

JavaGram – a language for Agile Programming

SHARAM HEKMAT
PRAGSOFT CORPORATION

This whitepaper provides an overview of JavaGram – a new programming language specifically designed for agile application development. As a Java derivative, JavaGram offers a number of novel features that simplify the task of developing user-interface-rich client-server applications, such as declarative programming, automatic remoting, asynchronous method invocation, and dynamic loading. Language constructs that support these features are described and their practical application for agile development explored.

1 Introduction

There has been growing interest in recent years in agile software development methods. The shortcomings of traditional, waterfall methods have been known for at least three decades, namely:

- The requirements for a complex system can rarely be specified fully and accurately in advance.
- Despite tight quality control, the output of each phase will contain gaps or flaws that won't be discovered until a later phase.
- User requirements will evolve during the course of a long project, thus making the end-result inconsistent with the latest requirements.
- The cost of fixing a requirement or design defect discovered later in the project is substantial.
- Given that a working system is not available until late in the project, there is little opportunity for user participation and feedback; this increases the risk of the system not being accepted by the users.
- These challenges often cause schedule delays and budget blowouts.

The agile approach attempts to address these difficulties by promoting a more iterative lifecycle, where emphasis is on prototyping, user participation, having a working system all along, and less documentation. The approach is best summarized by the agile manifesto (www.agilemanifesto.org), which places more value on:

- **individuals and interactions** over processes and tools,
- **working software** over comprehensive documentation,
- **customer collaboration** over contract negotiation, and
- **responding to change** over following a plan.

Successful adoption of the agile approach requires the overcoming of cultural, process, and practice barriers. Although these challenges are primarily non-technological, the use of suitable technology can be of substantial benefit. The reality is that most technologies available today are

not particularly well-suited to the agile approach, typically because they either predate the ‘agile age’ or ignore its dynamics.

JavaGram has been specifically designed to support the dynamics of agile development. Its conception is the result of seven years of experience gained from successfully practicing RAD and agile in commercial software projects. As such, it’s a language designed by a practitioner for practitioners.

1.1 Agility Criteria

So what makes a technology more suited to agile development than others? The answer to this question lies in the dynamics of practicing agile in real-life projects.

Agile relies heavily on iterative development. The first priority in a project is to produce a working prototype of a proposed system and then to use this as a vehicle for eliciting detailed user requirements. As requirements emerge, these are used to further refine and enhance the prototype. Enhancements are done as a series of mini development cycles where during each cycle we design, code, and test the next increment, and invite users to evaluate the outcome.

The cyclic and iterative nature of agile places great emphasis on going from requirements to working software rapidly. Speed is of the essence; otherwise cycles become slow and ineffective. Rapid delivery of the next cycle ensures that users will remain engaged and the project will not lose momentum. At the same time, it’s important to keep the project team small to avoid communication overheads and to minimize the need for detailed documentation, both of which will slow things down.

In order to speed up each cycle, we must not only have the means to design and code business functionality quickly, but also to rapidly turnaround defects raised during the testing phase. When this is not the case, testing can become hopelessly inefficient, as testers will spend most of their time waiting for critical fixes without which further testing can’t take place.

Equally important is the cross cycle speed. If each subsequent cycle takes longer than the previous one due to developers finding it difficult to add new functionality then momentum will be lost and eventually grind to a halt. It is therefore vital that the application under development lends itself to alteration and evolution. While this is primarily influenced by the quality and foresight of the original design, the underlying technology can go a long way in promoting good practices that avoid design inflexibility.

Finally, agile is all about simplicity. Simplicity is achieved by untangling complexity so that the essential is separated from the accidental. Once we’ve identified what’s essential, the least complicated way of getting there has the potential to deliver the best outcome. Without constant awareness of this, IT professionals have a tendency to be dazzled by technological complexity and run the risk of over-engineering their solutions.

1.2 Barriers to Rapid Development

In order to design a language that facilitates speedy development, we must consider the things that slow down developers; namely:

- **Language complexity.** The more complex a language is, the steeper its learning curve will be. Contrary to the popular opinion that a language's complexity is a function of its number of features and constructs, it's primarily a product of giving the programmer too many different ways of doing the same thing. Too much choice leaves the programmer in a situation where s/he has to constantly think about choosing the best approach, and this will slow down development. JavaGram avoids this pitfall by not attempting to be a totally versatile and general purpose programming language. For example, JavaGram offers only three container types – lists, vectors, and maps – each of which has only one implementation. So when the programmer sees the need for a container, no time is lost on deciding which one to use.
- **Plumbing overhead.** More than 50% of code in a typical application tends to be of 'plumbing' nature. This is code that doesn't implement business functionality but performs essential housekeeping. For example, data entered by the user into a screen typically needs to be extracted, validated, stored in a suitable data structure defined by the programmer, transmitted from the client to the server using a message defined by the programmer, unwrapped on the server-side, processed by interacting with a database, and so on. Typically, such data goes through a number of transformations where it's changed from one format to another, and yet another, until it finally can be acted upon. JavaGram greatly reduces the need for plumbing code by performing such tasks behind the scene without the programmer having to worry about them. This saves the programmer much valuable time, enabling him to focus on actual business functionality. Interestingly, because a lot of unnecessary transformation is avoided, the end-to-end process executes faster, thus also saving computational resources.
- **Procedural clutter.** Most languages (including Java) require the programmer to think procedurally. While this works well in some situations (e.g., implementing transactions), it hinders tasks that are better suited to a declarative style. GUI programming is one such task. Because GUIs are visual and typically hierarchical, a declarative notation can be far more expressive and convenient for implementing them. JavaGram adopts this style by allowing the programmer to define GUIs using a markup notation. This not only results in a lot less code, it also greatly enhances the readability of the code, to the extent that the GUI can be easily visualized by simply observing the code.
- **Static composition.** Most languages take a static view of the components that comprise a program and require all the referred components to be in place before the program can be executed. Because of the incremental nature of agile, this is a major inconvenience which forces the programmer to define placeholders and stubs to work around the issue. JavaGram overcomes this problem by allowing the programmer to dynamically load scripts. For example, if the action of a push button is defined by a dynamically-loaded script, the programmer will be able to run the program even if this script is not defined or is incomplete and has compilation errors. These errors will not surface unless the user actually pushes the button.
- **Complex object model.** In object-oriented programming, business objects can be a source of substantial complexity. Most of this complexity is in the underlying implementation of the business objects (e.g., persistence). However, from an agile development viewpoint, it's not the implementation that's important but the business functionality offered by the object. JavaGram reduces this complexity by offering a straightforward implementation model based

on a generic object which hides much of the underlying complexity. By sub-classing the generic business object class and using declarative schemas, the programmer can do away with time-consuming and error-prone activities such as implementing object persistence.

1.3 Barriers to Rapid Testing

The key to an effective and fast testing cycle is the ability to turnaround defects quickly. The usual pattern in system testing is that a tester runs a number of test cases and records observed defects. Some of these defects become showstoppers, preventing the tester from progressing any further. At this point the tester will need to wait until enough defects have been fixed and a new release produced so that testing can continue.

The main barrier to quickly delivering a test release is a slow build process. JavaGram addresses this by eliminating the need to produce an actual build. A release can simply consist of the correct versions of the scripts that comprise the application, extracted from a version control system and placed in a test application server. Compilation is not required, as the application server will incrementally compile the scripts on demand. This means that even a large application can be released in minutes.

Even better, for the majority of fixes, a complete release is not required. Developers can choose to release only the few affected scripts that fix the outstanding defects. JavaGram even allows an application to be hot fixed without restarting the server. Experience has shown that critical defects can be turned around shortly after they're raised by testers, thus enabling testers to continue their work with minimal disruption. Similar benefits are gained in production support.

1.4 Barriers to Rapid Evolution

A live application is best regarded as an evolving entity. The more the application is used, the more users will demand from it – active use leads to rapid evolution. When a new application is designed, it's almost impossible to foresee all the future demands that will be placed on it. At any point in time, it's only practical to consider features that are likely to be demanded in the near to medium term. Future (and especially unforeseen) demand is likely to stretch the application design to an extent that couldn't have been predicted.

Long term therefore, the malleability of the design becomes a critical issue. A design that is not accommodative of change will eventually break and disrupt the evolution process. So how can a design be made malleable? There are two competing views on this subject.

The first view argues that considerable flexibility should be built into the design from the beginning to make it future proof. This is intellectually appealing but in practice rarely successful. The problem with this approach is that devising extensive design flexibility can be very costly and invariably leads to design complexity, both of which go against the agile principles. Experience indicates that, in the long run, very few of such expensive flexibilities actually turn out to be used and the rest become a liability.

