Scripting in GIS Applications: Experimental Standards-based Framework for Perl

Alexandre Sorokine
Regional Science Institute
4-13 Kita-24 Nishi-2 Kita-ku
Sapporo 001-0024 Japan
Phone +81-11-717-6660
Fax +81-11-757-3610
sorokin@vtt.co.jp

Kurt Ackermann
Regional Science Institute
4-13 Kita-24 Nishi-2 Kita-ku
Sapporo 001-0024 Japan
Phone +81-11-717-6660
Fax +81-11-757-3610
kurt@vtt.co.jp

Research Session

Abstract

Scripting languages have long been utilized by GIS application developers to achieve higher levels of programming and shorter development times. Modern general-purpose scripting languages – like Tcl/Tk, Perl or Python – allow the smooth integration of various software components, while at the same time providing rich programming capabilities. Increases in processor speed and the development of industry-wide standards are removing obstacles to the proliferation of universal scripting languages in GIS applications.

This paper examines the feasibility, architecture and early results of the building of a framework of object-oriented modules for Perl that is designed to provide uniform access to various sources of spatial data from Perl scripts. The framework is independent of any particular GIS application and thus based on the object model detailed in the Abstract Specification by The OpenGIS Consortium. The framework is intended to be used for rapid application development, gluing various software components together and prototyping.

1. Introduction

Scripting languages have long since become commonplace in programming practice. Today most commercial software packages feature some kind of scripting capability and several general-purpose scripting languages – such as Perl, Python, Tcl/Tk and Visual Basic – are available for various platforms. Scripting languages are usually viewed as complementary to system programming languages like C, C++ or Java.

According to [9] scripting and system programming languages are fundamentally different. The main role of system programming languages is building data structures from scratch and defining operations with them on the level of memory words. The objective of scripting languages is intrinsically different and they are more intended for connecting various components together, assuming that extensive frameworks of such components already exist. The transition towards scripting languages can be viewed as akin to the transition from assembly languages to system programming languages, which started in the late 1950s. Indeed, scripting languages differ from system programming languages with respect to the same characteristics (instructions per statement and typing of variables) as system programming languages differ from assembly languages. In terms of the number of machine-code instructions per language statement, scripting represents a higher level of programming by bringing the number of instructions to the 100-1000 range compared to system programming, which, in turn, had increased the number of instructions per statement to 10 compared to assembly languages. In terms of typing, scripting languages have much weaker typing than system programming languages, thus facilitating the gluing of various software components. In this sense they are closer to assembly languages, which also have weak typing [9].

One of the major factors behind the observed proliferation of scripting languages is their ability to both increase the speed and reduce the costs of application development. Both advantages come at the expense of application speed, however this is well compensated for by faster processors. In addition to the vast gluing capabilities characteristic of them, scripting languages allow the creation of hybrid applications where computationally intensive elements are im-

plemented in such languages such as C or C++ and the remaining infrastructure is scripted.

This article suggests a new approach to how scripting may be used in GIS applications and describes a programming framework developed by the authors to illustrate this approach. This introductory section is followed by section 2, which depicts the authors' perception of the problems involved in building a generalized GIS interface for a general-purpose scripting language. Section 3 presents the proposed solution, a short overview of the existing technologies and popular general-purpose scripting languages and explains why certain design decisions were made. Section 4 expands on the framework architecture and important implementation and development process details. Characteristics and limitations of the resulting system and prospects for future development are discussed in section 5, followed by conclusions in section 6.

2. Problems

All major GIS software packages have long provided some kind of a scripting (or macro) language and those languages are widely used by GIS developers for both in-house and deliverable applications. For example, ARC/INFO AML was used to develop an extensive suite of user-friendly add-ons for the main system called ArcTools. ArcView Avenue is the foundation for a vast array of ArcView extensions developed by independent companies and individuals. MapBasic fulfills a similar role for MapInfo. A free software package GRASS uses a different approach, relying on integration with external scripting languages such as UNIX shell scripts and Tcl/Tk.

Despite being vital in many situations, application-specific macro languages usually lack features that would allow simple and effective integration with other applications and components. Typically, tools for integration with external applications are limited to the ability to load dynamic libraries and call their functions, issue a shell command or access an external database server. Capabilities for operations with spatial objects are naturally limited by the data model of a specific application. Rather often, application-specific macro languages lag behind in terms of syntactic richness, flexibility and expressiveness compared to mature languages.

On the other hand, general-purpose scripting languages like Perl or Visual Basic have evolved into sophisticated frameworks for integrating applications and components from various application domains. Interfacing a software package or component to a general-purpose scripting language benefits both of them:

- software application is augmented with a mature macro language
- capabilities of the application become available to the programmers in the target scripting language and the application may be viewed as an extension or module of that language
- through the scripting language it becomes possible to interface the software package to the vast library of modules or extensions of that scripting language and vice versa
- the dependency of a scripting programmer on a single software vendor is alleviated.

