MICE – Capturing Programming Styles in a Mixed-Initiative Coding Environment

Shilpi Rao[*], Vive Kumar[#], Marek Hatala[*], Dragan Gasevic[*]
[*]*School of Interactive Arts and Technology, Simon Fraser University, Canada*
[#]*Institute of Information Sciences and Technology, Massey University, New Zealand*
srao@sfu.ca, v.s.kumar@massey.ac.nz, mhatala@sfu.ca, dgasevic@sfu.ca

Abstract

*Programming Style refers to the ability to follow code conventions, to engineer code in a disciplined manner, to systematically debug code, to optimize code delivery through appropriate settings in the IDE (Integrated Development Environment), to regulate completion rates and quality of programming tasks, and finally to efficiently collaborate with other programmers and resources. This research investigates whether programming styles of individual programmers can be computationally recognized; If styles can be recognized by the machine, can they then be regulated so that programmers can reflect on their own programming styles; finally, can a mixed-initiative computational mechanism assist programmers to identify good programming styles and repair bad programming habits. The paper proposes a real-time architecture called MICE (**M**ixed-**I**nitiative **C**oding **E**nvironment) that has been developed to tackle this research question. The architecture uses ontologies to model-trace programming styles, employs rules to assist programmers to regulate their programming styles, and engages mixed-initiative scaffolding tactics and strategies to provide feedback.*

Keywords: ontology, model tracing, programming style, coding convention, debugging, self-regulated learning, mixed-initiative interactions, real-time processing

1. Research Outline

Computer programming potentially involves skills in *understanding* (application context and possibility), *planning* (design), *imaging* (imagination and visualization), *attitude* (acceptance of work involved and confidence in completing projects), *logic* (conceptualization, language use, and knowledge), *creativity* (artistry) and *work* (persistence, exploration, purpose and commitment) [24]. While applying these skills to develop code, programmers acquire individual styles. We define *Programming Style* as processes that a programmer adopts, over a period of time, to achieve specific programming goals. Programming Style, among other aspects, refers to the ability of programmers

- to follow code conventions,
- to engineer code in a disciplined manner [8][14],
- to systematically debug code [26],
- to optimize code development and delivery through appropriate settings in such as IDE (Integrated Development Environment),
- to regulate completion rates and quality of programming tasks,
- to efficiently collaborate with other programmers and resources, and
- to follow software engineering principles.

The abovementioned components mold the blueprint of what we consider as the Programming Style of an individual programmer. This research aims to develop a system to assist programmers regulate their coding styles when they code in Java using an Integrated Development Environment (IDE). The proposed system called MICE, abbreviated for Mixed-Initiative Coding Environment, targets three objectives:

- First, MICE aims to capture programming style components from interactions of programmers when they develop task-specific code. It exploits the Model-Tracing techniques to map interaction data to style components [2]. The traced data updates programmers' models accordingly.
- Second, MICE aims to engage programmers in Mixed-Initiative (MI) interactions. Mixed-Initiative strategies enable conversants, in our case, programmers and the MICE system, to contribute appropriate information, when it is best suited, towards mutually negotiated goals [11]. This means that both MICE and a programmer can share their respective goals with each other as well as initiate a feedback process independent of each other [1], [5], [12].
- Third, MICE aims to inculcate the theory of Self-Regulated Learning (SRL) in its feedback mechanism. SRL views learning as an activity that students perform proactively, rather than as a covert

event that happens to them in reaction to teaching [27]. MICE embeds an ontological representation of Zimmerman's SRL model, which includes the phases of forethought (planning, task analysis, self-motivation, goal setting), performance (self-control and self-observation), and self-reflection (self-judgment, self-reaction, self-evaluation, self-satisfaction) [25].

Targetting these objectives, MICE initiates one or more of the following feedback methods *at opportune moments*:

- Engage the programmer with a pre-defined conversation model [18], [20], [9], [10]
- Introduce the programmer to ready, able, and willing human helpers [15], [28]
- Provide the programmer with timely clues and hints [29]
- Scaffold the programmer in a guided practice session [4], [16]
- Offer the programmer SRL-specific feedback [22], [21].

We designed the MICE system as an ontology-centric framework consisting of two key ontologies: first, a Programming Style ontology that corresponds to the components of Programming Styles identified earlier; second, an Interaction ontology that captures interactions of a programmer within the MICE environment. The Interaction Ontology records interactions of programmers with the MICE system. Interactions of each programmer can be extracted from the ontology. The Interaction Ontology will be used to infer instances of specific programming styles using Description Logic or Production Rules. These inferred styles are updated in the Programming Style Ontology as and when they are recognized by MICE. We believe that each programmer may entertain one or more programming styles and MICE will be in a position to identify these styles by observing programmer interactions over a considerable period of time. In addition to these two, MICE also avails ontologies such as an ontology of a model of SRL, a Learner ontology, and a Time ontology.

