Submitted to "Software – Practice and Experience"

Persistent Objects in Procol – Illustrated with a GIS case

Peter van Oosterom[†], Chris Laffra[‡] and Vincent Schenkelaars[‡]

[†] TNO Physics and Electronics Laboratory
P.O. Box 96864, The Hague, The Netherlands
Email: oosterom@fel.tno.nl

[‡] Software Engineering Research Center
P.O. Box 424, Utrecht, The Netherlands
Email: {laffra, schenkel}@serc.nl

Persistent objects are objects whose contents may outlive the execution time of the program. This paper describes the process of introducing persistent objects in the object-oriented programming language Procol. The strength of persistent objects in an object-oriented programming language is the integration of a database system with a programming language. Persistent objects make the program development easier, because the programmer does not have to implement the explicit loading and saving of data. Besides the general functional aspects, special attention is paid to graphic applications in order to deal with their specific geometric requirements. For example, it must be possible to find, in an efficient manner, all graphic objects that fall within a given region. These issues, persistent objects and their geometric requirements, have not yet got the attention they deserve in the literature covering object-oriented graphic systems where modeling and functional aspects dominate. The concepts are illustrated with a case from class of systems which may be considered as typical for non-traditional applications; a Geographic Information System.

Keywords: OO, persistence, GIS, language design.

Introduction

The fact that the object-oriented approach is so successful in computer graphics, is mainly due to the system modeling capabilities that the object-oriented paradigm offers. The specialization relationships which exist between the graphic objects in a system, can be modeled with inheritance or delegation which are present in many object-oriented development environments. Geometric data types, such as points, vectors and matrices may be implemented by abstract data types using object classes [9, 15, 16, 34]. However, in practice this modeling power is not enough when implementing CAD systems or Geographic Information Systems (GISs) which deal with large data sets. Integrated database facilities are required to support the graphic application in an efficient manner.

In this paper we describe a solution that is based on the introduction of *persistent* objects in the object-oriented programming language called *Procol*. A detailed functional description of the C–based language Procol can be found in [49] and some implementation aspects in [50]. The resulting system is also extendable with new (persistent) types and operators. In itself this approach is not new and has been described by several other authors [2, 40, 45], but we also tried to make the system suitable for highly *interactive and graphic* applications. This goal is achieved by putting emphasis on both time-efficiency (offering techniques such as: navigation, index structures, and parallelism) and the recognition of multi-dimensional data. As an example, not only one dimensional, but also multi-dimensional index structures are provided in Procol. The emphasis in this document is on the inclusion of persistent graphic objects in the syntax and semantics of Procol and on the implications thereof for the technical realization, e.g., how Procol deals with problems such as referential integrity and associative searching.

We will state the problem area in section "The Need for Persistence." The alternative of using a persistent programming language is described in section "DBMS based GIS Architectures." three different architectures are described. We define our general requirements for persistence in an object-oriented language in section "Our Wish List."

Our first attempt to introduce persistence in Procol and comparisons with several other systems are described in section "The First Attempt." Building on this experience the following topics are discussed in more detail: referential integrity, (multi-dimensional) search problems, Procol extension alternatives and object instances of different sizes in sections "Referential Integrity" through "Object Instances of Different Sizes," respectively. In section "Persistent Objects in Procol" we present the solution as chosen in Procol. In nearly all the sections the requirements of graphics play an important role. The discussions are illustrated with examples based on graphic systems. In section "The GIS case example," the use of persistent Procol is illustrated with a GIS case. Finally, section "Further Research" indicates some topics where further research is required.

The Need for Persistence

The computer science research in the area of programming languages emphasizes programming constructs and data structures. One of the most popular paradigms is that of Abstract Data Types (ADTs). Object-Oriented Programming Languages (OOPLs) encapsulate this paradigm in an elegant manner using object types to describe the ADTs. Access from outside to the data inside an object instance[1] is only possible through the methods or procedures defined for that object type.

[1] In this document we will use the term *object type* to indicate a class of objects and the term *object* to indicate one instance. In case more emphasis is needed we use the term *object instance* explicitly.

The data stored in data structures (or objects) of a running program are in general volatile, that is, as soon as the program stops, the data are lost. However, in many applications the data itself are very important. An obvious solution is to save the data in a file by explicit write statements. The next time the program is started it first reads the data from file into the volatile data structures. Persistent objects make the program development more efficient, because the programmer does not have to worry about the reading and writing of data from and to disk. Also, the structure of the data file may become quite complex, resulting in possibly intricate parsing. Moreover, for large quantities of data this "file" solution becomes cumbersome during the execution of the program. Consider as an example an information system that registers bank accounts. A characteristic of this and many other information systems is that the objects are well structured, quite passive and occur in large quantities. Passive objects are objects that hardly ever send messages to other objects (except for replies); they only react to messages from outside. In the bank account example, an account object replies its current amount when asked for, and updates it when told so by a message from an authorized object.

Database Management Systems (DBMS) have been developed to deal with the large amounts of data mentioned above. DBMSs concentrate on the information representation and tackle related problems such as, integrity, security, redundancy, consistency, efficient searching, query formulation and concurrency control. A major drawback is that the DBMSs of the current generation are not extendable with new data types and operators. This makes the use of these DBMSs inconvenient in non-traditional applications that need support for other data types.

The database research community has recognized this deficiency and is now trying to design systems that are more open [17, 44, 62]. At the other side the OOPL research community has recognized the need for persistent objects. Now, these two worlds meet. In some cases these encounters result in conflicts because of very different and incompatible principles. For example, explicit (navigation) links amongst instances are considered harmful by the database community, because they are hard to maintain. However, these same links form the backbone in representing complex objects in OOPLs. On the other hand, sometimes the combination of the two research communities results in a nice symbiosis. An example of this is that the concurrency control problem in the database is solved by the model of an object that accepts messages one by one (as in Procol).

We are interested in interactive graphic applications, such as: CAD systems, VLSI Design, and GISs, see Figure 1. Common aspects in these systems are: interactivity, graphics, and large amounts of data. In [56] we showed that the object-oriented approach offers a good data modeling and design environment for GIS applications. In the same paper the need for persistent objects in our object-oriented programming language Procol, was identified.

DBMS based GIS Architectures

GISs have a clear need for persistent data storage. An alternative for using a persistent programming language to implement a GIS is using a (normal or volatile) programming language together with a database management system. Most commercial GISs are based on a DBMS, which is in most cases a relational database, such as Oracle, or Ingres [46]. One obvious drawback of the standard DBMSs is that they can not manipulate geographic[2] data. That is, there are no geometric attribute types (point, polylines, polygons) and operators (distance, intersection, circumference, area). Therefore, it is impossible to store geographic data in a natural manner and to pose queries such as: "Select all municipalities with more

[2] The geographic entities or objects in a GIS are based on two different types of data: *spatial* and alphanumeric *thematic* data. In turn, spatial data have two components: *geometric* and *topological* data [53].

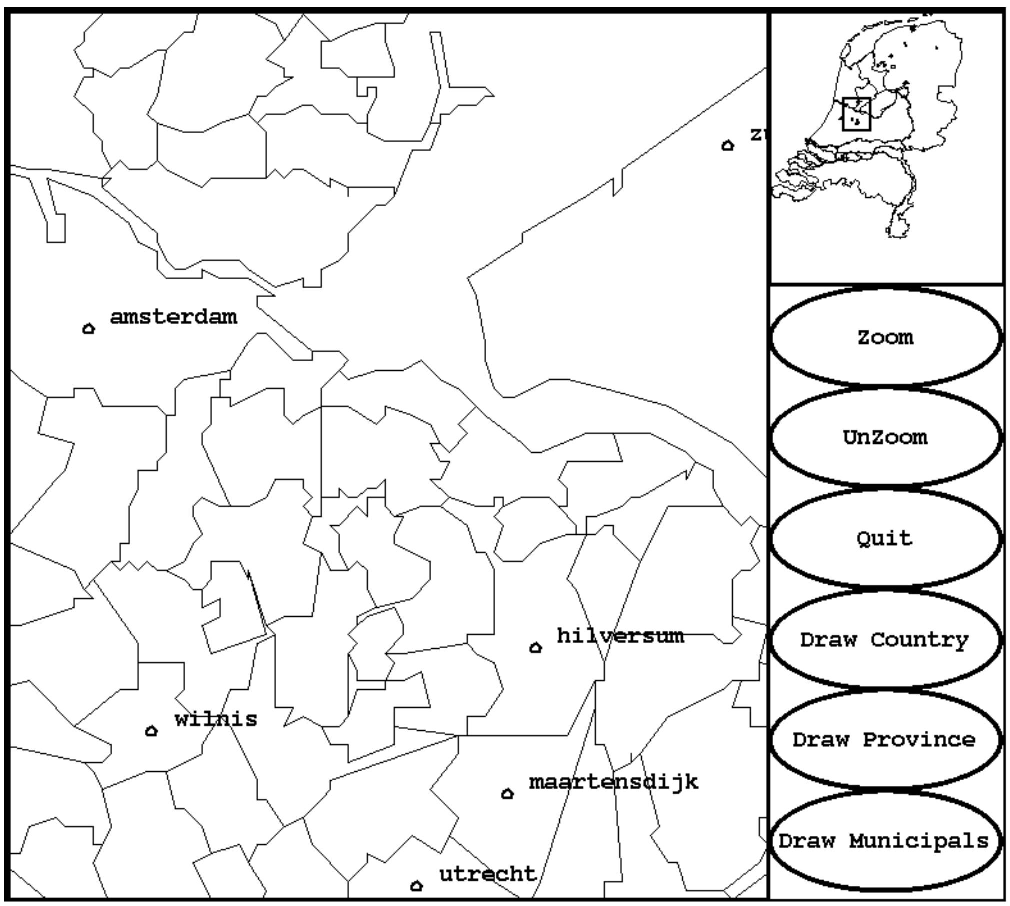

Figure 1: *A Geographic Information System*

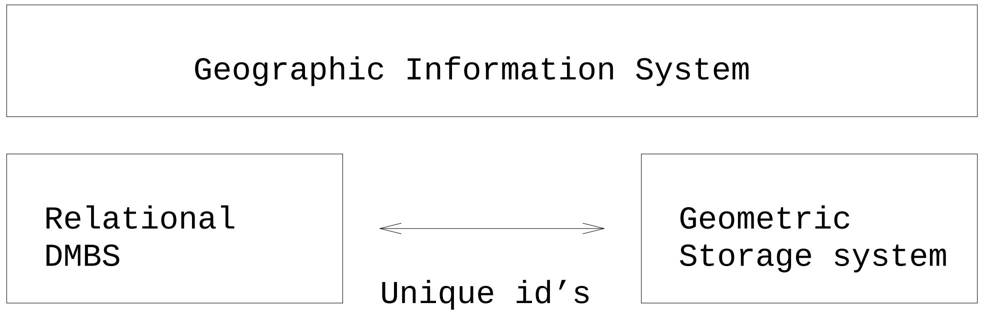

Figure 2: *The Dual GIS Architecture*

that 10,000 inhabitants and located within 3 kilometers from a lake." Further, these systems lack multi-dimensional access methods (or index mechanisms), which are obviously required as the size of geographic data sets is very large.

Different DBMS-based solutions have been suggested and implemented in order to overcome these problems. Three different types of system architectures can be distinguished: *dual architecture*, *layered architecture*, and *integrated architecture*. The term dual architecture was first introduced in [58][3]. In the next subsections, the pros and cons of these architectures are discussed. Note that it is not always easy to classify a specific system. For example *Smallworld GIS* [11] possesses characteristics of both the layered and the integrated architecture.

Dual Architecture

The most common type of commercial GIS architecture is the dual one. Dual architecture GISs have a separate subsystem for storing and retrieving spatial data, whilst thematic information is stored in a relational DBMS. This dual architecture is not conceptually elegant and also reduces the performance. An object that has both a thematic and a spatial component, has parts in both subsystems that are linked by a *common identifier*. In order to retrieve an object, the two subsystems have to be queried and the answer has to be composed. Figure 2 illustrates this dual GIS architecture. Typical examples of GISs with dual architecture are: ARC/INFO [19, 35] from ESRI, MGE [29] from Intergraph, SICAD [42, 43] from Siemens, and ARGIS 4GE [48] from Unisys.

