Radial Categories Applied to Object-Oriented Modeling — A Case Study for a Property Registration System[1]

Andrew U. Frank
Dept. of Geoinformation
Technical University Vienna
frank@geoinfo.tuwien.ac.at

ABSTRACT

Design of GIS applications must model current systems, e.g. for administrative and planning use, and include their legal aspects. The standard concentration on the static data structure in GIS — as captured with database design tools — is insufficient and an object-oriented approach is necessary to include the operations carried out. The combination of operations and data structure in a modeling tool is crucial to properly capture complex administrative or legal processing of documents, including the temporal aspects. This is clearly visible in the application selected here, namely real estate property registration, where one cannot succeed without modeling space and time.

Object-Oriented Modeling is *the* widely acclaimed method in software engineering. It is used for design and programming, but did not reduce the software crisis. The theory of object-orientation is simple, but the application to practical cases points to more difficulties than expected. The primary concept of object-orientation corresponds well with cognitive principles, but the details of practical object-oriented languages add confusion.

A non-trivial case study is used here to point out the issues and to propose an approach which is in line with human cognition. First the object-oriented concepts are revisited in the context of a class based functional programming language. It stresses the separation of behavior from implementation inheritance and allows design and rapid prototyping in the same language.

Human cognition uses radial categories with a central prototype. This is used to start the design of the property registry. Several refinement steps are then applied. This approach and the set of tools used allows to separate the code of refinement steps and result in very compact, formally checked code.

1. INTRODUCTION

The design of GIS applications is difficult, more difficult than other applications. It is generally thought that information systems about the real world (Shaw 1984), which represent space and objects in space and must related objects and events to time (Bobrow, Mittal, and Stefik 1986) are more difficult to design than others. GIS and most similar applications, for planning in the urban and regional environment and for town and public utilities application, fall in this category. This is corroborated by the difficulties application developers encounter. In this paper we use the design of a property registration system to make evident some of the lessons we have learned.

Object-oriented methods for software engineering are crucial for the design of GIS (Egenhofer and Frank 1987),(Worboys 1994). Spatial information systems deal with many objects which have somewhat similar properties, e.g. their existence in space and time, and the object-oriented method allows to use these similarities in the design process to make the design more regular and much smaller.

For application development, many tools are available but their power is limited and the software crisis continues, as the cover story from a popular journal recently indicated (Joch 1995). GIS

---

[1] Frank, A.U. 1996. "Radial Categories Applied to Object-Oriented Modeling: A Case Study for a Property Registration System". In Proceedings of *1st Int. Conference on Geographic Information Systems in Urban, Regional and Environmental Planning*, (Sellis, T., & Georgoulis, D., eds.), in Island of Samos, Greece (April 19-21, 1996), pp: 178-187.

application development uses often tools based on database engines (e.g. Oracles, Sybase). Tools for overall design (e.g. the OMT or OA method) are used by more experienced software engineering companies for large projects. Rapid prototyping intends to produce demonstration version of an application quickly. Substantive research has explored formal methods to support specification, but with limited practicability so far (Guttag, Horning, and Wing 1985, Guttag, Horowitz, and Musser 1978, Liskov and Guttag 1986).

Tools often concentrate on one or the other of the two aspects of object-oriented design, namely

- behavioral object-orientation of the design languages, and
- implementation object-orientation of the programming languages.

This makes translation between them difficult. Even at the level of programming languages, the object-oriented concepts differ, such that integration is difficult.

Recently, a substantive experiment to store cartographic data in an object-oriented database was undertaken, using one of the best researched object-oriented database management system (Bancilhon, Delobel, and Kanellakis 1992). Later, a port to another object-oriented database management system became necessary. The code was written in C++ and needed minimal change, "only" the description of the objects had to be translated to another object-oriented data description languages. This was surprisingly difficult and 9 month of work produced a version which compiled, but did not run properly.

