Enabling Integrated Development Environments with Natural User Interface Interactions

Denis Delimarschi, George Swartzendruber, Huzefa Kagdi
Department of Electrical Engineering and Computer Science
Wichita State University
1845 Fairmount St.
Wichita, KS 67260, United States of America
{dxdelimarschi, gnswartzendruber, huzefa.kagdi}@wichita.edu

ABSTRACT

The paper introduces the concept of applying Natural User Interface (NUI) interactions in the context of Integrated Development Environments (IDEs). Human voice and gestures are mapped to several IDE commends. A prototype tool is developed using the Microsoft *Kinect* hardware sensors and the available software development kits for Microsoft Visual Studio. A pilot study was conducted to assess the developed prototype. The results of the study suggest that it might be possible to apply natural interactions to a range of IDE capabilities.

Categories and Subject Descriptors

D.2.6 [**Programming Environments**]: Interactive Environments

General Terms

Experimentation, Measurement, Human Factors, Design

Keywords

Development Process Optimization, Gestures, IDE, NUI, Speech

1. INTRODUCTION

Natural User Interface (NUI) is an umbrella term that encompasses human-computer interactions with devices that are amenable to humans and their natural surroundings. The basic premise here is to let computers understand the innate human means of interaction (e.g. voice and gestures) and not induce humans to train to the language of computers (e.g., keyboard and mouse). To a large extent, NUI has become a popular paradigm with the widespread popularity and use of portable devices, such as *iPods* and Smartphones, as well as interactive entertainment consoles, such as *Xbox*. *Xbox* facilitates the application of natural habits and interaction approaches via the *Kinect* hardware sensor. This trend raises a number of interesting and important research questions in the context of software engineering research. These include, but are not limited to:

1. How well can NUI devices function with the IDEs that developers use?
2. Would developers find NUI tools acceptable as companions to their existing toolset of choice?
3. What impact would applying NUI have on the productivity and workflow of developers?
4. How much improvement, if any, does applying NUI offer to the effectiveness of software engineering tasks and software quality?

Currently, IDEs are largely limited to developer interaction via traditional hardware devices (e.g., the ubiquitous keyboard and mouse). Although developers are accustomed to these devices, there is a clear set of limitations that those devices do not solve, specifically related to extending the discoverability and accessibility of features offered. In this scenario, NUI leverages behaviors and habits that the user has already learned and does not require any special training – this includes voice commands, touch, and gestures. For example, a developer might want to iterate through breakpoints via a voice command, or use the visual application designer coupled with hand gestures to allow him to position different controls on different parts of the screen. Following this pattern, it is possible to recognize the developer's intent and attempt to pick or suggest the best course of action. For example, if the developer says *"analyze performance"*, the voice controller will recognize two key terms and attempt to locate capabilities in the IDE that perform analysis and possibly are related to performance.

It is a possibility that applying NUI to a software engineering environment could increase both the accessibility and discoverability of a wide variety of features available within an IDE. With the increased complexity and growing functionality of development tools on different platforms, software engineers often encounter issues accessing all those capabilities as well as leveraging them fast enough so as to not disrupt the development process. This work is directed to address these issues.

The software lifecycle is usually organized in several phases, including design, development, deployment, maintenance, and possibly retirement. Modern Integrated Development Environments, such as Microsoft Visual Studio, are

ICPC '14, June 2âĂŞ3, 2014, Hyderabad, India

capable of supporting any or all of these phases. These tools have extensible and flexible architectures that allow for their highly sustainable evolution. It allows third-party developers to build tools that can increase the efficiency of any or all the phases of the software evolution by being open to integrating third-party modules that impact the development process. Our tool, namely *KinectCommander*, was designed to take advantages of this extensibility framework.

