GE Energy

Smallworld Core Spatial Technology™ 4

Smallworld MAGIK™: The object oriented language for an object oriented world

GER-4235 (09/04)

Smallworld MAGIK: The object oriented language for an object oriented world

Abstract

In the late 1980s, application development was dominated by flat, procedural languages such as C, Pascal, Basic and so on. It was at the time that GE Energy Smallworld Software architects had the foresight to base a new GIS application on the foundation of a new object oriented technology that was beginning to emerge. Object oriented development offered the possibility of greater code reuse, encapsulation of data and functionality together with a natural and elegant way to model the complex real world assets that future enterprise GIS customers were expected to have. This new language was implemented using a novel virtual machine architecture that allowed code to be written once and executed, with little change, on a variety of platforms. So before C# and even before Java™, the Smallworld MAGIK™ programming language set the standard for enterprise GIS development. Today it continues to be the only true, pure object oriented language designed specifically to meet the unique set of challenges of enterprise GIS.

Life is a box of objects

Early GIS applications were based on procedural languages that implemented functionality as procedures and held data in separate structures. This demarcation of function and data might have been suitable for mass market applications such as word processing or spreadsheets but the Smallworld architects believed, fundamentally, this was the wrong approach for GIS applications.

Although the first GIS applications were expected to be relatively simple, they were also expected to become much more complex over time. In this context it would be important to be able to easily model an enterprise's complex assets and their convoluted interrelationships.

As applications became more complex, it became more important to shield the application developer from the internal implementation details using a well defined application programming interface (API). This approach would allow the underlying GIS to develop new technologies and functionality that might change the internal implementation, but, importantly, at the same time preserve the public API, reducing the effort and cost required to migrate code.

As applications became more powerful they required greater investment in their development and integration. To maximise the return on this investment, it was expected that code re-use would become a significant factor when attempting to reduce development costs, maintenance overhead and to achieve high quality applications.

Finally, it was expected that the GIS would eventually move out of mapping and engineering departments and adopt a more important role within the enterprise. In this context, such systems might be in operation for several years during which time new hardware platforms or operating systems might emerge and quickly become dominant in the marketplace. This meant that, ideally, development tools should be as platform independent as possible. This would safeguard any investment made already in existing applications.

An object oriented approach to an object oriented world

Using procedural languages to model complex systems is problematic to say the least. Increasing complexity tends to result in longer and longer lists of APIs with ever move cryptic names. New functionality demands often result in changes to an API that are often too subtle for most application developers to detect. In other cases system architects might generate a new API that although similar to an existing API is both named slightly differently and works slightly differently, making the application developers' choice all the more difficult.

Object orientated languages model real world objects as objects using classes. For example an electric

switch would be represented by a switch class. Each class might have one or more fields that can be used to hold attributes describing an instance of a class. For example, a particular switch might have a type, a manufacturer and a date when it was installed and so on. In addition to data, each class can implement the functionality that mimics the behavior of the real world object it is modeling. This functionality is exposed as a set of methods that forms the public API of the class. For example, a method might allow the switch to be turned off or on, or the name of its manufacturer to be accessed.

This approach is appealing to application developers. Complex systems implemented using procedural languages often result in a very large global API that quickly becomes difficult to use and unwieldy to manage. By splitting up this complex API and distributing it over several appropriate classes, it becomes much easier to manage and use. For example, application developers looking for switch functionality need only look at the switch class API and not be distracted by the APIs of other parts of the system.

Object oriented languages also allow application developers to easily model the real world relationships and dependencies. For example, a public electric utility might have several types of switches that share some common functionality. Using a procedural language, these would probably end up being modeled independently with little common code reused. A key feature of object oriented languages is inheritance. This powerful concept allows a child class to take on the characteristics of a parent class and supplement this definition with additional specific qualities. In the previous case, for example, the ability to turn a switch on or off might be a common piece of functionality that would be implemented on the parent switch class. This allows it to be reused by other switch classes that inherit from it. More specialized functionality that is peculiar to a particular type of switch would only be implemented on that specific class.

This ability to distribute a potentially complex API over several classes and reuse code through inheritance is a powerful advantage of object oriented language when compared to their procedural ancestors. Object oriented techniques offer dramatic improvements in development productivity, reductions in maintenance costs and higher quality applications.

A safe haven from technology

The Smallworld platform, from GE Energy, does not claim to have been first to introduce the idea of the virtual machine but the Smallworld platform was in fact the first to realize the practical advantages of the technology in enterprise GIS.

Most procedural based languages like C compile directly to what is called machine code. These are the simple instructions that are understood by a computer's processor and in turn execute functions implemented by the computer's operating system. Most consumer computers' operating system are based on the Windows© operating system and their processors are a derivative of Intel's family of x86 based processors. For consumer applications this is a perfectly acceptable approach, but for the enterprise it often represents a dependency on a single technology, which leaves some businesses decidedly uncomfortable.

The Smallworld MAGIK application code is not compiled directly into machine code, but instead is converted into an intermediate virtual machine code represented by what is called byte code. This code is independent of the underlying hardware platform or operating system. For each supported platform a small virtual machine is provided that interprets these byte codes and executes the appropriate machine code or operating system function when required.