MICE makes use of all the above mentioned ontologies to profile interaction data that are captured at run time. MICE then processes these data in a reactive manner and updates the facts in a production rule system. These facts trigger specific rules that initiate feedback processes.

The next section discusses the programming style components. Section 3 briefly discusses the architectures of MICE. Section 4 explains the key ontologies in MICE. Section 5 outlines what we mean by system-initiated feedback and how we plan to employ such feedback in MICE. The final Section of the paper concludes our views and also outlines the scope for future work.

2. Programming Style

In this section we define all the components of programming styles as well as give a research background for each of them. Before we describe them, we first introduce some important presumptions related to programming styles on which we based our research.

The first presumption is that a programmer does not have to adhere to a single recognized programming style. He/She can use one or more styles. Instead of attempting to stereotype the observed styles of a programmer to a particular type, the scope of MICE enables to identify a variety of styles exhibited by a programmer, over a period of time. Importantly, these observed styles are stored in an ontological form and MICE can communicate with the programmer about the changes in his/her programming styles over a period of time across different contexts.

The second presumption is that there is no one good programming style. The best-suited programming style/s may vary from programmer to programmer. The feedback given to programmers now-a-days is limited to syntactic errors and code conventions. Such feedbacks are mostly summative in nature; that is, they are provided not during code construction but mostly at compile time. Feedback could be given to programmers based on their code design, their code presentation style, and their debugging style [23], [7], [2] in a formative fashion.

We extend the notion of programming style in two main aspects: first, programming style is a process and hence can possibly change from context to context over a longer period of time; second, programming style includes a number of other factors outlined below, and a programmer can engage in one or more of these factors that determines his/her programming style.

2.1 Code Conventions

Code Convention is defined as the ability of a programmer to adhere to specific conventions prescribed for a particular language. For example, Java Code Conventions include usage of tabs, indents, blank lines, spaces, alignments, braces, wrapping, naming, file organization, documentation, language construct statements, and imports[1].

---

[1]http://jalopy.sourceforge.net/existing/links.html, http://java.sun.com/docs/codeconv/

Many Java code convention checkers/verifiers are available in public domain[2] as well as in the commercial market[3]. However, almost all these software only provide summative feedback as opposed to MICE's formative mixed-initiative feedback approach. Also, unlike MICE, the conventional checkers and verifiers do not allow programmers to change existing conventions to their liking.

2.2 Code Engineering

Code Engineering in MICE is defined as the ability of a programmer to construct code in a disciplined manner, preferably using sound software engineering principles. Normally, code engineering, among others, involves designing code, typing-in or pasting-in language constructs, compiling, version control, code refactoring, and using templates/patterns.

2.2.1 Code Construction

Kumar [14] and Doherty et al. [8] discuss a simple tool that recognizes code construction styles of programmers based on compile-time code segments (CT-SEG). Compile-time code segments are code (partial or complete) submitted to the compiler by programmers for verification of correctness. That is, every time a programmer submits code for compilation a version of the code (CT-SEG) is saved, thus enabling the tool to trace the ability of the programmer to incrementally construct code.

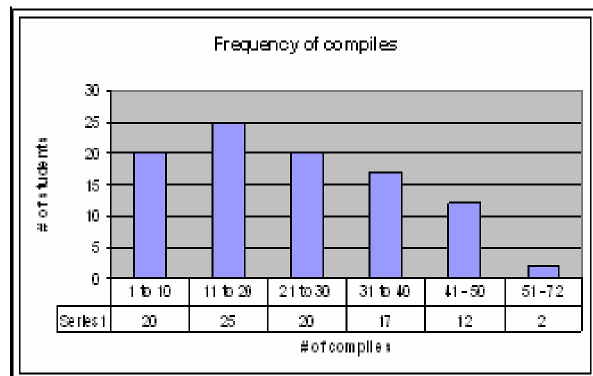

Figure 1. Distribution of frequency of compiles

A study conducted by Kumar [14] shows that the number of CT-SEG and the pattern of CT-SEG vary across programmers. That is, programmers compiled their code at varying time intervals. Figure 1 shows the distribution of frequency of compiles among 96 programmers in an experimental setup.

In the same study, Kumar [14] also observed the lines of code (LOC) between compiles. For example, some programmers compiled code at the end of subtasks while progressively increasing the number of lines of code (Figure 2).

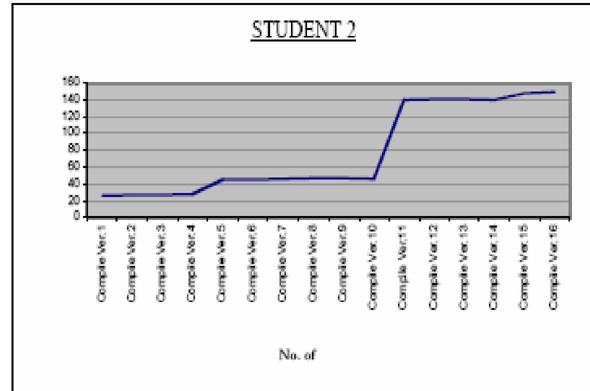

Figure 2. Variation in LOC between compiles

The study also showed code construction styles of programmers where they compiled only when the code for the entire task was completed. Also, there are programmers who tend to discard large chunks of code when the code developed so far failed to deliver results at subtask levels (Figure 3).