The advantage of the dual architecture is that it is partly based on a standard DBMS and that the storage and retrieval of spatial data can be efficient. However, this method has some severe drawbacks. The existence of two storage subsystems implies that query optimization is not possible to the full extent. A relational DBMS offers transactions that are atomic, durable, and serializable. Storing data outside the relational DBMS can result in losing the transaction semantics, because the two storage managers each have their own locking protocol. The final drawback of the dual architecture is that integrity constraints can be violated. For example, an entity can still exist in the spatial storage subsystem while it has been deleted from the relational DBMS.

[3] A similar classification was described by Bennis et al. in [6, 7]. They used the terms *partial DBMS architecture*, *shell architecture*, and *full DBMS architecture* for respectively dual architecture, layered architecture, and integrated architecture.

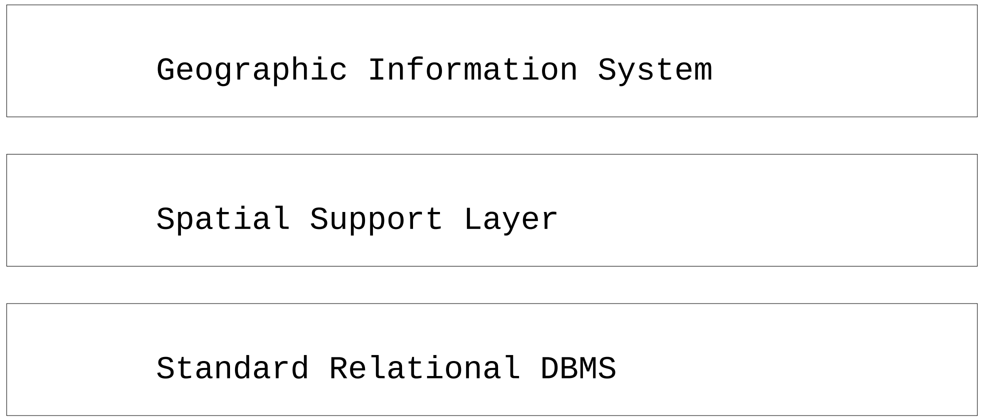

Figure 3: *The Layered GIS Architecture*

Layered Architecture

Most drawbacks of the dual architecture are caused by the fact that there are two storage managers each with their own responsibilities. It is possible to store the spatial data in the pure relational data model [57, 59]. This implies that the support for transaction semantics and integrity constraints is restored. However, in order to fit the data into the relational model, the coherent geographic entities have to be broken into multiple parts, which are stored in separate tables. Retrieving the original geographic entities has to be done by joins of relations, making the system slower and more difficult to use.

The user may be freed from formulating difficult queries by some "intelligent" translations in the layer on top of the standard relational database. This layer translates geographic queries (in "GeoSQL"; Geographic Structured Query Language) into standard SQL-queries and it may also implement spatial indexes. These indexes are implemented by means of auxiliary relations that contain the required index data. This makes spatial access faster, but the queries become even more complicated, because they have to use the auxiliary relations. This indirect implementation of an access method is less efficient than a direct implementation in the DBMS kernel. The layered GIS architecture is depicted in figure 3. System 9 [38] from Prime, GEOVIEW [60] from the University of Edinburgh, and SIRO-DBMS [1] from CSIRO Australia are characteristic examples of layered architecture systems.

Integrated Architecture

The inconvenient/inefficient mapping from the user data model to the relational tables can be avoided if more attribute types and access methods are added to the system. This solution is chosen in the integrated GIS architecture. In contrast to the other two types of architectures, this architecture cannot be based on a standard relational DBMS. It requires an extensible (and often object-oriented) DBMS. This is illustrated in Figure 4, where the spatial extension is completely embedded in the DBMS. Users may extend the DBMS with their own basic abstract data types (ADTs). An obvious drawback of this approach is that users have to implement their own types within the DBMS environment, which may be quite complicated. However, once this task has been performed, the advantages of this approach become clear. The implementation of the data model is easier, because the appropriate geometric types are available now. The formulation of spatial queries is directly supported in the extensible query language by means of added spatial operators such as distance, area, and intersection.

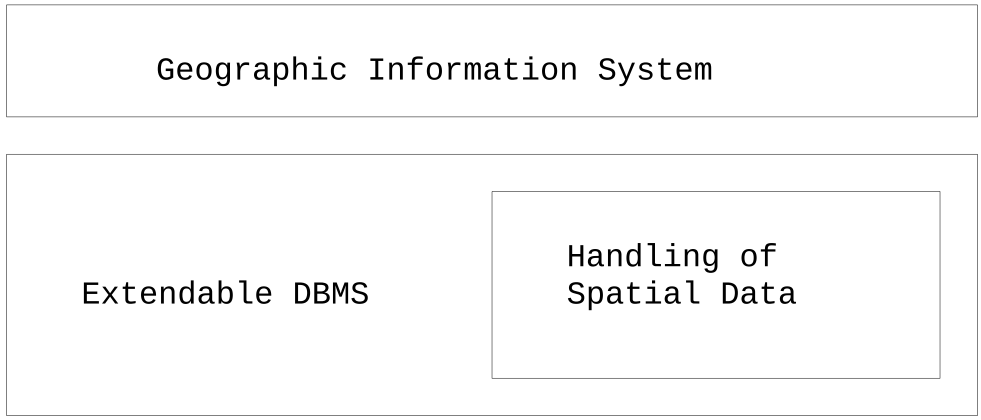

Figure 4: *The Integrated GIS Architecture*

Perhaps the most important advantage is the good performance of these systems. The direct implementation of data types in the kernel of the DBMS is very efficient. Another facility supporting system performance, is that one may provide own spatial access methods. The development of integrated GIS architectures depends on the availability of open DBMSs. A few characteristic examples of integrated GIS architectures are the research oriented systems GéoTropics [7] from the University of Paris VI and IGN France, TIGRIS [26] from Intergraph, and GEO system [58] from TNO The Netherlands. The latter is based on the open DBMS Postgres [44].

Our Wish List

The DBMS based solutions of Section "DBMS based GIS Architectures" all have in common that data must be transported from the DBMS to the application program and back if the changes of new data have to be stored permanently. As explained in the introduction, these transfers are a significant portion of the application implementation effort. Persistent programming languages try to remove this part of the implementation from the application. This section shortly discusses the major requirements for persistent objects in Procol. Some will be elaborated in one of the later sections, in which case the section referred to will be mentioned here. Note that these requirements may neither be orthogonal nor independent of each other.

r1 *Upward compatible.* Persistent objects have to be introduced with a minimal change to Procol. This implies that existing Procol programs do not have to be changed in order to be compiled by the new version of the Procol compiler.

r2 *Transparent persistent objects.* Persistent objects are treated by the application in the same manner as volatile objects. An exception may have to be made for incompatible database facilities, e.g., associative searching, that is object searching based on the contents or value of instances. Atkinson et al.[3] refine this by recognizing the following principles for persistent data (they assume several levels of persistence):

1. Persistence independence: the persistence of an object is independent of how the program manipulates that object. So, it has to be possible to call a procedure of which the actual parameters are sometimes persistent objects and other times volatile objects.

2. Persistence data type orthogonality: all objects are allowed the full range of persistence. This means that no matter how complicated the type is, its instances can still become persistent.

r3 *Complex objects.* This provides the system with sufficient modeling power, e.g., *part-of hierarchies.* In Procol, complex object types are defined by means of links (or references) to other object types that together define the complex object. The complex objects are static in terms of the object type structure and dynamic in the sense of the object instances.

r4 *Extendability with new ADTs.* This wish might be a trivial one in the context of OOPLs or object-oriented databases, but certainly not in the context of the traditional DBMSs. The definitions of the new persistent ADTs also have to be stored somewhere, if we want to be able to manipulate their object instances in a sensible manner.

r5 *Efficient handling of large numbers of objects.* Long-lived systems allow time for data to accumulate. This, combined with the fact that we aim at developing interactive systems, justifies this efficiency requirement to be even more important than in other systems. Not only efficient retrieval by object id (which is very important in OOPL and object-oriented databases, as used in navigation links) is required, but also efficient associative searching has to be possible. This is realized, as usual, by indexing techniques such as B-trees [5, 14] or hashing.

r6 *Object instances of different sizes.* A polyline or polygon has to be stored with a minimum of overhead, because of the required time (and space) efficiency in interactive systems. This implies that different instances of the same object type may have different sizes. To treat an object instance as a *unity* means that it is stored in a contiguous part of memory. This may seem to be an implementation issue, but it is very important and by putting it in our wish list we emphasize this. This topic is further discussed in section "Object Instances of Different Sizes."

r7 *Suitable for highly interactive and graphic applications.* The previous two wishes actually are part of this more general wish to make Procol suitable for this kind of applications. It has to be kept in mind that multi-dimensional data sometimes require other approaches than the data types encountered in traditional DBMSs. Also, the fact that Procol is designed as a parallel programming language should be exploited.

r8 *Exchangeable objects.* It should be possible to exchange object instances between different systems. Object instances created by one system must be directly applicable by other systems.

r9 *Deal with referential integrity in a satisfactory manner.* This is well-known problem in database and programming language research. The topic will be discussed in depth in section "Referential Integrity."

The First Attempt

In this section we describe our first (and quite simple) attempt to provide a mechanism for persistent objects, without trying to satisfy all the wishes of our requirements list. The problems we encountered give some insight in the complexity of introducing persistent objects. Then we briefly compare our first attempt with the approaches taken in some other systems.

Introducing Persistence

Normally, object instances are only present when the program is executing. Data will have to be loaded from a file or a database system into the (new) objects when the program is initialized. Just before the program stops, the data have to be saved again. This is, as argued in section "The Need for Persistence," an inconvenient method, especially in the case of applications with huge amounts of data that are not entirely needed in each session. An object-oriented step in the right direction is to store the state of objects themselves. This can be compared with making a *core dump* of a single object. When an object is saved, a "snapshot" of the object instance is made. Changes made after the save operation are not propagated to this snapshot.

The suggestion to store the objects themselves is not as simple as one might expect. This is because objects usually contain references (in attributes or local variables) to other objects. A reference to an object is an *id* (identification of the proper type), assigned to that object by the operating system when it was created with the Procol primitive *new*. In some situations it is useless to save an object without also saving the related objects.

The "snapshot" method is used in several other OOPLs. In systems offering multiple inheritance the object type that also has to be persistent, inherits this property from a general object type with methods to save and load the object. Egenhofer and Frank [18, 21] suggest the object type *db_persistent* with methods *store, delete, retrieve* and *modify*. ET^{++} [61] has an object hierarchy with the object type *object* in the top of this hierarchy. The object type *object* has methods called *PrintOn* and *ReadFrom* which enable transfer to and from disk. These solutions work fine as long as the object types contain no references to other objects but only simple attributes, such as for example an array of coordinates describing a polygon.

The persistent data in PS-algol [3, 36] are organized into one or more databases. Each database has its own root and may contain values of different (complex) data types. The data are "imported" into a program with the *open.database* procedure which returns a pointer to the root. The root has the form of a name-value table in which the value is usually a pointer to another data structure. The actual data are accessed by following these pointers and it is assumed that the programmer has to know the structure of the database (though this is nowhere stated in the PS-algol papers). Once imported, the data can be manipulated in the same manner as volatile data. The procedure *commit* propagates the changes made so far to the database, if it was open for writing. Everything that is accessible from the root is stored. This means that values may change and data (structures) be added or removed.

In the OOPL Eiffel [34] an object type that inherits from the object type *STORABLE* gets this kind of behavior by means of the methods *store* and *retrieve*. If the method *store* is invoked in object instance x, the whole object structure starting at x is dumped (in a special format) to a file, even if the referenced object types in x do not inherit from *STORABLE*. Depending on x and the object structure of the application, it is possible to store the whole object structure, or just a part of it. Basically, this solution has two drawbacks. First, the application programmer has to indicate when to save or load the objects explicitly. So, if the program is stopped before the save, the latest data are lost. Second, updating one object in an object structure can become very expensive if all related objects have to be saved also, even if they did not change.