The second view argues for simplicity. By keeping a design as simple as possible, we not only shorten the construction speed, we also make it easier for future developers to understand its make-up. The latter is far more valuable than some appreciate. Most practitioners would testify

that the biggest hurdle in tweaking a design is to first understand it. Once this is achieved, only a bit of developer creativity is required to work out a way of accommodating something new.

JavaGram adheres to the simplicity view. The language helps the developer to express things succinctly and with minimum clutter. Reduced plumbing code means that the vast majority of code actually represents business functionality. This facilitates understanding and makes it easier to work out how to best apply a change.

2 Salient Features

JavaGram builds on the strengths of Java and closely follows the Java syntax and semantics, including strong type checking. This should make it rather easy for a Java programmer to become proficient in JavaGram.

This section describes those features of JavaGram that characterize it as an agile programming language.

2.1 Server Centricity

Historically, programming languages have been designed with the assumption that application code will need to be installed in its target environment before it can be executed. The World Wide Web has been the most significant departure from this paradigm. Under this model, a web application is not pre-installed on the client side, but is incrementally downloaded in response to actions taken by the user. However, this dynamically-downloaded code (HTML, which may also include JavaScript and the like) is primarily concerned with presentation, and is generated by the actual application code residing on the web server (see Figure 1), hence the term *thin client*.

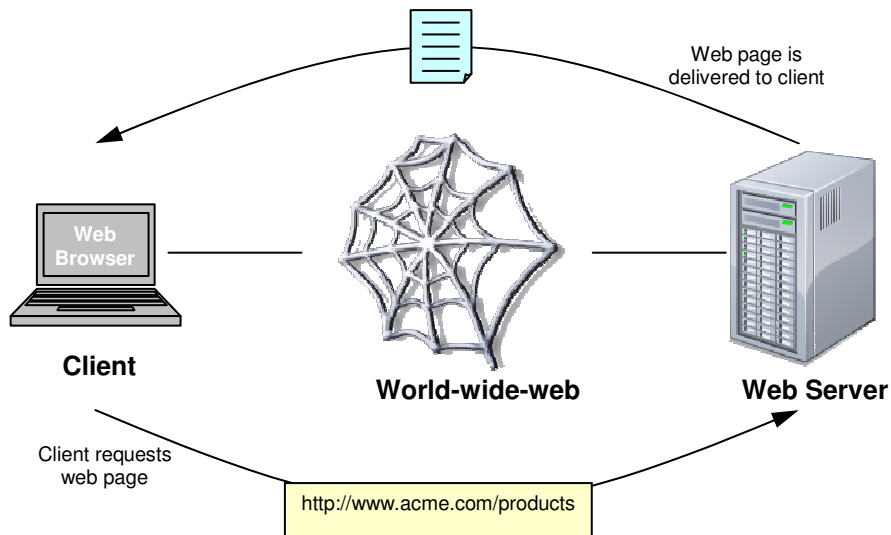


Figure 1. The web model.

The design of JavaGram was inspired by this model, with a key difference: rather than the server generating presentation code for the client, the server delivers actual application code to the client. Like the web model, the client initially contains no code at all, but rather is a shell capable

of receiving code and ‘interpreting’ it. In the web model, the client is a web browser; in the JavaGram model, the client is a compact runtime environment called JAG (see Figure 2).

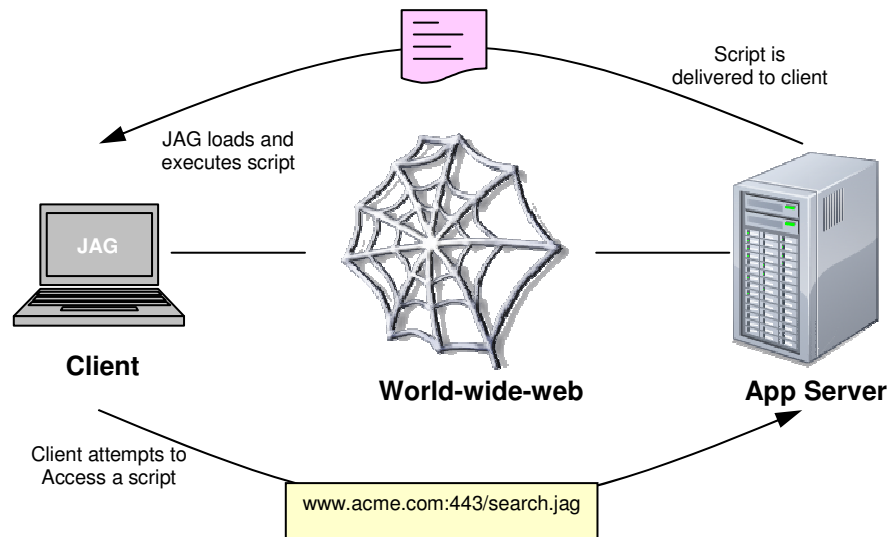


Figure 2. The JavaGram model.

A JavaGram client boots itself against a JavaGram server using an initial address (similar to a URL in a web browser), which identifies the server and the initial script. In response to this, the server creates a session thread to handle all subsequent communication with the client. The requested script is then sent to the client, which the latter loads and executes. During the course of execution, the script may refer to other scripts, which are sourced from the server in a similar manner. Therefore, the client code base is built incrementally according to user actions.

Like the thin client model, the server-centric model of JavaGram has a number of advantages over the traditional fat client model; namely:

- There is no need to install an application at the client end.
- An application can start quickly because the loading process is incremental.
- Release management is much easier, as a new version of the application doesn't need to be installed on every client, just on the server(s). The latter is much easier because servers are centralized and there is a lot less of them than clients – which are not only numerous but also often out of reach.

The JavaGram model also retains some of the advantages of the traditional fat client model; namely:

- Unlike the thin client model which is stateless, the JavaGram model is stateful. Because each client is allocated a dedicated session that lasts for the duration of the connection, the session accurately reflects the server-side state of the client. This eliminates the burden of additional programming normally required for thin clients to keep track of state information.
- JavaGram supports both synchronous and asynchronous requests, as opposed to the asynchronous-only thin client model.

- The programmer can delegate some of the application's processing to the client end (e.g., report generation, complex calculations) within the same code base. This provides more scope for load balancing and making sure that server(s) don't become a bottleneck.

For lack of a better term and to distinguish the JavaGram model from the fat client and thin client models, we'll subsequently refer to it as the *hybrid client* model.

2.2 Static and Dynamic Loading

JavaGram support static as well as dynamic loading of scripts. Static loading is suitable for specifying cross-script dependencies. For example, if a script named `Report` refers to classes defined in two other scripts named `Product` and `Customer` then the former should load the latter like this:

```
<load>
  "src/Product"
  "src/Customer"
</load>
```

This is called static loading and is somewhat similar to the `import` statement in Java, except that the scripts need not be locally present, but may be located on an application server.

However, in some situations we don't want a script to be loaded until it's triggered by an event (e.g., the user pressing a button). This is facilitated by dynamic loading using the `sys.load()` method, which can be called from within JavaGram code. As with static loading, the requested script may be local or be on an application server.

The JavaGram application server uses an implicit form of dynamic loading to serve remote calls received from a client. When a remote call (say, `Product.update(...)`), is received from a client, the underlying message includes the path of the script that contains the definition of the `Product` class. The server uses this path to dynamically load the script if it's not already loaded. This ensures that the loading of scripts on the server-side is automatic, demand-driven, and hence not a burden on the programmer.

2.3 Code Caching

To minimize client-server traffic, JavaGram employs a comprehensive caching mechanism, which maintains an active cache on both the client and the server sides (see Figure 3).

On the client side, the cache is somewhat similar to the local cache of a web browser, but is more deterministic. Whereas a web browser refreshes the pages in its cache based on their age, JavaGram requires an exact timestamp match. This is necessary in order to ensure the consistency of the versions of scripts that make up an application.

When a JavaGram client requests a script, the script is looked up in the local cache first. If not present, the script is sourced from the server, deposited into the local cache and given exactly the same timestamp as the server-side original, and then used. If a requested script is already present in the client-side cache then its timestamp is compared against the server-side original. If the timestamps match, then the client-side copy is deemed to be identical to the server-side original

and is used as is; otherwise, it's refreshed by copying the server-side original to the client-side cache. This scheme ensures that when a server-side script is modified, it will find its way to all clients that attempt to use it.