Among the most popular extensions for scripting languages are interfaces to database management systems (DBMS). For example, in the case of Perl, specific interfaces for many individual database servers had been developed independently. Now, however, most of the developer community's efforts are concentrated on the Perl DBI (Database independent interface for Perl), which enables a standardized approach towards accessing relational data in a manner similar to that of JDBC or ODBC. This type of database interface uses a standardized application programmers' interface (API) that is common through a series of drivers conformant to it. Drivers perform actual access to the database servers and typically are loadable dynamically at run time. APIs have become standardized to the point where no change in the calling script is needed to switch one underlying database into another, that is performed by loading another set of drivers.

Other common types of extension are web-related tools such as HTML or XML generators, parsers, interfaces to HTTP and mail servers. Perl is rightfully called a language of the World Wide Web. In our opinion, Perl success on the WWW can be attributed not only to its rich text processing capabilities, but to a great extent to its ability to easily integrate databases with web applications, owing to the availability of the above mentioned extensions. A substantial number of web applications are just frontends for database servers programmed in Perl (or another scripting language). Typically, such scripts translate requests to the web servers into database queries and then reformat the query result into HTML documents while performing a number of special tasks like generating raster images on the fly or user authentication.

There are several particularly notable trends in the area of GIS application software. Most major GIS software vendors are moving in the direction of enterprise-level GIS software infrastructure. This involves shifting from classical desktop GIS applications to a multi-tier model where spatial databases reside on servers and are shared by a variety of clients. Major products in this group are the Spatial Database Engine from ESRI Inc., Oracle Spatial from Oracle and spatial data handling extensions for the server products of many database server vendors.

Another trend is the proliferation of web-based GIS software. There is a plethora of commercial and other products available that are very well covered in the literature (for example, see [4]). Even though there are still very few web GIS applications of practical applicability, the amount of interest shown and investment being made by the industry inspires optimism. Hopefully with a new generation of web standards, such as Geography Markup Language [8] and OpenGIS Web Map Server Interface Specification [2], web GIS will be able to forge its way into the mainstream.

In light of this, development of a framework for a scripting language similar to the Perl DBI or JDBC but geared towards the handling of spatial data would be highly desirable. This paper examines the feasibility, architecture and early results of the building of a framework of object-oriented modules for Perl that is designed to provide uniform access to various sources of spatial data from Perl scripts. In the remainder of this article said framework will be referred to as GISI.pm, which stands for "Geographic Information Systems Interface" and .pm is a file name extension that is used to designate a Perl module.

In order for such a framework to achieve a comparably high level of capabilities for GIS applications as the Perl DBI or JDBC provide for database applications, access to the elements of spatial data structures must be provided. The fundamental problem for access to spatial data is a data model. There are several aspects to this problem.

First, frameworks like JDBC or the Perl DBI are built around relational databases that are already pretty well standardized and whose data model is very well studied and widely accepted. There is no such common data model in GIS, nor can it be built as a product of summing up the data models used in various GIS packages. Second, due to the sophistication of spatial data models there is no direct mapping of their elements into scripting language data structures as in the case of the relational data model. For example, in the spatial data model we have to deal with such notions as geometry or topology that do not have counterparts in standard programming language constructs. At the same time, such notions from relational databases as rows or data types map easily. Third, the same phenomena in GIS can be represented using absolutely different data models depending upon the requirements or analytical procedures sought (*e.g.* raster representations and vector models). More often than not these representations could be mapped into each other only with significant data loss.

In order to address the first aspect of the problem we have chosen the OpenGIS Abstract Specification [7, hereafter referred to as "the Abstract specification"] to be used as a common spatial data model. This specification is being developed from the very beginning to be a glue between data models of as wide a variety of GIS applications as possible, something which meshes very well with the purpose of scripting languages. The specification is recognized and supported by the industry and hopefully this will ensure its topicality for a long time to come. Also many vendors are expected to develop or bring existing products to compliance with the OpenGIS specification. Commonalities between the data models of the compliant products and our framework will simplify driver creation. Unlike some other attempts to suggest a common spatial data model, the OpenGIS Specification demonstrates healthy progress and continuous updating. Not the least reason for choosing the specification is its breadth, something which would be unrealistic for a limited group of developers to achieve.

To address the second aspect, we are developing not only the API itself but a hierarchy of abstract classes to represent the spatial data model in our framework. It has been decided not to address the issue of representing the same phenomena in several data models, at least at the current stage of development, but to shift it to GISI.pm extensions instead.

