Chapter 6

The JaM Machine API

6.1 Introduction

In some previous research work on definitive notations, prior to the implementation of the *Tkeden* interpreter, the practical emphasis was on the design and implementation of new definitive notations. These notations were specific to particular applications. The DoNaLD notation [ABH86] for line drawing and the SCOUT notation [Dep92] for screen layout were both developed with this aim in mind, as was Stidwill's partial implementation of the CADNO notation [Sti89]. For each notation, a translation process translates definitions in the application specific notation into EDEN definitions and data types through one-way communication along the UNIX command pipeline. In practice, interaction with scripts of definitions was inefficient because of the process of converting application specific data types into EDEN data types. This inefficiency was compounded by the slow process for the interpreted execution of EDEN code, in which translated data types typically require a myriad of EDEN interpreted operators and procedures to manipulate them.

The *Tkeden* interpreter does not support the command pipeline and, as a consequence, does not support the development of new application specific definitive

notations[1]. This problem is the motivation for the *Java Maintainer Machine Application Programming Interface* (JaM Machine API), a general purpose programming toolkit. The purpose of this toolkit is to allow a programmer to develop and implement new definitive script notations with data types and structures appropriate to the application in question. Applications developed with the API use compiled code for actions and update of definitions. This leads to more efficient execution of notations than the translation and interpreted execution mechanism of EDEN.

The most significant features of the JaM Machine API, which contribute new possibilities for the implementation of tools that support empirical modelling, are:

- Independence of data structures from the operators used to define dependency relationships between them. This is enabled by the object-oriented representation of the data types and operators. The JaM Machine API is a general purpose toolkit for the construction and integration of application specific definitive notations.

- Support for multi-agent interaction with scripts of definitions. Each definition has associated read/write permissions, supported by usernames and passwords for the secure identification of agents.

- Use of the Java programming language, with the following benefits:

  - multi-platform execution of empirical modelling tools without the need for compilation;

  - access to the class libraries of the Java APIs that support multi-threading, networking, graphical user-interfaces and integration of applications into world-wide-web browsers;

[1]Although the previous versions of EDEN are still in existence, they are not maintained.

– dynamic extension of compiled code for data types and operators of an empirical modelling notation on-the-fly.

The JaM Machine API update mechanism is based on the DM Model presented in Chapter 4. The general purpose nature of the JaM Machine API is realised through the object-oriented analysis of definitive scripts. This analysis is presented in Section 6.2 of this chapter along with a description of the classes that make up the API and guidelines for using them. Support for multi-user interaction within a single script is built into the API, based on a novel approach to agency similar to that used to protect file access on a multi-user computer system. This approach is presented in Section 6.3.

Scripts in definitive notations created using the JaM Machine API are known as *JaM scripts*, these scripts are written in *JaM notations*. A JaM notation is intended to be seen as a kind of intermediate, lower-level code for a higher-level definitive notation. To implement a notation such as DoNaLD [ABH86] with JaM Machine classes would require a parser for DoNaLD that performs some transformation from the DoNaLD script to a JaM script. As the JaM classes form an API, they can be integrated with any other Java software to provide dependency maintenance components within that software. Mechanisms for the incorporation of the dynamic type systems and dependency maintenance within other software are described in Section 6.4.

The final section of the chapter (6.5) shows a simple example of a JaM notation for arithmetic that is integrated into an *Arithmetic Chat* server/client application. The JaM Machine API is not developed specifically to support geometry and it is not intended that this programming toolkit is in any way similar to a programming API for geometry, such as *Djinn* [BCJ+95, BCJ+97]. The JaM Machine API is a general purpose programming toolkit for dependency maintenance.

A major case-study that uses the JaM Machine API is presented in Chapter 7, in which the API enables the creation of a definitive notation for the modelling of three-dimensional *empirical worlds*. These worlds have data types and operators to describe three-dimensional solid geometry.

6.2 Object-Oriented Analysis of Definitive Scripts

In the same way that a definitive script was analysed in Chapter 4 for the explanation of the DM Model, the underlying concepts of the JaM Machine API are based on the analysis of definitive scripts. These observations lead to the description of classes of objects that encapsulate the underlying components and mechanisms of dependency maintenance. The JaM Machine API is a set of classes in a Java package called "`JaM`". Rather than the one data type $\mathcal{Z}$ over which dependency maintenance is considered in the DM Model in Chapter 4, data types in JaM notations are represented as classes of objects that extend `DefnType`, an *abstract class*[2] in the JaM package.

Figure 6.1 shows the relationships between classes in the JaM package. The classes shown are the public classes [CH96] and those that are required for a thorough explanation of the API. If a class is connected to and on the right of another class then it extends the class to the left. Classes shown in the diagram in a box surrounded by a dashed line are abstract classes.

The breakdown of definitive scripts into classes of the JaM package is described below. Where appropriate, reference is made to the JaM specific definitive script shown in Figure 6.2 to explain how the analysis of a script leads to such a classification of classes. JaM scripts have similar properties to DM Model repre-

[2] An abstract class is one which cannot be instantiated to become an object as it contains stubs for the body code of some of its methods. Classes that extend an abstract class must implement these stubs. See [CH96] for more details.

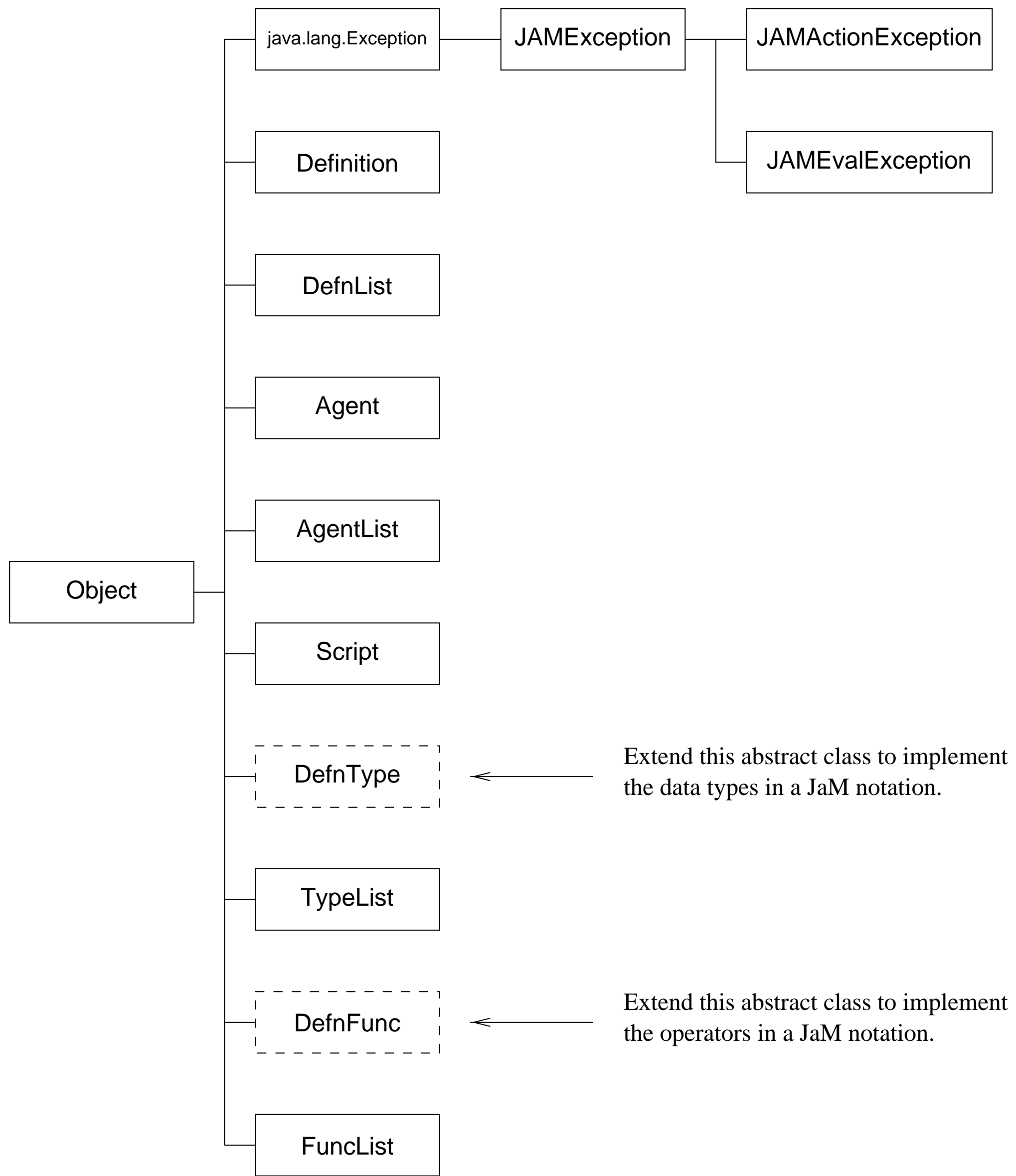


Figure 6.1: Class diagram for the JaM Machine API's package `JaM`.

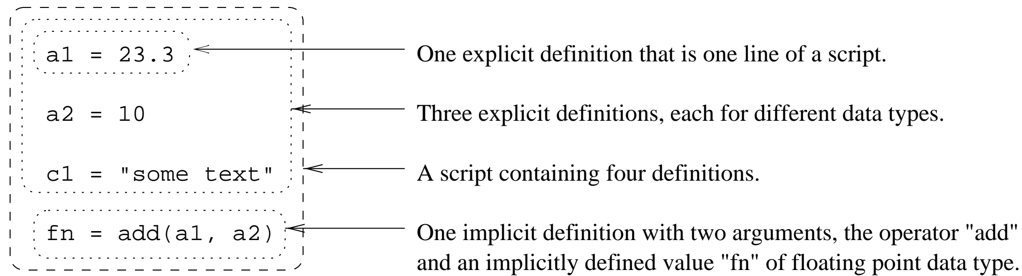


Figure 6.2: An example of a four definition JaM script in a definitive notation with its component features annotated.

sentable scripts (see page 104): operator composition is not allowed and arguments to operators must be other identifiers that already exist in the script. A programmer can implement a translating process to convert definitions from a notation with composition to a JaM notation using a parsing tool or a method similar to the *DoNaLD to DAM Translator* presented in Chapter 5 (Section 5.4). In contrast to this translation process, identifiers are not replaced by reference numbers as they are in Stage 3 of the DM Model transformation process (see page 107). Each definition in a JaM script is identified by a string.

