Kokomo: An Empirically Evaluated Methodology for Affective Applications

Derek J. Sollenberger
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206, USA
djsollen@ncsu.edu

Munindar P. Singh
Department of Computer Science
North Carolina State University
Raleigh, NC 27695-8206, USA
singh@ncsu.edu

ABSTRACT

The introduction of affect or emotion modeling into software opens up new possibilities for improving user experience. Yet, current techniques for building affective applications are limited, with the treatment of affect in essence handcrafted in each application. The multiagent middleware Koko attempts to reduce the burden of incorporating affect modeling into applications. However, Koko can be effective only if the models it needs to function are suitably constructed.

We propose Kokomo, a methodology that employs expressive communicative acts as an organizing principle for affective applications. Kokomo specifies the steps needed to create an affective application in Koko. A key motivation is that Kokomo would facilitate the construction of an affective application by engineers who may lack a prior background in affective modeling.

We empirically evaluate Kokomo's utility through a developer study. The results are positive and demonstrate that the developers who used Kokomo were able to develop an affective application in less time, with fewer lines of code, and with a reduced perception of difficulty than developers who worked without Kokomo.

Categories and Subject Descriptors

D.2.8 [**Software Engineering**]: Design—*Methodologies*; I.2.11 [**Artificial Intelligence**]: Distributed Artificial Intelligence—*Multiagent systems*

General Terms

Design, Experimentation, Human Factors

Keywords

Software engineering, Affective computing, Computational architectures for learning

1. INTRODUCTION

The term *affect* is used in psychology to refer to feelings or emotions. The term is also used more narrowly to describe an *expressed* or *observed* emotional response of a human to some relevant event. Expressed responses can be most easily demonstrated in a 3D virtual environment where virtual characters use gestures and dialog to emulate human emotional responses [7]. In contrast, we concentrate on the *observation* aspects of affective computing. In particular, we focus on enabling software developers to use affective computing techniques in order to create a superior user experience.

We take as our point of departure the Koko middleware developed by Sollenberger and Singh [17, 19]. Koko is a service-oriented middleware that observes and maintains a predictive model of a user's affective state, and thus relieves application developers from the challenge of maintaining such a model. However, Koko needs to be suitably configured so as to work effectively.

We propose a methodology called *Kokomo* that steps an application developer through a process for configuring Koko and incorporating it into an application. Interestingly, Kokomo is based on the notion of *expressives*, an important but little-known (especially in the agents community) class of communicative acts.

Further, we conducted a developer study to determine the actual and perceived benefits of Koko and Kokomo to application developers. Our hypothesis is simple: developers who use Koko and Kokomo can more easily construct an affective application than those who do not, while at the same time not diminishing the quality of their application. The study consisted of the same application assigned to groups of developers employing Kokomo (with Koko), Koko alone, and neither (just traditional techniques).

Our evaluation measured the ease of constructing an application both subjectively and objectively. We collected subjective developer feedback on their perceived difficulty via surveys which we obtained throughout the duration of the study. The feedback indicates a lower perception of difficulty for the affective portions of the assignment when using Kokomo than without it. Analysis of objective difficulty measures (code metrics and effort analysis) shows that Kokomo yields the best results on nearly every measure of code complexity and effort. However, the results were not uniformly strong for developers employing Koko alone (we revisit this point in Section 6).

It is important to note, that this work does not introduce any new theories or paradigms for the field of affective computing, but rather seeks to employ existing affective theories in a compelling new way. The main contributions of this work are focused on software engineering as we seek to expand the scope of agent-oriented methodologies into the realms of expressive communication and affective agents. Additionally, unlike many agent-oriented methodologies, we have subjected our methodology to an empirical evaluation in the form of a developer study where the developers are not the authors of the methodology.

Paper Organization

Section 2 provides a background on affect in computing and a summary of Koko as well as of Koko-ASM, a previous methodology.

Cite as: Kokomo: An Empirically Evaluated Methodology for Affective Applications, Derek J. Sollenberger and Munindar P. Singh, *Proc. of 10th Int. Conf. on Autonomous Agents and Multiagent Systems (AAMAS 2011)*, Tumer, Yolum, Sonenberg and Stone (eds.), May, 2–6, 2011, Taipei, Taiwan, pp. XXX-XXX.
Copyright © 2011, International Foundation for Autonomous Agents and Multiagent Systems (www.ifaamas.org). All rights reserved.

Section 3 describes the Kokomo methodology. Section 4 introduces the application used in the developer study. Section 5 describes our developer study in some detail along with an analysis of its results. Section 6 concludes with a discussion of the implications of our findings as well as some important directions for future work.

2. BACKGROUND

Smith and Lazarus [16] are credited with developing a theory of emotions, called *appraisal theory*, which has become the baseline model for affect in computational systems. As Figure 1 shows, an appraisal (for our purposes, an expression of affect) arises from how a user interprets or *construes* an event in the environment.