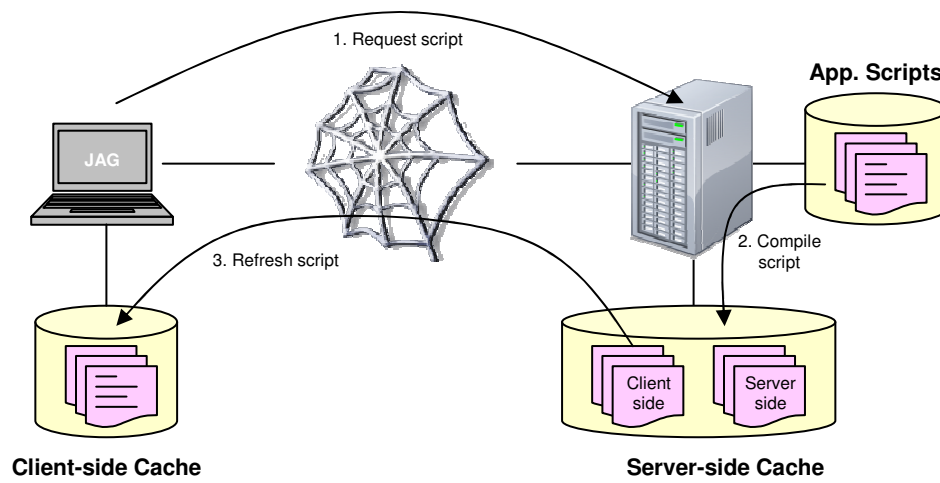


Figure 3. JavaGram script caching.

JavaGram also employs a server-side cache for a different purpose. Because a JavaGram script can contain code intended for clients and/or servers, each script is compiled to produce two variations – one for the client-side (from which irrelevant information such as the implementation of remote methods is removed) and one for the server-side (from which irrelevant information such as GUI code is removed). These binary files are compressed (to minimize transmission overheads) and deposited into the server-side cache. Furthermore, the files are given exactly the same timestamp as the original script from which they were compiled. The compilation process is dynamic, i.e., the script is compiled only when it's actually requested by a client or server and when the cached version is absent or out of date.

The fact that JavaGram servers dynamically compile code doesn't mean that an application build must be provided in source format. If a script is released in pre-compiled, binary format (e.g., for intellectual property reasons) then the server will generate the client-side and server-side binaries from the pre-compiled version of the script.

2.4 Built-in Types

JavaGram provides the following built-in types.

- Atomic Types:
 - `boolean` (similar to Java `boolean`)
 - `char` (similar to Java `char`)
 - `int` (similar to Java `long`)
 - `real` (similar to Java `double`)
 - `string` (similar to `java.lang.String`).

- `symbol` (like `string` except that multiple instances having the same representation are stored only once)
- `date` (date and time)
- `stream` (mechanism for performing IO with respect to a file, buffer, or socket)
- **Composite Types:**
 - `vector` (contiguous sequence of values, with random access)
 - `list` (sequence of values, without random access)
 - `map` (key-value pairs, with random access)
 - `object` (opaque instances of user-defined types, similar to `java.lang.Object`)
- **Pseudo Types:**
 - `vague` (can represent any type)
 - `native` (Java values with no equivalent type in JavaGram)
 - `void` (absence of a value)

Values for atomic and composite types can be created literally (except for `stream`) or programmatically. Here are some examples of literal values:

```
$name                // symbol literal
[#2007-12-25 14:35:40] // date/time literal
[10, 20, 30]         // vector<int> literal
[10, "twenty", 30]  // vector<vague> literal
[$one], [], [$two, $three] // vector<vector<symbol>> literal
$(1, 2.0, "three")  // list literal
[$name => "Bill", $dob => [#1977-08-20], $sex => "male"]
                    // map<symbol, vague> literal

class Person {
    string name;
    date dob;
    string sex;
    // methods...
}
[@Person name => "Alex", dob => [#1972-10-23], sex => "Male"]
                    // Person object literal
```

It's also worth pointing out that the JavaGram cast operator is `@`, and is more versatile than the Java cast operator. Examples:

```
Customer cust = obj @ Customer; // cast obj to Customer type
Customer cust @= obj;           // same as above
map m = sql.getRow@map(rs);     // instruct getRow() to return a map result
```

2.5 Object Orientation

The OO features of JavaGram are very similar to Java, with the following notable exceptions.

- Support for multiple inheritance.
- Support for remote methods and classes, both of which are managed transparently to the programmer.
- Support for text members (behave like methods and allow you to do advanced text processing).
- Support for GUI members (behave like class fields and allow you to define user interface components declaratively and hierarchically).
- Support for singleton classes.
- Support for object literals (class instances that are created at load time rather than runtime).
- Ability to limit the visibility of a class method to client or server side.
- Ability to specify default argument values for methods.
- No interface construct (abstract classes are used instead).
- Class definitions may not be nested.

Examples of some of these features appear in subsequent sections.

2.6 Multiple Inheritance and Mutual Classes

Multiple inheritance (MI) is a powerful design tool that, when used judiciously, can simplify a design and reduce development effort. Unfortunately, MI has attracted plenty of bad publicity due to complex realizations (e.g., C++) that programmers have struggled with. JavaGram attempts to remedy this using mutual classes:

- A derived class can have multiple base classes, provided at most one of them is non-mutual.
- All the base classes of a mutual class (if any) must also be mutual.
- Mutual classes that are inherited more than once in a class hierarchy (as in the ‘dreaded diamond’ problem) are treated as if they are inherited once. Therefore, an instance of the derived class contains only one instance of the mutual class’s fields. In this sense, mutual classes behave like virtual base classes in C++.

Let’s illustrate using some examples. Consider the following three classes for event handling.

```
class Event {
    public symbol getName () {
        return typeof(this);
    }
}

mutual class EventInitiator {
    protected map<symbol, vector<EventListener>> eventMap @= map();

    public void addListener (vague className, EventListener listener) {
        symbol eventName = typeof(className);
        vector<EventListener> listeners = eventMap[eventName];
    }
}
```

```

        if (listeners == null)
            eventMap[eventName] = listeners @= vector();
        sys.append(listeners, listener);
    }
    public void raiseEvent (Event event) {
        for (EventListener listener in eventMap[event.getName()]) {
            if (listener.consumeEvent(event))
                break;
        }
    }
}

abstract mutual class EventListener {
    // Return true to stop propagation.
    abstract public boolean consumeEvent (Event event);
}

```

The programmer can define new events by sub-classing `Event`; allow a class to raise events by sub-classing `EventInitiator`; and define event listeners by sub-classing `EventListener`, which can then be registered by calling `EventInitiator.addListener()`.

Now consider the following two classes.

```

abstract class View {
    // Base class for any screen-based representation...
}

class TableView extends View, EventInitiator, EventListener {
    // Provides a tabular, read-only view of data...
}

```

`TableView` is intended to provide a screen-based representation of tabular data. It sub-classes `EventInitiator` so that it can raise events (e.g., in response to the user selecting a row), and `EventListener` so that it can listen to events (e.g., changes to the underlying data). It also sub-classes `View` to inherit viewing functionality (such as mapping data to GUI widgets).

Because of **MI**, `TableView` will require minimal amount of code. In absence of **MI**, both `EventInitiator` and `EventListener` would have been defined as interfaces, requiring `TableView` to implement their methods.

Continuing further with our example, consider the following two classes.

```

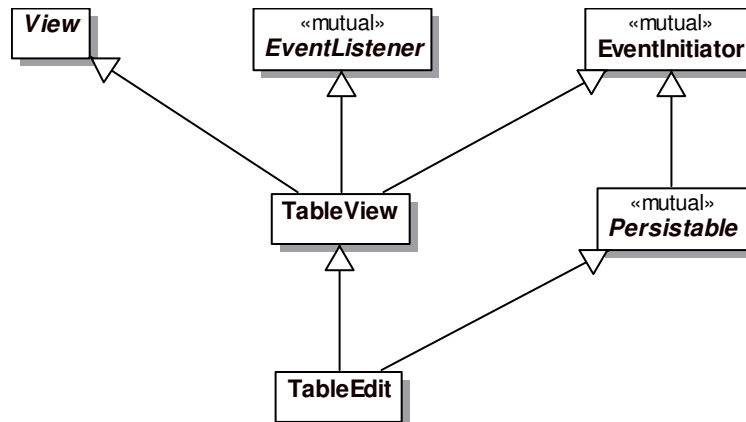
mutual abstract class Persistable extends EventInitiator {
    // Base class for any data that needs to be persistable
}

class TableEdit extends TableView, Persistable {
    // Provides an editable view of persistable tabular data
}

```

Persistable represents any data that needs to be persisted to disk storage. It sub-classes EventInitiator so that it can raise events in response to in-memory and on-disc data going out of sync or being reconciled. TableEdit is intended to enable the user to edit tabular data and permanently save such changes. Therefore, it sub-classes TableView and Persistable.

Here is the inheritance hierarchy for TableEdit.



Although EventInitiator appears twice in TableEdit's inheritance path, because the former is a mutual class, only one instance of it will appear in an instance of TableEdit. This is a desirable outcome (as only one EventInitiator is needed to manage the events) and addresses the first common problem of MI.

