course Student in 2004 and Media Network Center of Waseda University as a Research Assistant in 2005. His research interest includes support tools for learning operating method of applications. He is a member of IPSJ Japan.

Kazuhiro Fukaya received the B.E. and M.E. degrees in information and computer science from Waseda University, Tokyo, Japan, in 2003 and 2005, respectively. His research interest includes support tools for development of software with graphical user interface.

optimization and computer aided design. He is a member of IPSJ, JSSST, ACM and IEEE.

Yoshiaki Fukazawa received the B.E., M.E. and D.E. degrees in electrical engineering from Waseda University, Tokyo, Japan, in 1976, 1978 and 1986, respectively. He joined Department of Computer Science of Sagami Institute of Technology as a Lecturer in 1983 and Department of Electrical Engineering of Waseda University as an Associate Professor in 1987. He is now a Professor of Department of Information and Computer Science, Waseda University. His research interests include software engineering, program