In the course of a project to design a cadastral system for a country, we used object-oriented methods to analyze the database design and to assess different legal alternatives for the organization of a property registration system. From this effort and previous similar studies, some lessons for the use of object-oriented methods are reported here. The detail of the property registry in this example is kept to the level where it is independent of a particular legal system. The designed system reproduces the history of the parcel. The interpretation of this history to determine who owns a parcel requires to a legal interpretation which has different answers in different legal systems.

Design of an application is understanding a complex cognitive reality. Society has constructed a highly structured network of notions, rules and effects, e.g. to deal with real estate. These construction, in the legal, social, administrative domain do not follow standard mathematical logic, as little as does natural language. The tools provided to the application designer must correspond as much as possible to the cognitive structure to make his effort of translation simpler. The impedance mismatch caused by tools following a strict logic but conflict with elementary concepts of cognition aggravates his already difficult task.

Radial categories are a way to understand how humans build categories, where some prototypes are more typical members of a category then others (a robin is "more a bird" than a penguin). We propose to approach application design by identifying, designing and quickly test the prototypical case. From this refinement steps start. The changes for these were local, demonstrating the power of the approach used and the orthogonality of the tools.

2. OBJECTS = OPERATIONS + DATA

The basic notion of object-oriented design is the object, which has a specific set of properties and to which some specific operations can be applied. An object represents some real world object in the application domain. Object is a notion which is based on direct first hand and very extensive experience: human constantly interact with physical small objects (Lakoff 1987). From this experience we metaphorically transform (Lakoff and Johnson 1980) the notion of object to the non-physical domain, including the legal: the mortgage right is construed as an object and has many of the properties (even legal properties) of physical objects (Frank 1996).

The design of application is concerned with classes of objects which have the same behavior. Application designers talk about parcels, not a single parcel. In consequence the term 'object' is often used loosely to denote an object class or object type; to denote a single object, one speaks of an 'instance' or 'occurrence' of an object.

We propose to use nouns in the plural form (and capitalized) to denote object class. The plural form is the construct in natural language to transform a single object in an undetermined mass of similar objects (Langacker 1987).

2.2. Object classes

Object classes are formed following set logic: all objects which show the same behavior belong to a class. Objects may have an internal state, but modeled at the behavioral level are only the operations and their effects.

```
class Deeds d where                    -- the object d of the class of deeds
     draft :: ParcelID -> d ->  d      -- operations with their argument types
     sign :: Person -> Person -> Time -> d -> d
     signed, registered :: d -> Time
     register :: Time -> d ->  d
     isSigned, isRegistered :: d -> Bool
      seller, buyer :: d -> Person
      parcelAffected :: d -> ParcelID
```

The functional programming language used here is Gofer (Jones 1991, Jones 1994), very similar to Haskell (Hudak *et al.* 1992).

2.1. Categories in human cognition

Humans organize the real world with cognitive categories; these are not mathematical sets. Cognitive science has shown that human cognition follows a radial categories structure, where a central prototypical example characterizes object of this category best (Lakoff 1987). For birds, a robin is a "better" bird than a penguin or an ostrich. For the property registry, a building lot is a better example of a parcel than a small corner, owned by the telephone company to place its distribution equipment; a sales contract is the better example for the transfer of real estate property than a court order based on eminent domain.

2.6. Separate behavior, representation and implementation

Two tools must be available during design and implementation:

- The designer specifies object classes with operations.
- The implementor defines a data structure to hold the internal state and implements the operations as specified (using this data structure).

The major problem with many object-oriented tools is that these two aspects are not well separated and no specific constructions for one or the other are available within the same environment. “A class declaration specifies the representation of objects of the class and the set of operations that can be applied to such objects”(Ellis and Stroustrup 1990) and similarly for an object-oriented data model (Bancilhon, Delobel, and Kanellakis 1992, p. 77).

In contrast: In functional programming languages (Bird and Wadler 1989) of the Haskell tradition(Hudak *et al.* 1992), three notions are used:

class is used to describe an algebra without concern for the representation and implementation. A class describes only behavior.

data types describe representation.

instances of classes using data types give the implementation of the operations of a class for a particular data type. More than one implementation for a class is possible.

