Emotions as a Metaphor for Altering Operational Behavior in Autonomic Computing

R. Chandarana D.B. Skillicorn

December 2004
External Technical Report
ISSN-0836-0227-
2004-491

School of Computing
Queen's University
Kingston, Ontario, Canada K7L 3N6

Document prepared December 20, 2004

Abstract

The ability to change operational behavior in response to changes in both external and internal environment is an important aspect of autonomic computing. Managing such behavioral changes is challenging. We propose emotions as a useful mechanism for understanding the structure and use of behavioral changes, and present the design and implementation of the Emotion System, a stand-beside environment for ordinary programs. Both the metaphor and the implementation are designed to make it easy for software developers to integrate autonomic computing into their systems.

Emotions as a Metaphor for Altering Operational Behavior in Autonomic Computing

R. Chandarana and D.B. Skillicorn

Abstract

The ability to change operational behavior in response to changes in both external and internal environment is an important aspect of autonomic computing. Managing such behavioral changes is challenging. We propose emotions as a useful mechanism for understanding the structure and use of behavioral changes, and present the design and implementation of the Emotion System, a stand-beside environment for ordinary programs. Both the metaphor and the implementation are designed to make it easy for software developers to integrate autonomic computing into their systems.

1 Motivation

Software engineering concentrates almost exclusively on the functional properties of programs – what they do. As systems become more complex, operational properties of programs – how they do what they do – are becoming as important [2].

One reason why operational properties are harder to work with is that they are much more contextual than functional properties. It is straightforward (conceptually at least) to decide if a program computes the correct result; all that is needed is information about the state in which it is started. However, to compute whether a program will meet a given performance standard requires knowing about the hardware on which it will execute and the other programs, including the operating system, that will be executing at the same time. The performance of a middleware system may depend on hundreds of different factors.

Nor is it easy to make operational properties into functional ones by including them in specifications. A given program will typically be used in too many contexts for this to be a practical strategy.

Executing programs face two problems that prevent them responding to ambient conditions to meet operational requirements: they do not have access to all the information sources; and they do not have mechanisms for altering their behavior. Limited access to information is partly the result of the separation of concerns strategy used in software engineering. When the program was built, the software designer may have known information about the probable use of the program, but lacked a way to pass this through the compilation stage (some languages include features such as *pragmas* to help with this problem). For example, the software designer may have known or implicitly assumed something about the magnitude of typical arguments to a method. Arguments of different magnitudes may have very different runtimes – for example, floating point operations on some Intel processors run an order of magnitude slower for very large or very small operand magnitudes. Knowing that this is the first time a particular method has been executed, which is information known to the compiler, the runtime could execute it with increased caution. The most important path from ambient conditions to altered execution is at runtime itself – the runtime environment knows factors such as: system load, demand for the program's service, available memory, available network bandwidth and so on, all of which could be used to modify the way in which a program

is executed. The runtime system can also be aware of the program's state, and could modify the way it is executing based on that, for example handling transient exceptions by reexecuting code.

The contribution of this paper is the design of the Emotion System, a stand-beside system that collects information from all stages of the software construction process, especially runtime information; integrates it; and modifies the execution of the software to make it responsive to this extra information. The Emotion System represents one approach to autonomic computing.

The most attractive feature of the Emotion System is that it uses an integration mechanism interposed between collected state information and execution-altering actions. This permits it to gather information from local contexts aggressively, but assess its accuracy and usefulness from a global perspective. Hence, it is designed to make better and more stable judgements of whether and when changes in execution are required. Several new integration mechanisms were developed to make this possible.

The metaphor of emotions are readily understandable, which should assist software developers to see how to utilize the Emotion System as a part of their systems. The design of the Emotion System also allows them to use as much or as little of its functionality as they want.

2 Related Work

Manipulating the operational properties of software systems to make them responsive to their own behavior and their environments is the major goal of IBM's autonomic computing initiative [4, 6]. Several other companies have similar goals: HP's adaptive infrastructure, Microsoft's dynamic systems, and Sun's N1. IBM probably has the clearest vision, distinguishing four levels of autonomy: making systems self-configuring, self-optimizing, self-healing, and self-protecting. All of these approaches require the same basic infrastructure: channels for gathering information from the environment and the program itself; ways to integrate this information and map it to actions; and ways to effect these actions in the executing program. A solution to these problems requires much more infrastructure than the Emotion System, but the Emotion System contains each of the essential features in some form.

Aspect-oriented programming is an approach to passing information across (perhaps around) the information-hiding barriers of conventional software development [5]. Some program properties require systematic, small, but widespread actions throughout a program. Many of these actions are conceptually simple, but they create a maintenance problem because they are hard to find, once spread through the program. For example, if a formatted output file must be altered, many places in the program that generated it may need to be changed; altering global properties such as security also requires widespread changes. Aspect-oriented programming provides a mechanism to describe such a property in a single place, and have the requisite changes applied automatically throughout the program.

MILAN is a system that enables parallel programs to notice when more or fewer processors become available on a cluster system, and adapt their virtual parallel structure to match the available physical parallelism dynamically [1]. The dynamic adaption is based on a number of factors – the number of available machines (self scheduling), history and speculation of the machine risk factor (predictable scheduling), and ratio of jobs to machines (eager scheduling, dynamic granularity).

3 The Emotion System

The Emotion System consists of three parts:

1. A sensation collector that gathers information about the construction, compilation, and execution properties of each program;
2. An integrator for each emotion that translates sensations to actions using an integration and thresholding mechanism;
3. An effector that applies the actions to running programs appropriately.

An emotion represents one cohesive set of sensations, integration mechanism, and related effects; its role is as a kind of mental shorthand the connects what would otherwise be a diverse set of inputs and outputs to the Emotion System. Just as the human emotion of *anger* has a characteristic set of triggers, a level at which it becomes turned on (we talk about a person's boiling point), and a characteristic set of physiological and behavioral outcome, so the programmatic 'emotion' of novelty is triggered by factors such as execution of newly compiled code, and has outcomes such as more rigorous testing of runtime execution.

The Emotion System is designed so that a number of emotions can be used in an orthogonal way, without interacting with each other. An overview of the Emotion System is shown in Figure 1.