Referential Integrity

What happens if an object is deleted by its creator while other objects are still referring to this deleted object? A dangling reference is not a problem specific to persistent objects, it is a problem in the case of volatile objects too, but it manifests itself in a severe manner in combination with persistent objects. Assume that a persistent object contains a reference to a volatile object and the program is stopped. The next time the program is started, the reference to the volatile object is not valid any more (though it has not been deleted). By the way, dangling references can also occur in non-OOPL. For example, in C it is possible to have pointers to deleted data structures, which may be the cause of some severe errors in a program. Some systems guarantee *referential integrity*. An associated problem is that of an "unreachable" object, that is an object to which the last reference is lost. There are a number of possible approaches towards these problems:

- If we want to guarantee referential integrity, we at least have to be able to detect whether the integrity is damaged by the deletion of an object. This can be achieved by associating a *reference count* with each object instance. If the reference count has a value greater than zero, the object will not be deleted and the creator is notified of this fact. The reference count mechanism introduces overhead, because the counters have to be updated in each assignment to an object variable. Problems are introduced by cyclic data structures.
- A slightly different approach, but also based on a reference count, is followed in O_2 [4]. The object deletion is not refused but postponed until the value of the reference count is zero. The creator does not have to worry about trying to delete the object another time.
- Dangling references can not occur if we prohibit the deletion of objects. This approach is taken in for example GemStone [39]. In order to avoid congestion of the system, *garbage collection* has to be performed. Two well known methods for this are:
  1. Using a reference count: when the count becomes zero, the associated object instance is automatically deleted by the system.
  2. Performing a sweep through the object space (a directed graph) in order to detect which objects are unreachable. A disadvantage is that the sweep is performed periodically and during this operation the system can not be used by the applications. This can be avoided by using an incremental version of the sweep algorithm.
- The maintenance of the reference count introduces overhead; both memory usage and execution time increase. Clearly, it is more efficient to omit a reference count and directly delete the object at request. However, in this case the system is not allowed to reuse the *id's* of deleted objects for new objects. So, if a message is being sent to a deleted object, the system can detect this, and the sender will be notified. This strategy implies that the address of an object can not be used as its id, because when the object is deleted we want to be able to reuse that part of the memory space for new objects.

It may be clear by now that we are biased towards the latter approach. In the context of Procol, dangling references are probably programming errors and the detection of the illegal use of dangling references during run-time is an adequate solution. Finally, it is interesting to note that $PCTE^+$ [27, 28] offers links both with and without referential integrity. This is probably done for efficiency reasons. It is not stated in the $PCTE^+$ documents how the referential integrity is maintained.

Object Management

In order to solve the administrative problems associated with the use of object id's, there is a need of an *Object Management System* (OMS) that takes care of the (persistent) objects. One of the responsibilities of the OMS to keep the references in the object system consistent. To be more precise, an object system is consistent if [30]:

- No two distinct objects have the same identifier (unique identifier assumption). In other words, the identifier functionally determines the type and the value of the object.

- For each identifier present in the system there is an object with this identifier (no dangling identifier assumption).

Object Identity

A uniform object identification mechanism has to be developed, capable of dealing with objects shared by multiple programs, multiple users or even multiple computers (in a network). There should be a mechanism to indicate in which persistent objects one is interested, so one is not bothered by uninteresting objects of others. One possible method could be to organize the object instances in "datasets" which are put in the normal hierarchical file system. This limits the scope and makes the task of finding the right communication partner easier for the OMS.

In relational databases [13] an identifier key is formed by one or more user-supplied attributes. Value based matching is a transparent technique for expressing relationships. However, it provides no support for referential integrity at all. By contrast, OOPLs support the notion of object identity which is independent of the attribute values [37]. Khoshafian and Copeland [30] describe several techniques for implementing object identity and they conclude that using so called *surrogates* is the best technique. Surrogates are system-generated, globally unique identifiers, completely independent of the physical location and data contents of an object.

Searching

The objects as presented so far are not suited for associative search operations. That is, searching based on the contents of an object instead of using the object id to find an object. This is especially useful for a program that wants to use objects created by other programs, because the id's are unknown and have no semantic meaning. All that a program(mer) knows is about: object types (the kind of data he wants to use) and attribute values (restriction of instances).

Another use of associative searching is to solve the query: "How many inhabitants has the municipality with the name attribute 'Utrecht'?". We have to look at all the instances of the municipality object type until we have found the proper one. This is an $O(n)$-algorithm. However, this problem can be solved with an $O(log(n))$-algorithm, if a binary search is used. In a relational database, efficient searching is implemented by a B-tree [5, 14] for attributes on which an *index* is put. The B-tree has many useful properties, such as: it stays balanced under updates, it is adapted to paging (multiway branching instead of binary) and has a high occupancy rate.

The B-tree solution in an object-oriented environment is established by a set of auxiliary (system) objects. These objects do not contain the application data, but contain tree structures with references to the objects with the actual data. This B-tree has to be part of the OMS and, if possible, transparent to the "application" objects. Note that the OMS itself can be implemented in Procol as a set of objects.

There is some friction between the concepts behind the ADTs and the idea of associative searching, because associative searching requires knowledge of the internals of other objects. An object has to specify its query in terms of data-parts that are inside other objects. To limit the damage, only so called *visible* attributes may be used in the query. These visible

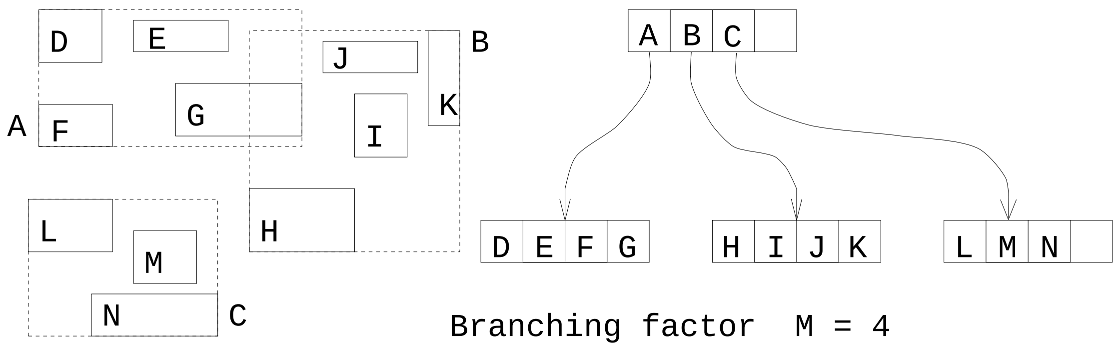

Figure 5: *The R-tree*

attributes become part of the specification of an object type (together with the actions of course), in contrast to the non-visible data-part which belong to the implementation. Note that an index may be put only on a visible attribute of an object type.

Multi-dimensional Data

The searching problem also applies to the graphic or geometric data. If no spatial structure is used, then queries such as "Give all municipalities within rectangle X" are hard to answer. A spatial data structure which is especially suited for the object-oriented environment is the R-tree [25]. This is because the R-tree already deals with objects; it only adds a minimal bounding rectangle (MBR) and then it tries to group the MBRs which lie close to each other; see Figure 5. This grouping process is reflected in a tree structure, which in turn may be used for searching. Several test results [20, 23] indicate that the R-tree is a very efficient spatial data structure.

Not all known spatial data structures [51] are suited for this purpose. For example kd-trees [8], quadtrees [41], R^+-trees, bsp-trees [52], cell-trees [24] and gridfiles, are more difficult to integrate in the object-oriented environment because they cut the geographic objects into pieces. This is against the spirit of the object-oriented approach, which tries to make complete "units, " with meaning to the user. The Field-tree [22], KD2B-tree and the Sphere-tree [54] are good candidates for integration in an object-oriented system, because they do not split the objects. Each of the spatial search structures has its own strengths and weaknesses, so that if several alternatives are offered by the system, an application can use the structure that fulfills its needs best.

In Procol, trees can be implemented in two different ways. The first method stores the entire tree in one single search object. The second method stores each node of the tree in a separate instance of the search object. The latter introduces overhead by creating a lot of search objects (nodes). However, it has the advantage of being suited for parallel processing in Procol because the search objects can run on parallel processors. This is useful for range queries: "Give all municipalities with more than 10.000 and less than 20.000 inhabitants." The appendix of [55] contains a part of the Procol implementation of the R-tree. In this implementation each node of the tree is represented by a separate object. In case of the R-tree this is a reasonable choice, because there is a fair amount of work in each node. The same would be valid for B-trees, but not for binary trees. In any case, for practical reasons, there has to be a separate search tree (index) for each attribute for which efficient searches are required. This has to be made clear to the OMS before the queries are posed.

It is possible that the value of an attribute changes, after the search tree has been created for that attribute. In that case, the tree may have become incorrect or inconsistent. This can be solved by sending an (implicit) message to the search tree object(s) in the OMS, just after the attribute has changed. Upon receipt of this message the search tree adjusts itself.

The Procol Extension

This section presents some issues concerning the syntax and semantics of the constructs which might be added to Procol for the support of persistent objects. This is done here without worrying how this can be achieved in our implementation of Procol. From the users point of view, this extension should be as small and simple as possible. First, we have to decide how to indicate that an object is persistent. Some alternative possibilities are:

- *Per Object Instance:* At the moment an object is created, it is decided whether it will be persistent or not. A convention can be made that objects created with the *new* primitive are volatile and the ones created with the *persistent* primitive are persistent.
- *Per Object Type:* At the moment the object type is defined it is specified whether all instances of this type are persistent or not. A modified keyword *OBJ* could indicate this: *PERSISTENT_OBJ*.
- *On the fly:* Make a volatile object instance persistent by applying a new Procol primitive *persistent*. Assume the variable *x* holds the id of an object instance; then this instance is made persistent by: *persistent x*. Note that there is a difference with the *save* operation of section "The first Attempt," because the values (states) of a persistent object are always guaranteed to be up-to-date. This approach has been chosen for Procol.

A combination of these approaches is also possible. In the language E [40] the programmer has to indicate per type (class) that instances are optionally persistent. The programmer has to decide per instance if it is really a persistent object instance.

The advantage of persistence per object type is that only once, during the object type definition, there is a difference for the application programmer between persistent and volatile objects. In the other solutions it is required to indicate that the object is persistent for each object instance. The major drawback of the latter choice is that two different types have to be defined if we want to use both the volatile and the persistent variants of basically one object type. In the case of strong type checking this means that we can not freely interchange the use of volatile and persistent objects as arguments in messages and procedure calls.

In order to get hold of persistent objects with unknown id, Procol will be extended with the *retrieve* primitive. Perhaps it is better to take the following approach towards the primitives *new, delete,* and *retrieve*: consider them as messages to the object types themselves ("class methods"). These are system objects (partly) responsible for the OMS tasks. These system objects have to maintain index structures if requested by sending them a *create_index* message.

The *retrieve* primitive has some resemblance to the *new* primitive, because it also assigns the id of an object to a variable of the proper type. Unlike *new*, *retrieve* will not execute the Init section, because that already happened when this object was created for the first time. The protocol (expression) of the object regulating access to the object is matched starting at the current (saved) state.

A *discrimination condition* can be used, because the object type information may not be specific enough. Of course only visible attributes can be specified in the condition. A *retrieve* returns the id of the object of the proper type for which the discrimination condition evaluates to *True*. If there is more than one object satisfying these criteria, only one is returned. If there is no object satisfying these criteria, a *NIL* object is returned.

It is a small step from the *retrieve* primitive to the associative search operation. In fact, it could be considered as an iteration over the retrieve operation. If fast replies are required, then in case of large set of objects, an index has to be used. This index could be a spatial index structure, e.g., for efficiently solving a "rectangle" query. There are several options for returning the answer of a search:

- Return one big set that contains the id's of all objects that satisfy the query. In case of large answers, a lot of temporary memory is required and it may take quite a while to generate the complete answer.
- Another strategy is first to state the query and then retrieve the answer one by one (or perhaps in buffers of a fixed size). The first part of the answer will probably be ready sooner than the complete answer would be. This promotes parallelism and is also quite important in an interactive application, because in that case the end-user can already see something on the screen.

The problem with the second solution for returning a search result is that other objects might interfere with the set of objects that belongs to the queried object type. "Third-party" objects could change values and add new instances or delete existing ones. We still have to investigate whether this can be solved by applying the right protocols in the system (OMS) objects. This has to be solved before we decide on the syntax of a search query.

Object Instances of Different Sizes

In section "Our Wish List" we saw that the wish to store a polyline or polygon with a minimum of overhead, implies that different instances of the same object type may have different sizes. So for example, the pure relational solution, presented by van Roessel [59] is not acceptable, because a polyline is scattered over several tuples in a table and first has to be aggregated before it can be used again. In [57] a solution in the context of the relational data model is presented.