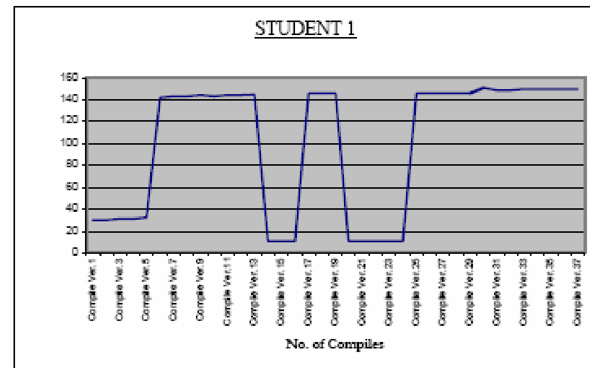

Figure 3. Variation in LOC between compiles

Yet another code construction style identified by the study shows how programmers develop code using different language constructs across compiles. Constructs from Java's Abstract Syntax Tree[4] such as comments, control structures, function declarations, variables, and so on were identified in each CT-SEG. The study showed that, for the same task, participants

[2] http://checkstyle.sourceforge.net, http://www.tiobe.com/jacobe.htm, http://pmd.sourceforge.net/

[3] http://www.jindent.com/

[4] https://jabstract.dev.java.net/

of the study, in general, employed a variety of language constructs for programming. Programmers changed the constructs significantly in between compiles when faced the task of debugging a considerably large numbers of errors and warnings. The study showed that a programmer's behavior can be tracked as a function of change in language constructs in specific debugging contexts. We believe that such syntax-level tracking can further be advanced to semantic-level tracking.

2.2.2 Code Quality

Code is expected to be of good quality. That is, it should be less complex, should have undergone rigorous testing, should have been refactored, and should have been critically analyzed by code experts. MICE enables programmers to reflect on how these factors affect the quality of their code.

By less complexity we mean, easy to read, easy to understand, easy to extend, easy to maintain, easy to test the code, and so on. Further, code can be made more readable and less confusing by removing unreachable methods and redundant fields from the code [30, 19]. Lines of Code (LOC) method and Function Points Analysis (FPA) are commonly accepted complexity determination methods[5] that can be easily incorporated in MICE. Rather than simply presenting the LOC and FPA results, MICE attempts to present this information to the programmer only when the moment is right. For example, the FPA value of a Java method under development along with a commentary on the trend of the programmer to write complex methods can be presented to the programmer when he/she attempts to send code for an official code review.

Testing plays an important role in determining the quality of code. More number of errors can be detected during testing if the test cases are more varied and explore more number of test paths. Inviting programmers to explicitly tag code, MICE can remind programmers about the importance of testing and also about their current testing habits. MICE can provide an analysis of the code with respect to testing, depending on pieces of code that have been tested. For example, programmers can tag pieces of code that have not been tested at all, that have been 'unit' tested thoroughly, that have been 'integration' tested partially , and so on.

Refactoring is yet another important aspect of coding. Refactoring is the process of rewriting a computer program or other material to improve its structure or readability, while explicitly preserving its meaning or behavior. In Software Engineering, the term *refactoring* means modifying source code without changing its external behavior[6]. Examples of refactoring include modifying all import statements in all java files when a file is moved from one package to another and changing all references to the class type when a class is renamed. Refactoring tools[7] save much manual work in coding that is necessary, tedious, and time-consuming. However, rather than simply performing refactoring behind-the-scenes, MICE externalizes refactoring outcomes and presents a summary of the same to programmers. This will be more helpful for students learning to program, with refactoring in mind, rather than for expert programmers.

In general, it is quite possible to critically and automatically analyze code using tools[8] and suggest good design and style improvements. Rather than simply and passively presenting these suggestions to programmers, MICE can present these suggestions at opportune moments with *explanations*[9] induced by the theory of SRL.

2.3 Code Debugging

Debugging is an art and is associated closely with code-engineering. However, because of the complexity involved in tracing programmers' debugging tactics and strategies, we treat code debugging outside the scope of code engineering. Typically, programmers employ a range of automated debugging techniques [26] that are listed below.