1.1 Usage Benefits

Integrating a NUI layer in a development environment carries several potential benefits. We discuss a few representative examples below:

1. **Increased feature discoverability**. Oftentimes developers do not have enough time or resources to spend on researching documentation, thus leaving large components of the IDE unused, even though those might benefit them in the process. With voice integration, the developer can search for specific capabilities and the middle layer will be able to detect and potentially infer features of the IDE that the developer wants to use to accomplish a specific task. This goes beyond simply calling the IDE search function and allows feature inference from a global tool feature list.
2. **Improved interaction time**. Instead of using keyboard and mouse input, the developer might produce voice commands that can activate a wide variety of features, such as debugging, performance analysis, set breakpoints and iterate through them or fall back on simple routine procedures, such as saving and opening solution files.
3. **Increased levels of accessibility of development tools for underprivileged categories of developers**. A core target group is disabled individuals who are not able to leverage standard input controllers, such as the keyboard and mouse, and are able to use voice to create and manage the development flow.

2. APPROACH

Using a set of proprietary libraries, we are detecting gestures and speech, translating them to commands that are recognizable by the IDE, and using them for scenarios such as debug process control or atomic program component creation. We describe our approach along with our tool, namely *KinectCommander*, which behaves as a proxy between native system calls and Microsoft Visual Studio. The action flow is shown in Figure 1.
The execution flow is split into five (5) core tasks:

1. **Capture.** The connected *Kinect* device is constantly capturing new audio and imaging data when activated. It is essential that new data is fetched with as low latency as possible – the hardware and the official SDK handle this aspect.
2. **Stage 1 processing.** At this point, the data is proxied to one of the designated channels, depending on the type. In the case of audio captured from the microphone array, the data will be passed to the Microsoft Speech SDK[1] to check whether the data can be transformed into a recognizable word or sequence of words.

[1]http://microsoft.com/en-us/download/details.aspx?id=14373

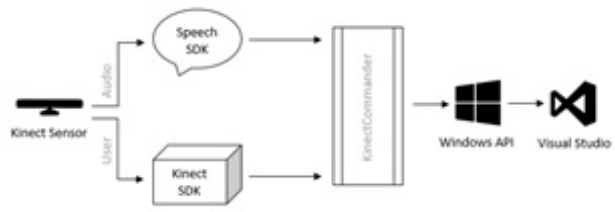

Figure 1: The execution flow for KinectCommander, where the originating hardware layer is using the proxy to determine which interactions triggers translate into viable commands, be that voice or gestures.

   For imaging data, the *Kinect* SDK picks up the processing responsibilities.
3. **Stage 2 processing.** Data is passed to *KinectCommander* to run against an internal database of commands, both for gestures and voice. The database contains bindings to environment features and potential activation combinations (e.g. keystrokes). Depending on the developer's setup, *KinectCommander* will default to a declarative engine, where data is declared in a series of XML files that can be built with a helper composition tool. Each XML file can be associated with an activity context (e.g. visual designer, code editor, and performance analyzer).
4. **Windows API communication.** At this point, *KinectCommander* has recognized either the audio or the gesture command and needs to pass it to Visual Studio. Because of the flexibility goal set for our application, instead of proxying the data through the Visual Studio pipe, the Windows API is used. In that way, third-party components that are leveraged by the IDE can be triggered outside the IDE itself.
5. **Visual Studio feature activation/trigger.** The Windows API will simulate the needed keystrokes to activate the requested feature, and Visual Studio will react accordingly.

We decided to create a clear separation of concerns, where voice commands are used to leverage quick commands and gestures are used to perform content manipulation tasks.

2.1 Voice Commands

Primarily, voice commands are mapped to elements of the IDE that need additional user input, such as text. These elements can also be triggered by voice commands that can be executed after the original command was recognized and processed. This capability is introduced to increase discoverability and let the user track down features and components they were not aware of yet. Voice command capture starts with the primary recognition of the word *'kinect'*. Examples of voice commands include:

1. **Find** – used to trigger the built in search and replace capability of the development environment. Can be used in the context of any project, independent of the selected programming language.

2. **Replace name** – used to find references to a specific object or variable name rather than a plain string reference and replace it in the context of the working project. Currently implemented to work with C# and F#.

3. **Add item** – opens the "Add New" dialog in the IDE that allows the developer to add an additional file or pre-built component to the project.

4. **Performance** – suggest several IDE capabilities that let the developer execute application performance diagnostics. Because of platform dependencies, the performance analysis tools will be independently selected for each project type.

Generic voice commands are used to activate menus and perform basic application manipulation commands, such as *File*, *New*, *Save All* and *Exit*.