Different sizes have an (enormous) impact on the implementation of persistent objects. In CO_2, the C implementation of O_2 [4], it was decided to prohibit object instances of different sizes. In contrast to this we would like to have persistent objects whose sizes may even change dynamically. For example, to make it possible to remove points from, or add points to, a polyline. However, this would even further complicate the implementation. A decrease of the size of an object is not too hard, but an increase of object size means that an object does not fit in its (contiguous) part of memory and the memory after this object is probably occupied by another instance. The object will have to be moved to another, larger, place, because we want to treat an object as a unity and do not want to split it. This would be impossible if the objects id is its address. However, this was already disapproved of because of reasons discussed earlier. In any case, growing persistent objects could introduce a lot of overhead.

A more feasible situation is that after the Init section, the size of an object may not vary any more. It is still possible to deal with dynamic problems. For example, use a pointer (id) to an object of type linked list. This object type has an “application” data-part and a pointer to the next list element. Each instance represents one list element and they all have the same fixed size. We can extend this approach and simplify our implementation of Procol, if we only allow the following data types as attributes: *Basic types* (int, char, float, ...), *References* (or links) to other objects, and *Arrays* with fixed size after the Init.

Persistent Objects in Procol

The question how to implement persistent objects in Procol can be divided into two sub-questions. The first is how to adapt the languages features (the external implementation). The second is how to implement this on the underlying platform (the internal implementation).

External Implementation

Our decisions regarding the external implementation of persistence in Procol include the introduction of the following new keywords:

1. **persistent** <object-id>
   in <dataset-key>
   With this statement a volatile object instance identified by <object-id> can be made persistent by coupling it to a dataset identified by <dataset-key>.
2. **volatile** <object-id>
   With this statement an object instance (identified by <object-id>), that has been made persistent before, can be made volatile. The result of this statement is that the persistent object instance is removed from its dataset.
3. **retrieve** <object-id>
   from <dataset-key>
   where <discrimination-string>
   and
   next <object-id>
   With this statement we can retrieve an instance of the same type of <object-id> from the dataset with identifier <dataset-key>, that satisfies the <discrimination-string>. If the dataset in question contains more than one instance of the required type, only one will be returned in <object-id>. Any successive execution of the **next** statement, will retrieve the *next* instance of the required type until all required instances have been retrieved from the dataset.

In general, <dataset-key> will be a string. This string can then be used to compose the filenames of the necessary dataset files. An example use of the provided persistence in the Declare section of an object type using the object `DRAWING`:

```
Declare

        object DRAWING drawing;

        allocate_drawing(char *name)
```

```
{
    new drawing;
    persistent drawing in name;
}
read_old_drawing(char *name)
{
    retrieve drawing from name;
}
```

Internal Implementation

We will now motivate our design decisions for the internal implementation. Procol has been developed and implemented on a network of Sun workstations running under SunOS Release 4. Five possible implementations of the extension of Procol with persistent objects were considered, (compare with [47]): *bare* implementation (using the Unix OS-interface calls: open, close, read and write), *shared mapped memory* (virtual files), *Ingres* [46] (or other relational DBMS), *Postgres* [44] (or other extendable DBMS), *PCTE*$^{+}$ [27, 28] (offers a powerful OMS derived from the Entity-Relationship Model of Chen [12]). See [55] for an more extensive discussion.

We have chosen for the mapped memory approach. A file is mapped directly by the Unix OS on the address space of a process. A major advantage is that there is no difference between the "stored" object instances and their "running" counterpart, at least not at the level of the Procol kernel. At OS level there is a difference and this is the same as the difference between virtual memory pages that are in main-memory and the ones that are swapped on disk. We expect this implementation to be very efficient. In order to gain some experience we are currently converting a local application that uses explicit read and write statements, into a mapped memory implementation. First test results indicate that the elapsed times decrease with about 30% in applications with many read and write statements. A disadvantage of the mapped memory approach is that we still have to do the memory management ourselves. In [55] it is explained in more detail, why we decided to use the mapped memory approach for the prototype implementation of persistent objects in Procol.

An object instance is identified by a *surrogate* [30], that is, the object id is not the actual address in memory but an indirection [45] to the actual locatation of the object. See figure 6 for a graphic demonstration of the process. Each surrogate contains an indication whether the object instance is volatile, persistent or deleted. When, during the execution of a Procol program, an object is referenced, we have to check the first part of the surrogate. If the object instance is volatile, the surrogate contains a key than can be used to retrieve the memory address of the instance variables. If the object instance is persistent, the surrogate contains a dataset identifier, and a key. With this dataset identifier and the key, the OMS is able to retrieve the actual memory address of the the instance variables of the persistent object in question.

The disadvantage of surrogates is that there is an extra indirection. However, surrogates have several important advantages. They allow objects to be switched between the volatile and persistent states, by modifying a part of the surrogate. We decided, based on the advantages and disadvantages mentioned in section "Referential Integrity," that the control of referential integrity is not required in Procol. However, the use of illegal references is signalled at run-time. This is done by inspecting the surrogate and taking the appropriate measures. If a persistent object contains a reference to a volatile object, this volatile object is not saved automatically. The proper way to program this case is to make the other object also persistent, if the referenced object is still needed in the future. The state of each object instance is stored as a single unit in contiguous memory. Instances of different sizes are no problem. The state

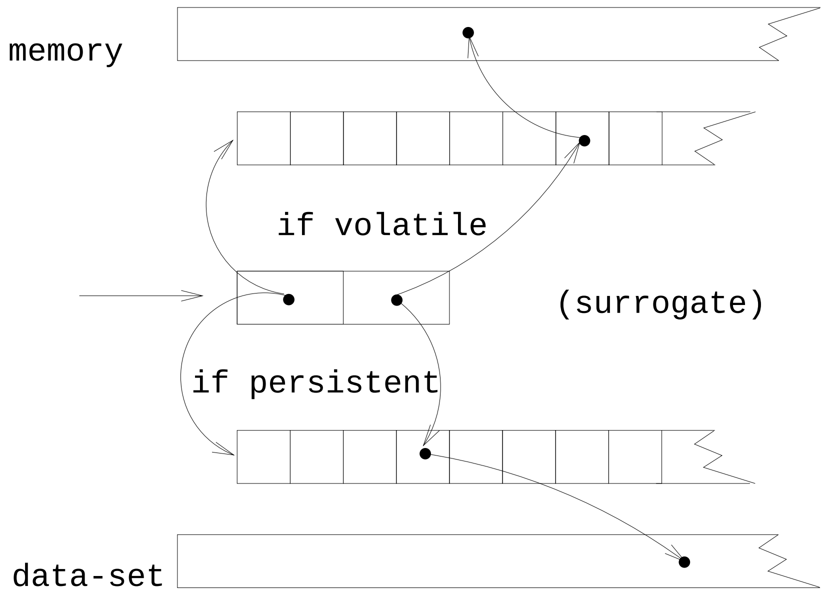

Figure 6: *Object Reference with Surrogates.*

also contains some additional data, for example the creator. The application programmer decides whether instances of the same object type are "stored together" without instances of other types in one dataset or if a dataset contains instances of different types. The latter is advantageous for representing complex objects in an efficient manner, because the instances of the different types that define a complex object are stored close together. Note that complex objects are important in CAD systems, because this is one of the main modeling tools.

The GIS case example

This section describes the design and implementation of a small GIS using persistent objects. The goal of this case is to investigate the behavior of persistent objects during design and implementation of an application. This case is an extension of the one described in [56]. To accomplish our goals we started by designing a non-persistent version and make the necessary changes for persistence later. With this approach we should be able to determine whether it is possible to change an existing non-persistent program into a program that does use persistent objects or that this change needs a different design approach. A GIS is, from this viewpoint, a non-traditional type of application. Every GIS must have three fundamental operations available for a user: display a map, select entities from a map and perform spatial calculations with one or more selected entities. Because the goal of the case was not to implement a complete GIS we only considered the first two operations in our GIS-application. The subject of the GIS is the administrative map of The Netherlands.

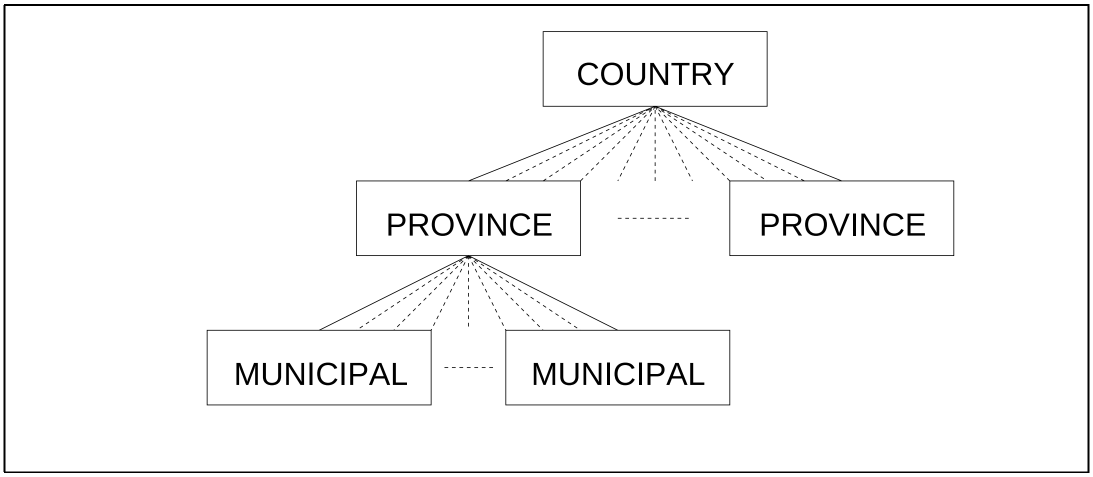

Figure 7: *Hierarchical structure of the map of the Netherlands.*

The design of the non-persistent version

In this case the objects are the administrative units. The smallest administrative units are called *municipalities* (see Figure 7). The main geometric attribute of these objects is a polygon describing the boundary of the municipality. Other attributes are the name and number of the municipality. We did not add any other thematic information, although it would not be difficult to add this data in additional attributes. All these municipalities are grouped together into twelve larger administrative units called *provinces* each of which contains a list of child municipalities. Each municipality has a reference to its parent province, this to create a full duplex reference. The main geometric attribute of a province is, just like the municipality, a polygon describing the boundary of the province. There is also a reference attribute to the parent object called *country*. The boundary of the province is also a part of the boundary of some municipalities and the some boundaries of the provinces form the boundary of the country.

To avoid redundant storage of the boundary information, another type of object called *chain* is introduced. A chain is a part of the border of a municipality that has a node at the begin and end point. A *node* is a point where three or more municipalities meet (water and foreign countries are considered to be special cases). The attributes of the chain object are the number of polyline points, an array of polyline coordinates and a reference to the left and right municipality. To create a full duplex reference, a list of child chains is added to all the municipalities. With this central storage of boundary information the geometric attribute can be removed from the municipality and province objects. So far, only an administrative logical hierarchy is present. To add a simple spatial hierarchy, we add a bounding box attribute to all these objects. This makes a hierarchical entity search possible. Most objects are skipped during an entity search, because a point can not be inside an administrative unit if it is outside its bounding box.

A summary of the functional requirements for the GIS are:

- Display the map by showing either country, or province, or the municipality borders.
- Allow zoom-in operations on the map. The user indicates a rectangle on the map. The GIS zooms in and redisplays the map.

- Allow a zoom-out operation on the map. The GIS zooms out to the original scaling and redisplays the map.
- Allow the user to select a municipality by clicking within its border. As a result the GIS should show the name of the municipality.
- Store the map (i.e. be an application that uses persistence).
- Allow the user to stop the GIS.

The implementation of the non-persistent version

The GIS contains two logically separate parts. One part translates the map data from a certain binary format into Procol objects. The other part is the actual GIS. These two parts form one program. Later in this article, in the persistent version of our GIS, these parts are actually split into two separate programs.