- Delta debugging – automatically narrows down the difference between a passing and a failing run
- Program slides – separates the part of a program or program run relevant to the bug
- Observing state – uses a debugger to observe the values of variables
- Watching state – uses a debugger to watch small parts of the program state to determine if they change during execution
- Assertions – uses comparison of observed values with the intended values when observing a program state

[5] http://www.verifysoft.com/en_code_complexity_meas ures.html, http://www.gotdotnet.com/Community/UserSamples/Details.aspx?SampleGuid=791516bd-b0ef-491a-be1e-0d622776197b

[6] http://en.wikipedia.org/wiki/Refactoring

[7] http://jrefactory.sourceforge.net

[8] http://www.serc.nl/people/florijn/work/designchecking/RevJava.htm

[9] To explain the basis of MICE's suggestions

Most of these automated debugging techniques do not capture debugging patterns over a period of time. In MICE, we are interested in observing how well programmers are able to debug code in between compilations. That is, the number of type errors and warnings produced by the compiler can be stored whenever a programmer submits code for compilation. MICE can track a programmer's errors and warnings across multiple compiles and record whether he/she tries to solve errors and warnings as soon as they appear, or debugs only a select few errors and warnings, or continues coding without correcting them.

Most programmers tend to ignore all warnings, but attempt to solve a few errors listed by the compiler [14]. From our personal experience, we recognize that expert programmers tend to develop a range of debugging skills; particularly skills that help them identify specific errors and/or warnings that maximizes their productivity. This research attempts to track errors and warnings resolved by programmers across compiles in an effort to identify the debugging styles of the programmers.

The study conducted by Kumar [14] indicates various patterns of debugging. Most of the participants in the study tried to eliminate errors as soon as they appeared and completely neglected the warnings. When these errors and warnings were compared with the LOC across various compiles, a pattern that indicated a marked change in LOC was observed. This change in LOC can vary from changing a few lines of code to changing or eliminating a major portion of the code, depending on the programmer's debugging style.

MICE records and presents observations on the debugging patterns of programmers that usually go unnoticed. Further, MICE also presents experts' debugging behavior, as case studies, to programmers.

At this time, the design of MICE is restricted to observing and recording debugging patterns of programmers. The correlational and causational effects that exist between code engineering and code debugging processes of programmers will be explored elsewhere, as part of the first author's thesis.

2.4 Optimal IDE Settings for Coding

The Integrated Development Environment (IDE) plays an important role in programmer productivity. An IDE is an environment that integrates multiple software engineering toolkits and presents the same to the programmer in a single interface. For example, the IntelliJ IDEA[10] IDE integrates and customizes a number of toolkits including project management, appearance, language editor, code compilation, compiler errors, colors and fonts, libraries, debugger, resources, IDE history, templates, plugins, and intention settings (Figure 4). One of the key goals of MICE is to be able to guide programmers towards an optimal IDE setting to suit their individual programming styles based on pre-defined models of IDE settings of experts.

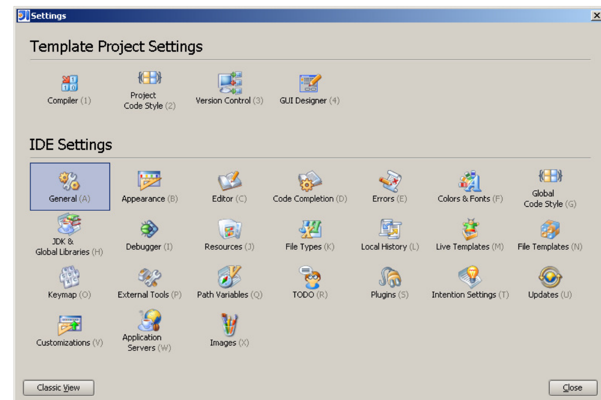

Figure 4. A settings of an IDE: IntelliJ IDEA

2.5 Regulating Coding Tasks

Programming tasks can be classified across different dimensions. Programming tasks and subtasks can be classified based on Vaid's [24] interpretation of cognitive tasks in programming that include *skills, planning, imaging, attitude*, *logic*, *creativity*, and *work*. Bloom's revised taxonomy could also serve to classify programming tasks in the cognitive dimension. For instance, a programming assignment could include components that explicitly demand students to exhibit their coding skills with respect to *remembering*, *understanding*, *applying*, *analyzing*, *evaluating*, and *creating* language constructs[11]. Leopard Tutor [13] classifies task under program readability, program understanding, program tracing, and program debugging.

Following the footsteps of the Leopard Tutor, MICE encourages programmers to construct task models before they start to code and also to track their progress with respect to the task model. Based on the time estimates provided by the programmers themselves, MICE presents proactive and non-intrusive feedback about the speed of their coding and a probabilistic

[10] www.jetbrains.com/idea

[11] http://rite.ed.qut.edu.au/oz-teachernet/index.php?module=ContentExpress&func=display&ceid=29

estimate of when the system expects them to complete the code.

2.6 Collaboration While Coding

Effectively collaborating with colleagues is crucial in extreme programming[12] and other agile software development methodologies[13], but is also important in normal coding environment. For example, a programmer may casually shout across the room for clarification on a particular type of bug or share code with a chat friend for an informal code review. A number of tools support collaboration in terms of chat, discussion boards, and so on. In our view, these tools support collaboration *passively*. By this we mean that these tools do not actively promote and guide users in appropriate and productive collaboration strategies and tactics. Morris et al [17] discuss ways in which the software can passively as well as actively promotes collaboration. Based on iHelp's model of code collaboration [3], the interaction interface of MICE has been designed so that programmers can share code with each other and critique the same under a guided environment.