3. Proposed solution

Several powerful and popular scripting languages may be considered for the outlined development. We think that the following three require specific mention: Perl, Python and Tcl/Tk. These are all free, mature, general-purpose, have large numbers of extensions and have been used for some GIS applications. In a practical sense, scripting languages may be characterized by such features as simple syntax, rapid development and implementation as compared with system programming languages, good GUI capability, cross-platform portability, extensibility, and the ability to be used either for standalone applications or to be embedded.

Scripting languages have often been categorized as having particular strengths, but some may not be widely applied outside of these areas. For example, Visual Basic is easy to learn, but is restricted to Windows environments. JavaScript's smooth integration with Web browsers is well known, but it is more or less limited to that environment. Of the more general-purpose languages Tcl is known for its GUI capabilities, Perl for its string-handling and Python for its object-orientation. Most can be embedded in applications to varying degrees or used standalone, and may be connected to other applications or protocols.

3.1. Tcl/Tk

Tcl (Tool Command Language), in combination with its graphical user interface toolkit (Tk), is probably the most popular scripting language. In addition to its proven GUI capabilities, Tcl/Tk displays considerable portability as its scripts can run on Unix, Windows and Macintosh. Further, its versatility can be attested to by the diversity of Tcl applications.

The fact that the Tk toolkit was written in Tcl is cited as proof of Tcl's greater GUI capabilities. Although both Python and Perl have packages to allow the use of Tk from their scripts, it will naturally work better with Tcl than with other scripting languages. Existence of the Tcl interpreter as a C library package argues greater embeddability than Perl, which was not designed to be embeddable.

Tcl/Tk has a history of being used for GIS applications. As was mentioned earlier, it was used to developed a GUI for the free GIS GRASS.

3.2. Python

Python shares the same common set of features with other general-purpose scripting languages, but differs from most in its fundamental object-orientation and because the core language supports creation of classes, use of multiple inheritance, definition of methods, operators overload and throw and catch exceptions. Its development has included influences from Modula, Lisp and Smalltalk. Python's high-level built-in data structures, dynamic typing and dynamic binding are features which give it considerable popularity for use as a "glue" language.

Another of Python's characteristics is its systematicity. That Python's clean, consistent syntax may not be as rich as that of other scripting languages is touted as one of its strengths, since more casual users will be less likely to be overwhelmed as they might be with a richer, and therefore more burdensome, syntax. The greater regularity and lack of shortcuts to be remembered are also illustrative of higher systematicity. Such characteristics equally result in a high degree of maintainability and facilitate the task of reading code, an activity which Python emphasizes over writing. There is a version of Python called JPython, which runs on top of the Java virtual machine, that is written in 100% Pure Java. Its features include seamless scripting for Java and the ability to use arbitrary Java classes.

A package called AVPython (`http://www.geocities.com/brucedodson.rm/avpython.htm`) has been developed to enable Python language support for ArcView GIS. It consists simply of a dynamic link library and an ArcView extension. The developer rationalized his decision to use Python as due to strong COM support and commonalities – syntax and readability – between Python and ArcView Avenue, while acknowledging that implementing Perl would have been almost as simple.

3.3. Perl

Sometimes the acronym Perl is said to stand for "Pattern Extraction and Reporting Language" and this is how Perl is mostly used and perceived by the programming community. Indeed, superiority of Perl in terms of text processing and pattern matching resulted in the reimplementation of its pattern matching capability in the majority of other mainstream programming languages. Perl can be characterized by almost any buzzword trait of a scripting language: it is easy to learn, fast to execute, and has been interfaced to an uncountable number of other systems and libraries.

Object-oriented features of Perl require special mention and explanation. Originally Perl was not object-oriented and the ability to aggregate data and methods in a single package has appeared only in version 5 in 1993. Even though Perl is not as well-known as Python for its object-oriented features, its object model is very powerful and the least restrictive. Also, it very naturally combines with a non-object-oriented style of programming. Unlike in most other languages, object-oriented features in Perl at large are not a part of the "core language" but rather built of elements that were already present in Perl for the purposes of structural programming. Objects in Perl are just references associated with methods from an arbitrary module. This is achieved using the Perl operator `bless`. Methods are distinguished from generic functions by receiving object references in the first position of the argument list [11].

Perl lacks many object-oriented notions such as abstract and static classes, protected and final methods or variables. All constructs that are really required in big object-oriented projects can be simply achieved by the development team following certain conventions. This corresponds very well to Perl philosophy. For example, private methods have to be implemented by just not mentioning them in the documentation.

This kind of loose object model has it pros and cons. In situations where the speed of development outweighs issues of code cleanliness, lack of restrictions may confer advantage. However, bigger and heavily object-oriented systems like GISI.pm have to be created very cautiously and strictly following the previously agreed conventions.

In GISI.pm development we have been using UML (Unified Modeling Language) charts to limn the system design. In our opinion UML is as good for Perl as for any other object-oriented language. However no UML mapping for Perl exists, so some minor adaptations are needed to be able to reflect Perl specifics. Later in this paper we use figures that adhere to the UML 1.1 specification [10].