The GIS part of the program contains three major sub-parts: a drawing area with input devices, a panel with pushbuttons and the data structure representing the map. The appropriate objects are declared in the `Declare` section, which additionally contains some auxiliary variables and a redraw function:

```
Obj     GIS
Declare object WINDOW           window, drawingarea;
        object IN_POINT         in_point;
        object IN_BOX           in_box;
        object PANEL            panel;
        object COUNTRY          country;
        object MUNICIPALIY      municipality;
        object READ             read;
        char   name[32];
        float  scale;
        int x, y, w, h, draw_mode, x1, x2, y1, y2;
        include "gis.h"
        redraw(new_mode)
        int new_mode;
        {
           draw_mode = new_mode;
           send drawingarea.Clear();
           send drawingarea.SetScale(x, y, scale);
           send country.Draw(drawingarea, draw_mode, x, y, w, h);
        }
```

The Protocol defines that the `GIS` object expects messages from `PANEL` and from the two input devices `IN_BOX` and `IN_POINT`.

```
Protocol  panel -> Zoom +
          panel -> UnZoom +
          panel -> DrawCountry +
          panel -> DrawProvince +
          panel -> DrawMunicipality +
          panel -> Quit +
          in_box(x1, y1, x2, y2) -> ZoomInArea +
          in_point(x1, y1) -> PointPick
```

In its Init section, the GIS object creates the window to hold the application, requests a subwindow from it to serve as drawing area, creates the panel with the pushbuttons, defines an initial input device for selecting a municipality, and finally creates the READ object which reads all binary data and creates the appropriate data objects. Note that no further actions of this object are activated; all reading is done in the Init section of the READ object.

```
30.  Init  x = y = 0;  w = 590;  h = 690;  scale = 1.0;
31.        new window(50, 50, w+180+12, h+8, w+180+12, h+8);
32.        send window.Open();
33.        request window.SubWindow(4, 4, w, h, w, h) -> (drawingarea);
35.        new panel(window, w+8, 4, 180, h);
36.        send panel.Enable();
37.        new in_point(drawingarea, 0, 0, w, h);
38.        new read;
39.        request read.GetCountry() -> (country);
```

At line 30 the auxiliary variables are give its initial values. The variables x, y, w, and h define the current viewport in the drawing area (after zooming in). The size and position of the viewport are saved to optimize drawing. That is why these values are passed to the country at line 19, whenever the map needs to be redrawn. The scale variable refers to the current scale of the drawing area. It is used at line 16 as a parameter to the SetScale message that is send to the drawingarea.

When the GIS object is deleted, it disposes of its child objects as shown in lines 43 and 44.

```
42.  Cleanup  send panel.Disable();
43.           del  panel, window, in_point;
44.           if (in_box) del in_box;
```

The corresponding Actions that are executed when the GIS object receives messages from PANEL, IN_BOX or IN_POINT are described next. To start with, the Zoom Action is called by the PANEL whenever the user presses the corresponding screen button. This Action ensures that the GIS object changes its interaction mode from *municipality selection* mode to *zoom-in* mode. Therefore, the current input device for selection, the IN_POINT object, is deleted at line 48. Then an object instance in_box is created at line 49. The user now has to input a rectangle on the map, indicating the area of interest.

```
46.  Actions
47.     Zoom = {
48.        del in_point;
49.        new in_box(drawingarea, 0, 0, 590, 690);
50.     }
```

After executing this Action, the map is of course not yet zoomed in. The actual zooming is done in Action ZoomInArea. After pressing the Zoom button on the panel, the user has to input a rectangle in the drawing area. The IN_BOX object will receive the mouse events originating from the drawing area and will draw a rubber box while the user is moving the mouse. When the user has completed the rectangle, the IN_BOX object will send the start point and the end point to its creator, which is the `GIS` object. The Action in which these coordinates are eventually received is the `ZoomInArea` Action:

```
ZoomInArea = {
   del in_box;
   new in_point(drawingarea, 0, 0, 590, 690);
   x += (int) ((float)x1 / scale);
   y += (int) ((float)y1 / scale);
   w =  (int) ((float)(x2-x1) / scale);
   h =  (int) ((float)(y2-y1) / scale);
   if (w>h) scale = 590.0 / (float) w;
   else scale = 690.0 / (float) h;
   redraw(draw_mode);
}
```

Important is that all the code for parsing events and for drawing rubber boxes is encapsulated into the objects `IN_BOX` and `IN_POINT`. The `GIS` object does not need to be concerned with these low level details. The same holds for the creation and management of the screen buttons. This is encapsulated into the `PANEL` object. What is left for the `GIS` object, is to allocate the correct resources and to define the control flow of the application without being bothered with distracting details.

The `UnZoom` Action restores the image of the map into a fully zoomed-out projection. Basically this Action serves as a reset function. For practical reasons, the `GIS` object does not remember previous zoom operations and cannot gradually zoom out. Of course, there is no particular reason why this could not be implemented in a more sophisticated version of the GIS application.

```
UnZoom = {
   x = y = 0;  w = 590;  h = 690;  scale = 1.0;
   redraw(draw_mode); /* draw in current drawmode */
   if (in_box) del in_box;
}
```

The panel has three buttons to draw the map, depending on the level of detail the user wants to see. The Actions called on activation of these buttons are:

```
DrawCountry  = { redraw(DRAWMODECOUNTRY); }
DrawProvince = { redraw(DRAWMODEPROVINCE); }
DrawMunicipality     = { redraw(DRAWMODEMUNICIPALIY); }
```

Furthermore, the panel has a button to stop the GIS application. When this button is activated, the `GIS` object will ask its `creator` to delete it.

```
Quit = { send creator.Stop(NORMAL); }
```

One of the main functions of the GIS application is represented by the PointPick Action. This Action is activated whenever the IN_POINT object has detected a selection by the user of the GIS application. The IN_POINT object produces two integers representing the point selected by the user. This point is given to the COUNTRY object at line 77. The COUNTRY object will return the municipality which contains this point, or it will return the NIL object whenever the user has selected a point outside any municipality (e.g. in foreign area). After receiving the municipality through the request, the GIS object draws the name of this municipality on the screen, together with an indication of the position that has been selected by the user. The names can be removed by redrawing the map.

```
76.     PointPick = {
77.        request country.GetMunByXY(x1, y1) -> (municipality);
78.        if (municipality) {
79.           request municipality.GetName() -> (name);
80.           send drawingarea.SetFontSize(18);
81.           send drawingarea.DrawString(x1, y1, name);
82.           send drawingarea.DrawEllipse(x1, y1, 1, 1);
83.        }
84.        else
85.           send drawingarea.Bell();
86.  }
87.  Endobj  GIS;
```

Layered interaction model

Graphical output and events generated by the user of the GIS are handled by a layered interaction model as described in [32]. Because the abstraction from physical events, increases by each layer added to the model, a highly extendable and high-level user-interface toolkit is achieved. Interaction tools are described as a composition of high-level abstract events. Interaction tools can be combined into abstract interaction panels. The use of Protocols allows specification of input patterns and sequencing. Because all access to the underlying window system is handled by a WINDOW object, we hide a lot of implementation details into a lower level. The individual layers of the model are depicted in Figure 8.

The WINDOW object forms layer 1. It forms the only link to the underlying window system. All drawing and event handling is done through this object. Therefore, the WINDOW object contains Actions for drawing lines, ellipses, text, etc. Event handling routines are also included.

The objects in layer 2 represent objects for managing events. Examples are KEY, MOUSE_1_DOWN, MOUSE_1_UP and MOUSE_MOTION. In fact, these objects form abstract representations of events processed by the WINDOW objects. The Protocol of a layer 2 object is as follows:

```
Protocol  WINDOW(x,  y) -> Event +
          ANY(window, x, y, width, height)  -> AddClient +
          ANY(window, oldx, oldy, oldwidth, oldheight,
              newx, newy, newwidth, newheight) -> MoveClient +
          ANY(window, x, y, width, height)  -> RemoveClient
```

Whenever an event of a certain type is generated by the underlying window system, the WINDOW object in which this event occurred will send it to the corresponding abstract event

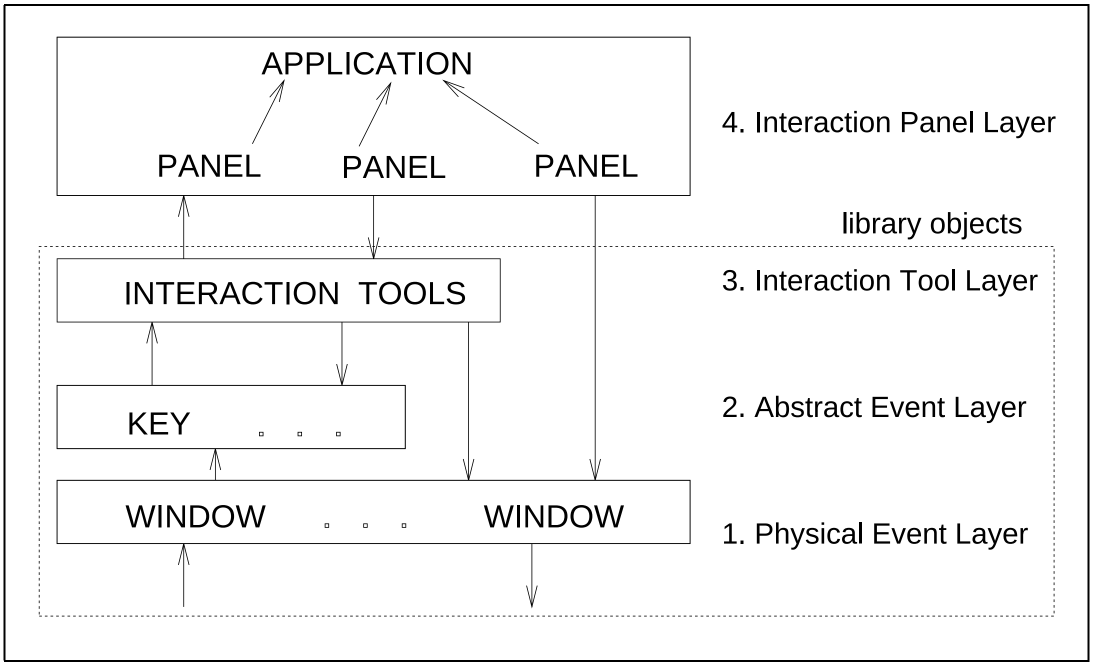

Figure 8: *A GIS design based on layered interaction model.*

object in layer 2. It is handled there in Action `Event`. Objects of level 3 and higher, interested in events of a given type, will register with the corresponding abstract event handler in layer 2, by sending the message `AddClient` to the appropriate type (see for example line 10 in the following object definition). Clients can un-register by sending `RemoveClient`. Clients can update their registration by sending `MoveClient`.

Typical objects found in layer 3 are interaction tools such as buttons, scrollbars, menus, etc. They are defined in high level abstract events found in layer 2. As an example we will show a definition of a button developed with the above mentioned layered model:

```
1.   Obj      BUTTON(object WINDOW window, int x, int y,
2.                  int w, int h, object ANY buttondrawer);
3.   Declare  int ex, ey;
4.   Protocol creator -> Enable ;
5.            MOUSE_1_CLICK(ex, ey) -> Pressed * ;
6.            creator -> Disable
7.   Init     send MOUSE_1_CLICK.AddClient(window, x, y, w, h);
8.   Cleanup  send MOUSE_1_CLICK.RemoveClient(window, x, y, w, h);
9.   Actions  Enable  = { send buttondrawer.DrawActive(); }
11.           Pressed = { send buttondrawer.Echo(); send creator.(); }
12.           Disable = { send buttondrawer.DrawInactive(); }
13.  Endobj   BUTTON;
```

The Protocol specifies that the `creator` of a `BUTTON` object instance has to enable the button first before it can be used. When this is done, the `Enable` Action at line 10 is activated. The object `buttondrawer` is told to draw its active representation. The id of the `buttondrawer` object is passed as an attribute (parameter) to an object of type `BUTTON` when it is created (see line 2). Furthermore, activation of the `BUTTON` is enabled by registering with the `MOUSE_1_CLICK` object type. Activation of the button works as follows: whenever the window system generates an event notifying that the user has pressed the first mouse button, it is received by a WINDOW object. The WINDOW object then sends a message to the appropriate abstract event handler. This object will in its turn send a message to the corresponding client without specifying an Action name. The receiving object (in this case the BUTTON) is free to dynamically bind the message to an appropriate Action. Therefore, line 5 actually binds the message to the Action Pressed. When the Action is activated, the BUTTON is highlighted at line 11 by sending the Echo message to the buttondrawer. Then the creator is notified by sending it a message, again without naming the Action. The creator can again bind this message from the BUTTON to any appropriate Action, possibly depending on its Protocol state.