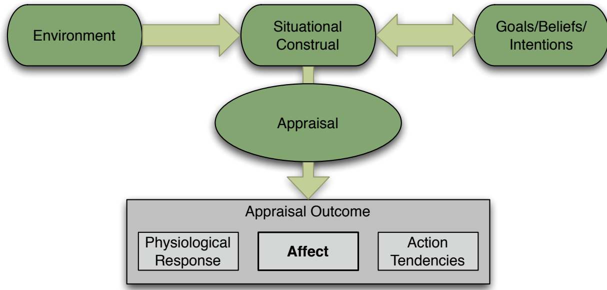


Figure 1: Components and process of an appraisal [16].

Further, the affective state contained within an appraisal outcome can be interpreted as a vector of discrete states, each with an associated intensity. For instance, upon successfully completing a task, a user could arrive at an appraisal that he or she is simultaneously *happy* with an intensity of α and *proud* with an intensity of β.

Appraisal theory finds broad application because of its computational nature. Its applications include educational games [1], similar to the task in our developer study. We note that affective computing extends beyond our discussion of appraisal theory. Picard [12] provides an excellent overview of the affective computing field which includes a more detailed discussion of appraisal theory.

A major challenge in applying appraisal theory is its domain dependence: it yields models of affect that are tied to a particular domain and context. The common practice when creating a new affective application have been to copy and modify an existing model to meet the specifications of the new domain. Recent approaches have begun to address this challenge. The EMotion and Adaptation (EMA) system [7] for modeling virtual characters incorporates a domain-independent model to be populated with the necessary context at runtime. CARE [10] does the same for modeling human subjects.

2.1 Koko

Koko is a multiagent and service-oriented middleware, which enables the prediction of a user's affective state [17]. Koko is not an affect model, but a platform within which (existing and future) models of affect can operate.

Koko provides an ontology of emotions and an ontology of the primitives for describing an application's state. A developer uses these ontologies to configure Koko by specifying (1) which emotions to model and (2) the relevant components of the application's state. From the configuration, Koko generates a model specific to the application. At runtime Koko receives inputs, described using the configured primitives, from the sensors and the application.

Koko supports appraisal theory models, as described earlier in this section. Appraisal theory dictates that each time an input is received an output must be produced. The output produced by a Koko model is a vector of probabilities, called an affect vector. The vector contains an element for each emotion as configured for the application. Each element is the probability that the user is currently experiencing the corresponding emotion. Koko makes each user's affect vector available to the appropriate applications.

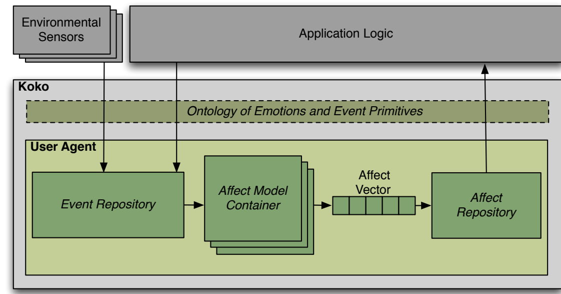


Figure 2: A Koko user agent (arrows represent data flow).

Koko follows the cognitive-based user affect modeling (CB-AUM) approach [8]. Koko supports one model per application for each user and exposes these models to applications using a formal set of interfaces [17]. A CB-AUM model relies heavily on machine learning techniques. Koko includes a repository for CB-AUM models from which a developer may choose the most appropriate model for a new application. The basic flow of such a model can be conceptualized in the following steps.

1. Seed the model with information about the user's environment. This seeding is based on the application-specific configuration.

2. Provide training data. Either the user or someone observing the user must record the user's emotional state in conjunction with data from the application or the sensors, preferably both. This is simplest if the application itself is appropriately instrumented, e.g., to query the user for their emotions. The data can optionally be acquired offline.

3. Learn a model from the training data. Koko does this using the Weka toolkit. In general, user affect modeling [8] makes heavy use of various machine learning techniques to adapt to the behavior patterns of individual users.

4. Provide probabilistic predictions regarding the user's affective state, represented in Koko as an affect vector, as described above.

Additionally, Koko is designed to support affective interactions among the user agents. The traditional Koko runtime is deployed in a cloud environment where there is one Koko agent per end user regardless of how many different Koko based applications are associated with the user. These agents are equipped to share affective information with other agents in the user's social circle, even across application boundaries.

2.2 Koko-ASM

Koko-ASM is a methodology for configuring a social (multiagent) affective application using Koko [18]. The configuration process of Koko is important because the affect models contained in Koko are dependent on the quality of information they receive from the application. The purpose of the methodology is to guide the application developer through Koko's configuration process in an intuitive manner.

Instead of simply listing the requirements for Koko's configuration, Koko-ASM uses the concept of *agent* interaction to guide the developer through the process. The methodology makes use of speech act theory [15] as a means of modeling communication between agents in a social environment. The primary focus of the methodology is to identify the communicative actions among agents and then through a series of steps decompose those interactions into the artifacts needed to configure Koko.