3. MICE Architecture

We present MICE's architecture under two categories: functional and technical.

3.1 The Functional Architecture of MICE

The functional architecture describes the flow of functionality in the system. As depicted in Figure 5, the flow starts with programmer interactions in an IDE. The current MICE software uses BlueJ as the IDE. These interactions trigger events to instantiate appropriate elements in MICE ontologies. Changes in ontologies trigger execution of rules. Specifically, the purpose of MICE rules are threefold:

1. rules are used to computationally recognize programming style components;
2. rules are used to identify opportunities for system-initiated interaction, opportunities such as programmers spending too much time debugging a piece of code or programmers consistently failing to construct task models [22];
3. rules are used to engage programmers in mixed-initiative dialogues [1], [5] with MICE. For example, the MICE system and the programmer can engage in a well-defined, role-playing conversational model when situations warrant such interactions.

Feedbacks of MICE are marshaled at real-time to the BlueJ IDE as well as to external systems (e.g., iHelp and gStudy). In return, events observed at external systems are recorded in the Interaction Ontology. For example, collaboration sessions in iHelp between programmers (via chatting, posting, and program-sharing) and gStudy events related to links-creation, highlighting, browsing, and searching can be recorded in the Interaction Ontology.

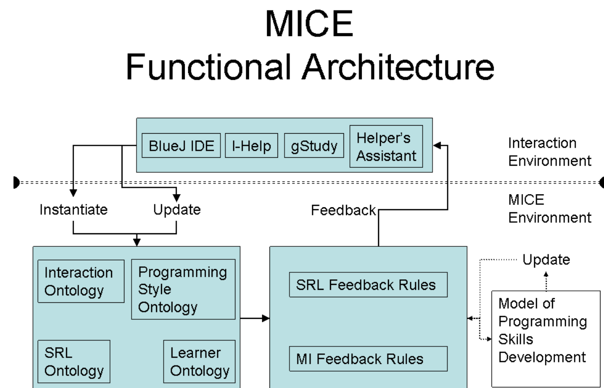

Figure 5: MICE Functional Architecture

The functional architecture also includes a module that accumulates the overall programming skill development, which can then be used to revise the rules. This part of the architecture has not been implemented yet. The double-dotted lines bifurcate the system into two parts – the interaction environment that contains the external interfaces for the programmers and the MICE environment that contains the model-tracing components.

3.2 The Technical Architecture of MICE

MICE presently uses the BlueJ IDE as the coding environment. Programmer interactions within BlueJ are tracked using extensions written by us. These *extensions* listen to BlueJ events such as compiling, clicking on a menu item, moving the mouse cursor, and so on. Once an event is triggered by the extension, Jena[14], a Semantic Web toolkit, instantiates or updates the Interaction Ontology at run-time.

Any change in the Interaction Ontology is recognized by MICE as a definite change in *facts* that are stored in the working memory of the JESS

[12] http://en.wikipedia.org/wiki/Extreme_programming
[13] http://en.wikipedia.org/wiki/Agile_software_develo pment

[14] jena.sourceforge.net

inference engine. Changes in facts automatically activate rules in JESS. A sequence of observable changes to the facts may trigger rules to detect the presence of a particular programming style and as a consequence instantiate the Programming Style Ontology. Further, the rules may also generate system-initiated feedback and send the feedback to external systems (e.g., iHelp, gStudy) via callbacks.

4 MICE Ontologies

The MICE architecture proposes to develop two key ontologies: a Programming Style Ontology and an Interaction Ontology.

4.1 Programming Style Ontology

As mentioned earlier, Programming Style consists of six components, namely, code conventions, code engineering, code debugging, optimal IDE settings for coding, regulating coding tasks, and collaboration while coding. The Programming Style Ontology includes these components as top-level classes. Subclasses are built within each top-level class to capture the presence of style components in programmer interactions. For example, the top-level class *code-convention* contains a subclass for *comment* that recognizes 3 styles of commenting code. They are, a) *comments-goody*, where the code has extensive comments from the programmer, b) *comments-versioning*, where the programmer minimally includes comments about the name of the code, author, date-of-creation, last edited, and so on, and c) *comments-nil*, where the programmer completely ignores to comment the code.

Other programming styles recognized by MICE include compile-for-pop[15], code-till-you-drop[16], hill-climbing-code-construction[17], end-of-days-debugging[18], end-of-world-debugging[19], and SRL-sincere[20]. The first author of the paper is in consultation with expert programmers to identify a number of other programming styles.