`Script` A `JaM.Script` class represents a set of objects that describe definitive scripts, including their current state and the mechanisms to update that state. A script object for the example shown in Figure 6.2 would contain a list of four definitions, data types for integer, floating point and string values and one operator "`add`". Instances also contain a queue of redefinitions awaiting a call to their `update()` method, which has the effect of updating the state of the script consistent with the definitions on the queue. This class is described further in Section 6.2.4.

`Definition` An instance of the `JaM.Definition` class represents one line of a definitive script. In the script shown in Figure 6.2, each definition is either of the explicit form "*identifier* = *value*" or the implicit form "*identifier* = *operator* `(`*ar-*

guments)". For every definition object, there is a variable for the string representing the *identifier* (`a1`, `a2`, `c2` or `fn` in the figure), along with variables to store the current *value* associated wit the definition (`23.3`, `10` etc.) in the current state of the script and additional information such as who owns the definition. Additional information is associated with an implicit definition: a reference to the defining *operator* (`add` in the figure) and references to the definitions that make up the sequence of *arguments* (`a1` and `a2`) to the operator. Definitions are discussed further in Section 6.2.1.

`DefnList` A `JaM.DefnList` represents an indexed list of zero or more definitions. An instance of this class contains a doubly-linked list data structure for references to objects which are instances of the `Definition` class. These lists are used to represent the script and queue in a `Script` class, dependency structures in the `Definition` class and the arguments to an implicit definition.

The construct "`(a1, a2)`" in Figure 6.2 is a textual representation of a list of definitions. More information on how this class is accessed is presented in Section 6.2.3 which describes operators in JaM scripts.

`Agent` Every definition in a JaM script can be associated with an instance of a `JaM.Agent` class who is the agent owner responsible for the definition. The use of agency in the JaM Machine is presented in Section 6.3 with the `Agent` class itself described in Section 6.3.2.

`AgentList` Instances of `AgentList` contain doubly-linked lists of references to instances of the `Agent` class. `AgentList` instances are used to establish an access control mechanism with username and password databases, as described in Section 6.3.

`DefnType` The data types of JaM notations are created by the implementor of a

notation by extending the abstract class `JaM.DefnType`. This process is described in Section 6.2.2 of this chapter. In the example script in Figure 6.2 the data types are floating point numbers, integer numbers and strings of characters.

`TypeList` Instances of `TypeList` contain doubly-linked lists of references to instances of classes that extend `DefnType`. This class is used to represent the current list of available types in an instance of the `JaM.Script` class. Three data types are illustrated in the example script. These three types would have a representation in the `TypeList`.

`DefnFunc` The operators of the underlying algebra over the data types of a JaM notation are created by the implementor of a notation by extending the abstract class `JaM.DefnFunc`. This process is described in Section 6.2.3. An operator in a JaM notation can be defined to map from the values of any sequence of arguments of the data types of that notation to the value associated with an identifier, that can also be of any type. This flexibility can be restricted to provide a more appropriate underlying algebra for the given type structure of an application specific notation. In the example script in Figure 6.2, the only operator is `add`, the argument list is a list of identifiers of ordinal types and the implicitly defined left-hand-side value for the *identifier* `fn` is a floating point value equal to `33.3`.

`FuncList` Instances of `FuncList` contain doubly-linked lists of references to instances of classes that extend `DefnType`. This class is used to represent the current list of operators available in an instance of a `JaM.Script` class. Only one data type is available in the example script and this would be the only element of the operator list in the script instance representing the example.

`JAMException` If an exception is caused by the execution of one of the methods in one of the other classes, a `JAMException` is thrown, requiring the code that called the method that there was a problem in its execution. More information on Java exception handling can be found in [CH96]. The exceptions thrown by methods of classes in the JaM package are tabulated in Appendix A.2.

The rest of this section of the chapter concentrates on the major components of the object-oriented analysis of definitive scripts. Where appropriate, the relationship between the DM Model presented in Chapter 4 and the JaM package classes is emphasised to highlight the use of the block redefinition algorithm (see Section 4.3.3) to update the state of JaM scripts.

6.2.1 Definitions

Two kinds of definitions are possible in a JaM notation, an implicit or an explicit definition. An explicit definition is of the form "*identifier* = *value*". When an explicit definition appears as a redefinition in a JaM script, its string description has the *identifier* token replaced by a name for the definition and the *value* replaced by a string of characters exactly describing the value of a data type supported in the JaM notation. From this string description, it is possible to infer which data type the `value` represents and to establish the data type associated with the *identifier*. Explicit definitions are like definitions in the DM Model that use the "$value_X()$" constant operator, which maps the associated value of any identifier to the constant value X.

An implicit definition in a script is of the form

identifier = *operator* (*arguments*)".

The *identifier* token should be replaced by a string name for the definition and the *operator* by a string name for an operator available in the current JaM notation. The

arguments to the operator are a comma-separated sequence of other identifiers whose names exist prior to this implicit definition in the JaM script. Implicit definitions establish the value associated with the *identifier* by evaluating the *operator* with the values in the sequence of *arguments*. The data type associated with the *identifier* in the script given by an implicit definition depends on the ordering of the data types in the list of *arguments*.

Implicit definitions in a JaM script are similar to implicit definitions in the DM Model. For each identifier there is an associated operator in the same way that for a reference in the DM model there is an associated mapping (F) belonging to the set $\{f \mid f : \mathcal{Z}^* \rightarrow \mathcal{Z}\}$ (see Sections 4.2.1 and 4.2.5). Also, for every implicitly defined identifier in a JaM script there is an associated sequence of arguments similar to the mapping (D) that takes a DM Model reference to a sequence of references. Every identifier in a JaM script has a current value in the same way that every DM Model reference can be mapped to a current value (V).

The data fields in a `JaM.Definition` class are shown in Table 6.1. In this table, the Java types for the fields are shown in the left-hand column, the identifier given to the fields in the class in the central column and a description of each field in the right-hand column.

6.2.2 Data Types

Conventional parsing methodologies for high-level languages use context-free grammars. A similar process is required for the parsing of values in explicit definitions in scripts, although the grammars used in a JaM script are simple. As every explicit definition in a JaM script can be observed to be of the form "*identifier* = *value*", it is easy to put in place a mechanism to parse the characters before the equals sign as a string identifier. More complicated is the process to match the string of characters on the right hand side of the equals sign to a regular expression describing a

Java Type	Name	Description
`String`	`name`	The *identifier* for an instance of the class in a JaM script.
`DefnType`	`value`	The *value* for an internal representation of a definition in its current state.
`DefnFunc`	`f`	The *operator* used to implicitly define the value of an instance of this definition. Set to `null` if the definition is explicit.
`boolean`	`fExists`	A boolean which is `true` if an instance of the class is implicitly defined and `false` if it is explicitly defined.
`DefnList`	`dependencies`	List of object references to definitions on which the value of this definition depends directly. This represents the sequence of *arguments* to the implicit definition. Set to `null` if the definition is explicitly given.
`DefnList`	`dependents`	List of object references to definitions that directly depend on this definition.
`int`	`knuthCounter`	The Knuth counter used in the block redefinition algorithm.
`Agent`	`owner`	The owning agent of this definition is a JaM script.
`boolean`	`ownerRead`	A `true` value if the `owner` can reference this definition in another implicit definition, `false` otherwise.
`boolean`	`ownerWrite`	A `true` value if the `owner` can redefine this definition instance, `false` otherwise.
`boolean`	`allRead`	A `true` value if agents other than the `owner` can reference this definition instance in their implicit definitions, `false` otherwise.
`boolean`	`allWrite`	A `true` value if agents other than the `owner` can redefine this definition instance, `false` otherwise.

Table 6.1: Data fields of a `JaM.Definition` class.

particular type and to parse the value into some internal representation.

A data type in a JaM notation is described by a simple context-free grammar that matches a pattern of characters. Each regular expression establishes a class of values that can be characterised and uniquely recognised by their string representation. Java supports a number of primitive data types and compound object data types that may be a more suitable and space efficient representation for a value given by a string. For example, the most efficient way to represent a string of characters that matches with a regular expression for a string in a Java application is through the use of a string type object. If a value matches with the regular expression for an integer, then the most efficient way to store this value is as a variable of the `int` Java data type. For a particular data type classification, it may be possible to find a translation methodology from a string of characters to an internal efficient data representation and from the internal representation back to a string of characters.

Data types in a JaM notation are defined by a programmer by implementing classes that extend the abstract class `JaM.DefnType`. Extended classes should contain data fields to hold an internal representation of the data of the type available to a user of a JaM script. The class must implement a method to match or reject a string description of a value given in an explicit definition in a script (`recognise()`), to parse this string description into its internal data representation (`parseIt()`) and to convert the internal data representation back into a string consistent with the current state of the value (`printIt()`). The programmer of sub-class of `DefnType` should ensure that a string for a value produced by the conversion of the internal representation of the data can subsequently be parsed back into the internal representation.

The methods that a programmer must implement within a class (X) that extends `DefnType` are shown in Table 6.2. Each row corresponds to the name of a method (**Name**). The types of arguments passed to each method are shown in the

column labelled **Passed**, the methods return type in the column labelled **Returns** and a description of the methods purpose in the column labelled **Description**.

Figure 6.5, located close to the discussion of its contents on page 219, presents code for a class that extends `DefnFunc` that is for the representation of an integer number data type in a JaM notation.

6.2.3 Operators

Only prefix operators feature in JaM notations. No infix or postfix operators are available. They are used to define the indivisible relationship between data values in a JaM script. Operators have a string name that corresponds to the name that they use in a script. JaM operators are conceptually similar to operators in the DM model. Every operator in the DM Model is a member of the set $\{f \mid f : \mathcal{Z}^* \rightarrow \mathcal{Z}\}$ for the one data type $\mathcal{Z}$. Each JaM script operator can conceptually map from any sequence of values of any data type X to a value of any data type Y.