3. KOKOMO

We now describe Kokomo, our proposed configuration methodology for Koko. Kokomo addresses the same goal as Koko-ASM, namely, to guide the application developer to configure Koko for an application. Koko-ASM and Kokomo share some commonalities: both are based on agents, both use the concept of agent roles, and both employ speech act theory.

However, Kokomo has some critical and fundamental differences in terms of how it achieves its goal. First, although Koko is designed to operate in environments ranging from those that are highly social to those containing only a single agent, Koko-ASM only considers applications that involve a high degree of interaction among Koko's user agents. Kokomo, in contrast, is broader in its scope and applies to all settings where Koko can be used. Second, Kokomo expands its treatment of agents beyond a Koko user agent (standing in for a user) to include both Koko user agents and virtual AI-controlled entities (e.g., virtual character). The third and final difference is that no two steps are equivalent across the methodologies. In fact, Kokomo adds two additional steps to provide more granular guidance for developers.

Upon completion of a design exercise with Kokomo, a developer will have identified or constructed all the artifacts needed to configure Koko for an application. Table 1 lists the steps used to create an affective application with Koko. The documentation below highlights the key concepts of Steps 1–5, which are the primary steps in Kokomo.

Table 1: The main steps of Kokomo.

#	Description	Artifacts Produced
1	Define the set of possible roles an agent may assume	Agent Roles
2	Describe the expressives exchanged between roles	Expressive Messages
3	Derive the emotions to be modeled from the expressives	Emotions
4	Describe a set of nonexpressive application events	Application Events
5	Construct the appropriate event definitions	Event Definitions
6	Select the sensors to be included in the model	Sensor Identifiers
7	Select the desired affect model from Koko's repository	Model Identifier
8	Register with Koko's runtime environment	Source Code

Step 1 requires the developer to identify the set of roles an agent may assume in the context of the application. Examples of such roles include teacher, student, coworker, enemy, and rival. A single agent can assume multiple roles and a role can be restricted to apply to the agent only if certain criteria are met. For example, the role of coworker may only apply if the two agents communicating work for the same organization.

Step 2 requires the developer to describe the expressive messages or *expressives* exchanged between various roles [15]. Searle defines expressives as communicative acts that enable a speaker to express their attitudes and emotions toward a proposition. Examples include statements like "Congratulations on winning the prize!" where the attitude and emotion is congratulatory and the proposition is winning the prize. Formally, the structure of an expressive is

$$\langle sender, receiver, type, proposition\rangle$$

The *type* of the expressive refers to the attitude and emotion of the expressive and the *proposition* to its content, including the relevant events. The *sender* and *receiver* are selected from the set of roles defined in Step 1. The developer then formulates the expressives that can be exchanged among agents assuming those roles. In the case that both the selected roles can only be assumed by artificially intelligent agents then formulate expressives only for communicative acts that are observable by a user agent. The result is a set of all valid expressive messages allowed by the application.

Step 3 requires the developer to select a set of emotions to be modeled from Koko's emotion ontology. The selected emotions are based on the expressives identified in the previous step. To compute the set of emotions, we evaluate each expressive involving a user agent and select the most relevant emotions from the ontology for that particular expressive. The selected emotions are then added to the set of emotions required by the application. This process is repeated for every expressive and the resulting emotion set is the output of this step.

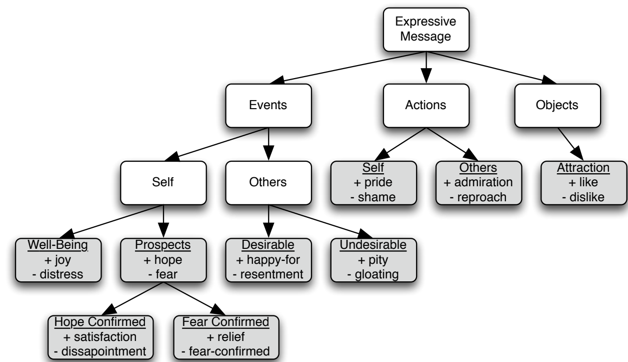


Figure 3: Expressive message hierarchy.

Koko offers support for expressives because it provides a well-delineated representation for affect. Koko can thus exploit a natural match between expressives and affect to help designers operationalize the expressives they use in their applications. The recommended approach to selecting an emotion is to structure Elliott's set of emotions [4] as a tree (Figure 3). Each leaf of the tree represents two emotions, one that carries a positive connotation and the other a negative connotation. Given an expressive, you start at the top of the tree and using its *type* and *proposition* you filter down through the appropriate branches until you are left with only the applicable emotions. For example, say that you have a message with a *type* of *excited* and a *proposition* equal to "I won the game." Now using the tree you determine that winning the game is an action the user would have taken and that *excited* has a positive connotation, so the applicable emotion must therefore be pride. In general, the sender and receiver would have different interpretations of the same event. For example, if the recipient of the above message is the agent who

lost the game, then the emotions that are relevant to the recipient would be admiration or reproach depending on their perception of the winner.