The hierarchical data structure

So far we only discussed the top level of the data structures for our GIS. As in the interaction model, the data structures used in our GIS are hierarchical too. The only object the GIS object communicates with is the COUNTRY object. The drawing and municipality selection is propagated through the hierarchy down to the chains.

In case of a point selection, the COUNTRY object activates the GetMunByXY action at line 13. At line 14 the first province is asked whether the point (x, y) is inside its bounding box. If not, the task to find the selected municipality is delegated to the first province (line 15). If the point is inside the bounding box of the first province, the province is asked to return the selected municipality (line 17). If the province cannot return a municipality, because the point is not inside any child municipalities of the province, the NIL object is returned. In that case the task to find the selected municipality is delegated to the first province (line 18). If the province does return a municipality, this municipality is sent to the sender of the GetMunByXY message (line 20). If there are no provinces in the children list of the COUNTRY object, or when a point outside any municipality is chosen, the NIL object is returned to the sender of the GetMunByXY message (line 23).

```
1.   Obj      COUNTRY
2.   Declare  object PROVINCE list;
3.            object MUNICIPALITY mun;
4.            int inside, x, y;
5.   Protocol .... +
6.            ANY(x, y) -> GetMunByXY
7.   Init ....
8.   Actions
        ....
13.     GetMunByXY = {
14.        if (list) {
15.           request list.InBox(x, y) -> (inside);
16.           if (!inside) @list.GetProvByXY(x, y);
17.           else {
18.              request list.GetMunByXY(x, y) -> (mun);
19.              if (!mun) @list.GetProvByXY(x, y);
20.              else send sender.(mun);
21.           }
22.        }
23.        send sender.(NIL);
24.     }
25.  Endobj   COUNTRY;
```

The PROVINCE object has three actions involved in the municipality selection. The InBox request of the COUNTRY object activates the action at line 33 and 34, which returns whether the point (x, y) is outside the bounding box. The GetProvByXY action at line 36 through

47, is almost identical to the `GetMunByXY` action of the `COUNTRY` object, except that the next province is represented by the object `next` instead of by `list`. The action `GetMunByXY` at lines 48 through 55, has a similar structure. The municipality, represented by `list` is asked whether the point `(x, y)` is inside its borders (line 50). If not, the task to find the selected municipality is delegated to that first municipality (line 52). If the selected point is inside, the municipality is returned to the requesting object (line 51). If this province has no child municipalities, or the selected point is not inside any child municipality of the province, the `NIL` object is returned (line 55).

```
1.   Obj      PROVINCE(....)
2.   Declare  ....
3.            object MUNICIPALITY list;
4.            object MUNICIPALITY mun;
5.            object PROVINCE next;
6.            int maxx, maxy, minx, miny,
7.            int x, y, inside;
8.   Protocol .... +
9.            ANY(x, y) -> InBox +
10.           ANY(x, y) -> GetProvByXY +
11.           ANY(x, y) -> GetMunByXY
12.  Init ....
13.  Actions
        ....
33.     InBox  = { send sender.(x<=maxx && x>=minx &&
34.                                y<=maxy && y>=miny);
35.     }
36.     GetProvByXY = {
37.        if (next) {
38.           request next.InBox(x, y) -> (inside);
39.           if (!inside) @next.GetProvByXY(x, y);
40.           else {
41.              request next.GetMunByXY(x, y) -> (mun);
42.              if (!mun) @next.GetProvByXY(x, y);
43.              else send sender.(mun);
44.           }
45.        }
46.        send sender.(NIL);
47.     }
48.     GetMunByXY     = {
49         if (list) {
50.           request list.Inside(x, y) -> (inside);
51.           if (inside) send sender.(list);
52.           else @list.GetMunByXY(x, y);
53.        }
54.        send sender.(NIL);
55.     }
56.  Endobj  PROVINCE;
```

The `MUNICIPALITY` object has two actions involved with searching. The `GetMunByXY` action at line 40 through 50 is identical to same action of the `PROVINCE` object. The action `Inside` at line 48 through 54 first checks whether the point is inside the bounding box (line 49). If so, a point-in-polygon test is executed (line 50 and 51). It requests the number of line cuts made with each of its borders when going from point `(x, y)` to point `(infinite, y)`. If the total number of cuts is odd, the point `(x, y)` is inside the municipality. Otherwise it is outside the municipality. If point `(x, y)` is not inside the bounding box, false is returned.

```
1.   Obj      MUNICIPALITY(....)
2.   Declare  object CHAIN_CONTAINER list;
3.            object MUNICIPALITY next;
4.            int total, inside;
5.            int x, y, maxx, maxy, minx, miny;
6.   Protocol .... +
7.            ANY(x, y) -> GetMunByXY +
8.            ANY(x, y) -> InBox +
9.            ANY(x, y) -> Inside
10   Init ....
11.  Actions
        ....
40.     GetMunByXY = {
41.        if (next) {
42.           request next.Inside(x, y) -> (inside);
43.           if (inside) send sender.(next);
44.           else @next.GetMunByXY(x, y);
45.        }
46.        send sender.(NIL);
47.     }
48.     Inside = {
49.        if (x<=maxx && x>=minx && y<=maxy && y>=miny) {
50.           request list.Cutting(x, y, 0) -> (total);
51.           send sender.(total%2);
52.        }
53.        else send sender.(0);
54.     }
55.  Endobj   MUNICIPALITY;
```

The CHAIN_CONTAINER object requests each of its elements the number of line cuts in the for-loop at line 21. In the case that the container is full, the counting of line cuts is delegated to the next container. The current number of cuts is stored in the variable total. This value gets updated in the CHAIN object.

```
1.   Obj      CHAIN_CONTAINER
3.   Declare  object CHAIN element[MAX_ELT];
4.            object CHAIN_CONTAINER next;
5.            int last, total, x, y, i;
6.   Protocol .... +
7.            ANY(x, y, total) -> Cutting
8.   Init ....
9.   Actions
        ....
19.     Cutting  = {
20.        for (i=0;i<last;++i)
21.           request element[i]].Cutting(x, y, total) -> (total);
22.           if (last == MAX_ELT) @next.Cutting(x, y, total);
23.           else send sender.(total);
24.     }
25.  Endobj  CHAIN_CONTAINER;
```

The number of cuts for a particular chain is calculated at line 21 and 22. The current number of cuts is added to the number of already calculated cuts and is returned to the requesting object (i.e. sender).

```
Obj      CHAIN( .... )
Declare  ....
         find_number_of_cuts() { .... }
         int x, y, total, cuts;
Protocol .... +
         CHAIN_CONTAINER(x, y, total) -> Cutting
Actions
   ....
   Cutting = {
      cuts = find_number_of_cuts();
      send sender.(total+cuts);
   }
Endobj  CHAIN;
```

Drawing the map on the display is not as complicated as the selection of a municipality. This is because drawing only needs to be propagated downwards. No information needs to go upwards. Below, the code fragments are given of the drawing actions of the `COUNTRY`, `PROVINCE`, `CHAIN_CONTAINER` and `CHAIN` data objects. The `COUNTRY` object receives the message `draw` from the `GIS` object. At line 13, it simply forwards this message to the provinces contained in the province chain.

```
1.   OBJ      COUNTRY
3.   Declare  ....
4.            object PROVINCE list;
5.            object WINDOW   drawingarea;
6.            int mode, x, y, w, h;
8.   Protocol .... +
9.            ANY(drawingarea, mode, x, y, w, h) -> Draw +
10   Init ....
11.  Actions
12      ....
13.     Draw = { if (list) send list.Draw(drawingarea,mode,x,y,w,h); }
14.  Endobj COUNTRY;
```

When a province receives the `Draw` message, it checks whether the bounding box intersects the viewport passed to it (line 29). If the two boxes do intersect, the `Draw` message is sent to the first municipality (line 30). At line 31 the `Draw` message is sent to the next province.

```
16.  Obj     PROVINCE(....)
17.  Declare ....
18.          object MUNICIPALITY list;
19.          object PROVINCE     next;
20.          object WINDOW       drawingarea;
21.          int maxx, maxy, minx, miny;
22.          int x, y, w, h, mode;
23.  Protocol  .... +
24.            ANY(drawingarea,mode,x,y,w,h) -> Draw +
25.  Init ....
26.  Actions
27.     ....
28.     Draw = {
29.        if (list) && !(maxx<x || minx>x+w || maxy<y || miny>y+h))
30.           send list.Draw(drawingarea,mode,x,y,w,h);
31.        if (next) send next.Draw(drawingarea, mode,x,y,w,h);
32.     }
33.  Endobj  PROVINCE;
```

The MUNICIPALITY object has exactly the same Draw action as the PROVINCE object. It sends the message to its first CHAIN_CONTAINER object and also to the next municipality.

The CHAIN_CONTAINER object receives the Draw message from its parent municipality and the Draw action at line 65 is activated. In this case the Draw message is not a result of a normal *send* statement to the containing chains and next container, but it is *delegated* to them [10, 31, 50]. This is a good example of the power of delegation. Because delegation is used, receivers of the Draw message, issued at line 67 and 69, think that the MUNICIPALITY object is the sender, instead of *chain_container*, which is the actual sender. This use of delegation is necessary to avoid drawing the map twice. As each chain has two parents, it receives from both of them the Draw message. However, only the message received from the left municipality is used to actually draw the chain. In order to achieve this, the CHAIN object will inspect whether the sending object is the same object it assumes to be its left parent (remember that a chain has a left parent, and a right parent). If this is the case, the chain gets drawn, else the message is ignored.

```
54.  Obj      CHAIN_CONTAINER
55.  Declare  ....
56.           object CHAIN           element[MAX_ELT];
57.           object CHAIN_CONTAINER next;
58.           object WINDOW          drawingarea;
59.           int last, mode, i, MAX_ELT;
60.           int x, y, w, h;
61.  Protocol .... +
62.           ANY(drawingarea, mode, x, y, w ,h) -> Draw +
63.  Init ....
64.  Actions
65.     Draw = {
66.        for (i=0;i<last;++i)
67.           @element[i].Draw(drawingarea,mode,x,y,w,h);
68.        if (last == max)
69.           @next.Draw(drawingarea,mode,sender,x,y ,w,h);
70.     }
71.  Endobj   CHAIN_CONTAINER;
```

The CHAIN object receives the Draw message and the Draw action at line 89 is subsequently activated. The sender is compared with its left parent. If they are identical, i.e. have the same id, the chain is drawn in the selected mode. The modes are: draw the country border, draw the province border and draw all borders. At lines 76 through 80 the function to draw the province borders is declared. At line 77 and 78 the left and right parent are asked to give their province id[4]. If the left parent and the right parent do not have the same id, the chain is drawn. At line 81 through 85 the function to draw the country border is declared. At line 85 and 86 the id of the left and right parent are requested. If at least one of them is the special municipality with id equal to 0, the chain is drawn. The municipality with id equal to 0 represents foreign countries and water.