A programmer defines new data types in a JaM notation by extending the abstract class `DefnType`, as described in Section 6.2.1. The string value of an explicit definition is converted through a method call into some internal data representation and there are methods to convert this back to a string representation again. Two mechanisms are available for a programmer to manipulate data by an operator:

1. It is possible to define JaM operators that convert the internal value representation of a sequence of arguments into strings again, perform an operation by manipulating these strings to form a new string. This can then be reinterpreted as a value of a data type that itself extends `DefnType`, by using the `recognise()` and `parseIt()` methods. This mechanism allows for the description of some novel operators, but it is slow and should be used sparingly.

2. Use the internal data representations of classes (fields) representing data types

Method	Passed	Returns	Description

X	`String`	X	A constructor method for an instance of X. The constructor method should always adhere to the following generic template for its code:
			`X(String name) {`
			`  super(name);`
			`  ref = 1372; // substitute real ref.`
			`}`

`makeNew`	—	`void`	A method that is used to return a new instance of the data type represented by class X. This method should always adhere to the following generic template for its code:
			`DefnType makeNew() {`
			`  return new X(this.name);`
			`}`

`recognise`	`String`	`boolean`	A method that returns `true` if the passed `String` object is matched as being a pattern of characters that can be converted into the internal data representation for the data type, otherwise returns `false`.
`parseIt`	`String`	`void`	A method that converts the characters represented by the passed `String` object into the internal data representation for the instance of the class for which the method was called. The method `recognise()` is guaranteed to have returned a `true` value for the same `String` prior to the invocation of this method.
`printIt`	`int`, `int`, `int`	`String`	A method that converts the internal data representation for the instance of the data type X for which it was called, into a `String` representation of the value. The three passed integers give hints to the method on how best to format this value string.
`action`	—	`void`	A method that can contain some procedural code that is executed every time a value of this data type is redefined or updated. The simplest acceptable implementation of the method is an empty block “{ }”.

Table 6.2: Methods that must be implemented for a JaM data type class X that extends `DefnType`.

directly.

For every implicit redefinition, the data types in the sequence of arguments need to be checked to see that they are appropriate for the operator. This process also determines the type associated with the identifier on the left-hand-side of the definition. Any type change by the redefinition of an existing definition d have to he propagated through the dependency structure to type check all the operators that define the dependents of d ($dependents(d)$ in the DM Model).

Operators in JaM notations are implemented by extending the abstract class `DefnFunc`. The programmer must implement the methods shown in Table 6.3, which include an argument sequence checking mechanism (`typeCheck()`) and a method for the evaluation of the value mapped to by the operator (`f()`). Figure 6.6, located close to the main dicussion of its contents on page 221, shows some annotated example code for a class that extends `DefnFunc` and therefore represents an operator in a JaM notation.

Both methods `typeCheck()` and `f()` are passed references to a `DefnList` object that represents the arguments sequence for the operator in a JaM script. Two public mechanism are available to access the `DefnList` class. The first is a public integer variable `length` in the class that is the length of the arguments sequence. The second is a method `arg(`n`)` that returns the n-th element of the arguments sequence as an instance of `DefnType` that represents the current value for the that argument. As an example of this method, to find the reference number for the type of the second argument in sequence `dl`, a programmer can use the method call `dl.arg(2).ref`.

Method	**Passed**	**Returns**	**Description**

F	`String`	F	A constructor method for an instance of F. Typically a programmer will only instantiate one of these objects per JaM script. The constructor method should adhere to the following generic code template.
			`F(String name) {`
			`  super(name);`
			`  ref = 1372;  // Reference optional`
			`}`

`typeCheck`	`DefnList`	`DefnType`	This method is used to check that the passed `DefnList` is a sequence of data types appropriate to the operator. If it is, the method should return an empty instance of an object of the data type mapped to by the operator F for the given sequence of types, otherwise the method should return `null`.
`f`	`DefnList`	`DefnType`	This method carries out the evaluation of the mapping from the data values in the argument list `DefnList` to the implicitly defined value for the associated identifier. A programmer can rely on the fact that the method `f` is called only after the method `typeCheck()` has returned a non `null` object reference. The value returned should be a new instance of a class that extends `DefnType` with its internal data representation set to values consistent with the operator as used in the JaM script. Also, this returned object should be an instance of the same class returned by the `typeCheck` method.
`action`	`Definition`	`void`	A procedural action that is executed directly after the method `f()` is called to update the implicit value of an associated *identifier*. The action is passed a reference to the current state of the `JaM.Definition` instance for that *identifier*.

Table 6.3: Methods that must be implemented for a JaM operator class F that extends `DefnFunc`.

6.2.4 Scripts of Definitions

A script of definitions containing n lines can be represented by a `DefnList` class of length n. A JaM script is written in a particular JaM notation with data types that can be represented by classes that extend `DefnType` and operators by classes that extend `DefnFunc`. The states of a definitive script can be represented in a class that contains fields for: the script of definitions, a queue of pending redefinitions, the notation data types, and the notation operators over these data types. A mechanism for updating the current state of the script is also required.

A `JaM.Script` class contains the data fields shown in Table 6.4. This includes: the current JaM script of definitions (`theScript`), a queue for redefinitions (`queue`), a list of available types for the script (`tlist`), and a list of available operators over these types (`flist`) that make up the underlying algebra for JaM notation in which the JaM script is written. All these data fields are private to a script and can only be accessed through method calls.

Scripts change only through redefinition. If a programmer chooses to use a JaM script as a component of their application the first procedural step is that they must create a new script. Then they should decide what data types and operators the scripts that they are to use require and add these to the component type lists and operator lists of their script. The script initially has no definitions and no definitions waiting in the queue for update, similar to the initial state of the DM Model[3]. The next step in interacting with a new script is to add definitions to the definition queue. Once an initial queue is established, this list of definitions should be considered as the update for the initially empty main script so that the queue of definitions becomes the first state of this main script of definitions. This process is subject to checks for typographic errors in the definitions on the queue, for cyclic

[3]See Section 4.2.5.

Java Type	Name	Description
`DefnList`	`theScript`	The current state of the JaM script of definitions represented by an instance of the `Script` class.
`DefnList`	`queue`	A list of redefinitions that will be used for the next call to the `update()` method. This will update the current state of definitions in the list "`theScript`".
`TypeList`	`tlist`	A list of data types available in the current JaM notation for the JaM script represented by "`theScript`". These data types are represented by instances of classes that extend `DefnType`.
`FuncList`	`flist`	A list of operators available in the current JaM notation for the JaM script represented by "`theScript`". These operators are represented by instances of classes that extend `DefnFunc`.
`AgentList`	`usersList`	A list of user information for user names and passwords, enabling multi-user support for script notations. This list contains all users who could possibly interact with a script at some point.
`AgentList`	`currentUsers`	This list contains users who are currently registered as *logged in* and hence are able to make new definitions and redefine existing definitions (subject to currently set permission values).

Table 6.4: Data fields (all are `private`) for a `JaM.Script` class.

dependency between definitions on the queue and script and for type clashes caused by implicit definitions in the queue.

Then the process for adding to the queue and then updating continues in a continuous loop, forming the interaction procedure for incremental update of the JaM script. When an update occurs, the definitions in the queue are checked and then used to modify the current state of script S. With reference to DM Model $\mathcal{M}_S = (A, F, D, V)$, the role of the set of references A is played by the "`theScript`" list of definitions. For each definition reference in this list: the "`f`" field represents the mapping from the reference to the operator, in a similar way to the DM Model mapping F; the "`dependents`" field represents the list of arguments of the operator, in a similar way to the DM Model mapping D; the "`value`" field represents the current value, in a similar way to the DM Model mapping V. The counterpart of the set K of redefinitions is the "`queue`" field in a JaM script class. The algorithmic procedure to update the current state of a JaM script is based on the block redefinition algorithm.

The most important methods that can be called by a programmer to interact with an instance of a `JaM.Script` class are shown in Table 6.5[4]. The table shows the name of a method, the types of the parameters that it is passed, the Java type of the value that the method returns and a description of the method. Several of the methods are passed an integer *cookie* value (see page 202 for the definition of a *cookie*) before other parameters. Where this is the case, this is a unique identifier for a user who is currently logged into the script and has requested the action prescribed by a call to the associated method. This allows the method to throw an exception (`JAMException`) if the user does not have permission to carry out the action. Multi-user support is described further in Section 6.3.

Once a new instance of the `JaM.Script` class is created for a particular JaM

[4]This is apart from the constructor method for a `JaM.Script` that is described in Table 6.7.

Method	Passed	Returns	Description
`addToQ`	`int`, `String`	`void`	This method is called with a `String` representation of a JaM script redefinition and adds this to the `queue` of redefinitions.
`update`	`int`	`void`	This method is called to update the main script (`theScript`) consistent with the redefinitions on the `queue` of pending definitions.
`addType`	`int`, `DefnType`	`void`	This method is used to add a data type to the `tlist` of an instance of a `JaM.Script` for use in definitions of `theScript` to create the JaM notation data types for the script. Though this method is normally called during the initialization of a script, it can be called at any time during the life of a script instance, dynamically extending the type structure of the JaM notation.
`addFunc`	`int`, `DefnFunc`	`void`	This method is used to add an operator to the `flist` of an instance of a `JaM.Script` to create the underlying algebra for the script. This method is normally called during the initialization of a script, but can be called at a later point to dynamically extend operators available in the underlying algebra of a JaM notation.
`printVal`	`int`, `String`	`String`	Convert the current value of the *identifier* with the same name as the passed `String` into a string of characters in the format "*identifier* = *value*".
`printDef`	`int`, `String`	`String`	Convert the current state of the implicit definition associated with the *identifier* name passed as a `String` into a string of characters in the format "*identifier* = *operator* (*arguments*)". If the definition is explicit, the behaviour of this method is the same as `printVal()`.
`printAll`	`int`	`String`	This method converts every definition in a script into a string containing a JaM script for all the definitions. If the definition is implicit then the implicit format is used as for `printDef` method, otherwise if it is explicit then the explicit format is used as for the `printVal` method.