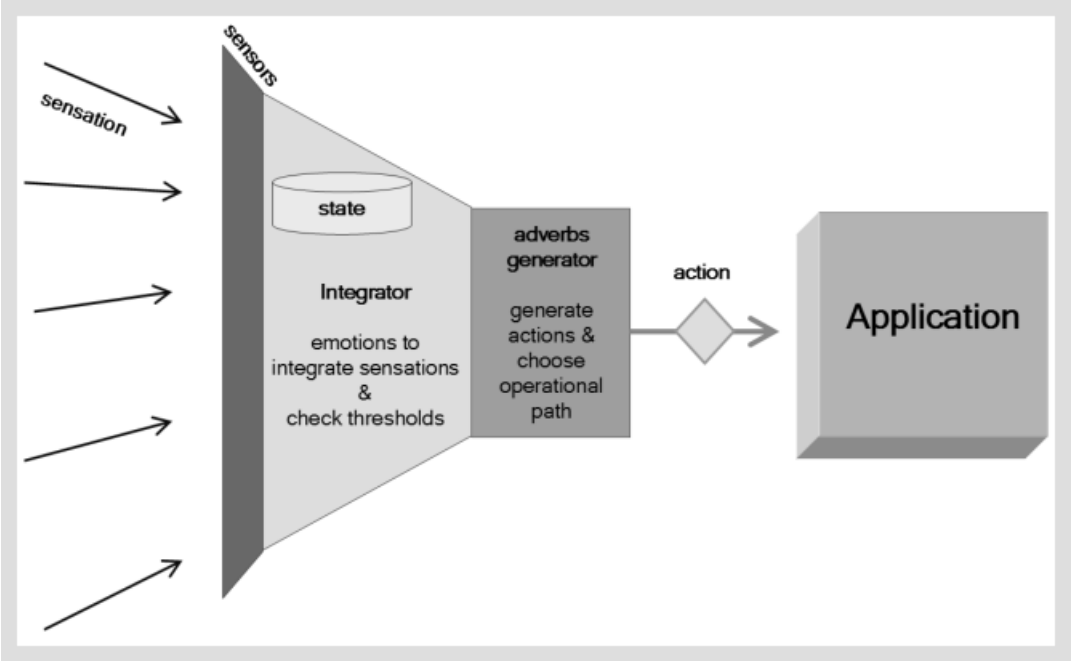

Figure 1: An overview of the Emotion System

3.1 Collecting sensations

Sensations are ways in which the software developer, the compiler, the environment, and the program itself can convey useful information ('hints') to the Emotion System. This control path

bypasses the conventional control path via program variables. The intent is to make it easy for any of these parts of the system to convey information that *may* be useful, without requiring that it necessarily be either correct or useful. This is important because the parts of the system that generate sensations have a limited view of the situation and it is onerous to require them only to send sensations when they are sure of their global correctness. For example, a software developer may know a useful piece of information in the context of a method, but does not know how often the method is invoked. Hence it is not clear how much weight to give to the piece of information in a single execution of the method.

The sensation mechanism is designed to encourage parts of the system to provide sensations whenever they might be useful; so that the default is to send the sensation. Whether or not the sensation has an effect is left up to the integrator.

Software developers send sensations by including metastatements in code. These have the form of a sensation name, and a value, which may be a function of program variables. Preprocessing the code converts these statements into code that sends the appropriate value to the Emotion system. For example, the sensation

```
[cautious, argsize - 1000]
```

which sends a cautious sensation for an unusually large argument magnitude becomes

```
EmotionMgr.getInstance ().sendSensation (
     Sensation.cautiousSensation,String.valueOf(argsize - 1000));
```

Other sensations are collected as side-effects of compilation. A stateful component of the Emotion System allows aspects of the execution history of each piece of code to be kept as well.

3.2 Integrating sensations

The integrator is sent sensations and must decide how to make use of them to change each emotion's state. The integrator approximates a global view of each program's action because it sees the full range of sensations. Its role is to decide how to weight each piece of information sent to it in making a global decision.

Two important properties are required for the integrator:

1. The integrator should not be manipulable, directly or accidently, by the sensations it is sent. At the point at which each sensation is generated , there is rarely enough information to judge how much influence this particular sensation should play in the decision to turn an emotion on or off. For example, a sensation in program code does not necessarily know how many times the code in which it lies will be invoked.

   It follows that each individual sensation should only play a small role in the overall decision and that it should have a kind of *idempotent* property – repeatedly sending the same sensation should not be treated as more significant than sending it once.

2. The integrator should be stable, so that small changes in sensation do not have a disproportionate effect on emotion state, and the system as a whole should not turn an emotion on and off too frequently.

To achieve these properties, we have designed two mechanisms for thresholding: a histogram-based technique, and a tug-of-war-based technique.

In the histogram-based technique, the integrator maintains a histogram data structure, with a fixed number of columns, say labelled from $-n$ to $+n$, and columns of fixed height, say m. This data structure is initialized so that each column has the same height. Empirically, choosing these initial heights to be $m/3$ seems to work well.

Each new sensation is mapped to a signal in the range $-n$ to $+n$ (damping or amplifying the relevant emotion). When a sensation value, say 2, is sent, the height of the second histogram is incremented by 1. All of the columns are now normalized (reducing their heights so that the total volume of all of the columns remains the same).

The threshold used to decide whether the emotion is turned on or off is determined by choosing a value from $-n$ to $+n$ and turning the emotion on if more of the histogram volume is to the right of the value, and turning it off if more of the volume is to the left.

This data structure has an appropriate idempotence property because repeatedly sending the same sensation quickly saturates the column corresponding to that value, and so new sensation messages have little effect. Choosing m to be about 5 seems appropriate to achieve this property. The data structure is also quite stable, because any single sensation message can only move the histogram mass slightly to right or left. Pathological behavior is still possible, but unlikely.

In the tug-of-war-based technique, the integrator maintains a single binary floating point value with limited precision, n. Each positive sensation signal of strength, say, i increments the value in the ith binary place from the end (a signal of strength n changes the first binary place, not the last); each negative sensation signal correspondingly decrements the value in the ith decimal place from the end. The corresponding emotion is turned on if the value is greater than or equal to 1, and is turned off otherwise. This simple strategy can be straightforwardly implemented using a single decimal floating point value. The tug-of-war approach is weaker than the histogram approach, but simpler to implement.

3.3 Effecting changes of action

The integrator informs the effector of changes in state for each emotion; the effector is responsible for changing the pattern of execution of the program in response. Some examples of changes in behavior are:

- reexecuting a method when an exception occurs (many errors in mature applications today are transient, and reexecution resolves them);
- executing a computation twice and checking that the result is the same;
- choosing different algorithms based on the values of arguments to them;
- performing more aggressive bounds checks;
- checking for a wider range of exceptions;
- logging more detailed information about application activity;
- loading an older version of a class when a new version generates an error;
- changing a parameter in a system call to achieve better performance;

- migrating the application to a new processor seamlessly;
- forcing the application to choose a new web service to interact with;

The basic mechanism that we use to effect different actions is the Java Platform Debugger Architecture (JPDA) which allows breakpoints to be inserted in the application with almost any granularity, and appropriate actions taken by the Emotion System in response. Of course, some of the actions listed above require the software developer to have provided multiple versions of a method or algorithm.

We describe this strategy of forcing applications to behave in different ways as *adverbial execution*: the response to the emotion of *caution* is to execute *cautiously*.

4 Implementation

The Emotion System is implemented as an independent component that can be attached to other applications at runtime – it is a stand-beside system.

The working of the Emotion System can be split into two parts – State Management and Adverbial Control. State Management is how the Emotion System maintains and integrates the state of the connected application, and how sensations from the application modify the state. It implements the sensation collection and the integrator part as described in Section 3. Adverbial Control is how the Emotion System reacts to events in the application and changes its runtime execution. It implements the effector part as described in Section 3.

4.1 Java implementation

The prototype Emotion System was developed using Java. The application and the Emotion System are run on separate virtual machines. This enables the Emotion System to be independent from the application and hence to be developed in a modular way. The virtual machine level of abstraction also gives the Emotion System a set of key methods that allow direct and powerful control over the application runtime. The high-level control flow of the Emotion System is shown in Figure 2.