2.2 Physical Gesture Commands

Gesture commands are mapped to direct visual manipulations in the integrated Visual Studio GUI designer, which allows the developer to drag controls on the surface with the help of skeletal detection of arm joints. Integration in the plain code editor is also possible. Examples of gesture capabilities include:

1. Move and resize specific sub-windows in the context of a project.

2. Placement and modification of built-in Windows components in Windows Forms and WPF projects during presentations.

3. Iterating through breakpoints by capturing minor horizontal head movement.

4. Scrolling through code following the developer's vertical and horizontal head movement.

Gestures can be re-programmed and easily re-mapped with the help of a custom-built gesture capture and storage tool.

3. A PILOT STUDY

The core goal of the pilot study was to test the prototype in the context of IDE commands that would be initiated by a developer with the help of voice and gestures.

3.1 Process

Five test subjects were asked to use voice and gesture commands to trigger a variety of elements related to the development process, in and outside the IDE. Participants are familiar with at least one programming language – though not necessarily the one used in the experiment – and an IDE. The *KinectCommander* service was running in the background, tracking body movement and speech. After a short introduction to the tool and some basic voice commands and gestures, each subject was given 15 minutes of active participation time to attempt controlling the IDE, while the researchers observed them in the same room.

Participants were given specific tasks, where they had to refactor pieces of existing code, trigger application performance analysis tools and add components to the existing project.

Overall, refactoring code components went without any issues, the biggest problem being the proper pronunciation and detection of object names and their respective counterparts. Adding third party libraries was speedier and more precise, given the fact that the voice commands were much shorter and exact – it was possible to infer the intended action given the available options even if the pronunciation was not correct.

Once the voice part of the experiment was completed, participants were asked to leverage gestures to manipulate the content of the IDE in terms of elementary operations such as closing or moving the debug and console windows, resize and place a given page in a different container as well as place pre-built Windows controls on an existing development surface. An introductory guide in the *KinectCommander* launch screen showed how to trigger the gesture mode and how to deactivate it, either by a gesture itself, a voice command or a built-in button in the IDE toolbar.

We have noticed that the best performance from the Kinect sensor is obtained when it is placed above the monitor, therefore being able to capture the developer's head as well as the upper body (arm) movements. Sensor placement did not impact voice capture as long as the sensor was in close proximity to the developer.

3.2 Observations

The study showed that the system has potential in meeting the objectives initially outlined. One of the core elements that proved to be the most useful in the context of the study was related to voice commands and discoverability of features previously not known by the developers that were using it. For example, the performance analysis tools would show up as a suggestion to developers that mentioned the term "*performance*". Otherwise, they would have to look for it through the menus in a new IDE. This also applied to a wide variety of refactoring tools that allowed variable name changing and reference updates.

The big challenge for this study was recognizing between voice commands and speech that was not intended to be covered by the tool. The typical development environment is rarely fully silent, with a lot of interference from potential third parties. This has proven to be an issue if words are spoken in addition to the existing commands that the developer himself has not triggered.

Gesture discoverability had a lower rate than voice, taking as many as 4 to 5 attempts to complete, compared to 2 voice command attempts.

Used in a standard development environment, both gestures and voice were successfully mapped to a predefined set of commands exposed through the IDE extensibility layer. The feedback received from the subjects indicated that for the test group, voice commands were more useful than gestures, being more intuitive and easier to infer based on previous knowledge.

4. DISCUSSION

4.1 Additional Applications

In addition to being applied as the interactive layer connected to an integrated development environment, *KinectCommander* can be easily extended and adapted to interact with other applications, such as productivity software or visualization tools. Modifications should be made to the com-

mand bindings and gesture reactions, however, not requiring major modifications to the tool itself, since GUI adaptation is handled through a custom-built learning tool.

4.2 Limitations

Current implementation is limited to the Microsoft Visual Studio Integrated Development Environment, running in the Microsoft Windows operating system. The NUI layer integrates with any project created through Visual Studio with a default configuration for commands and tools. This might have an impact on developers who want to leverage the tool but are working with a different language and/or operating system.

4.3 Challenges

One of the biggest challenges encountered are false positives in terms of voice recognition. The used speech toolset offers some leeway in terms of vocal capture and oftentimes a word that was said by the developer is recognized as a command even though it was not one. This can be potentially detrimental to the process in team development environments – a conversation does not always need to produce an executable command.