Table 6.5: Methods available for a programmer to communicate with an instance of a `JaM.Script` class.

notation, the data types and operators of that notation should be added to the instance to form the `tlist` and `flist` data fields. For each data type Y, a new instance of the class that represents Y should be created and this instance used in an argument to the `addType()` method[5]. The order in which this method is called is important as it affects the way in which data types are recognised in explicit definitions. The first data type added to the script is the last data type that a value string will be checked against for a true value from its `recognise()` method and vice versa. The process of calling the `addType()` method is similar to the need to carefully order the regular expressions in the input file to a scanner tool, such as "`lex`" [LMB92].

An example of this is shown in the code in Appendix A.3 where the call to `addType()` for the integer data type occurs after the call of the method for the floating point data type. This is so that numerical value strings that do not contain a period (full stop) character (or an "`e`" or an "`E`"), are recognised first as integers. If the string does contain a period it is not then recognised as an integer but as a floating point value.

Operators for the underlying algebra over the data types are added by calls to the `addFunc()` method. For this purpose, a new instance of the class representing the JaM notation operator should be created. The string name given to the instance should be the same as the name that will be used for the operator in the JaM notation. The `flist` field of an instance of a `JaM.Script` class is updated using the `addFunc()` method. The order in which the operators are added is not important[6].

[5]Only the `root` user can call the `addType()` and `addFunc()` methods. See Section 6.3 for more information.

[6]This is true as long as every operator has a different string name. No automatic check is in place to determine the uniqueness of these names.

6.2.5 Explicit Arguments to Implicit Definitions

One built-in feature of the JaM Machine API to assist a user interacting with a script is the ability to use an explicit value as an argument to an implicit definition. Consider the example where the `addToQ()` method of the `JaM.Script` class is called to introduce an implicit definition of the form "*id1* = *operator* (*arg1*, ..., *arg4*)". If the string of characters for *arg2* and *arg4* are explicit values instead of other identifiers that already exist in the script, then additional definitions are automatically created for these values. In the example below, the implicit definition is replaced by three definitions, two explicit and one implicit:

```
id1_2 = arg2
id1_4 = arg4
id1 = operator ( arg1, id1_2, arg3, id1_4 )
```

6.3 Multi-user Environments for JaM Scripts

A JaM script represents a set of definitions for values and persistent indivisible relationships between these values. These scripts are textual entities that can be shared between users by agreement that "*this script is the current state of our model*". Empirical modelling principles have been proposed as a good way in which to implement shared modelling systems to support methodologies such as concurrent engineering [BACY94b]. The JaM Machine API was the first tool to offer some technical support to a truly multi-user process for the creation and interaction with definitive scripts. If several different users are contributing to the definitions of a script simultaneously then it is useful to record who contributed which definition. By recording this information in instances of the `JaM.Definition` class, it is possible to ascribe ownership to each definition and manipulate this ownership so as to protect users who do not own a definition from redefining it.

Section 6.3.1 discusses the concept of permissions for controlling which definitions can be manipulated by which users, known as *definition permissions*. The mechanism used for controlling access to definitions is similar to that for controlling access to files on a multi-user computer operating system. The concept of agency used in the JaM Machine API is discussed in Section 6.3.2. In the following section, this is compared to the LSD notation [Bey86b], which is used for the analysis of agency in the modelling real-world situations and systems. The API can support some parts of the LSD notation in the implementation of models that use JaM notations.

6.3.1 Definition Permissions

One motivation for a modeller to construct a computer-based model of a real-world situation is as a mechanism to communicate qualities of the real-world referent through the model to other people. Another motivation is to assist in the process of thinking and reasoning about some real-world referent. In sharing a model, a modeller may wish to ensure that another user cannot undermine the integrity of the model. For example, in a model of a radio a modeller way not wish a user to be able to redefine the function of a control knob for tuning in stations so that it alters the volume of the sound instead. The modeller may also wish to protect definitions from accidental modification by themselves, as they are now committed to an explicit value or implicit indivisible relationship as observed from the referent.

For large models or models that require the skill specialisation of more than one person in their construction, a desirable environment is one which supports collaborative construction of models by more than one modeller simultaneously. In such an environment, modellers need to be able to protect their parts of the model to prevent accidental modification by users who do not comprehend their definitions and share other parts of the model with other modellers, either to allow them to

reference them in their own work or to allow them to interactively modify them.

For example, in the design of a new transistor radio, an electrical engineer designing the layout of a printed circuit board may wish to protect their layout of components from modification by the designer of the external case, but allow this designer to reference the current location of certain components on the board such as the volume control. The stylistic designer of the case may have control over the overall internal geometry of the case to which the electrical engineer must conform. Both modellers may have the ability to decide where to locate the headphone socket. All conflicts of interest need to be settled by negotiation between the two modellers.

When models are constructed as scripts of definitions, the text for these definitions can be easily shared between modellers through any means of textual communication. It is possible to associate each definition with a modeller by annotating the text. The scenarios discussed above can be classified as:

- modeller and user;
- modeller and commitment to components of a model;
- teams of collaborating modellers.

By associating with each definition information describing who has the ability to modify (redefine) or reference (request the value of) the definition, the scenarios listed above can be realised. The information associated with each definition is called the definition permission and, if implemented in a tool for dependency maintenance, can be used to control simultaneous access and interaction with one definitive script by several modellers and users simultaneously.

The concept of permissions associated with definitions in the JaM Machine API is based on an existing and proven method for users to share work and information while keeping some information private and personal. Multi-user computer

operating systems, such as UNIX [Joy94] and Microsoft's Windows NT [CS98], require a user to identify themselves prior to the start of an interactive session by typing a system-wide public username and secret password. During this session, all files which the user modifies and processes that they run are associated with their owning user. It is the job of users to make sure that they do not make available to others any files that are of a private nature and make public any information that they would like to share. It is also up to the user to protect important files so that they do not accidentally delete them or modify them once they are considered final. In this way, several users can interactively share the same devices such as disks, printers, processors or memory. Overriding control of the permissions to use a resource is given to a super-user, who is called "`root`" on UNIX or the "`Administrator`" on Windows NT.

On multi-user, computer-based filing systems, data files are protected on a file-by-file basis. To achieve the file protection on UNIX, each file has an associated *inode* that contains information such as date and time of the last modification of a file, the locations of a file on the physical disk surface and so on. It also contains some permission fields and owner information that describe information about which user owns the file, which group of users owns the file, whether the owning user can read, write/modify or execute the file, whether members of the owning group can read, write or execute the file and whether any other users can read, write or execute the file. Information similar to that recorded in an inode is associated with each definition by the JaM Machine API. In the current version, there is no support for groups and no concept such as the *execution* of definitions. For every definition, there is both an associated owner and permissions. The data fields in a `JaM.Definition` class are shown in Table 6.1.

JaM Machine permissions can be expressed as a string made up of four characters, of the form "`rwrw`". Each character is either as shown in the quotation

marks to indicate that the permission associated with the character is set, or a dash "`-`" to indicate that it is not set. For example, the first character is an "`r`" if the owning user has the right to reference the definition in the argument list to another definition, or can inspect the current value or implicit definition[7] in the current state of the script for the definition. If the user does not have these permissions, the first character is set to "`-`".

The second character for a definition permission refers to the owner's permission to be able to redefine the associated definition, explicitly or implicitly. It is a "`w`" if they can and a "`-`" if they cannot. The final two characters are the same as the first two except that they concern the permission for all non-owning users of a definition in a script to interact with the associated definition. The ways in which definition-owning agents can interact with scripts containing definitions are described in the next section. The default definition permissions set for a new definition in a script is owner reference and modify and non-owner reference only ("`rwr-`").

6.3.2 Definition-Owning Agents

Each potential definition owner is considered as an agent who can interact with a JaM script, contributing redefinitions. Every agent has a username and a password, the username uniquely identifying the agent. In the JaM package, agents are represented by instances of the class `JaM.Agent`. Every JaM script has a user list (`usersList`) of agents who are permitted to join interaction with it and a list of current users (`currentUsers`) that are actively interacting with the script (see Table 6.4).

To create a multi-user environment, a special super-user agent is required

[7] To inspect the current value of a definition, the `printVal()` method of `JaM.Script` is used. To inspect the current implicit definition, the `printDef()` method is used. See Table 6.5.

with the ability to add and delete other agents from the overall list of users. This super-agent is called "`root`" and every script must have one. When a script is initialized without an agent list specified, the only user is the `root` user. If a programmer wishes their script to only be interacted with by one agent at a time then this one agent should have username "`root`". This user still has to become active by *logging in* before they can call any of the other methods for an instance of a `JaM.Script` class.

Table 6.6 shows the methods in the `JaM.Script` class that support multi-user interaction. To create a new JaM script, a programmer first has to call the constructor method `Script`. Two variations to this method are available, depending on the types of the arguments that it is passed. To start a new agent database with only the `root` user or to initialize a script with only one agent, the method is called with an empty argument list. This generates a new instance of a script and the `root` agent must then use the `login()` method to start using it. This process assigns the `root` agent as currently active and generates a random and unique integer *cookie* that is used in other method calls of the script instance attributed to `root` until the agent logs out.

The code excerpt shown below demonstrates the single-user mechanism for the use of an instance of the `JaM.Script` class. The first two lines of the example code below shows the minimum code required to create a JaM script `s1`. The following lines contain code to add one data type and operator, add a new definition to the queue and update the current state of the initially empty script. When the `root` agent has finished interacting with the script, it is *logged out* using the `logout()` method.

```
Script s1 = new Script();
int root = s1.login("root", "PHDBOD");
⋮
s1.addType(root, new JaMInteger("JaMInteger"));
s1.addFunc(root, new JaMAdd("add"));
s1.addToQ(root, "a = 34");
s1.update(root);
⋮
s1.logout(root);
```

Notice from the code above how the `root` agent's *cookie* must be passed to all method calls to associate the method call with the `root` agent. In the example code, the identifier "`a`" is set to have owning agent `root`. At the end of the example, `root` is logged out using the `logout()` method and is no longer interacting with the script. The next time that the `root` agent logs on, they will be assigned a different *cookie* for reasons of security. It is by passing the *cookie* every time that a method such as `addToQ()` is executed that allows the system to check whether a particular user agent has permission to carry out a particular action, depending on currently set definition permissions.