If the *proposition* of the expressive message is composite or even ambiguous as to whether or not the *type* applies to an event, action, or object, then more than one path of the tree may apply. Such is the case when an agent conveys its mood via an expressive message. Mood is an aggregation of emotions and therefore does not have a unique causal attribution. For example, an expressive might convey that a agent is generally *happy* or *sad* without being *happy* or *sad* at something. Therefore, we do not select any specific emotion when evaluating an expressive pertaining to mood as the emotions that comprise the mood are captured when evaluating the other expressives. In other words, mood is not treated directly upon the reception of an expressive.

Step 4 requires the developer to describe a set of nonexpressive events in their application. A nonexpressive application event occurs when the user has an interaction with the application that may effect their emotions. In particular, we are interested in the emotions selected in Step 3. The nonexpressive aspect of these events is that they are not a direct result of interaction between two users, but rather of the user with their environment.

A developer could encode the entire state of the application as a set of events, but this is not necessary as we are only interested in the parts of the application that may effect the emotions we have selected. For example, the time the user has spent on a current task would likely influence their emotional status, whereas the time until the application needs to clear its caches or garbage collect its data structures is likely irrelevant.

We guide developers by helping them think about the interactions (direct and indirect) a user can have with their application and how those interactions relate to the set of emotions defined in Step 3. For example, if the application monitored phone calls and the user had not sent or received a call all day then that may affect the user's emotional state. Further, after identifying an event we must quantify it. In the previous example, the quantification could be the time since the last call was received.

Step 5 requires the developer to construct the appropriate event definitions using Koko's event ontology. The events described are a combination of the expressives in Step 2 and nonexpressive application events identified in Step 4. Each expressive identified in Step 2 is modeled as two event definitions, one for sending and another for receipt. The decomposition of a message into two events is essential because we cannot make the assumption that the receiving agent will read the message immediately following its receipt and we must accommodate for its autonomy.

Step 6 and **Step 7** both have trivial explanations. Koko maintains a listing of both the available sensors and affect models, which are accessible by their unique identifiers. The developer must simply select the appropriate sensor and affect model identifiers.

Step 8 completes the methodology by providing the developer with details on how to register their application within Koko's runtime environment. Given the artifacts gathered from the previous steps we can configure the application via the interfaces defined by the Koko middleware. Upon success, the Koko registration interface returns an *applicationID*, which acts as the identifier for the application. The developer uses the *applicationID* in all subsequent interactions with the Koko runtime, such as sending information about the user to Koko or querying for the user's current affective state.

4. APPLICATION FOR OUR STUDY

Our study involves the development of a math tutoring application for high-school students. This application centers around a dynamic lesson planning agent called the *virtual teaching assistant (vTA)*. The goal of the vTA is to sharpen the user's mathematical skills in a variety of areas, such as probability and geometry. The target audience is students preparing for the standardized end-of-grade tests that are given to US students at the conclusion of the 8th and 12th grades.

This application satisfies three essential criteria. First, it is simple enough to implement in a short period of time, yet complex and open enough that multiple solutions are possible. Second, the application can be compartmentalized into discrete units, thereby enabling us to compare individual components across all developers. Third, the education domain lends itself to affective interactions, where the role of affect in learning is well documented [14].

Let us further explain the role of affect in the learning process. The primary motivation comes from Csíkszentmihályi's theory of flow [2]. In highly simplified terms, the ideas is that individuals learn best in situations where they are neither bored (because the challenge appears trivial) nor frustrated (because the challenge appears impossible), but instead in an intermediary state called the *flow channel*. This concept has successfully (from an education standpoint) been applied in virtual learning environments [11].

4.1 Application Details

Figure 4 describes our application. At all times the user interface is under the control of either the virtual teaching assistant (vTA) or the testing algorithm. The remaining boxes represent the various services that are available to be used by either the vTA or the testing algorithm.

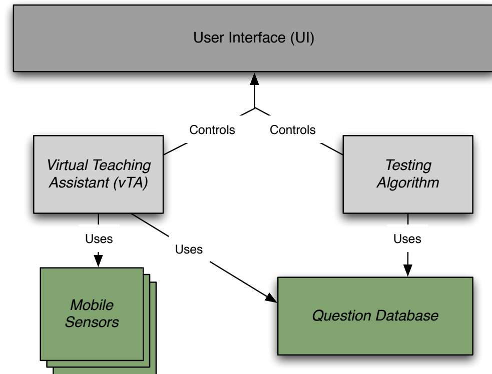


Figure 4: Application components.

The expected application usage is as follows. A user can elect to start a tutoring session from a welcome screen. A tutoring session is comprised of 15 mathematical problems where the first ten problems are selected by the vTA and the remaining five by a testing algorithm. The goal of the vTA is to select the ten best training problems in order to prepare the user for the test. The vTA has at its disposal a database of all possible questions as well as a set of physical sensors that can provide information on the user's current state. Using those sensors and by collecting other data such as the user's history, environment, and current affective state the vTA crafts a customized dynamic lesson plan for each user. The application presents the user with a series of problems (defined by the plan), who is then tested via the testing algorithm as a means of evaluating the effectiveness of the vTA's plan.