The second problem associated with MI is ambiguous methods calls. This is detected by JavaGram and can be overcome by explicit casting. For example, if a class C derives from classes A and B, both of which offer the same method foo() with identical signatures, the call c.foo() is reported as ambiguous, but can be resolved by coding it, for example, as c@A.foo().

2.7 Remote Methods

Remote methods (and remote classes) represent one of the most powerful features of JavaGram. They make the task of writing client-server programs exceptionally easy by removing the burden of having to deal with data communication, synchronization, hand-shaking, error handling, and so on. As a result, invoking a remote method on a server becomes as easy as invoking a local method. Hiding all this complexity has the added benefit of allowing the programmer to easily use the same code in different deployment models (standalone versus distributed).

Let's illustrate with an example.

```
class Person {
    int id;
    string firstName;
    string lastName;
    date dob;
    string sex;
    //...
```

```

}

class CRM {
    public remote static vector<Person> findPersons (Person criteria) {
        // Returns all persons matching criteria
    }
    //...
}

```

The method `CRM.findPersons()` is defined as **remote** (causing its implementation to reside on the server-side, and the client receiving only the signature of the method). When invoked (regardless of where it's invoked), it will execute on the server-side. The method is intended to interrogate a database, find a set of records that match `criteria`, and return them as a vector of `Person` instances.

Here is a sample call to this method which returns all male persons having the surname 'Smith':

```
vector<Person> matches = CRM.findPersons([@Person lastName=>"Smith", sex=>"Male"]);
```

This is what happens 'under the wraps' when a remote method is invoked:

- JavaGram determines the target of the method (i.e., the server that should handle the call).
- A message is composed that captures the call (method, its class, supplied arguments) and sent to the target server.
- The server dynamically loads the method's class (if not already loaded) and performs the call.
- The server composes a response that captures the returned value (or thrown exception) and sends this back to the client.
- The client processes the response to retrieve the returned value (or exception) and delivers it to the caller.

By default, the target of a remote call is the server from which the caller has sourced the class code (e.g., through a static or dynamic load operation). When a client is running against a single logical server (which can have multiple identical load-balanced physical instances) this will suffice. However, if the client is designed to run against multiple logical servers then the call can be targeted at a specific server. For example, in

```
stream svr = sys.client("localhost", 444, null, null);
Vector v = svr::CRM.findPersons(...);
```

`findPersons()` is targeted at the server denoted by `svr`.

JavaGram's exception handling works seamlessly across the client-server boundary. For example, an exception raised (on the server-side) by a remote method is delivered to the caller (on the client-side) as if it were a locally raised exception.

2.8 Remote Classes

Sometimes it's desirable to define an entire class as remote so that all its actual processing is handled on the server-side. Continuing with our example from the previous section, consider the following classes.

```
class Address {
    string street;
    string town;
    string state;
    string zip;
    //...
}

remote class Customer extends Person, Persistable {
    vector<Address> addresses;
    //...
    public void updateDetails (Person details) {
        // Updates customer's personal details
    }
}

class CRM {
    public remote static vector<Person> findPersons (Person criteria) {
        // Returns all persons matching criteria
    }
    public remote static Customer getCustomer (int id) {
        // Returns customer denoted by id
    }
    //...
}

class CustomerView {
    Customer customer;
    //...
    public void saveChanges (Person details) {
        customer.updateDetails(details);
    }
}
```

Here, `Person` has been sub-classed as `Customer` which, in addition to personal details, captures other customer information as well as providing persistence. Because `Customer` can have substantial information within it, we've decided to define it as a remote class so that this information will remain on the server-side and changes to it can be managed centrally. The remote method `CRM.getCustomer()` will look up and return a customer by its ID.

Next, we've defined a `CustomerView` class to allow users to view and edit customer information. The `saveChanges()` method of this class updates the personal details of the displayed customer by calling `customer.saveChanges()`. Because the latter is a method of a remote class, it's invoked remotely in a manner similar to remote methods.

JavaGram's 'under the wraps' handling of remote classes is as follows:

- The implementation of a remote class always remains on the server-side. When a client loads a remote class, it only receives a definition of its interface.
- An instance of a remote class can only reside on the server-side.
- When a client requests an instance of a remote class (e.g., by calling a remote method of another class), it receives a proxy object instead.
- Any operation performed by the client on a proxy object (field access or method invocation) is communicated by JavaGram to the server-side and applied to the real object instead.

In practice, a client has no way of knowing whether it's dealing with a remote method/class or a local one. This ensures that code designed to be deployed as client-server will also work when run as standalone. The obvious benefit of this is a simplified testing process – developers can develop and test standalone and later switch to client-server when the code is more fully developed.

2.9 Asynchronous Method Invocation

Asynchronous method invocation allows you to call a (local or remote) method such that you don't have to wait for it to complete. Execution proceeds to the next statement as soon as the call is made. When the call eventually completes, a callback method is invoked to complete the processing. If an error callback is also specified and the method throws an exception then that callback will be invoked instead.

Here is a simple example:

```
vector v = CRM.findPersons(criteria) -> viewer.show(v) -> viewer.error("failed");
```

The call to `CRM.findPersons()` returns immediately. Later, if the call completes successfully `viewer.show()` is invoked to display the result. Otherwise, `viewer.error()` is called to handle errors.

2.10 Declarative GUIs

JavaGram offers a completely different style of GUI programming to Java's Swing. Whereas GUI programming in Swing is procedural, JavaGram allows you to define a GUI declaratively. This has a number of advantages: you write a lot less code, the code is much more readable, and the code readily portrays the hierarchical structure of the GUI. As a result, creating sophisticated GUIs in JavaGram is much easier than in Java.

GUI members are defined using a markup notation. Semantically, however, GUI members behave like class fields, and therefore may be specified with any of the qualifiers allowed for fields.

We will continue with our example from earlier sections to illustrate. Here is a simple class that defines a panel for displaying a `Person`:

```
class PersonGui {
```

```

<Panel personView title="Personal Details">
  <Layout.gridBag/>
  <Lay row=0 col=0 weight=0.0>
    <Label title="First Name" align=$east/>
  </Lay>
  <Lay row=0 col=1 margin=2 fill=$horizontal>
    <Field.text key=$firstName/>
  </Lay>
  <Lay row=0 col=2 weight=0.0>
    <Label title="Surname" align=$east/>
  </Lay>
  <Lay row=0 col=3 margin=2 fill=$horizontal>
    <Field.text key=$lastName/>
  </Lay>
  <Lay row=1 col=0 weight=0.0>
    <Label title="Date of Birth" align=$east/>
  </Lay>
  <Lay row=1 col=1 margin=2 fill=$horizontal>
    <Field.date key=$dob/>
  </Lay>
  <Lay row=1 col=2 weight=0.0>
    <Label title="Sex" align=$east/>
  </Lay>
  <Lay row=1 col=3 margin=2 fill=$horizontal>
    <Combo key=$sex data=["Male", "Female"]/>
  </Lay>
</Panel>
}

```

PersonGui has a GUI member called `personView`, which is defined as a panel. The markup notation makes it easy to nest definitions and to specify property values. For example, the panel has a `title` property, is given a `gridBag` layout, and uses the `Lay` markup to arrange the panel widgets. The widgets are specified using `Label`, `Field.text`, `Field.date`, and `Combo` markups. Note how these widgets have been assigned `key` properties – these match the field names of the `Person` class to facilitate data binding.

The next class specifies the GUI representation for a `Customer` by extending `PersonGui`.

```

class CustomerGui extends PersonGui {
  <Panel custView>
    <Layout.border/>
    <Indirect ref={personView} lay=$north/>
    <Panel title="Addresses" lay=$center>
      <Layout.border/>
      <Pane.scroll>
        <Table addr format={ADDR_FORMAT} key=$addresses autoSize=true/>
      </Pane.scroll>
    </Panel>
  </Panel>

  static final vector<map> ADDR_FORMAT = [

```

```

        [$key=>$street, $title=>"Surname", $width=>100, $align=>$west]
        ,[$key=>$town, $title=>"Town", $width=>60, $align=>$west]
        ,[$key=>$state, $title=>"State", $width=>30, $align=>$west]
        ,[$key=>$zip, $title=>"Zip", $width=>40, $align=>$east]
    ];
}

```

CustomerGui defines a panel with border layout. Note how personView (defined in the base class) is referenced using an Indirect markup, and laid to the north of the parent panel. In the center of the parent panel, another panel is placed which contains a Table (for displaying customer addresses) wrapped in a Pane.scroll. The format of the table columns is specified by its format property which refers to a vector of maps, where each map specifies the format of one column. Again, note how each column format has a \$key entry – this specifies the name of the corresponding field in the Address class, for data binding purposes. The key property of the table itself denotes the addresses field of the Customer class, again for data binding.