Since the Smallworld MAGIK application's virtual machine is small, comparatively simple in operation and generic in construction, it is relatively straightforward process to provide new Magik virtual machines for new hardware platforms or operating

systems. Since the vast majority of the GIS application code is written in the Smallworld MAGIK programming language, most of the investment in an application can be safeguarded during migration to a new hardware platform or operating system when it emerges or at a point in time that is convenient to the enterprise.

This approach has allowed the Smallworld architecture to remain ahead of technology developments by providing support for early workstations from vendors such as SUN©, DEC©, HP© and IBM© (when GIS was very much still within the remit of the engineering department). It continues now with support for mass market PCs based on Intel processors and the Windows operating system and even to the recent emergence of open source platforms such as LINUX™.

Compile and go

Most procedural languages such as C and, to a lesser extent, even recent object oriented languages such as Java and C#, essentially still create applications using a basic three step process initially developed over 30 years ago. Modified code is first compiled, and then linked to one or more libraries of functionality after which the application is re-built into a form that can be used. This tends to be such a time consuming, cumbersome process that companies have been created offering a wide range of products and services to help manage the process.

The Smallworld MAGIK application's virtual machine supports on-the-fly incremental compilation negating the need for the link step and relegating the build step to the point at which the application is ready to be rolled out. This novel approach allows code to be modified even while the application is still running, yielding faster development and quicker testing and debugging. Members of a large development team can work with less dependency on each other resulting in improved productivity.

Another advantage specific to the enterprise environment is that the Smallworld MAGIK application also allows emergency patches to be applied often without the need to shutdown mission critical applications–particularly comforting in a 24x7 environment.

A brief comparison of peers

The Smallworld MAGIK language obviously is now not the only object oriented language available today (other vendors have finally realized what a good idea it was) and, as this paper has alluded to, other object orientated languages have similar features to Smallworld MAGIK. It is still worth identifying what the major differences are from a technical point of view.

- Smallworld MAGIK, Java and C# all compile code to intermediate byte codes that are executed by a virtual machine. Magik and Java are the only ones that are truly cross platform.
- Smallworld MAGIK, Java and C# have virtual machines that implement a background garbage collector that efficiently cleans up unwanted object instances automatically.
- Only Smallworld MAGIK supports multiple inheritance. Java and C# support single inheritance and provide interfaces to allow code to simulate multiple inheritance (though this approach often leads to code duplication).
- Java and C# are typed languages, Smallworld MAGIK is not (it is what is called polymorphic). There is much debate on this point: technically untyped object oriented languages are often regarded as more pure, whereas the use of types often yields more formal APIs.
- Only Smallworld MAGIK supports on-the-fly incremental compilation. Both Java and C# require some sort of compile-link-build.
- Only Smallworld MAGIK was designed from the beginning to support large, powerful and complex enterprise GIS applications.

A very brief tour of Smallworld MAGIK

The following sections give a very brief taste of the Smallworld MAGIK programming language.

Defining classes

Classes in Smallworld MAGIK parlance are called exemplar and are defined using the statement *def_slotted_exemplar()*. In this Smallworld MAGIK code fragment an exemplar called *person* has two slots (or fields) called *name* and *age* (these are initialized to " " and 0.0 respectfully). A second exemplar called *employee* is defined that inherits from *person*. *employee* has one additional slot called *salary* that is initially set to *0.0*.

```
def_slotted_exemplar(:person,
{
        {:name,""},
        {:age,0.0}
})
def_slotted_exemplar(:employee,
{
        {:salary,0.0},
},{:person})
```

Creating instances

Instances of Smallworld MAGIK exemplar are created by using the *_clone* statement and then correctly initializing the slots as appropriate by defining both a *new ()* and an *init ()* methods:

```
_method person.new(name,age)
        _return _clone.init(name,age)
_endmethod
_method person.init(name,age)
        .name<<name
        .age<<age
        _return _self
_endmethod
_method employee.new(name,age,salary)
        _return _clone.init(name,age,salary)
_endmethod
_method employee.init(name,age,salary)
        _super.init(name,age)
        .salary<<salary
        _return _self
_endmethod
```

A few hints: << means assignment (equivalent to using = in Java); the dot in front of each slot name allows access to the data in the slot; *_self* is a special statement that returns a reference to the instance of the object itself and *_super* is another special statement referring to the parent class (or super class).

This allows instances to be created from the Smallworld MAGIK command line:

```
e<<employee.new("David Brent",30,45000)
```

Implementing methods

Additional methods are implemented in a similar way as in this example which builds up the calculation of an employee's bonus based purely on age:

```
_private _method employee.age_as_percentage
        _return .age/100
_endmethod
_method employee.bonus
        _return .salary*_self.age_as_percentage
_endmethod
```

In this example the *_private* statement declares the method *age_as_percentage* to not be part of the public API of the *employee* exemplar thereby allowing its implementation to change if required whilst at the same time preserving the public API of the *employee* exemplar.

Conclusion

The early adoption of object oriented technology has endowed the Smallworld platform with an unmatched depth of experience and understanding of its benefits in the enterprise GIS environment. The Smallworld MAGIK programming language from GE Energy remains today the only true pure object oriented language designed from the outset to support the powerful, complex and mission critical GIS

applications demanded by today's enterprises. Key advantages such as rapid application deployment, cross platform support, code reuse and accelerated application development are all characteristics of a mature object oriented programming language called Smallworld MAGIK that was pioneered in the late 1980s.

Java is a trademark of Sun Microsystems, Inc.