[4] The province id is represented by an integer, and decided upon by the supplier of the map. The province id is not equal to the object id of the corresponding PROVINCE object.

```
Obj     CHAIN(object MUNICIPALITY m_left, object MUNICIPALITY m_right, ....)
Declare object WINDOW drawingarea;
        int left, right, x, y, w, h, mode;;
        draw_chain() {....}
        draw_prov() {
           request m_left.GetProvID() -> (left);
           request m_right.GetProvID() -> (right);
           if (left != right) draw_chain();
        }
        draw_country() {
           request m_left.GetID() -> (left);
           request m_right.GetID() -> (right);
           if (left==0 || right==0) draw_chain();
        }
Protocol  .... +
          ANY(drawingarea,mode, x, y, w, h) -> Draw
Actions
   Draw   = { if (sender == left) {
                 switch(mode) {
                    case 1: draw_country(); break;
                    case 2: draw_prov(); break;
                    case 3: draw_chain();
                  }
               }
   }
Endobj  CHAIN;
```

The translation of the binary data

Our approach of translating input data into Procol objects, still without considering persistence, is a top-down approach of constructing the administrative hierarchy. First the COUNTRY object is created. This object forms the root of the hierarchy. After the creation of the COUNTRY object the PROVINCE, MUNICIPALITY, and CHAIN objects are created at line 9 through 11 respectively, by the objects PROV_READ, MUN_READ and CHAIN_READ. The object CHAIN_READ has two scaling parameters for the geometric data. Note that there is only one action that can be executed by the READ object. The action is the GetCountry action, and is assumed to be called by the creator of the READ object instance only. In this example, the creator is equal to the GIS object described before. The only thing the action does, is to return the id of the COUNTRY object recently created (in the Init section).

```
1.   Obj      READ
2.   Declare  include "gis.h"
3.            object PROV_READ    prov_read;
4.            object MUN_READ     mun_read;
5.            object CHAIN_READ   chain_read;
6.            object COUNTRY      country;
7.   Protocol creator -> GetCountry
8.   Init     new country;
9.            new prov_read(country);
10.           new mun_read(country);
11.           new chain_read(595, 700, country);
12.  Actions  GetCountry = { send sender.(country); }
13.  Endobj   READ;
```

An impression of how the binary map data is read by the MUN_READ object is given in the next code fragment, assuming the PROVINCE and COUNTRY object already exist.

```
1.    Obj     MUN_READ(object COUNTRY country)
2.    Declare include "gis.h";
3.            object MUNICIPALITY municipality;
4.            object PROVINCE     province;
5.            int p_number, m_number;
6.            string name[20];
7.            start_municipality_read() { .... }
8.            last_municipality() { .... }
9.            read_next_municipality_number() { .... }
10.           ready_municipality_read() { .... }
11.   Init    start_municipality_reading();
12.           while(!last_municipality()) {
13.               read_next_municipality_number(&m_number, &p_number, name);
14.               request country.get_province(p_number) -> (province);
15.               new municipality(m_number, name, province);
16.           }
17.           ready_municipality_read();
18.   Endobj  MUN_READ;
```

At line 12 through 16 the municipalities are processed. The next municipality is read at line 13, returning the municipality id, the Province id and the name of the municipality. At line 14 the parent `PROVINCE` object is requested from the `COUNTRY` object. This `PROVINCE` object is needed as an attribute of the `MUNICIPALITY` object. The `MUNICIPALITY` object is created at line 15.

When each of the map objects is created, it inserts itself into its corresponding parent's child-list. These insertions are activated in the `Init` section of the objects. The lists are implemented as linked lists. The parent has a reference to its first child, and that child has a reference to the next child. This implementation does not work for the `CHAIN` objects because each chain has two parents. So a `CHAIN_CONTAINER` object is implemented in which a number of chains can be stored. These `CHAIN_CONTAINER` objects are linked together in the same way as the provinces and the municipalities.

The modification with persistent objects

As stated in this article the only needed changes in code should be:

- To create a persistent object the statement **persistent <objectname> in <dataset>** has to be inserted after the **new <objectname>** statement.
- Instead of creating the whole hierarchical administrative unit structure, the statement **retrieve country from "dataset"** has to be added.

The first step taken in the process of modifying our GIS into a persistent one was to split up the GIS and the data reading part into two separate programs. The final GIS program contains no read or write statements; it only refers to persistent objects by retrieving them from a dataset. As an aside, by splitting our GIS into two separate applications we show that persistent objects can be shared and transported between different applications. Figure 9 shows the design of the resulting programs.

The data translation program creates a dataset for every object type in the map structure; this shows that references to persistent objects can extend to datasets other than the one where the reference actually takes place.

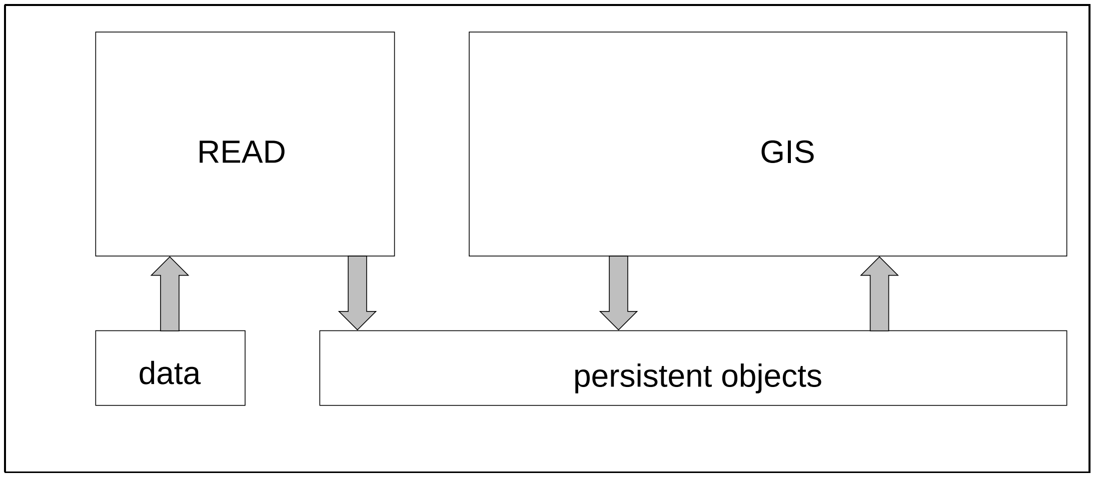

Figure 9: *Design of the example GIS.*

The resulting persistent data structure has the same hierarchical structure as the data structure in the non-persistent version. It has one single root, the country. The country has references to all existing provinces, and these provinces contain references to all municipalities. This type of data structure can be identified by one single reference root. Retrieval of the data structure is simple, as one only has to get hold of the root of the structure, in our case the `COUNTRY` object.

It is not a prerequisite of data structures that they should be hierarchical to be used in this persistent manner. The data structure itself may be cyclic, and the data structure may well have the form of a graph. Of importance is that we need to be able to identify a certain entrance to the data structure. In this example, the entrance is formed by the `COUNTRY` object. As it is the only instance of its type it is possible to retrieve it from the `"holland.pp"` dataset uniquely. It would have been possible to also define an extra entrance in the form of, say, an index structure on the name of the municipalities. This would have enabled the GIS to make selections based on municipality name and highlight the corresponding municipality on the map, for instance. This index structure could have been retrieved from a dataset based on its type, in the same manner we retrieved the country. In this case we would have had two entrances to the same data structure, and different interpretations and usage of the data structure could be envisioned. In the GIS context, two different indexing techniques are important. The first common search operation is based on geometric properties (e.g. by using an R-tree or our hierarchical bounding boxes approach). The second common search operation is based upon topologic properties of objects. A typical example of the type of queries involved is “give all municipalities that are adjacent to the current municipality”.

Retrieving persistent objects from a dataset

We saw that the data structure is retrieved from its persistent storage by retrieving the `COUNTRY` object instance (the root of the data structure) from its corresponding dataset (i.e. `"holland.pp"`). The assumption is that the dataset contains only one instance of the type `COUNTRY`, and issuing the `retrieve` primitive will produce the one and only `COUNTRY` object instance forming the root of our data structure. If the dataset contains more than one instance, one amongst them is returned (in practice the oldest one). If wanted, the `next` primitive can be used to retrieve the other `COUNTRY` objects from the dataset. Retrieving the country is the only thing the `GIS` object has to do to retrieve the data structure from its persistent storage. That is because the COUNTRY object contains references to a number of persistent objects. While accessing these references, for instance while drawing the map, the Procol runtime system will detect that these references are references to persistent objects. The corresponding datasets are then mapped to main memory (by the Operating System) and the persistent objects are automatically loaded. After retrieving the country, the GIS can send it messages, and the COUNTRY object can never determine whether it was created in the same program run as the GIS or that it has been retrieved at a later stage.

Only two lines of code need to be changed in the GIS object. The non-persistent version of the GIS reads in the country by creating a special READ object instance, shown in the following code segment.

```
1.   Obj     GIS
..
38.          new read;
39.          request read.GetCountry() -> (country);
...
```

The persistent version of the GIS, retrieves the COUNTRY object instance from a dataset, that is all that needs to be done to make the GIS persistent.

```
1.   Obj     GIS
..
38.          retrieve country from "holland.pp";
...
```

Of course, the objects created by the READ object need to be made persistent and stored in a dataset, before they can be retrieved in the manner shown above. To begin with, the READ object will not only create the COUNTRY object instance, but will also make it persistent. This is done at line 8 in the following segment.

```
1.   Obj      READ
2.   Declare  ....
..
7.   Protocol ....
8.   Init     new country; persistent country in "holland.pp";
9.            ....
12.  Actions  ....
13.  Endobj   READ;
```

Furthermore, the objects PROV_READ, MUN_READ, CHAIN and CHAIN_READ will also have to make the objects they create persistent. They therefore also contain persistent statements similar to the one descibe dat line 8. Summarizing, to make the non-persistent GIS persistent, one *retrieve* statement and five *persistent* statements suffice.

Practical problems

In the current implementation of the Procol compiler, it is not possible to use variable size objects. The size of objects is static for every instance of a specific object type, hence the size of object instances may not change after creation. In our case we have arrays of boundary coordinates, normally having different sizes. It is obvious that the linked list approach to implement arrays adds a lot of overhead in general. In our case we created an object with a static array large enough to contain the largest chain, because the maximum number of coordinate pairs of the chains was not very large. A flexible solution to this problem when dealing with large array's is to create an array object with a small static array, and link these objects together when the number of elements exceeds the size of the static array.

The natural place to link objects in a data structure together is in the `Init` section of the child object, as done in our non-persistent GIS. This is not possible when dealing with persistent objects. When an object is made persistent, its id is changed. This id change is not known by objects already referencing to this object; referential integrity can no longer be guaranteed. To avoid dangling references, all references to an object should be made after the object is made persistent. A more drastic solution is the introduction of *deep persistence*; i.e. when an object is made persistent, all its child objects are made persistent too. However, this method prevents the possibility of using volatile and persistent objects in one hierarchical data structure.

Another aspect that became apparent was an dataset management problem. The datasets are all registered in a dataset table (a kind of "yellow pages"). Each table entry contains a dataset number, and full path and filename where the dataset can be found. In this way it is possible to create two datasets with the same name in different directories. When more than one dataset is used to store the objects, the reference to an object instance which is stored in a different dataset, contains also the dataset-number of the dataset where that specific object instance is stored. Using this number, the dataset-table is consulted and the location of the dataset is extracted from there. But, when the datasets are copied or moved to another directory, the location in the dataset-table does no longer match the new location of this dataset. This problem arises because the user can move, copy, or delete datasets in the normal Unix shell. If the Procol runtime system would manage these tasks, like most relational database systems do, this problem would not occur. This runtime system is currently being developed.

Persistent program behavior

The translation program and the GIS application have been implemented and timed with `/bin/time` on a Sun Sparcstation IPC, with SunOS Release 4.1.1. and Sun OpenWindows. The different timings are summarized in Figure 10. Timing 1 shows the time needed by a non-persistent version of the GIS to start, to convert the map data into objects, to draw the entire map, and to immediately stop. Timing 2 indicates the time needed to start a persistent objects based GIS, to draw the entire map, and again to immediately stop. Of importance here is the difference between 1 and 2. Timing 3 refers to converting the map from binary data into Procol objects without making them persistent. Timing 4 refers to converting the map from binary data into persistent objects. Timing 5 show the time needed to start the GIS and without drawing the map, to stop immediately. The GIS uses persistent objects in this case. Timing 6 indicates the time needed to draw the entire map (excluding the time needed to start and stop the GIS). Finally, timing 7 indicates the time needed by the Procol run-time system to entirely retrieve the map from its persistent storage into main-memory.

These figures are shown to indicate the dramatic difference between timing 1 and 2. The difference is mainly introduced by the time needed to convert the (relatively unstructured)

```
 1.   40.1 seconds   read (start,convert,draw,stop)
 2.    4.7 seconds   persistent (start,retrieve,draw,stop)

 3.   36.4 seconds   read (into volatile objects)
 4.  425.6 seconds   read (into persistent objects)
 5.    0.8 seconds   start, stop
 6.    2.9 seconds   only draw
 7.    1.0 seconds   retrieve

calculation of 1 and 2:   1 = 3+5+6,    2 = 5+6+7.
```

Figure 10: *Timing numbers for the GIS using read and write statements versus persistent objects. Timing 1 is for a GIS with read and write statements. Timing 2 is for a GIS based upon persistent objects. Timing 3 to 7 indicate timing figures for individual steps in the process.*

data into a structured set of objects. As one can notice, converting these objects into persistent objects takes even more time in the current implementation, as the process of translating volatile objects into persistent objects is not yet fully optimized. Furthermore, the timing for 4 can be seen as the once-only time needed to build the database. After building the database (i.e. the cooperating set of persistent objects), it can be used in a relatively efficient manner (see timing 2).