It is up to the programmer who writes an application that uses a JaM script to prescribe how agents are going to interact with a script. It is possible to create completely computer-based autonomous agents, perhaps executing in different threads, that perform some regular or randomly-occurring redefinition. To protect the integrity of the redefinitions of an autonomous agent, any definitions that the autonomous agent shares should be associated with the definition permissions "`rwr-`". Other user agents may be connections to computer terminals with textual interfaces directly to the script and other agents may be interacting with the script through a prescribed graphical interface with buttons, tick boxes, choice menus and so on. The programmer of an application needs to ensure that all users log in and that every action that the agents subsequently takes results in a method call to the

script with the agent's *cookie* as the first argument.

The information about agent owners for definitions and current definition permissions is not recorded in the textual version of a JaM script. This information must be requested using the `inspectDef()` method of the `JaM.Script` class or can be inferred by the exceptions thrown by calls to methods of this class (see Appendix A.2). A database of usernames and passwords that can be used to instantiate new objects of the `JaM.Agent` class can be saved to disk by the `root` agent using the `savePasswords()` method (see Table 6.7) and loaded in for use in new script instances using the `Script` constructor that is passed a filename (see Table 6.6). Agents can be added and deleted from a script instance by the `root` agent only, using the `addUser()` and `delUser()` methods.

Ordinary agents other than the `root` agent have methods that when called with their *cookie* allow the modification of passwords (`password`) and definition permissions (`permissions`), as shown in Table 6.6. An agent can change their own password using the `password()` method of an instance of a `JaM.Script` class by passing their current *cookie* and password followed by a new password repeated twice[8]. An agent can change the definition permissions of a definition that they own using the `permissions()` method that alters the permission fields inside an instance of a `JaM.Definition` class, which are shown in Table 6.1.

6.3.3 Agents in the JaM Machine API and the LSD Notation

The LSD notation [Bey86b] is a specification notation for the description of agency in models of systems that are concurrent and/or reactive. The notation is not directly executable and describes agents in terms of: identifiers in a definitive script that are associated with their current state (*state variables*); state variables for other

[8]The password argument is repeated in this method call to encourage good practice in user-interface design by programmers.

Method	Passed	Returns	Description
`Script`	`String`	`void`	Constructor method for a new instance of a `JaM.Script` class. The `String` argument should represent the pathname to a file containing a user database for the new instance of the script class.
`Script`	—	`Script`	Constructor method for a new instance of a `JaM.Script` class. The new instance will have one super-user called "`root`" with a password "`PHDBOD`".
`login`	`String`, `String`	`int`	This method is passed a username and password and, if the user exists in the password database (`usersList`), returns a *cookie* integer that the user can then use to perform all their interaction with the script. The user is added to the list of logged in users (`currentUsers`).
`logout`	`int`	`void`	Log the user with the passed integer *cookie* value out from being able to interact with the instance of a script. The user is removed from the list of currently logged in users (`currentUsers`).
`addUser`	`int`, `String`, `String`	`void`	Add a user to the list of users who can interact with the instance of a script (`usersList`). Only the `root` user can do this and must provide their current *cookie*, a username for the new user and and initial password for the new user.
`delUser`	`int`, `String`	`void`	Remove a user from the list of users who can interact with this instance of a script (`usersList`). Only the `root` user can do this and must provide their current *cookie*, and the username for the user to delete.
`permissions`	`int`, `String`, `boolean`, `boolean`, `boolean`, `boolean`	`void`	Method by which the owner of a definition can change its current permission values. A user can only change the permission of a definition that they own. The first argument is the user's current *cookie*, followed by the name of the definition, then four boolean values corresponding to `ownerRead`, `ownerWrite`, `allRead` and `allWrite` fields respectively (see Table 6.1).
`password`	`int`, `String`, `String`, `String`	`void`	Method for a user to change their own password. The first argument is the users current *cookie*, the second their current password and the third and fourth should be the new password repeated twice.

Table 6.6: Methods for the `JaM.Script` class for controlling multi-user interaction with an instance of the class.

Method	Passed	Returns	Description
`inspectDef`	`int, String`	`String`	Format internal information about an instance of a `JaM.Definition` class and return it as a string. Returned information is "*identifier owner permissions* `->` *dependents*".
`savePasswords`	`String`	void	Save current username and password database for an instance of a `JaM.Script` to an encoded file. This file can be subsequently reloaded using the `Script` constructor method.

Table 6.7: Utility methods in the `JaM.Script` class.

agents that they can observe (*oracles*); state variables for other agents that they can redefine (*handles*); definitions derived implicitly from their current state and their oracles (*derivates*); guarded redefinition actions that represent the agent's protocols to take action based on their current state and derivates (*protocol*). An LSD specification cannot be interpreted operationally without providing additional runtime information, such as priorities for the order in which the guards for redefinitions are examined and, if true, the order that the a guarded actions (redefinitions) are executed. Ness discusses the significance of LSD analysis in connection with software development in detail in his thesis [Nes97][9]

The agency and definition permissions in the JaM Machine API can be used to support agency in models specified in the LSD notation. In this section, one possible strategy for the interpretation of an LSD specification in a JaM script is presented. The list below takes each component (state, oracle, handle) of an LSD notation for any agent **A** in a specification. Suggestions are given in this list for ways in which a programmer can handle the agent specification when implementing it with the JaM Machine API. The first step is that the programmer should create

[9]Recent and yet-to-be published work on the implementation of an *LSD Engine* was carried out by Alexander Rikhlinsky at the Moscow Engineering Physics Institute as part his MSc. thesis. This work was supervised by Adzhiev.

an instance of a `JaM.Script` with usernames and passwords for agents with the same names as used in the LSD specification.

`state` All state variables for **A** in the LSD specification should be owned by agent `A` in the internal representation of the JaM script. The permissions for these state variables should be initially set to owner reference and modify "`rw--`".

`oracle` All oracles for **A** that are state variables for another agent should have non-owner reference permission initially set to true "`rwr-`".

`handle` All handles for **A** that are state variables of another agent should have non-owner modify permissions initially set to true "`rw-w`".

`derivate` Derivates for agent **A** are implicit or explicit definitions in a script that are either state variables for the agent, handles for state variables of another agent or an internal definition private to the agent. In the case that the definition is internal and private, it should have its non-owner reference/modify permissions set to false "`rw--`". Once derivates are introduced and committed as an accurate description of the observed behaviour of the real agent, their definition permission can be set to disallow any further modification "`r---`".

`protocol` The implementation of the protocol is up to the programmer of the application. One possibility is that each guard is presented to a human user who is representing the interaction of **A** with the model when it evaluates to true and the user takes responsibility for determining the order of execution of the associated redefinition action, or whether the action should be carried out at all! Alternatively, an autonomous agent running as a thread can be introduced to periodically sample the values of guards and make the associated redefinition actions when the guard is true.

In this way, a programmer can use the multi-user support in JaM to assist in some aspects of implementing models based on LSD specification by making sure that the definition permissions are set as strictly as is possible, to meet the conditions of the list above. However, once an identifier in a script is an oracle or a handle for one agent other than the owner, it is open for redefinition by any other non-owner agent and does not provide protection of the definition consistent with the LSD specification.

By way of illustration, consider the excerpt of an LSD specification describing the second-hand and minute-hand agents of a clock given in Figure 6.3a. Other agents exist in the clock specification, such as an oscillator that increments the state variable `seconds` of the second-hand agent. An interpretation of this specification as a JaM script[10] is shown in Figure 6.3b, in which the definitions are split between the agent owners `second_hand` and `minute_hand`. The permissions for these definitions are shown in the right-hand column of this table. Notice how the derivates are kept private to the owning agent. Figure 6.3c shows some Java code for the `run()` method of a Java thread that could be used to implement the `protocol` section of the second-hand agent, with a script instance called `s1` and an integer *cookie* variable for the currently active second-hand agent identified as `second_hand`[11].

The `minutes` identifier in Figure 6.3b has non-owner modify permission so that the second-hand agent can update the value through its protocol. Nothing now prevents another agent other than the `second_hand` agent from modifying the values of minutes which may or may not be part of the whole LSD specification. The programmer should provide a level of protection for such a definition in their own code if they wish to enforce strict adherence by uses to the specification.

[10] The implicit definitions using the `eval` operator take the first argument as a description of a template expression tree and substitutes the second argument as `$1`, the third argument as `$2` etc., prior to evaluation.

[11] The thread sleeps for 0.1 seconds before performing its next check of the guards.

```
agent second_hand() {
state
  seconds sec_length
  sec_angle xsec ysec
handle
  minutes
derivate
  sec_angle = (pi / 2)
    - (pi * seconds) / 30,
  xsec = sec_length * sin(sec_angle),
  ysec = sec_length * cos(sec_angle)
protocol
  seconds == 60 ->
    minutes = minutes + 1,
  seconds > 60 ->
    seconds = 1
}
```

```
agent minute_hand() {
state
  minutes min_length
  min_angle xmin ymin
oracle
  sec_length
handle
  hours
derivate
  min_length = 0.75 * sec_length,
  min_angle = (pi / 2)
    - (pi * seconds) / 30,
  xmin = min_length * sin(min_angle),
  ymin = min_length * cos(min_angle)
protocol
  (minutes == 60) ->
    hours = hours + 1,
  (minutes > 60) ->
    minutes = 1
}
```

a

Owner	Script	Permissions
second_hand	`seconds = ... // Handle for oscillator agent`	r-rw
	`sec_length = ... // Handle for designer agent`	r-rw
	`sec_angle = eval("(pi / 2) - (pi * $1)/30", seconds)`	rw--
	`xsec = eval("$1 * sin($2)", sec_length, sec_angle)`	rw--
	`ysec = eval("$1 * cos($2)", sec_length, sec_angle)`	rw--
minute_hand	`minutes = ... // Handle for second_hand agent`	r-rw
	`min_length = eval(0.75 * $1, sec_length)`	rw--
	`min_angle = eval("(pi / 2) - (pi * $1)/30", minutes)`	rw--
	`xmin = eval("$1 * sin($2)", min_length, min_angle)`	rw--
	`ymin = eval("$1 * cos($2)", min_length, min_angle)`	rw--

b

```
void run {
  String smin;
  while (true) {
    synchronized(s1) { // get a lock on object s1
    if (s1.printVal(second_hand, seconds).endsWith("60"))
      {                                   // check first guard
        smin = s1.printVal(second_hand, minutes).lastToken();
        s1.addToQ("minutes = " + (Integer.valueOf(smin) + 1));
      }                   // add redefinition to queue if true
  smin = s1.printVal(second_hand, seconds).lastToken();
  if (Integer.valueOf(smin) > 60)        // check second guard
    s1.addToQ("seconds = 1");            // add redef. if true
  s1.update(second_hand);  } // update s1 and release lock
  Thread.sleep(100);
  }
}
```

c

Figure 6.3: LSD specification with a JaM script representation and thread implementation of a `protocol` section.