Now let's write a simple program for testing our classes.

```

singleton class TestApp extends GuiApp {
    <App app lookAndFeel=$windows>
        <Frame mainFrame title="Sample" width=400 height=250 event=frameHandler>
            <Indirect ref={new CustomerGui().custView}/>
        </Frame>
    </App>

    public TestApp () {
        super(mainFrame);
        gui.bind(mainFrame, testData);
    }
    protected void frameHandler (native comp, symbol event) {
        if (event == $close)
            super.exit();
    }
    public static void main () {
        TestApp.singleton.run();
    }

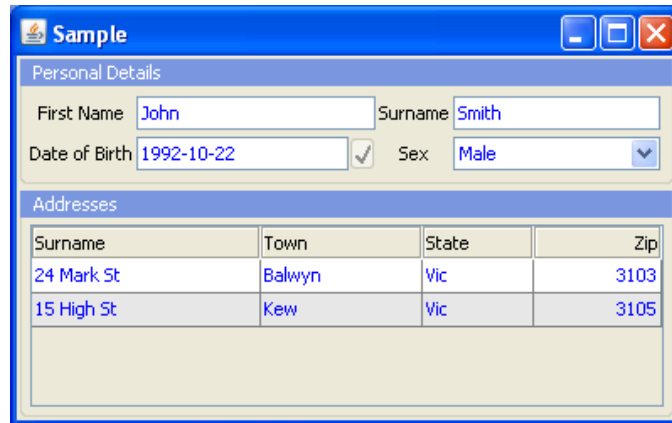
    Customer testData = [@Customer
        id=>1, firstName=>"John", lastName=>"Smith",
        dob=>[#1992-10-22], sex=>"Male",
        addresses=>[
            [@Address street=>"24 Mark St", town=>"Balwyn",
                state=>"Vic", zip=>"3103"]
            ,[@Address street=>"15 High St", town=>"Kew",
                state=>"Vic", zip=>"3105"]
        ]
    ];
}

```

TestApp sub-classes GuiApp – a library class that provides basic GUI application functionality. It uses App and Frame markup to define a simple application frame. The event handler for the frame is a local method which exits the application in response to a window close event. Note how the constructor calls gui.bind() to bind the frame to testData (a locally-defined instance of Customer

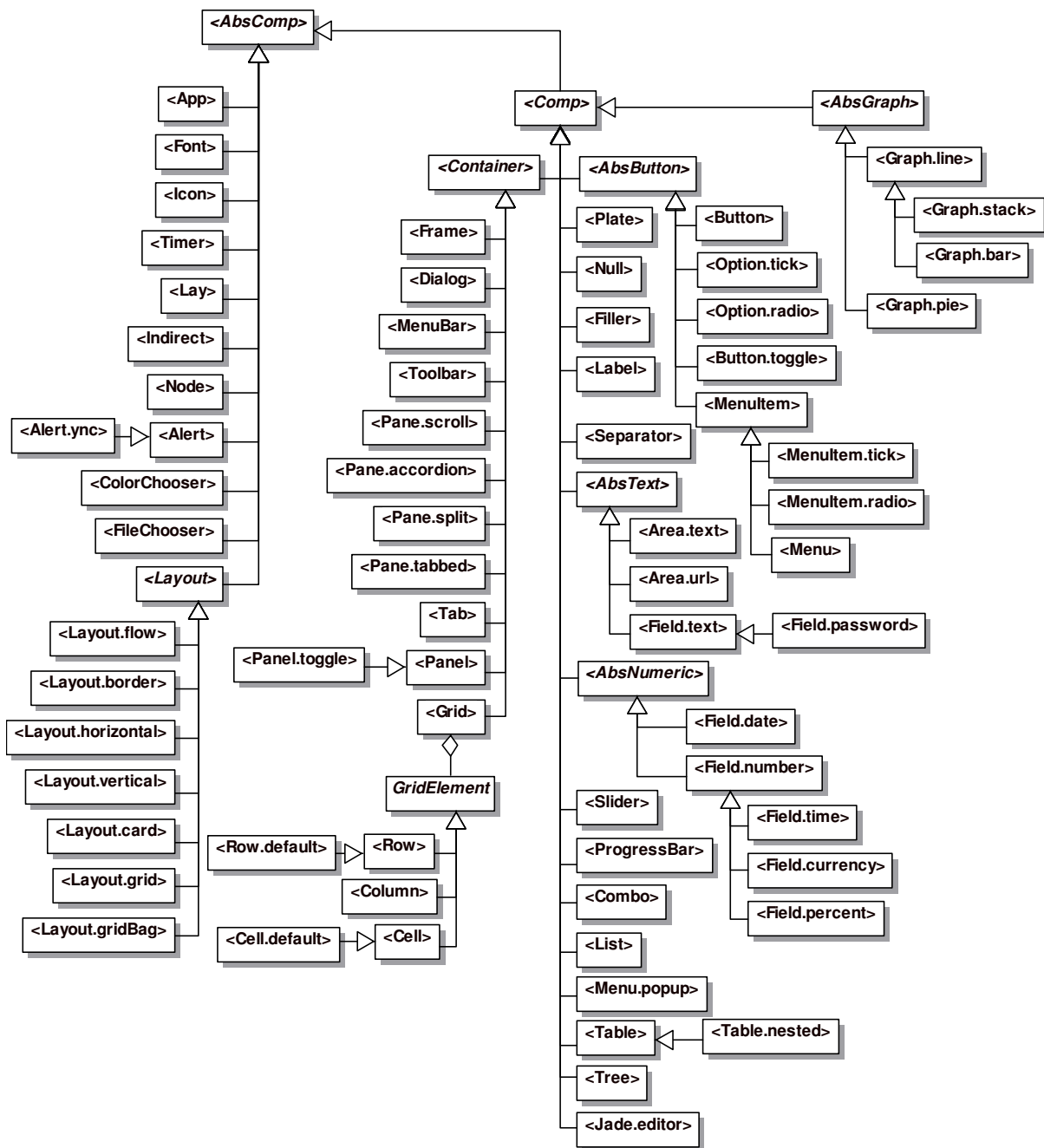
which also contains a couple of `Address` instances). Finally, because `TestApp` is a singleton class, the main method simply uses `TestApp.singleton.run()` to create and run an instance of it.

Here is what the application GUI looks like:



If you compare the code for this application to an equivalent Java application, you will find the difference quite striking. The Java version will be much longer and less readable.

It's not practical to describe the full extent of the JavaGram GUI notation in the limited space of this paper. Instead, the following UML diagram provides a quick summary of what it covers.



The GUI markup notation can also be extended by the programmer. For example, to support spider graphs, you can sub-class `jag.gui.tags.GuiAbsGraph` (the underlying Java class for `<AbsGraph>`) and implement a class named `jag.gui.tags.GuiGraph_spider`. If the class file for this is added to the class path for JavaGram, then the markup `<Graph.spider>` will seamlessly become available to the programmer. JavaGram uses reflection to achieve this.

2.11 Parameterized Text

Programs often have to do some level of text manipulation. Examples are:

- Program-generated email messages

- Composing SQL commands
- Generating (screen or document based) messages and reports

In most programming languages, this is done through string concatenation (e.g., using the `+` operator or the `StringBuilder` class of Java). The end result is rather messy and difficult to visualize due to the procedural nature of the code.

JavaGram offers two facilities to overcome this. The first is called delayed strings and is useful for simple text parameterization tasks. Here is an example:

```
sys.println($"found {n} records in {s} seconds");
```

Assuming that both `n` and `s` are variables, this will produce an output such as:

```
found 2000 records in 0.05 seconds
```

The rules are quite simple: a delayed string is preceded by a `$` and can contain arbitrary expressions enclosed in braces. Unlike normal strings, which are created at load time, a delayed string is created at runtime by evaluating and substituting the embedded expressions.

For more elaborate tasks, JavaGram provides text class members. A text member is like a method and can be used to handle parameterized text. It is defined using the markup notation, and can be prefixed with method qualifiers. For example:

```
class Foo {
    static public
    <text string newAccEmail (string name, string id, string passwd)>
        Dear {name},
        This is to confirm that your account has been setup.
        Your Userid is: {id}
        Your Password is: {passwd}
        You will be required to reset your password when you first login.
    </text>
    //...
}
```

The call

```
sys.println(Foo.newAccEmail("John Smith", "john", "xyz21!"));
```

will return the following string:

```
Dear John Smith,
This is to confirm that your account has been setup.
Your Userid is: john
Your Password is: xyz21!
You will be required to reset your password when you first login.
```

Like GUI markup, the text markup notation is extensible. In fact, JavaGram provides a number of such extensions for SQL handling, as discussed in the next section.