Each developer is provided working code for all components of the application with the exception of the vTA. In this manner, we reduce nonaffect-related variability in the solutions and in our mea-

surements of quality and effort, and force our developers to focus on the logic of the application. Using the theory of flow as motivation, the developers are responsible for designing and implementing their own customized vTA.

Now we describe each component of the application from the developer's perspective. Some components are used by the vTA and others serve as an outlet for information produced by the vTA.

Problem Set (Question Database)

The problem set is a subset of a question bank that is maintained by the National Center for Educational Statistics (http://nces.ed.gov). We extracted all multiple choice mathematics questions asked to 8th through 12th grade students based on data collected by the National Assessment of Educational Progress.

Each problem in the dataset contains an *ID*, *question*, *grade level (targeted)*, *content area* (properties and operations, geometry, analysis and probability, or algebra), *difficulty* (percentage of students who correctly answered the question on national standardized tests), and *answer*. The questions are text based, some with a greyscale illustration, and include five multiple choice options as answers.

A developer can retrieve a set of questions based on any permutation of the problem components in a manner analogous to SQL queries, with which our intended developers are familiar. Or, a developer can request a randomly selected problem that meets specified criteria, e.g., a random geometry problem that fewer than 20% of students answered correctly.

User Interface

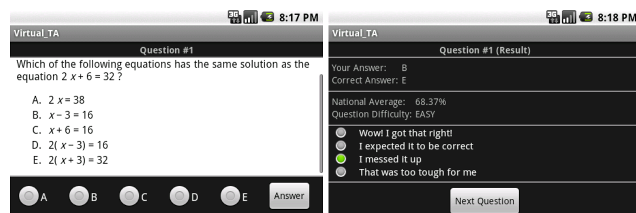
As Figure 4 shows, the vTA controls the user interface (UI). The developer determines the UI via predefined control points, which help limit developer effort and provide uniformity across implementations. Figure 5 shows the two primary screens of the four screens in the UI. Using these basic screens the vTA may configure the user experience with the following four actions.

Pre-Session Questionnaire. Just before a training session begins, the vTA may present the user with a set of up to three questions, and use the answers received to initialize the lesson plan for the user.

Ask Question. After initialization, the UI requests a question from the vTA, who responds with the ID of a question, which the UI displays on the screen.

Record Result. The UI conveys the user's answer to the vTA and shows the user the correct answer.

Post-Question Feedback. After each question, the vTA can optionally ask the user one multiple choice affect-related question, which can provide the vTA with better insight into the user's current affective state.



(a) Question Presentation (b) Result and Feedback

Figure 5: The application's user interface.

The vTA controls the user interface for the first ten problems to train the user, at which point the testing algorithm assumes control and asks the final five questions. The testing algorithm interacts the same as the vTA, but may additionally display the most recent user's history on the application's start page.

Mobile Sensors

A unique advantage of building a mobile application is the availability of sensory inputs. Most smart-phones today come equipped with sensors such as GPS, accelerometer, microphone, and proximity sensors. Access to data provided by these sensors is useful for determining the environment in which the user is operating as well as for evaluating the emotional state of the user. Using this data, developers can track the performance of users in specific environments and customize the lesson plan accordingly.

We employ Android as our development platform. Each developer had access to an Android phone with real sensor inputs. Android provides utility classes for a variety of sensors to use interfaces for extracting data from the sensors.

Virtual TA

The virtual TA is at its core an intelligent agent that maintains a reasoning engine. It is in this engine where the developer must use the available tools, such as sensors and the question database, to craft a unique lesson plan for the user. The vTA has one simple goal: *Create the lesson plan for the user that gives them the best chance of success when tested.*

In general, a vTA could carry out complex reasoning, such as based on the user's previous standardized test scores and available national statistics. However, allowing such extensive approaches would have introduced too many variables into the evaluation. To reduce the variability, we instructed all developers to consider only three categories of information (exam, environmental, and emotional data) in their reasoning algorithms to formulate a lesson plan.

Exam data is any data that can be computed from the user's past performance. It is up to the developer how complex the analysis of the exam data is, but they must at least consider the previous question and the overall performance of the user on the previous exam.

Environmental data is any data gathered from a physical sensor. For instance, by using the microphone on the device a developer could monitor how well the user performed in noisy environments. The developers must integrate data from at least one sensor into the reasoning logic of their vTA.

Emotional data is qualitative feedback from the user on their current status, which can be obtained after each math question as a means to determine how the user felt about the last question. A developer would use this feedback along with the theory of flow as a basis for updating the lesson plan. Developers are required to use such feedback to influence the next question as well as store the feedback for future analysis when selecting questions.

The developer must construct a reasoning algorithm that uses the data to generate a lesson plan. In addition to writing the algorithm, each developer was required to provide written documentation detailing how the algorithm functioned.

Testing Algorithm

The testing algorithm selects the five questions used to evaluate the user at the end of each training session. The algorithm considers the question category, difficulty, and recurrence, and asks no more than two questions from any category. Therefore, since there are only four categories, the test always covers a minimum of three categories. The algorithm also ensures that the national average

for correctly answering at least one question is above 70% correct (easy) and one is below 30% (hard). Finally, the algorithm ensures that no question is repeated in the next three exams for the user.