6.3.4 Extension of Agency in JaM Scripts

In this section of the chapter, a mechanism for associating ownership and permissions to reference/modify definitions has been discussed. This mechanism is limited to associating one owning agent to each definition and four permission bits, but is sufficient for proof of concept. The data fields of the `JaM.Definition` (Table 6.1) and `JaM.Agent` classes can easily be extended to include additional information such as group ownership and permissions for definitions, date and time of last modification, size of the definition in memory, and any other useful definition management information similar to that included in a files *inode* on a UNIX system. The methods of the `JaM.Script` class can also be extended to increase the flexibility of the management of interacting users in a multi-user script environment, via methods to control group access and the transfer of ownership of definitions. In the future, support should be provided for representing hierarchies of definitions in directories of a filing system or tables of a database [Bow93].

6.4 Implementation Mechanisms for JaM Scripts

Many implementation mechanisms for applications that support or integrate JaM scripts are available. These include:

- the standard single-user mechanism (as illustrated on page 202);
- the dynamic extension of the data types and operators of a JaM script on-the-fly during execution of the script interpreter, where the data type and operator structure is not fixed at any point. This model is described in Section 6.4.1.
- distributing interfaces to scripts across several workstations with client/server systems. This is described in Section 6.4.2.

- using scripts that include data types for other scripts. This mechanism is known as the *scripts within scripts* mechanism and is presented in Section 6.4.3.

6.4.1 Dynamic Extension of JaM Notations On-the-fly

A JaM script offers the same level of open-ended functionality to a user as the EDEN interpreter in the manipulation of definitions in scripts but does not allow for the redefinition of operators on-the-fly as is possible in EDEN. The data types and operators in JaM scripts are implementing by a programmer by extending some classes of the JaM Machine API. These are added to the notation during execution of one of the implementation mechanisms, which allows the notation to be different for specific applications. Hence the types and operators of a JaM script recognised by the API's parser are not rigidly defined prior to the initialization of a JaM script. This allows a user who is also a competent Java programmer to introduce new data types and operators over those data types during interaction with a script. This provides the open-ended functionality of the EDEN interpreter to extend and redefine operators, albeit in a comparatively clumsy manner[12]. The mechanism also extends EDEN's functionality as new explicit data types can be introduced on-the-fly.

The *dynamic data type and operator extension* model for implementing applications based on JaM scripts involves a modification to step 8 of the standard model. The implementation can continue to call `addToQ()` and `update()` repeatedly to create new definitions and redefine existing definitions. At any point in this process new types and operators can be added by following the step-by-step process.

1. Create a new instance of the `JaM.LoadJaMClasses` class. Within this list, it assumed that this instance is identified as "`jcl`".

[12]The classes must be compiled into Java bytecode before being loaded into the running Java Virtual Machine and then added to the type of operator list of a JaM script instance.

2. To load a data type class that extends `DefnType` or an operator type that extends `DefnFunc`, call the method "`jcl.loadClass("`*full_classname*`")`", which returns an object `c` of type `java.lang.Class`.

3. Cast a new instance of this class to `DefnType` if it represents a data type and to a `DefnFunc` if it represents an operator to be added to the current script instance, eg. "`DefnType dt = (DefnType) c.newInstance();`".

4. If the loaded class `c` represents a new data type, call the `addType()` method for the script instance. If the loaded class `c` represents a new operator for the script, call the `addFunc()` method for the script instance.

5. The newly loaded data type or operator is now available for use in calls to the `addToQ()` method in the dynamically extended JaM notation. Newly compiled classes may require significant testing and may generate runtime errors.

The JaM programmer can build up libraries of classes that they wish to use in several different applications. By using the dynamic extension of scripts, a programmer can create an environment where a user can experiment with different data types and operators. In a geometric design application such as the *empirical world builder* presented in Chapter 8, the dynamic extension model can be used to introduce new classes of shape types during a modelling session without the need to stop, link the object code for the application and restart.

6.4.2 Multi-user Client/Server Implementation Mechanisms

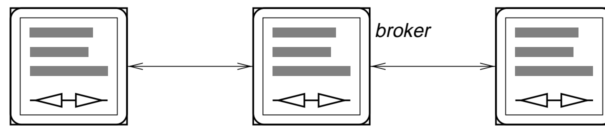
For agents to collaborate simultaneously in the interactive construction and experimentation with a JaM script as described in Section 6.3, it is necessary to provide separate interfaces for each agent. The core Java programming API provides several mechanisms to enable a programmer to construct multi-user applications. This in-

cludes the creation of threads to handle concurrent components of processes and to take advantage of multi-processor parallel computer systems. Support for TCP/IP networking is available within classes that implement servers to listen and accept network connections, and clients that open communication sockets to these servers. Network support for modelling with JaM scripts is the topic in this section.

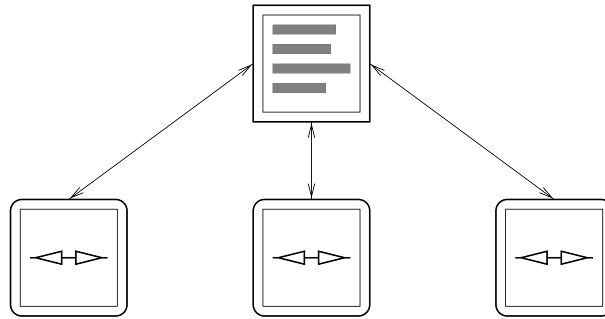
The distribution of interfaces to JaM scripts over networks with server/client implementation mechanisms provides the basis for a programmer to construct applications such as multi-user spreadsheets (cf [Nar93]) and tools to support concurrent engineering. The users of these applications can share a workstation at different times of the day, cooperate in an interactive session in the same office or potentially anywhere on an interconnected TCP/IP network, including over the Internet [Kro94].

A *JaM client* is defined as an interface through which an agent can perform redefinitions for a JaM script. This interface is not necessarily textual, it could be a graphical user interface or a connection to a real-world sensor such as a digital thermometer. A *JaM server* is a Java application containing at least one instance of a JaM script class. The standard implementation mechanism, where only the `root` agent interacts through a single JaM client with a script is the simplest of all client/server models. Methods of classes in an application based on the standard model can call the methods of the script instance and pass the root agent *cookie* (see page 202) to every method call. The JaM client in this case can either be an object in the current implementation (such as a text area) or in a separate general application such as a `telnet` client connected to a JaM server socket network port, through which the server provides interaction with its script.

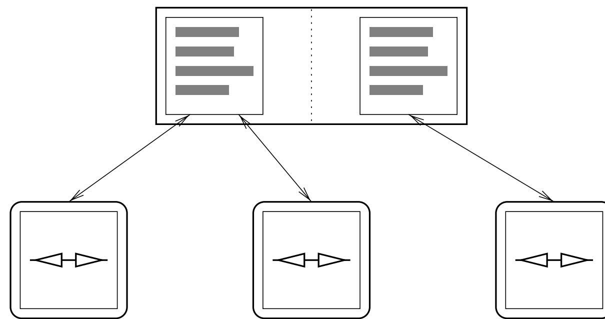
When there are several concurrent agents, many configurations that can be considered for the connection of JaM clients and JaM servers. Figure 6.4 shows three possible configurations. In this figure, boxes with a thick border represent



a) Peer-to-peer, self synchronising combined server / clients



b) Dumb clients talk to a central, intelligent, single-script server



c) Dumb clients connected to an intelligent multi-threaded, multi-script server

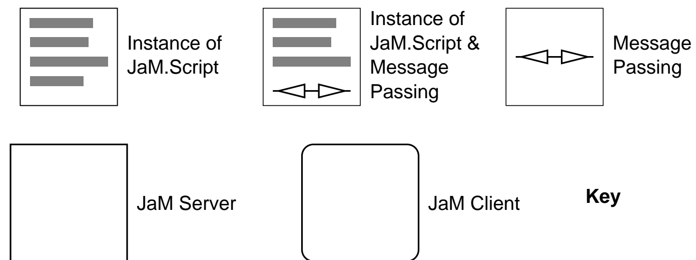


Figure 6.4: Client/server implementation models for multi-user JaM scripts.

a process running on an independent Java virtual machine, either on the same or different computer systems. The thinly bordered boxes represent either JaM script instances or implementation of message passing mechanisms over network sockets for definitions and interactive communication between processes. The figure has three sections 6.4a, 6.4b and 6.4c for three models, with a key shown beneath. Each model is described in the list below.

6.4a - Each JaM client has its own combined JaM server and therefore its own script instance in this *peer-to-peer* model. The programmer must implement a synchronisation mechanism to ensure that any definition that exists in the script of the JaM client of one agent is also propagated to the other agents. Similarly, the current *logged-in* status of agents must be distributed across all scripts and that each script is for the same JaM notation. One client can exists with a greater degree of *intelligence* than the others. This can act as a broker for the synchronisation information and as a super-user agent with higher-level privileges to modify scripts on the other clients than available at the clients themselves.

6.4b - A dumb client is a JaM client in an application that has no associated script instance as part of its code. In this example, three interface clients are connected to a JaM server process that can handle the interaction of several agents simultaneously interacting with the same JaM script instance.

6.4c - Multi-threaded JaM servers can contain more than one JaM script instance and provide a way for agents using dumb JaM clients to choose whether to start by instantiating a new script or to join in interaction with an existing script. The JaM server may also offer a user agent more than one JaM notation that they can instantiate a JaM script object for and then use interactively. The figure shows two agents at dumb JaM clients interacting with the same

script and another agent interacting with a separate script independently.