2.12 Database Interaction

In JavaGram, interaction with databases is facilitated by the `sql` pseudo class. Additionally, a number of parameterized text constructs are provided to make SQL formation straightforward and consistent with the declarative style of JavaGram.

Let's illustrate these using a few simple examples. Recall the `Person` class from earlier sections, and suppose that a database table called `person` has been created to provide persistence for it, having the following SQL schema (keys not shown):

```
table person:
  id integer not null
  first_name varchar(16)
  last_name varchar(16)
  dob date
  sex varchar(8)
```

Here is a class that retrieves persons based on surname:

```
singleton class TestSql {
  <text.sql string sqlFindPerson (string lastName)>
    select * from person
    where last_name = {`lastName}
  </text.sql>

  public void testFind () {
    query ($mysqlDb) {
      native rs = sql.executeQuery(sqlFindPerson("Smith"));
      while (sql.next(rs))
        sys.println(sql.getRow@Person(rs));
    }
  }
  public static void main () {
    TestSql.singleton.testFind();
  }
}
```

The text method `sqlFindPerson()` formulates the required SQL command. The `<text.sql>` markup behaves like the `<text>` markup, except that it supports additional properties and allows automatic escaping of text parameters. So, for example, `{`lastName}` is expanded to `'Smith'`. `testFind()` performs a query by calling `sql.executeQuery()` and passing the SQL returned by `sqlFindPerson()` to it. This returns a result set which we then iterate through using `sql.next()`. Each row is extracted by calling `sql.getRow@Person(rs)`. The casting here instructs `sql.getRow()` to return each row as a `Person` instance, thus avoiding a lot of unnecessary code to convert the row data to `Person` fields. It matches column names against class fields by automatically converting, for example, `first_name` to `firstName`.

Note the use of the `query` statement – it assigns a database connection to the thread as the enclosed section of code is executed, thus eliminating the possibility that the connection might be used by another thread. The `$mysqlDb` argument of the `query` statement refers to a database connection pool specified elsewhere in the application configuration, which contains an entry like this:

```
[
  $database =>
    [$mysqlDb =>
      [$name => "MySQL", $url => "jdbc:mysql://localhost/test"
      , $user => "root", $password => "password"
      , $timeout => [$login => 30, $query => 20]
      , $driver => "com.mysql.jdbc.Driver"
      ]
    ]
  //...
]
```

The program produces an output like this:

```
[@Person dob=>[#1992-05-16], firstName=>"John", id=>1, lastName=>"Smith",
sex=>"Male"]
```

An easier method of doing queries and updates is to use the `<text.sql.query>` and `<text.sql.update>` markups, as shown in the following example.

```
singleton class TestSql2 {
  <text.sql.query native sqlFindPerson (int id) db=$mysqlDb>
    select * from person
    where id = {`id}
  </text.sql>

  <text.sql.update int sqlUpdatePerson (Person p) db=$mysqlDb>
    update person
    set first_name = {`p.firstName},
        last_name = {`p.lastName},
        dob = {`sys.format(p.dob)},
        sex = {`p.sex}
    where id = {`p.id}
  </text.sql>

  public void testFindAndUpdate () {
    transaction ($mysqlDb) {
      native rs = sqlFindPerson(1);
      if (sql.next(rs)) {
        Person p = sql.getRow@Person(rs);
        sys.println(p);
        // Modify person:
        p.firstName = "Peter";
        p.dob = [#1981-12-25];
        sqlUpdatePerson(p);
      }
    }
  }
}
```

```

        // Check modifications:
        rs = sqlFindPerson(1);
        if (sql.next(rs))
            sys.println(sql.getRow@Person(rs));
    }
}
}
public static void main () {
    TestSql2.singleton.testFindAndUpdate();
}
}

```

Unlike `<text.sql>`, `<text.sql.query>` performs the query when invoked and returns a result set. Similarly, `<text.sql.update>` performs the update and returns an update count.

Note the use of the `transaction` statement – it behaves like the `query` statement but, in addition, groups a series of SQL statements into a single transaction.

The `testFindAndUpdate()` method retrieves a person, changes his first name and date of birth and retries it again to confirm that the changes have taken place. It produces an output like this:

```

[@Person dob=>[#1992-05-16], firstName=>"John", id=>1, lastName=>"Smith",
sex=>"Male"]
[@Person dob=>[#1981-12-25], firstName=>"Peter", id=>1, lastName=>"Smith",
sex=>"Male"]

```

For repetitive database operations, you can create a prepared statement and reuse it, as illustrated by the next example.

```

singleton class TestSql3 {
    Person p;

    <text.sql.prepare native sqlInsertPerson () db=$mysqlDb>
        insert into person (id, first_name, last_name, dob, sex)
        values ({?p.id}, {?p.firstName}, {?p.lastName}, {?sys.format(p.dob)}, {?p.sex})
    </text.sql.prepare>

    public void testInsert () {
        transaction ($mysqlDb) {
            native stmt = sqlInsertPerson();
            p = [@Person id=>3, firstName=>"Linda", lastName=>"Boyd",
                dob=>[#1991-02-13], sex=>"Female"];
            sql.execUpdate(stmt);
            p = [@Person id=>4, firstName=>"Tim", lastName=>"Bold",
                dob=>[#1982-07-10], sex=>"Male"];
            sql.execUpdate(stmt);
        }
    }
    public static void main () {
        TestSql3.singleton.testInsert();
    }
}

```

```
}
```

`<text.sql.prepare>` returns a prepared statement. Note that rather than passing `p` as an argument to `sqlInsertPerson()`, we've specified it as a class field. The reason is that placeholders such as `{?p.id}` are not evaluated at the time of the call to this method, but later, when `sql.executeUpdate()` is called on the statement.

One of the readability benefits of the JavaGram style of SQL programming is that all SQL commands are localised to `<text.sql...>` markups and are hence easily identifiable. This is superior to the traditional style where SQL is freely sprinkled throughout the code.

2.13 Serialization and Parsing

A useful feature of JavaGram is that any value (other than `stream` and `native` type values, but including class instances) can be serialized to clear text, as well as parsed without any extra programming effort. In this sense, JavaGram is similar to Lisp (where the `print` and `read` functions are capable of serializing and parsing anything). The resulting benefits are:

- Complex data structures (such as meta data) can be pre-created in code. This is convenient as well as self-documenting.
- Programs can be debugged more easily.
- Composite data can be stored in a single database column and retrieved easily, thus facilitating much simpler data models.
- The data exchanged between client and server ends can be easily viewed in a readable textual format.

JavaGram provides these method for serialization and parsing:

- `sys.print()` for serializing data.
- `sys.read()` for parsing data.
- `sys.serialize()` for serializing code as well as data (to string format)
- `sys.parse()` for parsing code as well as data (from string format)

The source/target of `sys.read` and `sys.print` is always a stream, which in turn may represent standard input/output, a file, a socket connection, or a memory-based buffer.

The ability to parse data or code 'on the fly' can be very useful in some programming situations. It's especially useful in agile programming, as it can substantially reduce coding and maintenance effort. We'll explore a practical example of this in the next section.

2.14 Streamlined Data Modelling

Earlier, we looked at examples of providing persistence for the `Person` class. If we were to continue with traditional entity relationship (ER) modeling, we would also provide a table for `Address` and one for capturing the one-to-many relationship between `Person` and `Address`, or have a foreign key in the `address` table to depict a person. This style of ER modeling, while offering

the greatest SQL composition flexibility, is rather time consuming and maintenance intensive – changes in the object model (e.g., adding a new field to `Person`) would typically require changes to the data model. Agile development requires a smarter approach.