Perl has a rather limited history of use in GIS, although it is sometimes used as an interface language for packages like the MapScript scripting interface for the MapServer – a "CGI-based application for delivering dynamic GIS and image processing content via the World-Wide Web" – at the University of Minnesota (`http://mapserver.gis.umn.edu/`).

3.4. Why Perl

Our decision to choose Perl was influenced by several factors. One of the major factors was team expertise with the language and the fact that a stash of unconnected Perl scripts had already been developed at our company for a variety of GIS data processing tasks.

The second important advantage of Perl is a plenitude of readily available language modules, which seems to be wider than exists for other languages. These modules spared us a noticeable amount of time and effort through not having to implement those functions that had already been created by other developers. Moreover, such a wide variety of modules would allow us to interface our framework to something that we had not already thought of. In addition, the Comprehensive Perl Archive Network (CPAN, `http://cpan.org/`) and the module of the same name allow automated download and update of Perl modules installed on a local workstation. Thanks to CPAN we need not fear complexity of the installation procedure of our modules by a user even if a lot of dependencies upon other modules are present.

We were first attracted by the Perl DBI that provides standardized access to many database servers and which has already been mentioned in this paper. From a development perspective we are relying on the following modules:

- Tie::IxHash – provides implementation of an ordered hash array; that is, a data structure whose elements may be accessed using either hash keys or array indices
- URI – a generic unified resource identifier (URI) that we are using to implement driver specifiers
- A group of modules that implements design patterns as described in [3]; helps developers to facilitate good object-oriented programming practices with Perl
- FreezeThaw – support for Perl serialized objects.

In terms of potential applications areas that our framework can be interfaced to, we have considered several modules:

- mod_perl – allows the writing of Apache web server modules in Perl
- CGI – simplifies writing cgi scripts in Perl
- a group of modules which provide access to and operation on raster images (Imagie::Magic and others)
- modules that facilitate integration into various distributed computer environments, such as CORBA and OLE/COM on the Windows platform; in future, these modules would provide a relatively simple path for interfacing GISI.pm to other applications conforming to the OpenGIS implementation specifications [5] and [6]
- a vast array of modules to read and generate XML and other markup languages
- user interface toolkits: PerlTk, Gtk-Perl, X11-Motif, PerlQt.

4. Architecture

GISI.pm is an object-oriented API (Application Programming Interface) for accessing spatial data from Perl scripts. GISI.pm is intended to define a set of classes and conventions to provide a consistent interface to access spatial data independent of underlying format or access method. GISI.pm is based on the OpenGIS Abstract specification Version 4 [7].

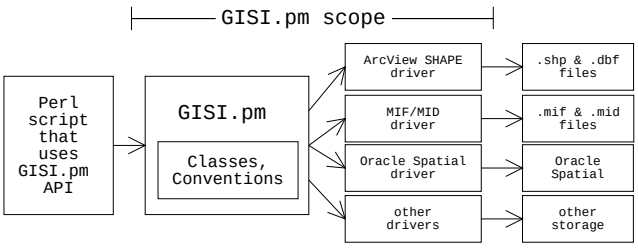

Figure 1. Architecture of a GISI.pm application

GISI.pm is an interface layer providing communication between Perl scripts utilizing the GISI.pm API and drivers that provide actual access to the sources of spatial data (Fig. 1). In the application architecture, which uses Perl as a glue between various pieces of the GIS software and libraries written in a variety of programming languages, the role of GISI.pm is that of a data access layer. All effort has been made to keep GISI.pm thin and limit its scope and purpose to data access. An additional requirement for GISI.pm is the ability to preserve the maximum information that comes with a dataset. In this regard our ultimate goal is to provide access to all information carried in spatial datasets (geometry, attributes, metadata, etc.) as defined in the Abstract specification.

4.1. Architecture overview

GISI is implemented as a set of Perl modules comprising the core GISI.pm and optional drivers. GISI.pm itself performs the following actions: loading of the specified driver(s), dispatching method calls to the drivers, and error checking and handling. All other actions should be implemented outside of GISI.pm. The drivers implement support for a given type of spatial data source, which can be a file in some format, a database server or almost anything else.