4.1.1 Application integration

There are four stages of interaction between the Emotion System and the application.

Compile time

The application program is preprocessed at compile time to add or modify code to:

- initialize contact with the Emotion System at a remote destination;
- send sensations, with relevant information, to the Emotion System;
- query the state of the Emotion System;

The preprocessing transforms the emotion tags into actual code that performs the Emotion System interaction at runtime. In the prototype, TXL [3] was used to perform the preprocessing.

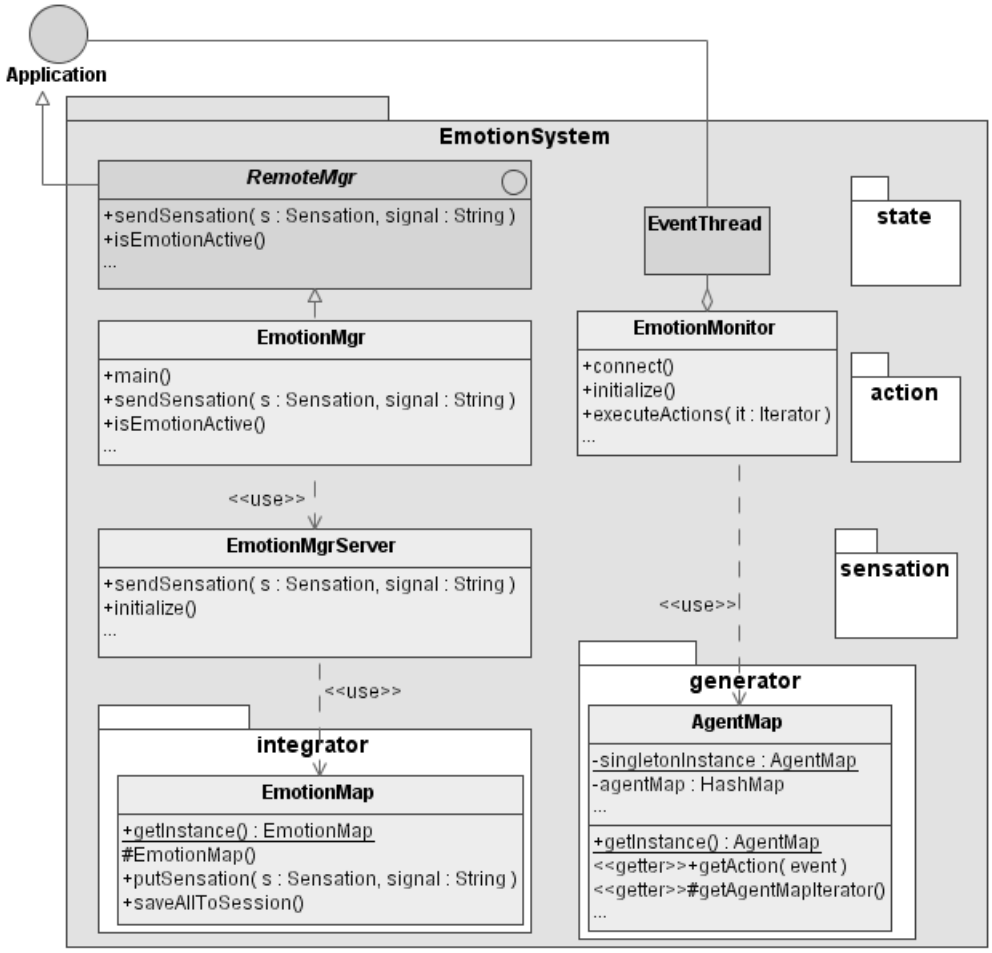

Figure 2: Emotion System high-level control flow

Runtime connection setup

At runtime, the application initializes two connection streams with the Emotion System. The first connection stream is an **input/output channel** that allows the application to send sensations or query the state of the system. The second connection stream is a persistent, threaded **control channel** that allows the Emotion System to monitor and control the runtime execution of the application.

The input/output channel is implemented using Java Remote Method Invocation (Java RMI) [9]. Using the predefined remote interface to the Emotion System, the application can access the communication stream. The control channel is implemented using the Java Platform Debugger Architecture (JPDA) [8]. The Emotion System initializes the server-side component of JPDA. The application uses Java command-line parameters to initialize the JPDA connection, enabling the Emotion System to monitor and control the runtime execution.

Runtime communication

The runtime communication is a two-way transfer of messages or control statements that are initiated by the application. Application sensations are evaluated at runtime and sent to the Emotion System using the input/output channel. Automatic event triggers such as exceptions, method entry, etc, are monitored by the Emotion System using the control channel. Any of these application-originated messages or triggers can induce a set of response actions by the Emotion System.

Connection termination

When the application terminates, a final termination event is received by the Emotion System. This causes the Emotion System to back up the current state of the application. We utilize a JPDA event – *VMDisconnect*, to terminate the Emotion System.

4.2 State management

State management is the process of maintaining, integrating and modifying the state of the application stored within the emotion system. The process gets its input from both the application, and as feedback from the Emotion System itself.

The manager is the point of the entry in the Emotion System for the sensations. It is responsible for decoding the messages sent by the application and passing them to emotion handlers.

The manager is implemented using Java RMI. The class *EmotionMgr* implements the *Remote* class and is initialized on a predefined port. At runtime, the application connects to the specific port and uses the methods declared in the *EmotionMgr*.

To enable a server-side use of the sensation-handling methods, the implementation of the *EmotionMgr* methods is done in *EmotionMgrServer*. The methods of *EmotionMgrServer* can be called directly by the other classes of the Emotion System. The class also serves as the internal initializer of various services (*AgentMap*, *EmotionMap*, *UserOptions*, and *EmotionMonitor*).

4.2.1 Emotions

Emotions represent a specific configuration of the application state controlled by a subset of sensations. The structure of emotions is shown in Figure 3. To aid a broadcast of sensations to various emotions and perform other common tasks, an *emotion map* is used. The emotion map maintains the list of emotions in the system and is the mediator between emotions and the rest of the system.

Each emotion has an integration mechanism, environment map, state bean, and a unique id. On receiving a sensation, the emotion integrates the sensation using the integration mechanism and takes into account miscellaneous information using the environment map. Each emotion can be either active or passive, and this is decided by the combination of the integration mechanism and the environment map.

The environment map is a store of extra information received during the history of the application: exception locations, compile time, and so on.