Of importance here is that applications based upon persistent objects do not contain read or write statements to load or save objects. Furthermore, the data file that serves as input to the application need not be parsed (as done in timing 1 in our example). This is because the persistent objects contain references to each other and the inherent structure in the datafile has been determined before and translated into persistent object references.

Although speed increases when introducing persistent objects, the size of the datasets is, as a drawback, increasing too. This interchange of speed and size is not specific to persistent objects, it is, for example, a general problem for compilers; code is optimized for speed or size. In our case, the size of the map data in binary format is approximately 4 times smaller compared to the size of the datasets containing the persistent objects of our map data structure. This can become a problem when dealing with very large quantities of binary map data.

Further Research

Besides being an object-oriented programming language, Procol is also a parallel programming language. It is possible that objects run in parallel on multiple processors. We have only used this in a few examples. Clearly, this topic deserves more attention and more research is needed in the context of highly interactive and graphical systems.

It is a small step from one single user Procol program with persistent objects to a system with multiple users, at least conceptually, because each object has its own protocol which regulates the communication. It should not matter from which program a message originates. However, we will have to reconsider some of the concepts.

The provided query facilities are very limited; with *retrieve* it is only possible to get hold of one “starting” object id of a specified type or to perform the selection of instances from one set (object type) at a time. More complex queries have to be programmed into the objects (the Procol program). Attention has to be paid to avoid object types becoming too specific. That is in contradiction with one of the basic principles of databases of data being independent of applications. More research in this area is necessary.

References

[1] D.J. Abel. SIRO-DBMS: A database tool-kit for Geographical Information Systems. *International Journal of Geographical Information Systems*, 3(2):103–116, 1989.

[2] Malcolm P. Atkinson and O. Peter Buneman. Types and persistence in database programming languages. *ACM Computing Surveys*, 19(2):105–190, June 1987.

[3] M.P. Atkinson, P.J. Bailey, K.J. Chisholm, P.W. Cockshott, and R. Morrison. An approach to persistent programming. *The Computer Journal*, 26(4):360–365, 1983.

[4] F. Bancilhon, G. Barbedette, V. Benzaken, C. Delobel, S. Gamerman, C. Lécluse, P. Pfeffer, P. Richard, and F. Velez. The design and implementation of O_2, an object-oriented database system. In *Advances in Object-Oriented Database Systems, 2nd International Workshop on Object-Oriented Database Systems, Bad Münster am Stein-Ebernburg, FRG*, pages 1–22, September 1988.

[5] R. Bayer and E. McCreight. Organization and maintenance of large ordered indexes. *Acta Informatica*, 1:173–189, 1973.

[6] K. Bennis, B. David, I. Morize-Quilio, J.M. Thévenin, and Y. Viémont. GéoGraph: A topological storage model for extensible GIS. In *Auto-Carto 10, Baltimore*, pages 349–367, March 1991.

[7] K. Bennis, B. David, I. Quilio, and Y. Viémont. GéoTropics database support alternatives for geographic applications. In *4th International Symposium on Spatial Data Handling, Zürich, Switzerland*, pages 599–610, July 1990.

[8] Jon Louis Bentley. Multidimensional binary search trees used for associative searching. *Communications of the ACM*, 18(9):509–517, September 1975.

[9] E.H. Blake and S. Cook. On including part hierarchies in object-oriented languages, with an implementation in Smalltalk. In *ECOOP '87*, pages 41–50, 1987.

[10] Jan van den Bos and Chris Laffra. "Procol - An Object Language with Protocols, Delegation and Con- straints", *Acta Informatica*, 28:511–538, August, 1991.

[11] Arthur Chance and Richard G. Newell and David G. Theriault. An Object-Oriented GIS – Issues and Solutions. In *Proceedings EGIS '91, Second European Conference on Geographical Information Systems*, pages 179–188, April 1991.

[12] Peter Pin-Shan Chen. The entity-relationship model – Toward a unified view of data. *ACM Transactions on Database Systems*, 1(1):9–36, March 1976.

[13] E.F. Codd. A relational model of data for large shared data banks. *Communications of the ACM*, 13(6):377–387, June 1970.

[14] Douglas Comer. The ubiquitous B-Tree. *ACM Computing Surveys*, 11(2):121–137, June 1979.

[15] Brad J. Cox. *Object-Oriented Programming – An Evolutionary Approach*. Addison-Wesley, Reading, Mass., 1986.

[16] Walter C. Dietrich, Jr., Lee R. Nackman, Christine J. Sundaresan, and Franklin Gracer. TGMS: An object-oriented system for programming geometry. *Software – Practice and Experience*, 19(10):979–1013, October 1989.

[17] Max Egenhofer and Andrew Frank. Panda: An extensible DBMS supporting object-oriented software techniques. In *Database Systems in Office, Engineering, and Scientific Environment, New York*, March 1989.

[18] Max J. Egenhofer and Andrew U. Frank. Object-oriented modeling in GIS: Inheritance and Propagation. In *Auto-Carto 9, Baltimore*, pages 588–598, April 1989.

[19] ESRI. ARC/INFO – The Georelational Model Revisited. *ARC News Winter*, page 9, 1989.

[20] Christos Faloutsos, Timos Sellis, and Nick Roussopoulos. Analysis of object oriented spatial access methods. *ACM SIGMOD*, 16(3):426–439, December 1987.

[21] Andrew U. Frank. Multiple inheritance and genericity for the integration of a database management system in an object-oriented approach. In *Advances in Object-Oriented Database Systems, 2nd International Workshop on Object-Oriented Database Systems, Bad Münster am Stein-Ebernburg, FRG*, pages 268–273, September 1988.

[22] Andrew U. Frank and Renato Barrera. The Field-tree: A data structure for Geographic Information System. In *Symposium on the Design and Implementation of Large Spatial Databases, Santa Barbara, California*, pages 29–44. Lecture Notes in Computer Science 409, Springer Verlag, July 1989.

[23] Diane Greene. An implementation and performance analysis of spatial data access methods. In *Proceedings IEEE Fifth International Conference on Data Engineering, Los Angeles, California*, pages 606–615, February 1989.

[24] Oliver Günther. *Efficient Structures for Geometric Data Management*. Number 337 in Lecture Notes in Computer Science. Springer-Verlag, Berlin, 1988.

[25] Antonin Guttman. R-trees: A dynamic index structure for spatial searching. *ACM SIGMOD*, 13:47–57, 1984.

[26] John R. Herring. TIGRIS: Topologically integrated Geographic Information System. In *Auto-Carto 8*, pages 282–291, 1987.

[27] Independent European Programme Group – Technical Area 13 (IEPG TA-13). PCTE$^+$ C Functional Specification Issue 2, July 1988.

[28] Independent European Programme Group – Technical Area 13 (IEPG TA-13). Introducing PCTE$^+$, 1989.

[29] Intergraph. MGE – The modular GIS environment, July 1990. Brochure.

[30] Setrag N. Khoshafian and George P. Copeland. Object identity. In *OOPSLA'86*, pages 406–416, September 1986.

[31] Chris Laffra. *Procol, an object language with protocols, delegation, persistence and constraints*. PhD-dissertation, Erasmus University Rotterdam, planned in 1992.

[32] Chris Laffra and Jan van den Bos. "A layered object-oriented model for interaction". In *Advances in Object-Oriented Graphics I*, Proceedings of the first Eurographics workshop on object-oriented graphics, Königswinter, Germany, Springer, 1990.

[33] Chris Laffra and Peter van Oosterom. Persistent graphical objects. In *Advances in Object-Oriented Graphics I*, Proceedings of the first Eurographics workshop on object-oriented graphics, Königswinter, Germany, Springer, 1990.

[34] Bertrand Meyer. *Object-oriented Software Construction*. Prentice Hall, London, 1988.

[35] Scott Morehouse. The architecture of Arc/Info. In *Auto-Carto 9, Baltimore*, pages 266–277, April 1989.

[36] R. Morrison, A.L. Florianis, A. Dearle, and M.P. Atkinson. An integrated graphics programming environment. *Computer Graphics Forum*, 5:147–157, 1986.

[37] Norman W. Paton and Peter M.D. Gray. Identification of database objects by key. In *Advances in Object-Oriented Database Systems, 2nd International Workshop on Object-Oriented Database Systems, Bad Münster am Stein-Ebernburg, FRG*, pages 280–285, September 1988.

[38] Lars-Ole Pedersen and Richard Spooner. Data organization in system 9. Technical report, WILD Heerbrugg, 1988.

[39] D. Jason Penney and Jacob Stein. Class modification in the GemStone Object-Oriented DBMS. In *OOPSLA'87*, pages 111–117, September 1987.

[40] Joel E. Richardson and Michael J. Carey. Persistence in the E language: Issues and implementation. *Software – Practice and Experience*, 19(12):1115–1150, December 1989.

[41] Hanan Samet. The quadtree and related hierarchical data structures. *Computing Surveys*, 16(2):187–260, June 1984.

[42] Matthaeus Schilcher. Interactive graphic data processing in cartography. *Computers & Graphics*, 9(1):57–66, 1985.

[43] Siemens Data Systems Division. SICAD the geographical information system for modern mapping, August 1987. Brochure.

[44] Michael Stonebraker and Lawrence A. Rowe. The design of Postgres. *ACM SIGMOD*, 15(2):340–355, 1986.

[45] Andrew Straw, Fred Mellender, and Steve Riegel. Object management in a persistent Smalltalk system. *Software – Practice and Experience*, 19(8):719–737, August 1989.

[46] Sun Microsystems, Inc. SunINGRES Manual Set, January 1987.

[47] D.C. Tsichritzis and O.M. Nierstrasz. Fitting round objects into square databases. In *ECOOP '88*, pages 283–299, August 1988.

[48] UNISYS. Open geographic information systems forum, September 1989. Brochure.

[49] Jan van den Bos. Procol – A protocol-constrained concurrent object-oriented language. *Information Processing Letters*, 32:221–227, September 1989.

[50] Jan van den Bos and Chris Laffra. Procol – A parallel object language with protocols. In *OOPSLA '89, New Orleans*, pages 95–102, October 1989.

[51] Peter van Oosterom. Spatial data structures in Geographic Information Systems. In *NCGA's Mapping and Geographic Information Systems, Orlando, Florida*, pages 104–118, September 1988.

[52] Peter van Oosterom. A reactive data structure for Geographic Information Systems. In *Auto-Carto 9, Baltimore*, pages 665–674, April 1989.

[53] Peter van Oosterom. *Reactive Data Structures for Geographic Information Systems*. PhD thesis, Department of Computer Science, Leiden University, December 1990.

[54] Peter van Oosterom and Eric Claassen. Orientation insensitive indexing methods for geometric objects. In *4th International Symposium on Spatial Data Handling, Zürich, Switzerland*, pages 1016–1029, July 1990.

[55] Peter van Oosterom and Chris Laffra. "Persistent graphical objects in Procol", In *TOOLS 2*, the *TOOLS '90* proceedings, pp. 271–283, Paris, June 1990.

[56] Peter van Oosterom and Jan van den Bos. An object-oriented approach to the design of Geographic Information Systems. *Computers & Graphics*, 13(4):409–418, 1989.

[57] Peter van Oosterom, Marcel van Hekken, and Marco Woestenburg. A geographic extension to the relational data model. In *Geo '89 Symposium, The Hague*, pages 319–333, October 1989.

[58] Peter van Oosterom and Tom Vijlbrief. Building a GIS on top of the open DBMS "Postgres". In *Proceedings EGIS '91, Second European Conference on Geographical Information Systems*, pages 775–787, April 1991.

[59] J.W. van Roessel. Design of a spatial data structure using the relational normal forms. *International Journal of Geographical Information Systems*, 1(1):33–50, 1987.

[60] T.C. Waugh and R.G. Healey. The GEOVIEW design. A relational data base approach to geographical data handling. *International Journal of Geographical Information Systems*, 1(2):101–118, 1987.

[61] André Weinand, Erich Gamma, and Rudolf Marty. Design and implementation of ET^{++}, a seamless object-oriented application framework. *Structured Programming*, 10(2):63–87, 1989.

[62] Andreas Wolf. The DASDBS GEO-Kernel, concepts, experiences, and the second step. In *Symposium on the Design and Implementation of Large Spatial Databases, Santa Barbara, California*, pages 67–88. Lecture Notes in Computer Science 409, Springer Verlag, July 1989.