4.2 Interaction Ontology

As the programmer starts to code, most of his/her interactions with the BlueJ IDE and gStudy are populated in the Interaction Ontology. We are currently in the process of bringing iHelp interactions into the ontology. The interactions are captured in terms of system events. Events that are currently being tracked in BlueJ and populated in the interaction ontology are:

- Compile Event – For every compile event, the Compile class stores a new instance that contains the compile ID, timestamp of the compile, pointers to the next and the previous versions of the compile code, line numbers of the content (code) added/deleted/modified, and LOC while compiling.
- Run Event – Similar to the compile event, when a programmer executes the code, a run event creates an instance of the Run class with the run ID, timestamp of the execution, pointers to the next and the previous versions of the execution, and LOC.
- Errors and Warnings Event – Compilation of code also creates instances of the Error and Warning class that gets instantiated with information about the compile ID and the number of errors and warnings produced by the compile. Further, the event also identifies the types of errors and warnings produced by each compile leading to a cumulative summary of the types of errors and warnings for a programmer.
- Added/Deleted Constructs Event – When a programming language construct is added or deleted an event is triggered that updates and an instance in Added Construct class or Deleted Construct class. Various language constructs at various levels of abstraction, such as comment, if-else control structure, while loop, for loop, do-while loop, template, switch, return, class, throw, functions, expression statements, declaration statements, function definition statements, and so on, can be identified using JavaML[21]. JavaML, an XML-based source code representation for Java programs, is used to uncover the deep structure of the program from a piece of code at various levels of abstraction[22]. The information stored for each event includes the type of construct added/deleted, the associated compile ID, and the content that have been added or deleted. The information stored for each even includes the type of construct added/deleted, the associated compile ID, and the content that have been added or deleted.

MICE can receive a number of events from external applications such as gStudy and iHelp. For example,

[15] Programmer compiles code at every opportunity such as every time he/she takes a sip of pop

[16] Programmer develops code non-stop for longer terms and compiles only toward the end of coding

[17] Programmer constructs code incrementally with reasonable number of breaks, compiling intermediate code from time to time

[18] Programmer starts to debug only at the very end of task completion

[19] Programmer debugs at every opportunity

[20] Programmer sincerely adheres to SRL and regularly reflects on his/her programming habits

[21] http://www.badros.com/greg/JavaML/

[22] The abstraction levels have not been defined yet

gStudy can be modified to record the following events to be distributed when a programmer attempts to *read* a programming task, *refers* to various online resources, *comprehends* the task in terms of a code design, and *chats* with a fellow programmer to validate the design. Events that are recorded in the ontology are listed below

- Link Event – This event is triggered when a programmer creates a link between the two sections of content. The content could be a description of the programming task of any multimedia (text, graphics, audio, and video) resource that the programmer is referring to. This event triggers and instantiates data such as link type, link created from, link created to, and so on.
- Highlight Event – This event is triggered when the programmer attempts to highlight any portion of the content. Further, the programmer can attach qualitative clues such as 'important', 'doubt', 'to discuss', and so on.
- Browse Event – This event records the links that the programmer has followed while performing the coding task.
- Search Event – This event records when the programmer engages in a search activity within a custom-built search tool.
- Chat Event – This event is recorded when the programmer chats with another programmer from within gStudy's gChat tool. The gChat tool also enables the programmers to use pre-built, semi-constructed, or freely formulated queries. The event records who chatted with who, when, for how long, on what content, using what qualitative queries, sent what responses, and so on.
- Posting Event – This event is triggered when the programmer posts an article to the custom-built discussion board or visits the discussion board to read and respond to others' postings. This event records the type of participation by the programmer, the time of the participation, and so on.

In summary, the Interaction Ontology is populated with events observed in the interaction environment; further, the Programming Style ontology is instantiated wherever a pre-defined style is recognized by the MICE system. The next section discusses the types of feedback facilitated by MICE.

5. Feedback

The very premise of MICE is that, unlike other programming environments, the system can monitor a programmer's interactions across various tools and interpret these interactions to various levels of abstractions of programming styles. Further, MICE can proactively engage the programmer in a SRL-induced dialogue.

MICE's feedback can be passive (programmer-initiated query) or active (system-initiated suggestions to the programmer without the programmer asking for it).

If the programmer is taking too much time to code, or alternatively writing the code and deleting most of it, or getting too many errors and warnings regularly, or some other pre-defined programming styles detected by the MICE system, MICE can step in and proactively interact with the programmer, with appropriate feedback.

The rules (the antecedent part) that trigger such proactive feedback incorporate variables that correspond to the instantiated programming styles as captured in the Programming Style Ontology. Also, the rules (again, the antecedent part) can incorporate variables that correspond to the stages of SRL that a programmer can go through. The consequent part of the rules can suggest that the programmer use alternative coding style/s or perform a SRL-specific activity.

MICE is not intrusive. Programmers may or may not accept MICE's feedback. Further, they can question about the reasoning (summary of the antecedents) that lead to the feedback.

The programmer is encouraged by the system to use the principles of SRL [25]. The programmer is encouraged to plan, then self-reflect, and then self-evaluate pieces of code. For example, reading, taking notes, making flow-chart designs, and engaging in chats prior to coding can be construed of as planning. If the planning stage is not performed by the programmer, as observed by MICE, the system can then suggest (by providing a system-initiated, non-intrusive feedback) the programmer to perform specific planning actions before starting to code. Similarly, observing how often the programmer compiles, executes, and modifies code according to errors can determine if the programmer is engaged in self-reflecting and self-evaluating.