4.2.2 Persistence of state

The complete Emotion System is mapped into two layers of persistence (Figure 4):

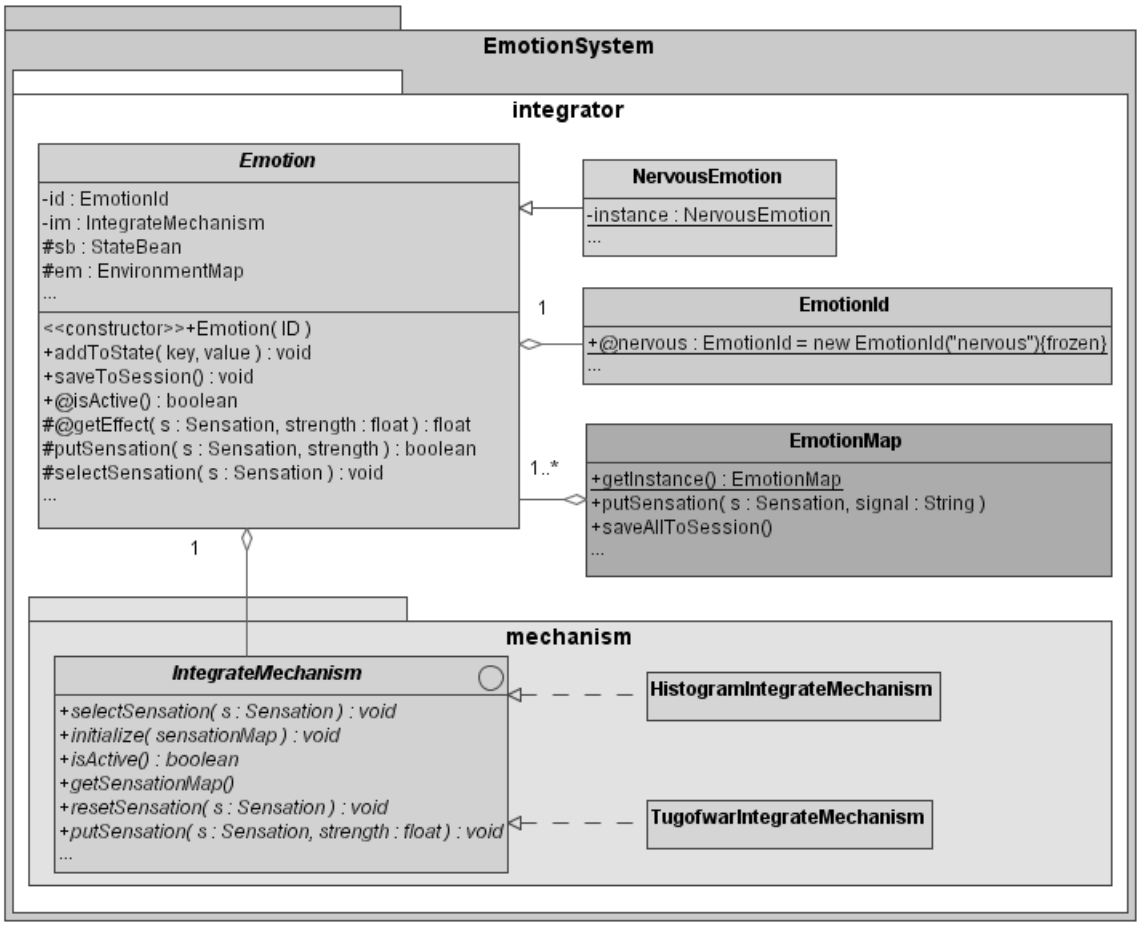

Figure 3: Emotions in the Emotion System

Sensation layer The strengths of the subset of sensations for a given emotion.

Environment layer The history of miscellaneous information received along with the sensations pertinent to the given emotion.

A *StateBean* is used to encompass these two layers of information for each emotion. Using an index system, an *Emotion* can then retrieve its own *StateBean* at the start of a session. Each component in the sensation layer is identified by the combination of the sensation id and the integration mechanism id. Each environment layer component is identified by a predefined constant.

The persistence of state is achieved by using the *StateContainer*. The *StateContainer* is an interface that defines methods to store and retrieve *StateBean* objects. The *StateContainer* is implemented using JDO (Java Data Objects) [7].

4.3 Adverbial control

Adverbial control is the process by which the Emotion System controls and monitors the application (Figure 5). Adverbial control works by monitoring a predefined set of events, and generating

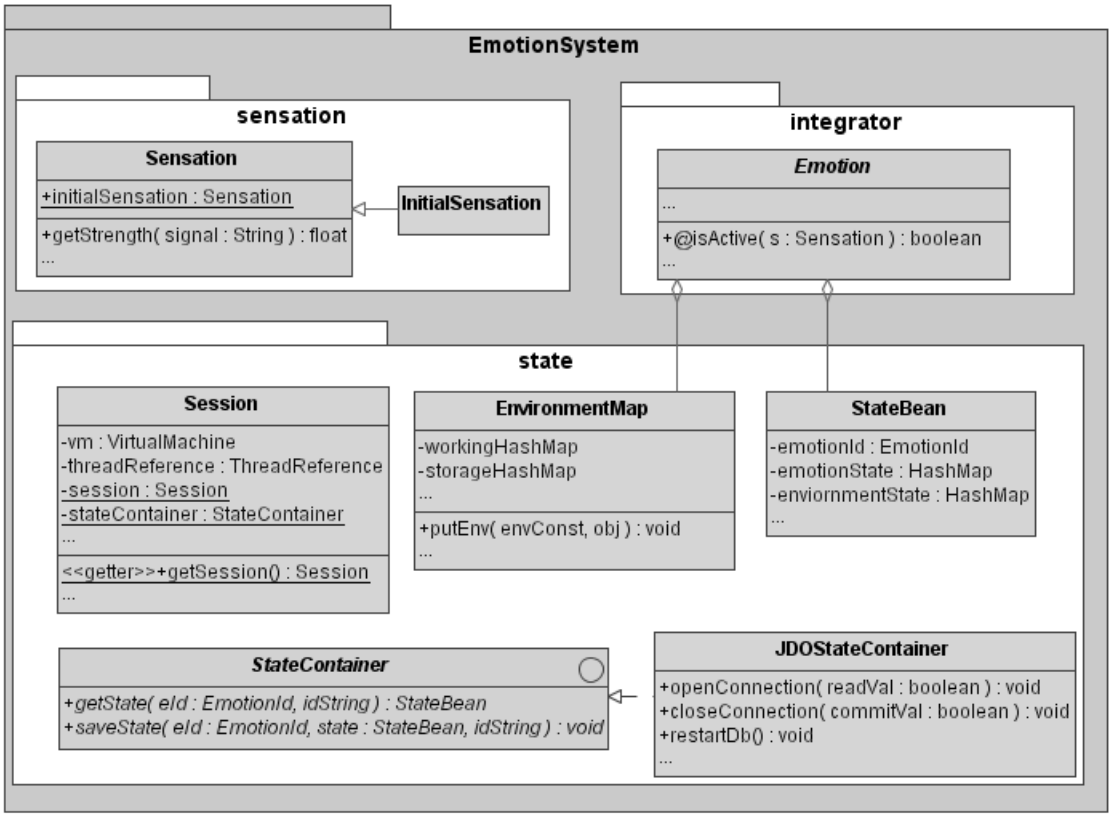

Figure 4: Emotion System state management

appropriate actions.

There are different kinds of event that can act as breakpoints to perform the adverbial control. Each event allows for a varied level of granularity and flexibility. The following are the major kinds of events.

Method entry/exit - allows runtime modification at the method entry or exit.

Step event - allows runtime modification at every line of code executed.