The details of this algorithm were provided to the developers so they were aware of the testing strategy. Further, we store each user's past performance on exams and make it available to the vTA.

5. EVALUATION

We selected 30 students to participate in the developer study. Each developer had experience programming in Java, and had no prior experience in affective computing. The developers were paired into teams of two and given basic programming assignments in order to evaluate their programming abilities. The assignments were evaluated by a third-party reviewer who ranked each team based on its programming proficiency.

Additionally, each developer was given a survey asking them to rate their software engineering experience and describe projects they had worked on. Over 97% of the developers had a minimum of two years experience programming in Java, but for 88% of developers the initial programming assignments were their first exposure to programming in the Android operating system.

Using the results of the developer surveys and the programming assignments, we ranked each developer with respect to their peers. Using those rankings, we then divided the teams into three groups so as to equalize the estimated programming ability across the groups. Each team within a group was assigned a variation of the same task. The control group did not have access to Koko or Kokomo, one of experimental groups had access to Koko, and the second experimental groups had access to Koko and Kokomo.

All groups are given identical instructions regarding vTA. The Koko and Kokomo groups were additionally provided instructions for using Koko and Kokomo, respectively. Each team was given four weeks in which to complete the assignment. They had to design and develop the vTA agent using either the components provided (e.g., question database and user interface) or any API provided by the Android SDK.

We conducted a baseline quality check for each application. This check ensured that all applications ran appropriately and did not omit any of the features in the required feature set. This test was done in order to strengthen our claim that the introduction of Koko and Kokomo does not diminish the quality of the application. All applications from all three groups passed the baseline check. In fact, we observed that applications using Koko and Kokomo had additional features, in particular, affective features that were not present in the control groups applications. However, our statistical measures disregard such extra features.

Each team was evaluated in the same manner regardless of its group. We evaluated the study in both objective and subjective ways. The objective portion involved measuring the time spent by each developer and various measurements on the source code. The subjective portion was based on periodic surveys of the developers to measure their perceptions regarding their feelings on the complexity of each aspect of the project and the utility of the tools provided.

In-study surveys. These were completed by each developer every time they worked on the project. A minimum of one survey was required for each day (and within one day) the developer worked on the project. The survey collected information such as the time spent on each component of the application as well as their perceptions and comments on difficulty.

Post-study surveys. There were completed by each developer at the end of the development cycle to describe their perception of the entire project. Additionally, this survey was used as a mechanism for developers to recommend improvements to the project and the tools.

The remainder of this section focuses on the evaluation of our hypothesis. Our hypothesis stated that developers who use Kokomo can more easily an construct affective application than those who do not, while at the same time not diminishing the quality of their application. The following two sections evaluate the objective and subjective metrics respectively. Finally, after viewing each category of metrics independently we take a holistic view of the results to determine if our claim is satisfied.

5.1 Objective Results

Measuring the complexity of a project based on the resulting source code is nontrivial because there are no definitive measures of complexity. Complexity can relate to the size of the code base and also to less measurable notions such as extensibility and maintainability. Here we adopt some well-known metrics from software engineering [5, 13], and use them to analyze each project independently. Finally, we compare the metrics across all three developer groups. Figure 6 highlights the results for four of the most well-known metrics. In all cases except Figure 6(d), the lower value indicates a better result.

CC McCabe's Cyclomatic complexity [9] indicates the number of "linear" segments in a method (i.e., sections of code with no branches) and therefore helps determine the number of tests required to obtain complete coverage. It also indicates the psychological complexity of a method.

NoLm The number of levels per method reflects the number of logical branches each method has on average. NoLm is a key factor in determining code readability, and is also used to determine how well the code adheres to in object oriented design patterns.

NoS The number of statements in the project is a common measure of the amount of time spent developing and the general maintainability of the code.

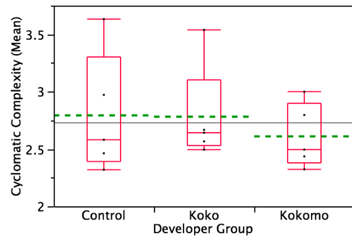
NoM The number of methods in the project reflects increasing modularity and readability of the code (for a fixed NoS value).

We had the developers log the time they spent on each portion of the assignment. Figure 7 gives an overview of the time spent on the project as a whole as well as its affective components specifically. It is important to note that though the total project time did not drastically decrease with Kokomo, the percentage of time spent on affect fell sharply. Additionally, the drastic reduction of time and minimal variance for Kokomo in Figure 7(c) illustrate the benefit of the methodology.

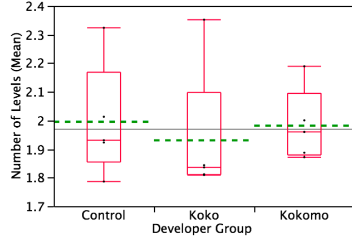
5.2 Subjective Results

Several survey questions asked the developer to either rank components against one another or rate the individual components on a difficulty scale. Table 2 shows the average difficulty rankings given by each development group for the five key components of the project. You can see that the introduction of both Koko and Kokomo reduced the developer's perception of difficulty for the affective aspects of the project.