In each of the example models, the programmer needs to make sure they handle the ordering of calls to the `addToQ()` methods and the subsequent `update()` method to ensure that the behaviour of the script is synchronised and predictable[13]. It is also important that a programmer ensures that any exceptions thrown by new definitions on the queue of definitions, such as reporting typographic errors in a definition, or exceptions thrown by an update such as the detection of a cyclic dependency, are sent to the appropriate agent who caused the exception.

6.4.3 Scripts Within Scripts

It is possible to have an instance of a `JaM.Script` class as a data field in a class that extends `JaM.DefnType`. In other words, it is possible to create a data type in a JaM notation for the representation of JaM scripts. Every script has a textual description and so is an appropriate data type to use in a JaM notation. It is also possible to define operators that maintain implicit dependencies between scripts. For example, consider a notation where a script with identifier c is defined persistently and indivisibly as the concatenation of script a and script b. In this *scripts within scripts* model, it is possible for a programmer to consider ways in which to implement definitive notations that contain the higher-order dependencies introduced in Section 3.2.2. An implicitly defined script need not be in the JaM notation as the script that contains its definition.

In this way it becomes possible to define generic templates for scripts that, with some variable parameters, can be instantiated to create new scripts whose structure depends implicitly on the value of these parameters. This is similar to

[13] An example of how to do this in the thread code shown in Figure 6.3c using the Java `synchronized` statement [Fla96]. The method grabs a lock on the script object `s1` until the update is completed successfully and the lock is released at the end of the block.

the graph construct available in the DoNaLD notation, of which the speedometers shown in Figure 3.5 are an example.

6.5 A Simple Illustrative Example - *Arithmetic Chat*

The case-study presented in this section is for a simple JaM notation with two data types and one operator. The code for the classes that extend `JaM.DefnType` and `JaM.DefnFunc` is presented and explained. These data types are then integrated into an instance of a `JaM.Script` class, the script for which can be shared between several users through a mechanism similar to an interactive Internet chat program, such as *Internet Relay Chat* (IRC) [Kro94]. The two data types are integers and floating-point numbers and the operator is addition. These are to be used in a proof-of-concept application in which agents collaborate and discuss scripts of definitions over these two data types with the one operator.

Data Types

The arithmetic chat application has two data types, one for representing integer values and the other for representing floating point values. To represent these as a JaM script requires two classes that each extend `JaM.DefnType`, one to represent integers called `JaMInteger` and one to represent floats called `JaMFloat`. The first step in the process of creating these data type classes is to decide on an appropriate internal data representation in Java for these data types. For these particular data types this is a trivial process as both have direct representation in Java, the `int` type for integers and the `float` type for floating point numbers.

The next stage is to embed this internal data representation in as the data field(s) in the class that represents that data type. Figure 6.5 shows annotated source code for the `JaMInteger` class with the one data field "`int d`" that will hold

the internal data representation for an instance of the class. (The `JaMFloat` source code is very similar to the `JaMInteger` code and so is omitted). The internal data representation in this class is "`float d`".

The ordering of methods in a class is not significant but the class must contain implementations for all abstract methods of the superclass together with a constructor method. One possible way to proceed with implementing a data type class is described here. Having chosen the internal data representation, a unique integer reference number for the data type should be chosen and the constructor method for the class added as its first method. This should be followed by the `makeNew()` method that follows the standard template described in Table 6.2.

Two methods are required for conversion to and from internal data representations and strings, `printIt()` and `parseIt()` respectively. The `printIt()` method is passed information about the current formatting of the output line on which the string will be printed. These formatting parameters are the current number of spaces inserted as an indent after a new line character, the current position along the line that the string returned will be appended and the length of the current line. In this method, a programmer should ensure that the length of the string describing the internal data representation does not exceed the length of the line, taking account of the current position. If it does then a new line and indent should be inserted. The `parseIt()` method is passed a string that has already been recognised as of the correct type and converts it into the internal data representation. In the example this is done using the Java core API static method `Integer.parseInt()`.

The programmer also needs to implement a method to decide whether a passed string can be converted into the internal data representation (`recognise()`) and a piece of code describing an action (`action`) to be taken every time a value of this type is updated. In the `recognise()` method shown in Figure 6.5, the string passed is checked to see that it is a sequence of digits, preceded by an optional minus

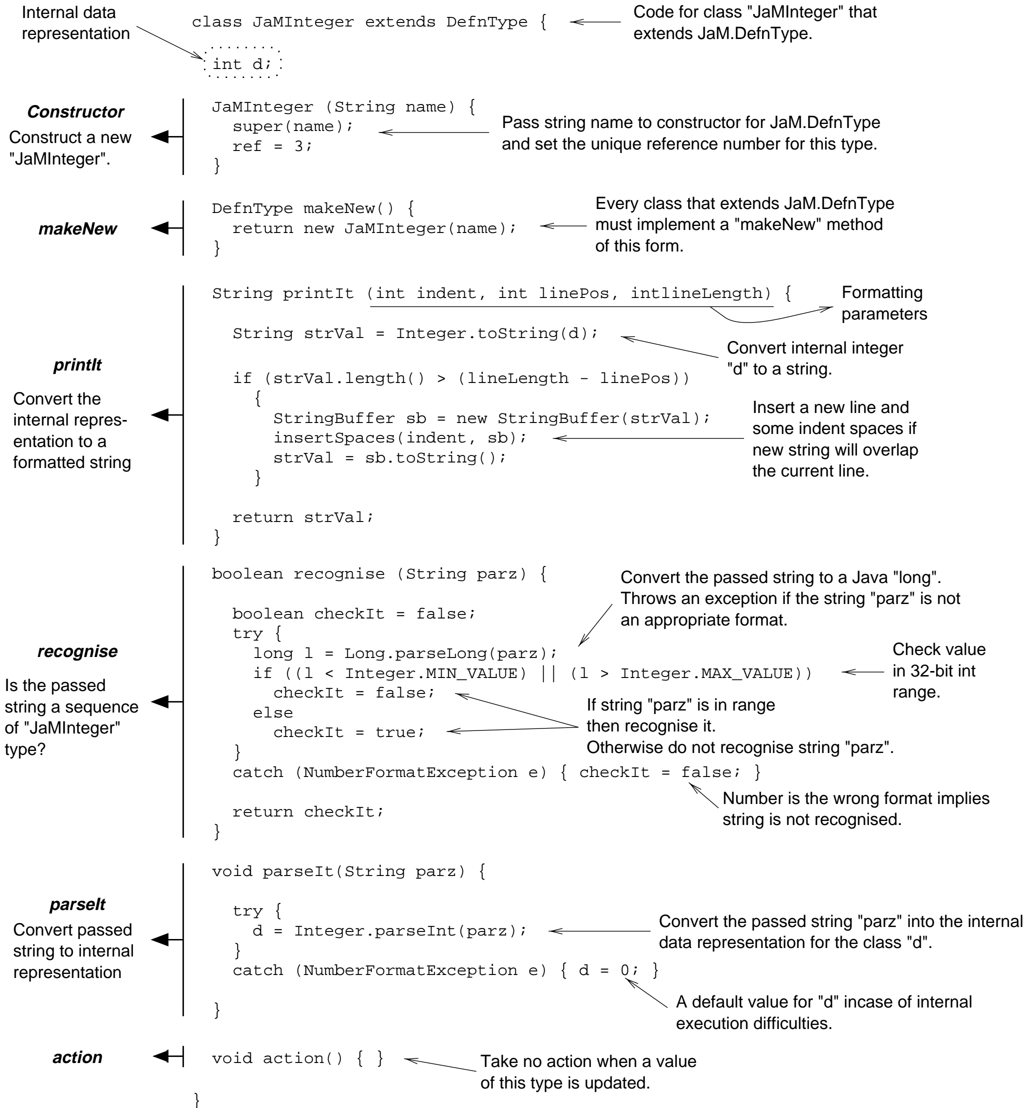


Figure 6.5: Annotated code for the `JaMInteger` class for the representation of integer values in JaM script instances.

sign, using the `Long.parseLong()` method which throws an exception if its string argument contains any other alphanumeric characters than those that match with the regular expression for long integers. If the exception is thrown then the method catches it and returns false, in other words the number string was not a sequence of digits and could not be converted into the internal data representation. A check is carried out to see that the number is within the range of an integer before returning true to say that the passed string is recognised.

In this example class, the `action()` has no code body and therefore no effect. The action is intended to be used in other applications by data types such as graphical lines that need to be redrawn every time their value is updated. Once the class is written, it can be compiled and then used in conjunction with other classes of the same package.

Operators

The example application presented has only one operator in its script for defining implicit number values by summing a list of other value. The source code for the methods of class `JaMAdd` for this operator can be easily modified to provide other standard arithmetic operators such as subtraction, multiplication and division. To implement this JaM script operator, a programmer must extend the `DefnFunc` class and implement the code for the body of its abstract methods.

The source code for one possible version of class `JaMAdd` over the two data types represented by `JaMFloat` and `JaMInteger` is shown annotated in Figure 6.6. The class contains a static lookup table for the reference numbers of the types that may be acceptably used as arguments to the operator in a JaM script. This is followed by a constructor method that calls the generic constructor method in the `JaM.DefnFunc` class. No data fields are required in a class that extends `DefnFunc`.