Exception event - allows runtime modification to be made if an exception has been generated.

Watchpoint - allows runtime modification to be made if a given variable is modified.

VMConnect/VMDisconnect - allows appropriate actions to be executed when the application virtual machine connects or disconnects from the emotion system.

ClassLoad - informs the Emotion System that a class is being loaded into the application virtual machine (usually during the first use of a class's method or attribute). This provides opportunity to request special breakpoints (e.g. during initialization of variables), or more flexible step events (e.g. every 5 lines of execution).

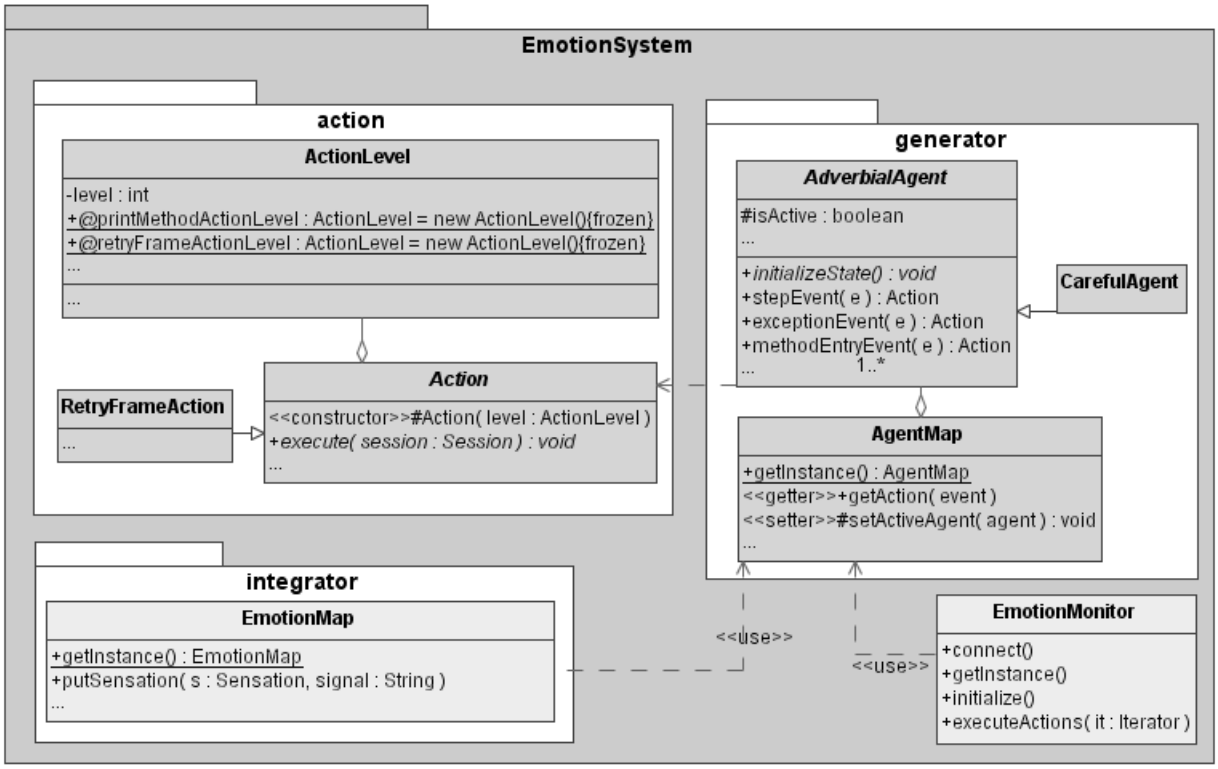

Figure 5: Classes for adverbial control

When the application virtual machine connects, a corresponding *EventThread* is created to monitor the JPDA (control channel) connection. Using JPDA, the Emotion System has access to various features of the application's virtual machine:

Suspend – If the event received by the *EventThread* is being monitored, then the application can be suspended while appropriate actions for runtime modification are performed on it.

Location – The location allows the Emotion System to determine where the application has been suspended. This is particularly useful in the case of exceptions, to determine the exact location of the exception.

StackFrame – The Emotion System has access to the *StackFrame* and the corresponding *Local-Variables* at each frame of the application memory. The Emotion System also has access to the *ThreadFrame* that allows it to control the execution of threads and methods. The control is also available up to the *Class* level, which allows the entire class code to be redefined.

There is overhead to monitoring at a fine level (for example each line), or suspending the thread for any event. Hence, the monitoring level is set dynamically by the *AdverbialAgent* based on the state of the *Emotion* objects in the system.

4.3.1 Implementing adverbial control

To implement adverbial control, we use actions that perform single direct modification of the runtime execution. A complete adverbial control task might consist of a group of such actions. Having single direct actions has the following benefits:

- Individual, cohesive actions that perform single tasks are easy to implement.
- The Emotion System can perform high-level control so that effect of an action is not duplicated.
- The Emotion System can compare actions and perform them in an stable order. The goal is to have actions execute without being corrupted by the effects of previous actions.

We use the command design pattern to create an *Action* interface. Each implementation of the *Action* interface performs a single direct modification or query of the application runtime. The *Action* class itself implements the *Comparable* class and this property is used in comparing different actions.

4.3.2 Adverbial control based on the state

AdverbialAgent is the base class of adverbial control. It defines the default response to any application events as null (no effect). Each instance of the *AdverbialAgent* implements a specific adverbial control. It defines a set of actions (response) to necessary events (request). The *AdverbialAgent*, like the *Emotion*, can be active or passive. The state of the *AdverbialAgent* is the result of a boolean function on one or more emotion states.

AgentMap is a control class for the various instances of *AdverbialAgent*. When an event is received, the *AgentMap* queries all the active *AdverbialAgent* objects with a request for a response. The response of the *AdverbialAgent* is a set of *Action* objects. These are sorted and executed on the application virtual machine by the *EmotionMonitor* using the JPDA.

4.4 Emotion maps

Each emotion map identifies a sequence of inputs, integration configuration and the corresponding effects.

4.4.1 Example sensations

The following are some of the possible sensations and the format in which they are used. In terms of the implementation, each of these sensations are classes that extend the *Sensation* abstract class.

Initial sensation. This sensation is used to inform the Emotion System about the last compile time of the enclosing class. The *strength* of the sensation signifies the power of each initial sensation. Since the sensation has an inverse effect on the emotions (the emotion turns inactive after x sensations of the same timestamp), the strength determines the period of time for which the sensation will be in effect.

Warning sensation. The sensation is used to inform the Emotion System about the probability of a possible exception. The *strength*, in general, signifies the amount of certainty of the sensation.

Unusual sensation. This sensation is used to identify unusual values (based on the expected expression results). The sensation is sent along with the result of a dynamic expression, that is used by the Emotion System to interpret the extent of unusualness in context of the sensation.

Exception sensation. This sensation occurs when an exception is caught and some action is taken. The objective of the sensation is to prevent exception handling in an infinite loop. Unique location info is sent along with the sensation, that identifies where the exception was caught.

Object sensation. This sensation stores information at 'object level'. Unique object identification is sent along with the sensation. All information received under the unique object identification is integrated into a single emotion.