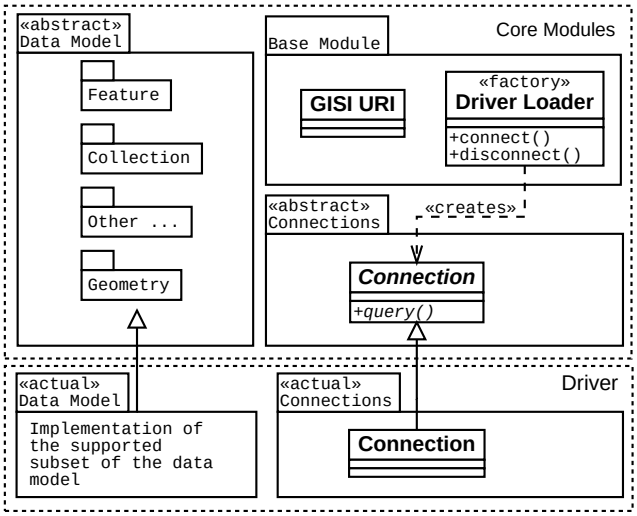

Figure 2. GISI.pm elements

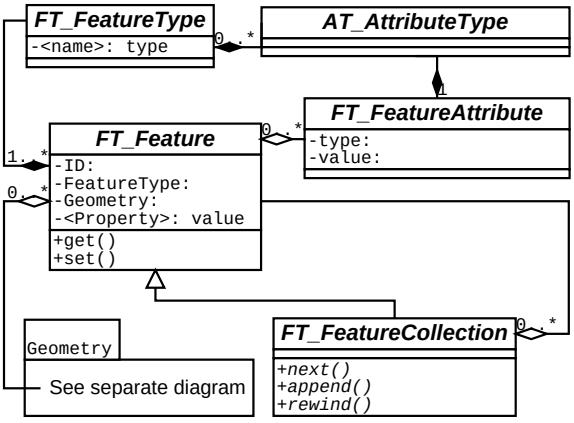

Figure 3. Implemented subset of the OpenGIS object model (simplified)

The basic elements of the core GISI.pm are the driver loader, feature collection identifier, an abstract class that represents a connection and a framework of abstract classes that represent the spatial data model (Fig. 2). Each driver consists of a set of Perl modules that implement a connection class and those classes of the spatial data model that are supported by the driver. There are also several utility classes in GISI.pm, including classes which perform functions common for many drivers, such as parsing, and various factories and iterators which support the spatial data model.

GISI.pm's operation is rather similar to that of the Perl DBI. The driver loader is a static class that implements two significant methods – `connect()` and `disconnect()`. `connect` receives an object or a string that specifies a data source and returns an object that implements a connection. Optionally, driver-specific parameters may be specified. The connection class has methods for retrieving or creating feature collections. The latter can be iterated through with the help of cursors or appended with new feature objects, if allowed.

4.2. Spatial Object Model

In terms of the object model of spatial data, as was mentioned earlier, GISI.pm relies on the OpenGIS Abstract Specication Version 4 [7]. Only a subset of this object model, which is described below, has been implemented (Fig. 3).

The basic unit in the OpenGIS object model is a *feature* (`FT_Feature` on Fig. 3). A feature is an abstraction of a real-world phenomenon and may be associated with a location relative to the Earth. Each feature is specified by its attribute set and belongs to some recognizable type (feature type, `FT_FeatureType`). Feature type defines a list of attributes which distinguish that feature. In such a list each attribute is characterized by an attribute name and an attribute type (`AT_AttributeType`). Features are organized into feature collections (`FT_FeatureCollection`). Features of a single feature collection have the same type (in the original specification types that are inherent of the feature collection feature type). One of the attributes can be of a special type "Geometry" that designates feature positioning relative to the Earth's surface.

Geometry (Fig. 4) objects represent sets of points in a particular coordinate system. The sets of points are represented by `DirectPosition` objects. All geometry objects are inherent of the `GM_Object` class. Basic subclasses of `GM_Object` are geometric primitives (`GM_Primitive`) and aggregates (`GM_Aggregate`). Primitive geometries include points, curves and surfaces (designated as `GM_Point`, `GM_Curve` and `GM_Surface`) that represent 0-, 1-, and 2-dimensional objects respectively. The specification also defines 3-dimensional objects (solids), which have not been implemented. Points (`GM_Point`) contain a sole positioning object that consists of a single coordinate tuple. A curve (`GM_Curve`) is an ordered set of curve segments (`GM_CurveSegment`) and a surface (`GM_Surface`) is an ordered set of surface patches (`GM_SurfacePatch`). Aggregates (`GM_Aggregate`) are geometry objects that consist of other geometry objects. Multiprimitives (`GM_MultiPrimitive`) contain geometry objects of a single subclass of the `GM_Primitive` class.