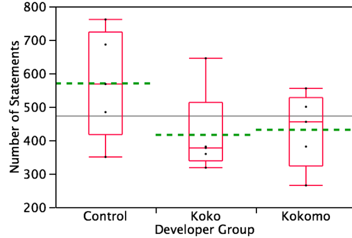
Further, we asked developers to individually rate each component of the assignment on a difficulty scale of 1 (least) to 5 (most).



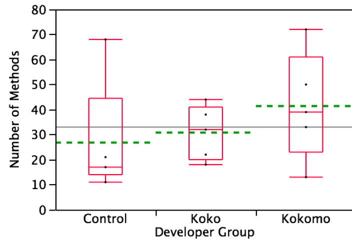
(a) Cyclomatic Complexity



(b) Number of Levels



(c) Number of Statements



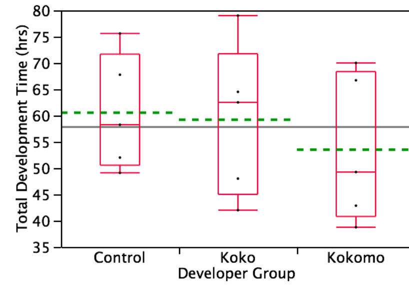
(d) Number of Methods

Figure 6: Software metrics from our study.

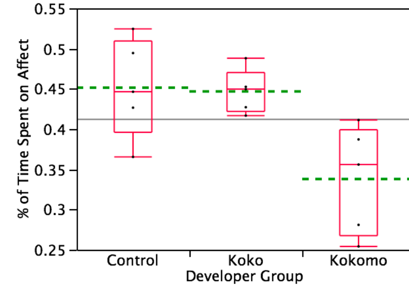
As shown in Figure 8(a), 90% of the developers using Kokomo rated the difficulty of the affective component as a 1 or 2. However, 70% of the control development group gave the same component a rating of 3 or higher. Finally, Figure 8(b) shows how developers rated the benefit obtained from the Koko tools on a benefit scale of 1 (none) to 5 (high). Of the developers using Kokomo, 70% perceived Koko to offer a high benefit to them (a rating of 4 or 5) whereas only 40% of those without Kokomo gave a similar rating.

5.3 Conclusions

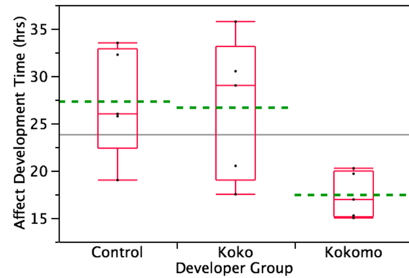
Using both objective and subjective data, we have empirically shown that the introduction of Kokomo results in both perceived and tangible benefits for developers. With respect to software metrics the applications written using Kokomo had either equivalent or better results than those written without Kokomo. The use of Kokomo resulted in significant time savings and also significantly reduced the time variance for constructing the affective component.



(a) Total Time



(b) Percentage of Time



(c) Affective Time

Figure 7: Development time metrics.

The reduction of the time variance is important in that it enables developers to more accurately budget their time. More accurate time estimates in turn enable them to better scope the cost of incorporating affect into an application.

Further, in all cases where developers were asked their perceptions of the project those using Kokomo had a more positive perception than those who did not. This positive perception of Kokomo combined with the tangible benefits make a compelling case for its practical adoption by software developers.

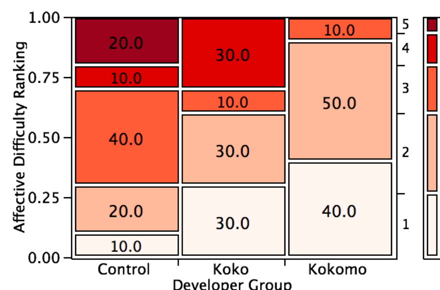
6. DISCUSSION

The most perplexing result of our study was that the data showed that the introduction of Koko without Kokomo did not result in a marked improvement. Upon evaluating the source code and developer surveys, we realized that several developers who were only given Koko were not able to completely grasp the affect modeling concepts that underlie Koko. As a result, they were unsure of how to configure the Koko middleware. Whereas the above observations strengthen the case for Kokomo's usefulness, we plan to improve our documentation regarding Koko for future studies and deployment.

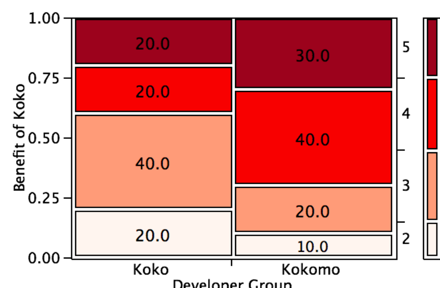
Existing agent-oriented software engineering methodologies specify messages at a high level and therefore are not granular enough to

Table 2: Difficulty ranking of project components.