4.4.2 Example emotions

The following are some of the emotions that maintain state for the target application. In terms of the implementation, each of these emotions are classes that extend the *Emotion* abstract class.

Novel emotion. This emotion maintains information about how novel the target application is, by keeping track of how mature the application is, noting the number of times the application has executed since its last compilation. The initial sensation is the only sensation integrated by this emotion.

Cautious emotion. This emotion represents how cautiously the application should be executed. The warning sensation, unusual sensation and exception sensation are integrated by this emotion. The integration determines if errors are unusually likely.

Confident emotion. This emotion represents the degree to which the application is confident in the current execution. The sensations that are integrated by this emotion are – credit sensation, unusual sensation.

Self emotion. This emotion represents the emotion value of a single object, as identifies by the object sensations. It maintains different instances of integration mechanism for object sensations with different identifiers.

4.4.3 Example adverbial agents

The following are some of the adverbial agents that generate effects on the application runtime execution. In terms of the implementation, each of these adverbial agents are classes that extend the *AdverbialAgent* abstract class.

Fresh agent. This agent provides runtime enhancement that will be useful for initial executions of a class. The frequency of monitoring events is high. The events of method entry/exit, exceptions, and class loading, are monitored. The effects are: printing log information, and providing aggressive exception handling and recovery. This agent depends on the state of the novel emotion.

Careful agent. This agent provides special exception-handling mechanisms. The only event that is monitored is the exception event. When an exception occurs, a recovery mechanism is deployed to try and respond. This agent is dependent on the state of the cautious emotion.

4.4.4 Example actions

To understand the practical implementation of the effects generated by the adverbial agents, we shall look at some of the actions in context of the Java implementation of the Emotion System. There are two kinds of actions – (type 1) actions for the requests of events to be monitored, and (type 2) actions for performing tasks in response to events. Some of the type 1 actions are:

- *ClassLoadRequestAction* enables the Emotion System to receive an event notification when a new class is loaded in the application execution.
- *SetBreakpointAction* sets breakpoints at the method entries so that this information can be logged as debug information.
- *ExceptionRequestAction* enables the Emotion System to receive an event notification when an exception occurs in the application.

Some of the type 2 actions are:

- *RetryFrameAction* changes the stack frame order to execute the current method again.
- *PrintVariablesAction* prints the method names entered along with useful information.

5 Examples

We shall illustrate the use of emotions to achieve the following:

Dynamic log information. Dynamic log information is generated based on the last compile time of each class in the software. The key advantage over using debug flags is that re-compiling one class automatically generates log information pertaining to the use of the class anywhere in the system, while keeping the overhead down with reference to additional log information.

Aggressive exception handling. Programmers tend to catch generalized exceptions and forget to perform appropriate actions or don't know what they should be. We shall look at the case of a runtime exception (unchecked in Java), and see how it can be caught in the debug stage of the software, while it is ignored in the later executions of the software.

The emotion map used will be {*initial, novel, tentatively*} – i.e. the *novel* emotion integrates the *initial* sensations, based on the status of which the *tentatively* agent controls the flow of execution.

5.1 Example emotion: {initial, novel, tentatively}

The target application prints out integers stored in an array.

Listing 1: Target application

```
1  public class arrayBound {
2      private static RemoteMgr emgr;

4      public static void main(String[] args)  {
5          initRemoteMgr(args);
6          int[] intArray = {0,1,2,3,4,5};
7          try{
8              for (int i = 0; i <= intArray.length; i++) {
9                  System.out.print(i + "=" + getArrayVal(intArray,i) + ",\t");
10             }
11         }catch(Exception e){
12             //do nothing
13         }

15     }

17     private static int getArrayVal(int[] iArray, int i){
18         return iArray[i];
19     }
20 ⊞ ..........................................................................
```

Due to the incorrect comparison (<=) in Line 8, there will be a runtime exception in Line 9. The generalized exception is caught in Line 11. However, there are no actions to take for that exception. The normal execution of this class gives the output shown in Listing 2.

Listing 2: Target application output

```
1 java arrayBound noemotions

3 0=0,    1=1,    2=2,    3=3,    4=4,    5=5,
4 Process terminated with exit code 0
```

Not surprisingly, if this was run in a unit test, no error would be detected. This is because the generalized exception handler catches the exception and does nothing in response. However, the use of the same code in a larger context might have serious consequences since the code does not match the requirements.

Listing 3 is the output of the target application when connected to the Emotion System. As shown in the Line 1, the connection transport and port number are passed as command-line parameters to the Java virtual machine.

The most important part of the listing is the output on Line 3. The value of element 6 in the array is shown to be 5, which is actually the value of element 5. However, this value is not printed (Line 3 of Listing 2). This is because, without emotions, the exception is caught in Line 11 of Listing 1 and no action is taken. In test, the code seems to be working as required. On the other hand, with aggressive exception handling, the exception is caught, recognized as an `ArrayOutOfBoundsException`, and dealt with appropriately.

Listing 3: Application with emotions

```
1 java arrayBound -Xdebug -Xrunjdwp:transport=dt_socket,address=127.0.0.1:6001

3 0=0,    1=1,    2=2,    3=3,    4=4,    5=5,    6=5,
4 Process terminated with exit code 0
```

Listing 4: Emotions system output

```
1 java -Djava.rmi.server.codebase=http://localhost:8080/matrix/classes/
2       -Djava.security.policy=java.policy
3       EmotionSystem.EmotionMgr

5 : EmotionMgr bound in registry
6 : Argument:[]
7 : Setting up application log:CLIENT_040808_181317.log
8 : StateBean[novel] received
9 : Environment ->
10 :        >COMPILETIME = Tue Dec 14 13:04:46 GMT-05:00 2004
11 : Retrieved sensation [initial] >
12 : >-10:0,-9:0,-8:0,-7:0,-6:0,-5:0,-4:1,-3:0,-2:0,-1:0,
13      0:0,1:0,2:0,3:0,4:0,5:0,6:0,7:0,8:0,9:0,10:10
14 : EmotionMap initialized.
15 : EmotionMonitor is ready to accept connections on port:6001
16 : Connection received
17 : ID=EmotionSystem.generator.FreshAgent| CLASS=EmotionSystem.generator.FreshAgent received.
18 : ID=EmotionSystem.generator.FreshAgent| CLASS=EmotionSystem.generator.FreshAgent set active
       .
19 : ACTION:EmotionSystem.action.ExceptionRequestAction
20 : ExceptionRequests added with SUSPEND_ALL=true
21 : ACTION:EmotionSystem.action.MethodExitRequestAction
22 : MethodExitRequest added with SUSPEND_ALL=false
23 : EVENT : com.sun.tools.jdi.EventSetImpl$VMStartEventImpl|active Agents:1
24 : EmotionSystem.sensation.InitialSensation received with
25      strength = Tue Dec 14 13:04:46 GMT-05:00 2004#7
26 : strength = -7.0:transformed = -4:index = 6
27 : Sensation[initial] restructured - reduced by [0.5]
28 : -10:0,-9:0,-8:0,-7:0,-6:0,-5:0,-4:1,-3:0,-2:0,-1:0,
29      0:0,1:0,2:0,3:0,4:0,5:0,6:0,7:0,8:0,9:0,10:5
30 : Emotion [novel]:Active
31 ⊞ ..........

32 :: EVENT : com.sun.tools.jdi.EventSetImpl$MethodExitEventImpl|active Agents:1
33 : FreshAgent: MethodExitEvent received.
34 : ACTION:EmotionSystem.action.PrintVariablesAction
35 :: ENTERING PrintVariablesAction : execute
36 ⊞ ..........

37 : EVENT : com.sun.tools.jdi.EventSetImpl$ExceptionEventImpl|active Agents:1
38 : ACTION:EmotionSystem.action.RetryFrameAction
39 : Pop frame at method:getArrayVal
40 : EXITING ABNORMALLY at EventThread
41 : added to Envmap:COMPILETIME:Tue Jul 15 13:04:46 GMT-05:00 2004
42 Process terminated with exit code 1
```