Based on the requirements for the CRM application, we might conclude that the only retrieval scenarios required for `Customer` are via its ID, surname, or date of birth. We might also conclude that addresses never need to be handled outside the context of a customer. This enables us to capture `Customer` data in just one very simple table.

```
class Customer extends Person {
    vector<Address> addresses;
    //...
    static public
    <text.sql.update int createTable () db=$mysqlDb>
        create table if not exists customer (
            id integer not null,
            last_name varchar(16) not null,
            dob date,
            details text,
            primary key (id), index (last_name, dob)
        )
    </text.sql>

    public
    <text.sql.update int insert () db=$mysqlDb>
        insert into customer (id, last_name, dob, details)
        values ({`id}, {`lastName}, {`sys.format(dob)}, {`sys.serialize(this)})
    </text.sql.update>

    static private
    <text.sql.query native _getById (int id) db=$mysqlDb>
        select details from customer
        where id = {`id}
    </text.sql.query>

    static public Customer getById (int id) {
        query ($mysqlDb) {
            native rs = _getById(id);
            if (sql.next(rs))
                return sys.parse(sql.get(rs, $details)@string) @ Customer;
        }
        return null;
    }

    static public void main () {
        transaction ($mysqlDb) {
            createTable();
            testData.insert();
            sys.println(getById(1));
        }
    }
}
```

```

static Customer testData = [@Customer
    id=>1, firstName=>"John", lastName=>"Smith", dob=>[#1992-10-22], sex=>"Male",
    addresses=>[
        [@Address street=>"24 Mark St", town=>"Balwyn", state=>"Vic", zip=>"3103"]
        ,[@Address street=>"15 High St", town=>"Kew", state=>"Vic", zip=>"3105"]
    ]
];
}

```

The `createTable()` method creates a table where the first three columns capture the indexed attributes. `insert()` inserts a `Customer` instance into the table. Note how `sys.serialize(this)` is used to serialize the entire object into the last column. Conversely, `getByID()` uses `sys.parse()` to reconstruct the `Customer` object by parsing the text in the last column.

The program produces the following output:

```

[@Customer dob=>[#1992-10-22],firstName=>"John",id=>1,lastName=>"Smith",
sex=>"Male", addresses=>[[@Address state=>"Vic",street=>"24 Mark St",
town=>"Balwyn",zip=>"3103"], [@Address state=>"Vic",street=>"15 High St",
town=>"Kew",zip=>"3105"]]]

```

The maintenance advantage of this approach should be obvious: adding new fields to the `Customer` class will not affect the data model and will have minimal impact on the code.

Some readers are likely to ask: how well would this perform? The answer is: better than expected. The efficiency of the JavaGram runtime environment ensures that the dominant bottleneck is database IO, not serialization or parsing.

For lack of a better term, we call this approach *streamlined data modelling*. The initial reaction of experienced ER practitioners to this approach is often dismissive: ‘this goes against normalization and won’t perform.’ Our practical experience however tells us that it has a large part to play in agile development. We’ve built large and successful applications with only a dozen underlying tables, which have required minimal changes despite volatile and evolving requirements. Our most valuable learning has been that one size doesn’t fit all, so the choice of a data model is best guided by the actual user requirements. Analysing user requirements will quickly reveal where the performance hot spots are, and these can be easily accommodated through traditional ER modelling. For everything else, a streamlined approach will deliver a rapid, inexpensive, and easy-to-maintain solution.

We can quantify this in rough terms. If you have a class that is likely to have millions of persistent instances that require to be retrieved in a number of different ways involving its relationships to other classes (i.e., needing SQL joins) then traditional ER modelling is your best bet. On the other hand, a class whose instances are in the thousands, or one that doesn’t involve complex relationship-based retrievals, is adequately served by the streamlined approach.

There is another angle to the performance argument that often gets overlooked. A traditional, narrow view of performance sees it purely in terms of code execution speed. A broader, agile view would see it in terms of both development cycle and execution speed. Programs are not things that are developed once and used for years to come. They take much time to develop and

even greater time and effort to keep operational while accommodating evolving requirements. Ignoring development speed in the overall equation is therefore unrealistic.

2.15 Business Objects

Earlier sections showed how persistent business objects can be realized using JavaGram's OO features and the `sql` pseudo class. JavaGram also provides a pseudo class (`bom`) and a library class (`Object`) that together provide a convenient framework for working with business objects that require persistence. We'll continue with our earlier example of `Customer` and show how this class can be defined using `Object`.

```
class Customer extends Object, Person {
    vector<Address> addresses @= vector();

    static final map tableSpec = [
        $db => $mysqlDb,
        $table => $customer,
        $columns => [
            $generatedKey => $id,
            $id => $id,
            $last_name => $lastName,
            $dob => $dob,
            $rest => [$fields => [$firstName, $sex, $addresses], $compress => true]
        ]
    ];

    public Customer (Person p) {
        super@Object(tableSpec);
        id = p.id;
        firstName = p.firstName;
        lastName = p.lastName;
        dob = p.dob;
        sex = p.sex;
    }

    public void addAddress (Address addr) {
        sys.append(addresses, addr);
    }

    static public void main () {
        Customer bo = new Customer([@Person id=>1, firstName=>"John",
                                   lastName=>"Smith", dob=>[#1992-10-22], sex=>"Male"]);
        bo.addAddress([@Address street=>"24 Mark St", town=>"Balwyn", state=>"Vic",
                       zip=>"3103"]);
        bo.addAddress([@Address street=>"15 High St", town=>"Kew", state=>"Vic",
                       zip=>"3105"]);
        bo.persist();
        vector<Customer> found @= Object.find(Customer,
                                             [$like => [$lastName => "Sm%"]]);
        sys.println(found);
    }
}
```

The key difference between the earlier examples and this style is that the latter is more declarative. By sub-classing `Object`, we avoid writing explicit SQL for `Customer`. Note how the desired organization for the `customer` table is specified by the `tableSpec` map. In addition to the database connection pool and table name, this map specifies the organization of the columns. The underlying table has four columns: `id` (which is auto generated), `last_name` (which maps to the `lastName` field), `dob` (which maps to the `dob` field), and `rest` (which encodes the values for `firstName`, `sex`, and `addresses` fields in a compressed format).

Once a customer object is created (or modified), it is saved to the database by calling its `persist()` method. The `find()` method provides a versatile method of lookup using one or more fields, with partial or exact matches. Other useful methods such as `delete()` and `refresh()` are also supported.

The program produces the following output:

```
[[@Customer dob=>[#1992-10-22],firstName=>"John",id=>1,lastName=>"Smith",sex=>"Male",
addresses=>[[@Address state=>"Vic",street=>"24 Mark St",town=>"Balwyn", zip=>"3103"],
[@Address state=>"Vic",street=>"15 High St",town=>"Kew",zip=>"3105"]]]]
```

The obvious advantage of this style of defining business objects is that, at least for vanilla operations, no direct SQL needs to be written.

2.16 Java Interoperation

JavaGram provides a simple facility for interoperating with its underlying implementation language (Java). The `sys.java()` method uses reflection to allow any Java class or method be accessed, and automatically maps data between JavaGram types and Java types. This method is rarely used, but is a useful last resort when the programmer needs to do something that JavaGram doesn't directly support.

3 Implementation

JavaGram has been developed in pure Java and is therefore platform independent. The implementation is packaged in three parts:

- JavaGram runtime environment (JAG).
- JavaGram Development Environment (JADE)
- JavaGram standard scripts

3.1 JavaGram Runtime Environment

JAG is packaged as a single JAR file (`JAG.jar`). It's very compact (currently under 2.5MB uncompressed) and provides a complete environment for deploying and running JavaGram applications, including:

- Parsing and interpreting JavaGram code
- Compiling JavaGram code into binary format
- JavaGram application server (for deploying JavaGram servers)

- JavaGram proxy server (for failover and load balancing application servers)

Every JavaGram client or server is deployed using JAG. A server deployment also requires the JDBC drivers of the databases accessed by the server, and a configuration file which specifies the server settings (for security, data compression, database connection pools, session management, etc.).

All JavaGram servers are generic and exceptionally easy to setup and deploy (taking only a few minutes). The same application server instance may serve many different applications. JavaGram code is deployed into an application server by simply dropping it into a nominated folder – no server startup is required – and the server does the rest.

One of JavaGram's important promises is that a client needs to be installed only once. The process is like this. The user downloads and runs a small installation program (which installs JAG.jar, a small Java keystore, and a shortcut to an application's URL) on the user's machine. When the user runs the shortcut, the application is automatically booted (scripts are demand-downloaded as necessary and executed). Application maintenance is totally transparent to the user:

- When a new version of the application is released to a server (i.e., revised scripts), these revisions automatically find their way to the client installations.
- If a new version of JAG.jar is released to a server, this can be automatically propagated to all clients that depend on the changes in this JAR, causing the new JAR to be automatically downloaded by the affected clients, and replacing the old JAR.

Furthermore, application code changes that leave class schemas unchanged (i.e., only affecting the implementation of methods) can be deployed without restarting a server or any client. This allows emergency hot fixes to servers without disruption to users.

JavaGram application servers are highly scalable. Multiple servers can be deployed in either a failover or load balanced configuration. Where load balancing is used, traffic is routed via one or more proxy servers, which in turn match clients against server instances based on their load.