Rank	Control	Koko	Kokomo
1 (Easiest)	Heuristic	Heuristic	*Affect*
2	Programming	*Affect*	Heuristic
3	Exam	Programming	Environment
4	*Affect*	Environment	Programming
5 (Hardest)	Environment	Exam	Exam



(a) Affective Difficulty Ranking



(b) Koko's Perceived Benefit

Figure 8: Subjective developer distributions.

support the expressive communication employed by Kokomo [3]. Further, Kokomo's applicability has a much narrower scope than these methodologies since it is restricted to affective applications that may or may not employ multiagent systems in their implementation. These distinctions are simply the result of a difference in focus. It is quite possible, given the narrow scope of Kokomo, that it could be integrated with broader methodologies in order to leverage their existing processes and tools. For example, many methodologies have detailed processes by which they help developers identify all possible messages that are exchanged among agents. Kokomo would benefit by integrating such processes, thereby making it easier to identify the expressive messages.

Further, model-driven development methodologies, such as INGENIAS [6] or ASEME [20], are relevant. Kokomo could benefit from such methodologies by using a specific agent modeling description language to describe the expressive interactions among agents. Such a model could be used to produce the source code needed to configure the Koko runtime automatically. This would reduce the amount of time spent translating the expressive messages into a format understood by Koko and would also facilitates improved agility in developing such as in changing the agent interaction model without having to rewrite the source code.

Koko is a multiagent middleware supporting affective interaction among agents, but this study involved only a single agent. This choice was intentional in that it enabled us to demonstrate that unlike Koko-ASM, the Kokomo methodology could be used in single agent environments. In the future, we plan to demonstrate the utility of Kokomo in multiagent settings by using it in social applications that take full advantage of Koko's social design.

7. REFERENCES

[1] C. Conati and H. Maclaren. Modeling user affect from causes and effects. In *User Modeling, Adaptation, and Personalization 2009*, volume 5535 of *LNCS*, pages 4–15, Berlin, September 2009. Springer.

[2] M. Csikszentmihalyi. *Finding flow*. Basic Books, New York, 1997.

[3] S. A. Deloach, M. F. Wood, and C. H. Sparkman. Multiagent systems engineering. *Int'l J. Soft. Eng. Know. Eng.*, 11(3):231–258, 2001.

[4] C. Elliott. *The Affective Reasoner: A Process Model of Emotions in a Multi-agent System*. PhD, Northwestern, 1992.

[5] N. E. Fenton. *Software Metrics: A Rigorous and Practical Approach*. PWS Pub, London, 2nd edition, 1997.

[6] I. Garcõa-Magarino, J. Gomez-Sanz, and R. Fuentes-Fernandez. Model Transformations for Improving Multi-agent Systems Development in INGENIAS. In *Proc. AOSE*, pages 25–36, 2009.

[7] J. Gratch and S. Marsella. A domain-independent framework for modeling emotion. *J. Cognitive Systems Res.*, 5(4):269–306, Dec 2004.

[8] C. Martinho, I. Machado, and A. Paiva. A cognitive approach to affective user modeling. In *Affective Interactions*, *LNAI* 1814, pages 64–75, Berlin, 2000. Springer.

[9] T. McCabe. A complexity measure. *IEEE Trans. Soft. Eng.*, 2:308–320, 1976.

[10] S. McQuiggan, S. Lee, and J. Lester. Predicting user physiological response for interactive environments: An inductive approach. In *Proc. AIIDE*, pages 60–65, 2006.

[11] S. W. McQuiggan, J. P. Rowe, and J. C. Lester. The effects of empathetic virtual characters on presence in narrative-centered learning environments. *Proc. SIGCHI*, pages 1511–1520, 2008.

[12] R. W. Picard. *Affective Computing*. MIT Press, Cambridge, MA, 1997.

[13] R. S. Pressman. *Software Engineering*. McGraw-Hill, 6th Edition, New York, 2005.

[14] P. A. Schutz and R. Pekrun, editors. *Emotion in Education*. Elsevier, Boston, MA, 2007.

[15] J. R. Searle. *Speech Acts*. Cambridge U Press, Cambridge, UK, 1970.

[16] C. Smith and R. Lazarus. Emotion and adaptation. In L. A. Pervin and O. P. John, editors, *Handbook of Personality*, pages 609–637, New York, 1990. Guilford Press.

[17] D. J. Sollenberger and M. P. Singh. Architecture for affective social games. In *Proce. Workshop on Agents for Games and Simulations*, *LNAI* 5920, pages 79–94, Berlin, 2009. Springer.

[18] D. J. Sollenberger and M. P. Singh. Methodology for engineering affective social applications. In *Proc. AOSE*, pages 49–60, 2009.

[19] D. J. Sollenberger and M. P. Singh. Koko: An Architecture for Affect-Aware Games. In *J. AAMAS*, In press.

[20] N. Spanoudakis and P. Moraitis. Model-driven agents development with ASEME. In *Proc. AOSE*, 2010.