Listing 5: Application's log file

```
1 INFO : LOG NAME:CLIENT_040808_181317.log
2 ⊞ ..........

3 FATAL: Exception occurred: java.lang.ArrayIndexOutOfBoundsException.
4 FATAL: Attempt to access element:6 from array of length:6
5 FATAL: Element access point changed to:5 and access retried.
```

Listing 4 is the output of the Emotion System, when it is connected to the target application.

Initialization. Lines 5 to 15 refer to the initialization phase of the Emotion System. The following components are initialized by the system:

1. `EmotionMgr` - is responsible for the input/output channel (using Java RMI).

2. Command line arguments - that provide parameters to control the initialization of the Emotion System.
3. Persistent state - the persistent state is loaded into the emotion system.
4. Emotions state - all the emotions objects are initialized with the persistent state received.
5. `EmotionMonitor` - is responsible for the control channel (using JPDA). This channel is open for connections on a predefined port.

Agent initialization. Once the connection from the application is received, the agents are set up. During the agent initialization process, the application virtual machine is suspended. As shown in Lines 16 to 22 of Listing 4, the agents are initialized. Each agent queries the corresponding set of emotions and determines whether it is active or passive. The agent initialization process also generates a set of actions that set up requests for required events during application execution. This is why agent initialization happens after the connection is received. As shown in Line 22, method exit requests are added.

Sensation integration. Lines 24 to 30 refer to the sensation being received and integrated by the Emotion System. The initial sensation is received from the application via the input/output channel. The complete sequence of operations after the sensation is received is performed while suspending the application virtual machine. The sensations are first transformed by the emotion. This process parses the extra information passed with the sensation. In the case of this experiment, the timestamp is extracted and compared with the timestamp in the emotion environment. The integration mechanism of the emotion filters the sensations it has been defined to receive and factors in the strength of the sensation. The result of the pass shown in Listing 4 is that the emotion is still active.
The working of the integration mechanism can be seen in Lines 11 to 13. The map for the initial sensation is retrieved from the persistent storage. The column number 10 has the value of 10, while the column number -4 has the value of 1. Each new sensation adds value to the negative spectrum of the histogram, as shown in Line 24. As the values go higher, to maintain the idempotent property, the sensation map is restructured. In this case the positive column with the large values are trimmed down to a great extent. When the median of the histogram becomes less than the mean ($mean = 0$), the status for that sensation becomes inactive. As the only sensation for the novel emotion, this renders the status of the emotion inactive.

Action effect. Lines 32 to 35 show the `PrintVariablesAction` being executed on a `MethodExitEvent`. This action writes log information to the application log file created in Line 7. The `ExceptionEvent` is caught and reacted upon in Lines 37 to 39. The output generated as a result of the `ExceptionEvent` is shown in the Listing 5.

Shutdown. Lines 40 to 41 refer to actions performed on receiving the `VMDisconnectEvent`. This event notifies the Emotion System that the target application has stopped executing. The emotion system reacts by storing the current emotion state in the persistent storage and performs other closing actions (e.g. closing the application log file).

The ability to dynamically turn on debugging features allows users to test the application automatically, that is without inserting debugging code that has later to be removed. The overhead of the Emotion System goes to zero, as the application matures.

5.2 Example emotion: {warning, cautious, carefully}

In this experiment, we shall look at a possible solution to the problem of transient exceptions. The experiment also shows the case of a distributed effect (an effect in a method or component far from where the sensation is collected).

The emotion set of {warning, cautious, carefully} will be used for this experiment. The *careful* agent is responsible for switching on exception monitoring and taking appropriate actions in response. The state of the careful agent relies on the *cautious* emotion that integrates the *warning* sensations.

Listing 6 is the target application that simulates the transient bug. Line 22 is the place where the transient bug occurs. This bug is thrown if static variable $q > 0$. The bug is transient in nature, since the method on reexecution will have the value $q = 0$. We insert a warning sensation (role of a programmer's hint) that defines the possible scenario that activates the warning.

This application is faked, as a real transient error is hard to produce on demand. However, the benefit of the Emotion System for the target application does not depend on the prior knowledge of the transient error. The integration mechanism handles repeated sensations and can predict the accuracy of the sensation from the given context.

Listing 6: Transient bug application

```
1  public class transientBug{
2      static int q = 1;

4      public static void main(String[] args) {
5          int x = 5, y = 5, z;

7          initRemoteMgr(args); //initialize remote manager

9          //send warning sensation that depends on certain conditions
10         sendSensation(Sensation.warningSensation,(q > 0 )?"1":"0");

12         z = multiply(x,y);
13         System.out.println(x + " * " + y + " = " + z);

15     }
16 ⊞ ..........

17     public static int multiply (int i, int j){
18         System.out.println("q = " + q);
19         try {
20             if(q--- > 0){
21                 System.out.println("ERROR: Throwing exception.");
22                 throw new Exception(); //simulate the transient bug
23             }
24         } catch (Exception e) {
25             return 0;
26         }
27         return i * j;
28     }
29 ⊞ ..........
```

Listing 7 shows the runtime output of the application without emotions. Because of the transient bug, the application fails and produces the wrong output (Line 3).

Listing 7: Target application output without Emotion System

```
1 java transientBug -noemotion q = 1
2 ERROR: Throwing exception.
3 5 * 5 = 0
```

Listing 8 is the output from the Emotion System connected to the target application. To give the Emotion System a blank slate of state information, it has been started with the command line option to reset it (Line 1).