Positioning is represented by the descendants of the coordinate point class (`CoordinatePoint`), which is a tuple of coordinates in *n*-dimensional space. Class `Direct-Position` is intended to associate coordinates with a spa-

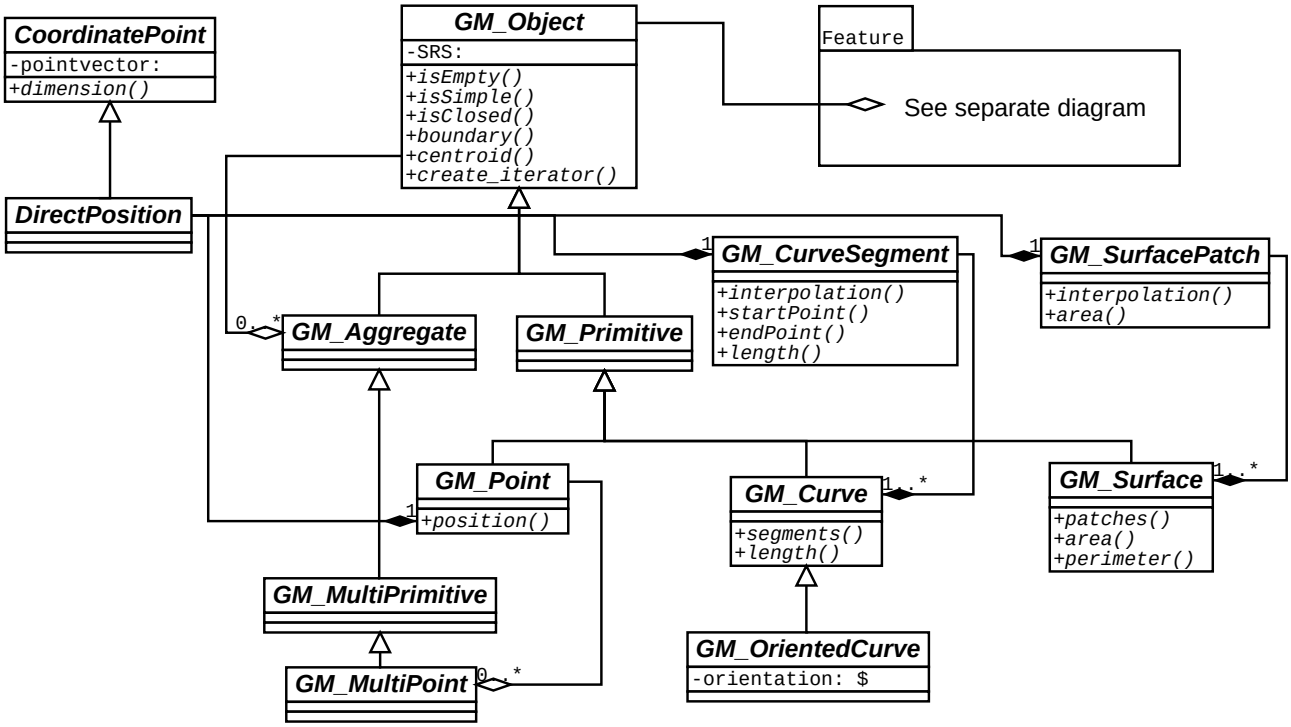

Figure 4. Implemented subset of geometry model (simplified)

tial reference system. Specific interpretation of the dimensions is done in the subclasses of `DirectPosition`.

4.3. Implementation Details

In order to straightline the `use` [1] statement in custom scripts and reduce load on the Perl interpreter, the GISI.pm API is grouped into several Perl modules. In most cases script programmers have to specify only a single `use GISI;` statement. Other non-driver modules may need to be specified if a programmer is going to instantiate his own features, geometry or positioning objects.

There are several utility class hierarchies that are included in GISI.pm. In addition to standard methods, access to the elements of geometry objects can be done using geometry iterators. There are two kinds of iterators: one to iterate through primitive geometries and another to iterate directly through positioning information. Geometry objects can be created with the help of geometry factories. In most cases there is no need to create special factory classes in Perl since passing a class name to a function automatically allows parameterization of object instantiation. However, a special factory class is needed for geometry objects to unify their creation directly from positioning information as creation of curves and surfaces involves the additional step of creating curve segments and surface patches. Feature collection can be traversed using one of the cursor classes. There are several types of cursors providing various additional features like caching, filtering and others. In all cases the use of iterators, cursors and factories is a preferred method for accessing the elements of or creating a GISI.pm data structure. Also there is a special hierarchy of classes that represent attribute types.

4.4. Drivers

GISI.pm drivers must comply with the definitions as shown in Fig. 4. Due to the outstanding gluing capabilities of Perl, drivers can be written either in Perl itself or a mixture of Perl and other languages such as C. For example, the driver for ArcView SHAPE files that comes with GISI.pm is built on top of a freely available library written in C (ArcView Shapelib by Frank Warmerdam, available from `http://gdal.velocet.ca/projects/shapelib/`). Typically, writing a driver in C involves a two-layer implementation with one layer interfacing a library to Perl and the other layer interfacing raw Perl data structures into GISI.pm objects.