A number of other types of feedback can also be initiated by MICE. A programmer might want to chat with a human helper to solve his/her doubts. MICE, in collaboration with the iHelp system, can find a ready, able, and willing helper.

Further, MICE can provide feedback specific to data that are captured under various events such as 'compile event', 'run event', 'error and warnings event', and 'added/deleted constructs event'. For example, a programmer's coding style can be pictorially presented as a compile ID vs. time plot. Similarly, the debugging

practices of the programmer can also be pictorially captured and presented.

The pointers to the next and previous versions of the compiled code enable one to traverse across the programmer's compilation behavior. Any change in the code in between compiles leading to a new set of errors and warnings can be observed and be presented to the programmer as causes of new errors and warnings.

One of the simplest types of feedback in MICE is based exclusively on the type of qualitative note, highlight, or link created by the programmer in gStudy. For instance, a programmer can create a note or link of type *important*, *doubtful*, *contradictory*, and *supporting*. Such qualitative notes allow the MICE system to identify the stage in the SRL process, as well as the type of feedback to the programmer.

Another example is related to highlights. Depending on the type of tag associated with a highlight, MICE can determine where a programmer is having difficulty or what of the programmer considers as important. This information can be used to produce feedback based on how many other programmers found similar material difficult or important.

In concluding this section, we can say that MICE is designed to provide real-time user-initiated or system-initiated feedback at all stages of coding, including comprehension, design, development, deliberation, testing, and reviewing.

6. Conclusion and Future work

MICE is a system developed for the programming domain. However, the domain can be extended to places where quality of the product is to be increased, collaboration and sharing is needed, training users to improve their technique, encouraging users to use self-reflect and self-evaluate, and/or classifying the users based on their style determined by their interactions with the system and give them feedback accordingly.

In this paper, we presented a generic yet practical definition for Programming Style in terms of components namely, code conventions, code engineering, code debugging, optimal IDE settings for coding, regulating coding tasks, and collaboration while coding.

MICE is designed to provide real-time feedback to programmers based on their programming styles. Further, MICE is also designed to incorporate the principles of SRL as part of its feedback. The feedback from MICE can be programmer-initiated or system-initiated. Further, we plan to incorporate models of mixed-initiative dialogues to enable MICE to engage programmers in constructive dialogues.

We plan to collect data by conducting an empirical study. The study is aimed at the following questions:

- Are programming styles of individual programmers computationally recognizable?
- Can programming styles recognized by the system be regulated so that programmers can reflect on their own styles?
- Can a mixed-initiative computational mechanism assist programmers to identify good programming styles and repair bad programming habits?

We believe that the study will show the effectiveness of theory-oriented mixed-initiative feedback in coding. Further, the ontological capture of programming styles can be seen as a knowledge repository of programming experiences and hence is shareable across academic boundaries.

7. References

[1] Allen, J. F., Mixed-initiative interaction, IEEE Intelligent Systems, 1999, pp. 14-16.

[2] Anderson, J. R., Reiser, B. J, "The Lisp Tutor," Byte, 1985, vol.10, pp.159-75.

[3] Brooks, C., Panesar, R., Greer, J., "Awareness and Collaboration in the iHelp Courses Content Management System," In Proceedings of the 1st European Conference on Technology Enhanced Learning, Crete, Greece, 2006, pp. 34-44.

[4] Chan, T., Integration Kid: A learning companion system, 12th International Joint Conference on Artificial Intelligence (IJCAI'91), Sydney, Australia, 1991, pp. 1094-1099.

[5] Cohen, R. and Allaby,C. and Cumbaa, C. and Fitzgerald, M. and Ho, K. and Hui, B. and Latulipe, C. and Lu, F. and Moussa, N. and Pooley, D. and Qian, A. and Siddiqi, S., User Modeling and User-Adapted Interaction, chapter What is Initiative?, Kluwer Academic Publishers, 1998, pp. 171-214.

[6] Collins, J., Greer, J., Kumar, V., Mccalla, G., Meager, P., Tkatch, R., Inspectable user models for just-in-time workplace training, The Sixth International Conference on User Modelling (UM'97), Italy, 1997, pp. 327-337.

[7] Corbett, A.T., Anderson, J.R., The Lisp Intelligent Tutoring System: Research in Skill Acquisition, In Larkin, J., and Chabay, R. (Eds.) Computer assisted instruction and intelligent tutoring systems: Shared goals and complementary approaches. Hillsdale, NJ: Lawrence Erlbaum, 1992.

[8] Doherty L., Shakya J., Jordanov M., Lougheed P., Brokenshire D., Rao S., Kumar V.S., Recognizing Opportunities for Mixed-Initiative Interactions in Novice Programming, AAAI Fall Symposium on Mixed-Initiative Problem-Solving Assistants, accepted for publication, 2005.