Listing 8: Emotion system connected to target application

```
1 java (...) EmotionSystem.EmotionMgr -reset
2 ⊞ ..........

3 : EVENT : com.sun.tools.jdi.EventSetImpl$VMStartEventImpl|active Agents:0
4 : Sensation received:warning
5 : strength = 1.0:transformed = 3:index = 13
6 : -10:0,-9:0,-8:0,-7:0,-6:0,-5:0,-4:0,-3:0,-2:0,-1:0,
7   0:0,1:0,2:0,3:1,4:0,5:0,6:0,7:0,8:0,9:0,10:0
8 : Emotion [cautious]:Active
9 : ID=EmotionSystem.generator.CarefulAgent| CLASS=EmotionSystem.generator.CarefulAgent set
      active.
10 : ACTION:EmotionSystem.action.ExceptionRequestAction
11 : ExceptionRequests added with SUSPEND_ALL=true
12 : EVENT : com.sun.tools.jdi.EventSetImpl$ExceptionEventImpl|active Agents:1
13 : CarefulAgent: exceptionEvent received
14 : Emotion [cautious]:Active
15 : ACTION:EmotionSystem.action.RetryFrameAction
16 : Pop frame at method:multiply
17 ⊞ ..........
```

The warning sensation sent in Line 10 of Listing 6 is received in Line 4 of Listing 8. This sensation is sent with the strength of 1. However, internal transformation results in the value of 3. The transformation converts the user-specified sensation strength to a range suitable for the integration mechanism in use (the histogram mechanism in this case). The sensation map and consequently the emotion (caution) is turned active (Line 8). This activates the careful agent and causes appropriate action(s) to take place (Lines 9 to 11). The action in this case is the request to monitor exception events in the application virtual machine.

The exception in Line 22 of Listing 6 is received in Line 12. The only active agent, the careful agent, responds with the action to retry the frame. This effectively executes the method again and overcomes the transient bug.

The result of the Emotion System on the application is shown in Listing 9. The exception is thrown as shown in Line 3. On reexecuting the method, the value of q matches the requirements and hence the method completes successfully. The result is now correct, as shown in Line 5.

Listing 9: Target application output with emotion system

```
1 java (...) transientBug
2 q = 1
3 ERROR: Throwing exception.
4 q = 0
5 5 * 5 = 25
6 Process terminated with exit code 0
```

An interesting aspect of this experiment is that, though the method is reexecuted, the static data structure is not modified. This is because of the fact that the action just retries the current

`stackframe` – the unit corresponding roughly to each method call. Therefore, the static data structure, and for that matter any data structure at lower stackframes (earlier methods), is not modified.

Though not displayed in the listings above, one more integral part of the exception-catching process is saving the location of the exception in the environment. This prevents catching exceptions in an infinite loop. Once an exception of the same location is reached for a predefined number of times, the emotion becomes inactive. This is accomplished through a negative sensation (feedback to the Emotion System) sent each time an exception is reached at the same location.

The most useful feature of this experiment is that the hint did not have to predict where the exception would take place, and where the corresponding effect should be generated. This enables programmers to give hints at the programming stage, without worrying about the certainty of the hint, or the corresponding solution.

6 Discussion

The use of emotions presents an approach for 'state-based operation execution', and demonstrates some of the benefits of this approach. However, we realize that there are some limitations and issues that have to be understood, and possibly dealt with during the future work.

With reference to the demonstration in Section 5.2, the key assumption is that the reexecution of the method does not have side effects. This approach assumes that persistent objects remain in a stable and correct functional state during the reexecution of the method.

One possible approach to overcome this limitation is to create save-points before the method entry, where save-points store the entire state of execution (variables, thread states, method locations, etc.). Before the reexecution of the method, the state is reverted back to the most recent save-point.

Another approach is to assume that the change of persistent objects during reexecution of the methods is vital for the recovery approach (as evident in the demonstration of Section 5.2). Since the exception itself signifies the presence of an error, the recovery is now a heuristic approach towards stability rather than correctness.

In the current implementation, there is a single threshold that triggers the change between two different states of an emotion. The main reason for this is that it is hard to define multiple responses to different levels of emotion; but there is no difficulty, in principle, in defining multiple thresholds for an emotion.

Experiments with the Emotion System have shown that the overhead it imposes is proportional to the number of active emotions and the effect of the adverbial agents. For example, with no emotions active, the overhead of the Emotion System is close to 0. With the *careful* emotion active, exception event monitoring is turned on. This results in increase of the overhead of the Emotion System.

6.1 Applications of the Emotion System

Realizing the limitations and issues of the Emotion System, there are obvious applications where the Emotion System is not appropriate. These include real-time systems, systems that require certification (e.g. security certification), and critical systems that require precise behavior.

Some of the possible applications of the Emotion System are:

Maintenance/developing. During these stages, useful operational modification can be achieved without having to modify the actual software.

Applications that require robustness. Defining emotion maps, along with sensations that predict possible exception scenarios, the applications can achieve robustness with little overhead.

Network applications. Emotions can integrate sensations about network activity, and predict the possibility that the current packet is intrusive.

Post development enhancements / what-if choices. New algorithms or enhancements can be added and chosen based on pre-defined scenarios, without modifying interfaces of methods or components. This is especially useful in case where the actual code cannot be changed, as in case of code libraries.

7 Conclusions

Some of the challenges faced by autonomic computing are: how to collect appropriate information about program behavior, especially when programs are large and built by many people; how to modify program behavior; what the mapping between information and changed behavior should be; and how to present autonomic systems so that humans can design to take advantage of them.

The Emotion System is a partial solution to all of these challenges. Information is collected both automatically (time since compilation) and from software developers. However, information from all sources is treated as speculative rather than authoritative, acknowledging that, in most contexts, the global usefulness of local information cannot be judged locally.

The mapping from information to changed behavior is modelled by emotions, which act as integrators of positive and negative 'hints' and which apply thresholding, so that changed behavior is stable and not easy to manipulate.

Modification of program behavior is done by adapting mechanisms originally designed for debugging, allowing an arbitrary granularity of intervention. Some behavioral changes can be constructed automatically, but the need for software developers to provide variant operational execution strategies is also permitted.

Finally, the metaphor of emotions is designed so that software developers may understand the mechanisms in a natural way, both with respect to their provision of sensations, and their understanding of variant operational behaviors.

References

[1] A. Baratloo, P. Dasgupta, V. Karamcheti, and Z.M. Kedem. Metacomputing with MILAN. In *Heterogeneous Computing Workshop*, pages 169–183, 1999.

[2] R. Berry. Trends, challenges and opportunities for performance engineering with modern business software. *Software, IEE Proceedings*, 150(4):223–229, 2003.

[3] J. R. Cordy, C. D. Halpern-Hamu, and E. Promislow. TXL: A rapid prototyping system for programming language dialects. In *Computer Languages*, volume 16, pages 97–107, 1991.

[4] J. Kephart and D. Chess. The vision of autonomic computing. *IEEE Computer*, 2003.

[5] R. Laddad. Aspect-oriented programming will improve quality. *IEEE Software*, 20(6):90–91, 2003.

[6] R. Sterritt and D. Bustard. Autonomic Computing - a means of achieving dependability? In *Proceedings of the 10th IEEE International Conference and Workshop on the Engineering of Computer-Based System*, pages 247–251, 2003.

[7] Sun Microsystems, Inc. *Java Data Objects Specification*. `http://java.sun.com/products/jdo/`.

[8] Sun Microsystems, Inc. *The Java HotSpot Virtual Machine - Technical White Paper*. `http://java.sun.com/products/hotspot/docs/whitepaper/Java_Hotspot_v1.4.1/Java_HSpot_WP_v1.4.1_1002_1.html`.

[9] Sun Microsystems, Inc. *Java Remote Method Invocation - Distributed Computing for Java*. `http://java.sun.com/products/jdk/rmi/reference/whitepapers/javarmi.html`.