The `typeCheck()` method checks an argument sequence represented as a

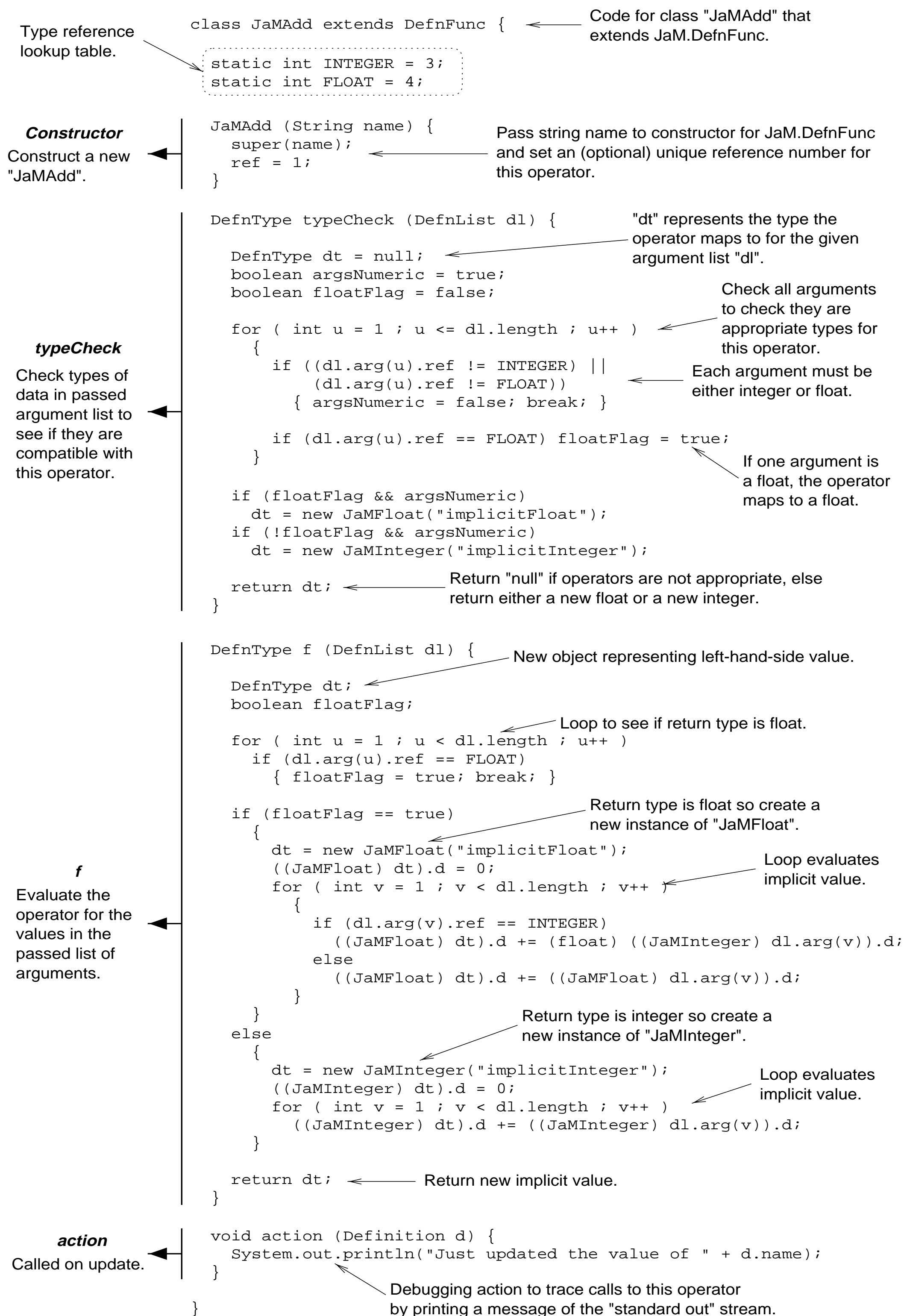


Figure 6.6: Annotated code for the JaMAdd class for summing together a sequence of numerical arguments.

`JaM.DefnList` to see if it contains arguments appropriate to a particular operator. In the example implementation there are only two data types and these are both acceptable arguments to the operator. Future reuse of the operator may require its use in JaM notations with data types such as strings that are not appropriate data types. To allow for reuse, the class has a check built in to check that its arguments are all numerical. No limit is imposed on the length of the sequence of arguments. If one or more of the arguments to the operator is a floating point number then the operator returns a floating point number (a `JaMFloat`), otherwise if all the arguments are integers then the operator returns an integer (a `JaMInteger`).

The method `f()` contains the code that will evaluate the value mapped to by an implicit definition based on this operator. This method requires access to the fields in the objects representing the arguments to the operator. It is necessary to cast these objects to their original type, as specified by their type reference number before the internal values can be retrieved. In the example code in Figure 6.6, if the return type of the operator is a `JaMFloat`, as will have already been established by the `typeCheck()` method, it can be inferred that at least one of the arguments to the operator is a float. To access the internal data values of the arguments to an operator, it is necessary to check whether they each represent integer or floating point values and cast the object appropriately, prior to summing their values. If the type mapped to by the operator is integer then it can be inferred that all the arguments must be integers and so only one cast to the `JaMInteger` is required in the summing procedure.

The `JaMAdd` code in Figure 6.6 includes an `action()` method that is called every time this operator is evaluated. One use for this method is as a debugging tool for a programmer to trace the execution of their implementation from within method calls to the JaM Machine API. This is illustrated in Figure 6.6, where every time the value of an identifier defined by the `JaMAdd` operator is updated, the identifier

is printed on the standard output stream. If a programmer requires access to the internal data fields of an instance of a `JaM.Definition` class in their application, a similar mechanism can be used to pass a reference of a definition object processed by JaM into other non JaM Machine API areas of the implementation.

Integration of Scripts into an Application

A proof of concept tool to allow several agent users to interact simultaneously with scripts of definitions for a JaM notation containing floating point and integer data types with one operator add has been constructed. This uses the source code shown in this section in Figures 6.5 and 6.6 to describe the data types and operator for the notation, and is based on the client/server model shown in Figure 6.4b. The source code for the server is shown in Appendix A.3 and the client is a standard `telnet` client program, generally available on operating systems that support the TCP/IP networking protocol.

When the server is run, a new instance of the `JaM.Script` class is created along with a server that listens for connections on a specified port. The implementation adheres to the same procedural order as the standard model described in Section 6.4 until step 6. When the server receives a connection request from a client, it creates a new thread to handle that connection. The user connecting through a client is requested to type their name. This is used as a new username and password for the agent database of the script and the user is automatically logged in. From then on, every line that the user types is echoed to all other concurrent users with their name preceding what they typed. Certain sequences of characters also perform actions.

// *definition* Add the *definition* to the queue of definitions for the central script and call the update method. If there is an exception, this is reported to all

users.

`value` ***identifier*** Request the current value associated with the *identifier* as a string. The value appears in all clients, except when the *identifier* does not exist. In this case, an exception is reported to all users.

`defn` ***identifier*** Request a textual version of the current implicit definition associated with the *identifier*. The value appears in all clients, except when the *identifier* does not exist. If this is the case, an exception is reported to all users.

`?` ***identifier*** Request some additional internal information about the definition associated with the *identifier*, including its current owner, definition permissions and dependencies. This information appears in all clients, except when the *identifier* does not exist. In this case, an exception is reported to all users.

`exit` Close the connection to the client. The server application continues to run.

Once the server is started, user agents can connect. All they need to make a connection is a `telnet` client, specifying the port number to which the server is attached. Figures 6.7 and 6.8 shows an interactive chat session between two users Jane and Mark. Together they construct and inspect a four definition script while holding a textual discussion. Any line of text typed by Jane or Mark is indicated in their respective consoles by a "`=>`" arrow symbol. At the end of the session, Jane tries to redefine the value for `c`, a definition that is owned by Mark, and is refused permission to do so as the default permission set for a new definition is "`rwr-`".

The behaviour of this server is trivial but does demonstrate the potential for sharing JaM scripts amongst clients. Any other JaM data types and operator classes that exist can be instantiated and used in a similar chat application. The `telnet` client can be replaced by a more complex and specialised client that includes

```
=> jane@britten > telnet britten 8861
   Trying 137.205.225.29...
   Connected to britten.
   Escape character is '^]'.
=> Welcome to Arithmetic chat.  Please enter your name > Jane
   Hello Jane, you may now chat!
   Welcome to new user Jane.
   Welcome to new user Mark.
   Mark: Hi Jane! Here is a line of text that will be echoed to you and me.
=> Hello Mark. How are you?
   Jane: Hello Mark.  How are you?
   Mark: Well.  Shall we try some sums ... how about adding 23E3 and 2431.
=> // a = 23E3
   Jane: // a = 23E3
   Mark: // b = 2431
   Mark: // c = add(a, b)
=> value c
   Jane: value c
   script > c = 25431
=> defn c
   Jane: defn c
   script > c = add(a, b)
=> ?b
   Jane: ?b
   script > b Mark rwr- ~> c.
=> How about summing all three values together.
   Jane: How about summing all three values together?
   Mark: // d = add(a, b, c)
   Mark: value d
   script > d = 50862
   Mark: Why don't you redefine the value of c ...
=> // c = -12e-2
   Jane: // c = -12e-2
   Jane: JaMException: addToQ: It is not possible to modify this definition: c
   The modifying agent is not the owner and write protection is set.
=> Oh .. I do not have permission to do this!
   Mark: exit
=> exit
   Jane: exit
   Connection closed by foreign host.
   jane@britten > _
```

Figure 6.7: Jane's console view of an *Arithmetic Chat* with Mark.

```
=> mark@gem >  telnet britten 8861
   Trying 137.205.225.29...
   Connected to britten.
   Escape character is '^]'.
=> Welcome to Arithmetic chat.  Please enter your name > Mark
   Hello Mark, you may now chat!
   Welcome to new user Mark.
=> Hi Jane! Here is a line of text that will be echoed to you and me.
   Mark: Hi Jane! Here is a line of text that will be echoed to you and me.
   Jane: Hello Mark.  How are you?
=> Well.  Shall we try some sums ... how about adding 23E3 and 2431.
   Mark: Well.  Shall we try some sums ... how about adding 23E3 and 2431.
   Jane: // a = 23E3
=> // b = 2431
   Mark: // b = 2431
=> // c = add(a, b)
   Mark: // c = add(a, b)
   Jane: value c
   script > c = 25431
   Jane: defn c
   script > c = add(a, b)
   Jane: ?b
   script > b Mark rwr- ~> c.
   Jane: How about summing all three values together?
=> // d = add(a, b, c)
   Mark: // d = add(a, b, c)
=> value d
   Mark: value d
   script > d = 50862
=> Why don't you redefine the value of c ...
   Mark: Why don't you redefine the value of c ...
   Jane: // c = -12e-2
   Jane: JaMException: addToQ: It is not possible to modify this definition: c
   The modifying agent is not the owner and write protection is set.
=> exit
   Mark: exit
   Connection closed by foreign host.
   mark@gem > _
```

Figure 6.8: Mark's console view of an *Arithmetic Chat* with Jane.

interfaces for graphics and sound which better support interaction with JaM scripts. Because the dependency maintenance is integrated by the JaM concept into a programming API, a programmer can use it in anyway they wish, in combination with any other Java classes and programming APIs.