[9] Ferguson, G. and Allen, J. and Miller, B., TRAINS-95: Towards a Mixed-Initiative Planning Assistant, Proceedings of the Third Conference on Artificial Intelligence Planning Systems (AIPS-96), Scotland, 1996, pp. 70-77.

[10] Karen, L. and Myers, W. and Tyson, M. and Wolverton, M. J. and Jarvis, P. A. and Lee, T. J. and desJardins, M., PASSAT: A User-centric Planning Framework, Proceedings of the 3rd International NASA Workshop on Planning and Scheduling for Space, 2002.

[11] Hearst, M.A., "Mixed-Initiative Interaction," IEEE Intelligent Systems, Sept/Oct, 1999, vol. 14, no. 5, pp. 14-23.

[12] Horvitz, E. and Breese, J. and Heckerman, D. and Hovel, D. and Rom-melse, K.,The Lumiere Project: Bayesian User Modeling for Inferring the Goals and Needs of Software Users, Proceedings of the Fourteenth Conference on Uncertainty in Artificial Intelligence, Madison, WI, 1998, pp. 256-265.

[13] Kemp, R., Todd, E., Krsinich, R., Testing the Effectiveness of the Leopard Tutor under Experimental Conditions, Proceedings of the 12th International Conference on Artificial Intelligence in Education (poster), pp. 839-841, 2005.

[14] Kumar, V.S., An instrument for providing formative feedback to novice programmers, In Proceeding of the Annual Meeting of American Educational Research Association (AERA), Division I – Education in the professions, Paper session –Relationship between teaching and learning (13.032), 71, San Diego, CA, USA, 2004.

[15] Kumar, V., Helping the Helper in Peer Help Networks, PhD Thesis, University of Saskachewan, Canada, 2001.

[16] Lesgold, A., Lajoie, S., Bunzo M., Eggan, G., SHERLOCK: A coached practice environment for an electronics troubleshooting job. Computer Aided Instruction and Intelligent Tutoring Systems, USA, 1992, pp. 201-238.

[17] Morris, R., 1 Church, H1., Hadwin, A.F1., Gress, C.L. 2, & Winne, P.H., The Use of Prompts, Roles and Scripts to Support CSLC in gStudy, Annual Meeting of the Canadian Society for the Study of Education, York University, Toronto, ON, May 26-30, 2006.

[18] Pateras, C. and Chapados, N. and Kwan, R. and Lavoie, D. and Tremblay, R., A Mixed-Initiative Natural Dialogue System for Conference Room Reservation, Proc. EuroSpeech'99, 1999, pp. 1275-1278.

[19] Tip, F., Sweeney, P., "Practical Extraction Techniques for Java", ACM transactions on Programming Languages and Systems, Vol. 24, No. 6, Nov 2002, Pages 625-666.

[20] Ramakrishnan, N. and Capra, R. and PerezQuinones, M. N., MixedInitiative Interaction = Mixed Computation, Proceedings of the 2002 ACM SIGPLAN workshop on Partial evaluation and semantics-based program manipulation, Virginia Tech, 2002.

[21] Samin B., "Effects of Self-Regulated Learning in Programming", Masters Thesis, Simon Fraser University – Surrey campus, Canada, 2004.

[22] Shakya, J., Knowledge Engineering and Knowledge Dissemination in a Mixed-Initiative Ontological Framework, MSc Thesis, Simon Fraser University, Canada, 2005

[23] Spohrer, J.G., and Soloway, E., Analyzing the high frequency bugs in novice programs, In Soloway, E., and Iyengar, S. (Eds.), 1986, pp. 230-251.

[24] Vaid, J., Cognitive Psychology and Computer Programming, http://www.rtis.com/nat/user/jfullerton/school/psyc345/program.htm, 1998.

[25] Zimmerman, B. J., "Becoming a self-regulated learner: An overview," Theory into Practice, 2002, vol. 41, 2, pp. 64-72.

[26] Zeller, A., Why Programs Fail: A Guide to Systematic Debugging, ISBN 1-55860-866-4, Morgan Kaufmann, NY.

[27]. Winne, P. H. (1997). Experimenting to bootstrap self-regulated learning. *Journal of Educational Psychology*, *89*, 397-410

[28] Brooks, C., Panesar, R., Greer, J., "Awareness and Collaboration in the iHelp Courses Content Management System" In Proceedings of the 1st European Conference on Technology Enhanced Learning, Crete, Greece, 2006 (forthcoming)

[29] Morris, R., 1 Church, H1., Hadwin, A.F1., Gress, C.L. 2, & Winne, P.H., The Use of Prompts, Roles and Scripts to Support CSLC in gStudy, Annual Meeting of the Canadian Society for the Study of Education, York University, Toronto, ON, May 26-30, 2006

[30] Tip, F., Sweeney, P., Practical Extension Techniques for Java, ACM Transactions on Programming Languages and Systems (TOPLAS), Nov 2002, Volume 24, Issue 6, pp.625-666.