Another challenge is internationalization. Currently, *KinectCommander* is tested and designed for the English (US) cultural conventions and linguistic accent. The system will not work as intended with heavy international accents and other languages. This limitation does not apply to gesture recognition.

5. RELATED WORK

The NUI concept is not new, and we rely on research performed in the context of other domains beside software engineering.

Marsic *et al.* [5] discuss the generalized implementation of a system that can be controlled with the help of voice commands, gestures and facial expressions. Leopold *et al.* [4] discuss a system implementation that relies on speech recognition and pen input, correlating individual object tags and pen gestures with pre-defined behaviors. Unlike *KinectCommander*, it does not include body gestures but rather emulates that behavior through pen input.

As we were developing the prototype, one of the primary questions we asked is whether the offered capabilities are precise enough for the user to feel any difference in the amount of work accomplished with the help of a radically different input system. Begel *et al.* [1] mentioned that it is possible that the developers are less efficient by using a natural interface instead of typing. This is related to the fact that typing is already ingrained in every developer's habits, while more natural interactions might not be as standard in the development process. We discovered that one of the aspects of the system we are working on, gesture recognition, was tested in an experimental environment by Fang [2] in the context of the two-string problem that demonstrated that ultimately spatial pattern matching for an arbitrary group of individuals has similar precision to the scenario where a standard input component, such as the mouse, was used.

The core of *KinectCommander* remains its voice-interpreting engine that translates decoded language elements into visible actions. An implementation of a basic voice interaction layer on top of an existing operating system was documented by Kader *et al.* [3] but relies on a very primitive control mechanism that requires the user to be very clear and explicit about the intentions in the actionable area. *KinectCommander* does not have strict command rules and allows the developer to customize the set of actions based on an unlimited number of commands, as long as those can be detected with a relatively high precision by the underlying SDK.

6. CONCLUSIONS & FUTURE WORK

We introduced the concept of applying Natural User Interface (NUI) interactions in the context of Integrated Development Environments (IDEs). We developed a prototype tool and conducted a pilot study to assess its feasibility. Despite the challenges and current limitations, we believe that it has the potential to improve development, tool discoverability, and ease of use. To make the tool more reliable and applicable, we have set forth the following goals:

1. Implement a more reliable speech recognition mechanism that would allow for localization.
2. Integrate learning capabilities that would allow the system to adapt to the user's pronunciation and gesture speed.
3. Extend the system beyond Visual Studio, on a variety of operating systems.

More details on the project, including video demonstrations, are available on the Wichita State University Software Engineering Research Lab website[2].

7. REFERENCES

[1] A. Begel and S.L. Graham. An assessment of a speech-based programming environment. In *Visual Languages and Human-Centric Computing, 2006. VL/HCC 2006. IEEE Symposium on*, pages 116–120, Sept 2006.

[2] Wei-Chieh Fang, Yu-Lun Lin, Feng-Ru Sheu, and Nian-Shing Chen. Exploring problem solving performance through natural user interfaces. In *Advanced Learning Technologies (ICALT), 2013 IEEE 13th International Conference on*, pages 232–234, July 2013.

[3] M.A. Kader, B. Singha, and M.N. Islam. Speech enabled operating system control. In *Computer and Information Technology, 2008. ICCIT 2008. 11th International Conference on*, pages 448–452, Dec 2008.

[4] J.L. Leopold and A.L. Ambler. Keyboardless visual programming using voice, handwriting, and gesture. In *Visual Languages, 1997. Proceedings. 1997 IEEE Symposium on*, pages 28–35, Sep 1997.

[5] I. Marsic, A. Medl, and J. Flanagan. Natural communication with information systems. *Proceedings of the IEEE*, 88(8):1354–1366, Aug 2000.

[6] Shairaj Shaik, Raymond Corvin, Rajesh Sudarsan, Faizan Javed, Qasim Ijaz, Suman Roychoudhury, Jeffrey G. Gray, and Barrett R. Bryant. SpeechClipse: an Eclipse speech plug-in. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 84–88, 2003.

[2] http://serl.cs.wichita.edu/?page_id=166