3.2 Compilation

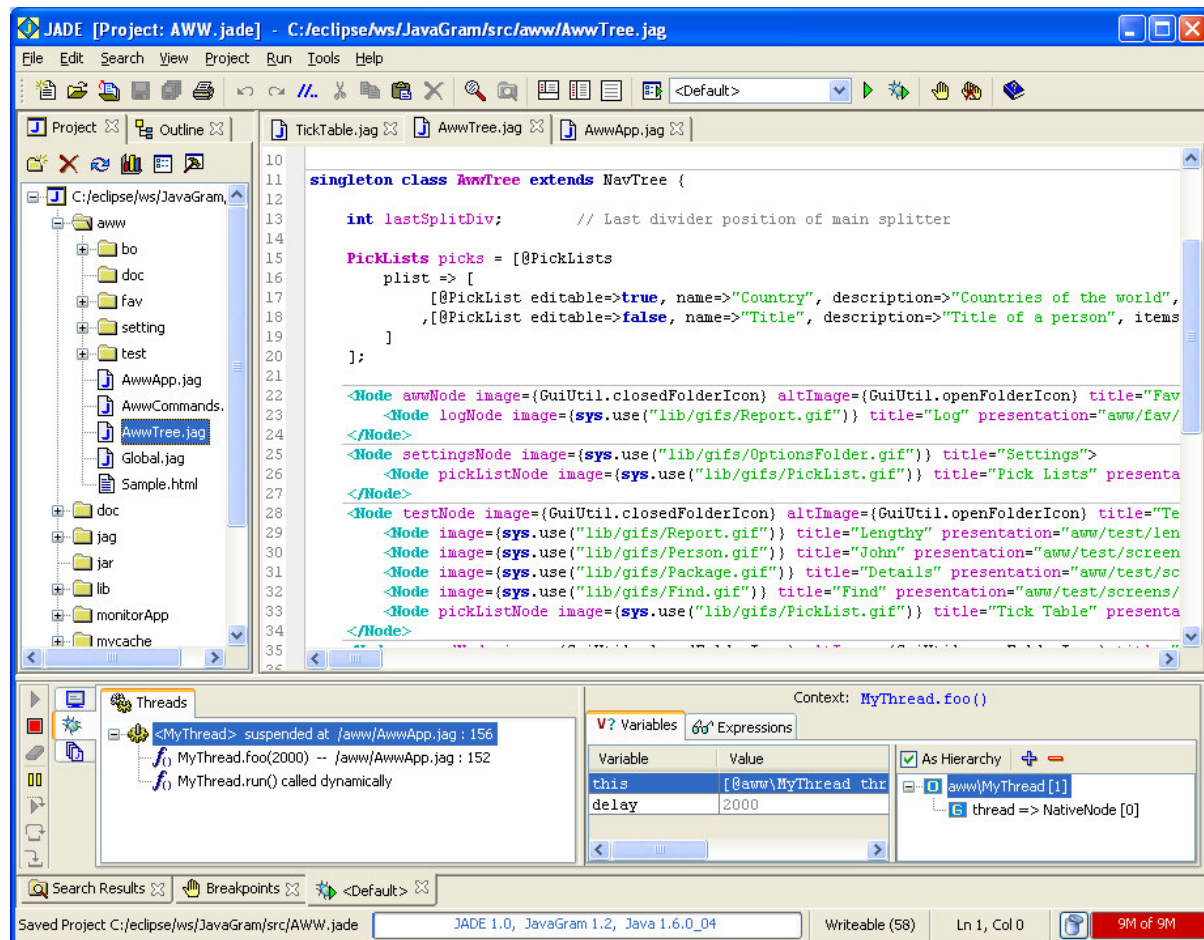
The JavaGram compilation process is straightforward. A script is parsed, analyzed, and converted to an equivalent byte format. When a server internally compiles a script, it produces two compiled versions, one for client-end and one for server-end. Either version excludes information that's not relevant to its intended target environment. Scripts can also be explicitly compiled, in which case a single version is produced that's equivalent to the source. Explicit compilation is suitable for cases where an application is to be deployed in binary rather than source format (e.g., for intellectual property reasons).

The JavaGram parser can parse scripts in source or binary format. The latter has the advantage of being more secure and more efficient – the parser has to do a lot less work.

The binary codec used by the compiler is version controlled, so if a client tries to load a script that's been encoded using an outdated codec, this is detected, causing a refresh from the server.

3.3 JavaGram Development Environment

The JavaGram development environment (packaged as JADE.jar) provides a productive visual environment for developing, debugging, and running JavaGram applications.



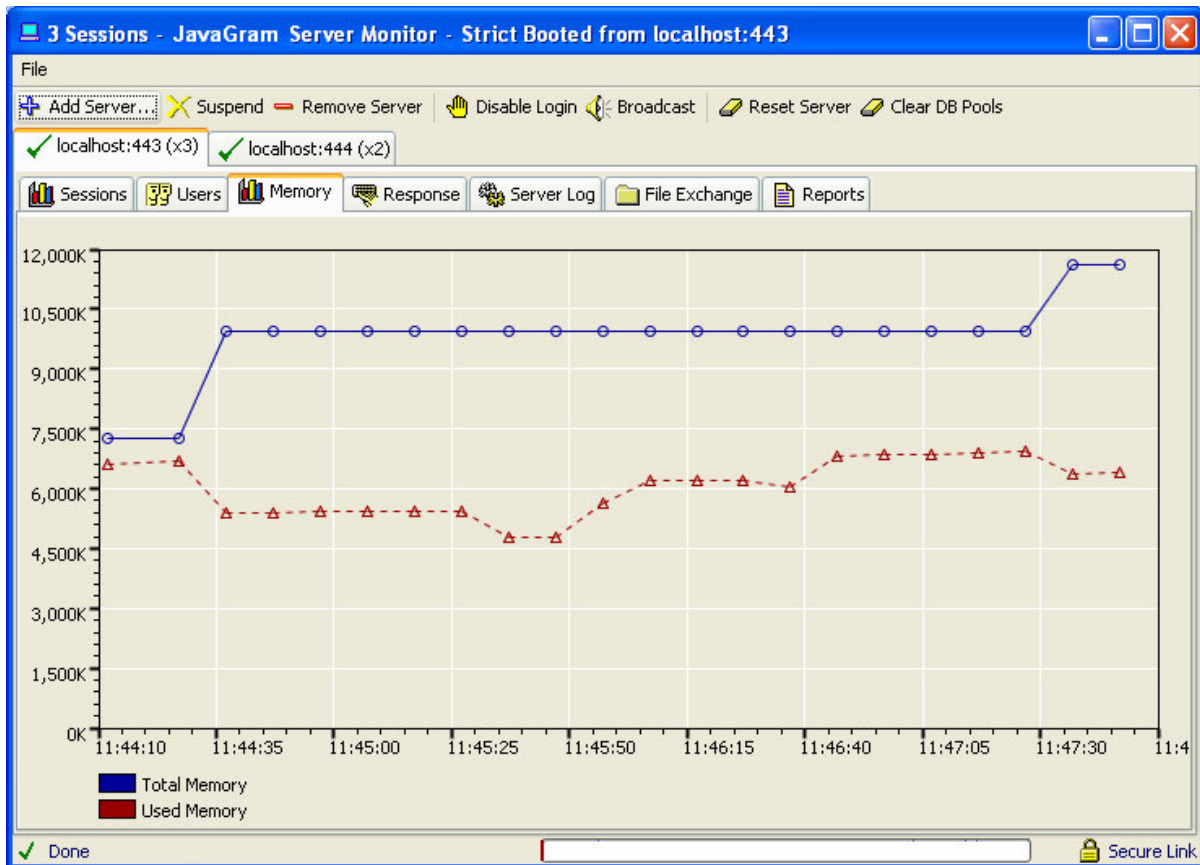
Key feature of JADE are:

- Project setup and view
- Customizable syntax-colored editing of JavaGram code
- Outline view of classes in terms of their members
- Ability to run servers, clients, and standalone deployments
- Ability to debug code, including setting up breakpoints, viewing the runtime stack, watching variables and expressions, and code insight
- Single and multi-file search and replace functionality
- Project compilation

Future versions of JADE will build upon this and provide more advanced functionality such as re-factoring.

3.4 Server Monitor

The server monitor is a utility that allows deployed JavaGram servers to be monitored. It's written purely in JavaGram.



Key features of monitor are:

- Monitor multiple application servers from one location
- View active sessions in a server; view each session's runtime stack, and terminate problematic sessions
- View all users connected to a server; disable additional logins; send a broadcast message to all users
- View memory usage and server latency graphically
- View the server log
- Upload/download files to/from servers
- Generate various reports

4 Future Development

A major area planned for future development is Adobe Flex integration. The goal is to allow the same code base to be used to deploy an application as desktop or browser-based, without loss of

functionality. Flex has been chosen for this purpose as its GUI design is fairly consistent with JavaGram's style. This initiative will involve the integration of the JavaGram application server with a web server, and the automatic translation of JavaGram GUI notation to Flex notation.

~ ~ ~