Interfacing Perl to C is easy, owing to SWIG. The interface compiler SWIG (Simplified Wrapper and In-

[1] `use Module;` in Perl has the same meaning as `#include <module.h>` in C

terface Generator, `http://www.swig.org/`) is commonly used to facilitate the connecting (gluing) of programs written in system programming languages, like C, C++ and Objective-C, with scripting languages, particularly Perl, Tcl/Tk and Python. SWIG uses the declarations containing function and method prototypes commonly found in C/C++ header files to generate the glue code (wrappers) needed by scripting languages for accessing the underlying C/C++ code. One more advantage of SWIG is independence of Perl implementation; therefore, with the next major upgrade to Perl, when the Perl core is expected to be rewritten in C++, and after SWIG is updated to that release, only recompilation will be needed.

Currently (May 2000), the following drivers of general interest have been implemented: ArcView SHAPE, several ARC/INFO `generate` command formats, MapInfo MIF/MID and CSV (comma separated value) and DBASE formats for tabular data.

4.5. Development strategy

From the very beginning GISI.pm was geared towards practical needs such as application prototyping, file format conversion and related tasks. Because of this condition, the development team could not engage in pursuing the goal of implementing all desired (and even essential) features at once. The problem is exacerbated by a certain level of interdependencies of the blocks of the system. To address this problem we are following a development path similar to the "incremental-build model" described in [1, Chapter 19]. The idea of this approach is to firstly develop a system skeleton, then create stubs instead of system components and gradually replace them with actual working code – this way there is always something operational. Owing to Perl's weak typing and dynamic nature, creation of stubs is relatively easy and was attempted in the case of the feature identifier module. There are several relevant parts of the Abstract specification that have not yet been implemented, or which have only been implemented partially.

As for the Geometry package, the current implementation lacks any topology classes. In the specification topology is presented as a parallel hierarchy of [0..3]-dimensional objects. Metadata and spatial reference systems are not implemented either, both of which are of considerable practical value. The biggest missing element is the coverage. In the Abstract specification a coverage is defined as a "2-dimensional metaphor for phenomena found on or near a portion of the Earth's surface". Gridded coverages are of particular interest as they represent such commonly used categories of spatial data as digital elevation models, images and physical fields.

Identifiers are only partially implemented. Currently, a feature identifier is expected to be a scalar value or a reference. Feature collections have driver URIs as their identifiers. There is also a weak requirement for an identifier to be unique within the scope of a feature collection. What to assign to the feature identifier is at the discretion of the driver. If a data source has a notion of feature identifiers, then its value is assigned to the GISI.pm feature identifier. In other cases an attribute value or feature sequential number can be used as a feature identifier, which should be specified upon driver initialization.

Similar to Perl itself, our development strategy features a strong test suit. Elaborate testing is of special importance for Perl scripts since, due to the dynamic nature of Perl, most errors expose themselves only during run-time. There is a set of tests for each method of each module to ensure that each component works properly.

To ensure proper interaction between components we are planning to use reference datasets and to check if they are properly converted to and from various formats. The datasets should be representative of all types of information and all situations supported by GISI.pm. For this purpose we have developed a relatively simple script (gisconvert.pl) that is at the same time a general-purpose GIS format converter. The intention of the script is to be able to convert properly from one format to another with minimal format-specific information provided.

5. Results, Limitations, Future Development

At the current stage of development (May 2000) we have a working system that has shown itself useful in the following situations:

1. Conversion of rarely used and unique formats: it is a very common situation that data exists in some format that is either outdated, supported by none of the available packages or support of that format is broken; with the help of GISI.pm writing only one half of the converter would be enough.

2. Correction of improperly formatted files: this often happens even to commercial converters when a file produced by one software package in the format of another package is not accepted by the target package.

3. Conversion of commonly used file formats: even though there are quite a number of commercial and free utilities for this task, the advantages of having GISI.pm are that it is cross-platform, free, fast and capable of handling non-standard situations.

4. Exploration of and collecting statistics on the datasets without converting them into any particular package.

5. Custom geometric transformations and projection conversion without using commercial packages (with the help of PROJ.4 originally developed by USGS and currently available from `http://www.remotesensing.org/proj/`).

As was mentioned earlier, benefits of scripting languages come at the cost of slower application speed. However, scripts written with the help of GISI.pm are not as slow as was originally expected. For example, in converting common file formats, GISI.pm is faster than some commercial products. Profiling results show that most of the time is usually spent in parser methods. Parser implementation that we are using is far from being effective, so we think that enhancement is possible.

There are serious shortcomings in the existing version of GISI.pm. First of all, lack of the implemented blocks such as metadata or spatial reference systems support mentioned earlier does not allow the building of practically usable drivers for many GIS formats. The second problem is insufficient error handling that probably should be modeled after the Perl DBI. In some cases lack of support for concurrent access to the dataset impedes usability.

There is no doubt that GISI.pm is currently in its alpha-version stage. Transition to the beta stage will be achieved when the following elements are implemented (in no particular order):

1. Topology support
2. Spatial reference systems
3. Metadata
4. Feature identifiers
5. Gridded coverages
6. Internationalization and Unicode

The obvious direction for future development is to support more file formats and data sources. The development team has experience with interfacing GISI.pm predecessors to relational databases with spatial data extensions. It would also be interesting and relatively easy to develop drivers for non-spatial vector formats like, for example, SVG [12].

6. Conclusions

The first conclusion is that the approach used in GISI.pm works and is effective. It is far more effective than writing haphazard scripts to fulfill the same circle of tasks. GISI.pm has already saved our company a noticeable amount of working hours by encouraging code reuse thus reducing spending on GIS related programming.

Based on our experience we can suggest at least one enhancement for the Abstract specification, related to the possibility that feature collection may be a redundant class. In the current version of the specification, feature collection is a subtype of feature that is used to aggregate other features. In the GISI.pm framework a feature collection most closely corresponds to a statement handle or result set. However, in this case it should not be a subclass of feature even though it may possess some geometric characteristics such as a bounding box. More confusion will be generated if the notion of feature collections is applied to coverages, especially grid coverage features.

Our suggestion is to use just ordinary features in place of feature collections. Traversing of feature collections can be performed with greater flexibility by defining various iterators (cursors) on the level of implementation specifications.

We especially want to mention that the geometry specification has made remarkable progress, and today is much more flexible and mature compared to simple features.

Concerning Perl object-oriented features, we would suggest including in either the core language or developing modules the capability to perform a variety of object-oriented checks such as for function arguments, private variables, final methods and others. All these capabilities should be optional and switchable on demand. This will allow script and driver programmers to perform fewer manual checks on the code to ensure its object-oriented cleanliness, while preserving the ability to write code fast when there is a need.

In general we think that scripting in GIS has a bright future for componentized, glue-oriented applications and prototyping. Frameworks similar to GISI.pm, if developed for a variety of scripting languages, can be used to glue together various applications and components: data access and storage systems, simulation models, computationally intensive algorithms, web and user interface applications.

References

[1] BROOKS JR, F. P. *The Mythical Man-Month: Essays on Software Engineering*. Addison-Wesley, 1995.

[2] DOYLE, A. *Web Map Server Interface Specification*, 1.0.0 ed. Open GIS Consortium, 35 Main Street, Suite 5, Wayland, MA 01778 USA, April 2000. `http://www.opengis.org/techno/specs/00-028.pdf`.

[3] GAMMA, E., HELM, R., JOHNSON, R., AND VLISSIDES, J. *Design Patterns: elements of reusable object-oriented software*. Addison-Wesley, 1994.

[4] LIMP, F. W. Mapping hits warp speed on the world wide web! *Geoworld 12*, 9 (September 1999), 32–42.

[5] OPEN GIS CONSORTIUM. *OpenGIS Simple Features Specification For CORBA*, 1.0 ed. 35 Main Street, Suite 5, Wayland, MA 01778 USA, March 1998. `http://www.opengis.org/public/sfr1/sfcorba_rev_1_0.pdf`.

[6] OPEN GIS CONSORTIUM. *OpenGIS Simple Features Specification For OLE/COM*, 1.1 ed. 35 Main Street, Suite 5, Wayland, MA 01778 USA, May 1999. `http://www.opengis.org/techno/specs/99-050.pdf`.

[7] OPEN GIS CONSORTIUM. *The OpenGIS(tm) Abstract Specification*, 4 ed. 35 Main Street, Suite 5, Wayland, MA 01778 USA, March 1999. `http://www.opengis.org/public/abstract/`.

[8] OPEN GIS CONSORTIUM. *Geography Markup Lamguage (GML)*, v1.0 ed. 35 Main Street, Suite 5, Wayland, MA 01778 USA, April 2000. `http://www.opengis.org/techno/specs/00-029.pdf`.

[9] OUSTERHOUT, J. K. Scripting: Higher level programming for the 21st century. *IEEE Computer 31*, 3 (March 1998), 23–30. `http://www.scriptics.com/people/john.ousterhout/scripting.html`.

[10] RATIONAL SOFTWARE AND OTHERS. *UML Notation Guide*, 1.1 ed., September 1997. `http://www.rational.com/uml/`.

[11] WALL, L., CHRISTIANSEN, T., AND SCHWARTZ, R. L. *Programming Perl*, 2nd ed. O'Reilly & Associates, Inc, September 1996.

[12] THE WORLD WIDE WEB CONSORTIUM. *Scalable Vector Graphics (SVG) 1.0 Specification, W3C Working Draft*. 545 Technology Square, Cambridge, MA 02139 USA, March 2000. `http://www.w3.org/TR/SVG/`.