Behavior = Operations. The designer is concerned with the behavior. She needs to express, for example, that a deed must be signed before it can be registered or that it can be registered only once. This is best described in terms of the outcome of an operation applied to an object (or several operations applied in sequence). This is often called an axiomatic style of specification writing (Liskov, and Guttag 1986). It is based on the view of an object class as an multi-sorted (universal) algebra (Birkhoff and Lipson 1970). In an executable language, these axioms must be expressed in a constructive form, for example:

```
register d t r = if not isSigned d then  error "unsigned deed cannot be
registered"
                 else if isRegistered d then error "deed is already registered"
                      else store d t r
```

Representation = Data. Programmers are concerned how to represent the internal state of an object and how to implement the operations on this representation. Programming languages and database management system data description languages are typically geared towards this end.

Here data types will be capitalized nouns in the singular, single occurrences start with a lower case letter and are either proper names (e.g. *Andrew Frank*) or a possibly abbreviated noun followed by a number.

Example:

```
data Deed = Deed Person Person ParcelID Date Date
                 seller, buyer, parcel, date signed, date registered
aDraft, aSignedDeed :: Deed             -- two object of type Deed
andewFrank, peterMiller :: Person       -- two objects of type Person
aDraft = draft (ParcelID 7)
a signedDeed = sign aDraft andrewFrank peterMiller
               — a aDraft of a Deed is signed by the seller and the buyer
```

Implementation. The code to execute the operations is based on the representation. Operations, which do not depend on the representation can be expressed directly in the class, but at least the basic update and observer operations must be coded in an instantiation, which declares that object with this representation belong to this class:

```
instance Deeds Deed where
  draft pa = Deed unknownPerson unknownPerson pa NYT NYT unknown
  signDeed seller buyer time (Deed p1 p2 pa s r id) = Deed buyer seller pa time
r id
  register rt (Deed p1 p2 pa s r id)  = Deed p1 p2 pa s rt
  registered (Deed p1 p2 pa s r id)  = r
  isRegisered (Deed p1 p2 pa s r id) = not (r==NYT)   -- NYT = not yet
```

3. GENERALIZATION AND INHERITANCE

Inheritance is one of the key ideas in object-oriented design: Superclasses are formed from classes which show some communality: Parcels are one type of Areas, others are Forests, Cities, Lakes. The objects of the subclass have all the properties and operations of the superclass. This can be shown as a hierarchy and follows the famous hierarchical classification of the animal and plant kingdoms by Linnee.

In artificial intelligence, such generalization hierarchies are called ‘is_a’ hierarchies, because each object of the subclass is an object of the superclass: e.g. each dog is a mammal. But also two classes of objects can have the same operations, if they form a ‘part_of’ (or aggregation) relation. The assembly responds to the same operations than the part (a ‘pars pro toto’ schema): One starts a car by starting its engine.

A *deed* is registered in the registry by registering it in the book of deeds. In this case the data type *Registry* and *TheBooks* both respond to the operation *‘registerDeed’* and *historyOfParcel*. This implies that *Registry* and *TheBooks* both belong to the class of Registries.

```
class Registries r where
        registerDeed :: Deed -> r -> r
        ownerOfParcel :: ParcelID -> r -> Person
        parcelsOfOwner :: Person -> r -> [ParcelID]
        historyOfParcel :: ParcelID -> r -> [Deed]
        historyOfParcel’ :: (Deed -> Deed -> Bool) -> ParcelID -> r -> [Deed]
                                        -- accepts a sort order as a parameter
instance  Registries Registry where
        registerDeed deed (Registry now rs) =
                Registry (now’) (registerDeed (stamp now’ deed ) rs)
                        where now’ = tick now
        historyOfParcel p (Registry now rs) = historyOfParcel p rs

instance Registries TheBooks where
        registerDeed deed (TheBooks  ds) =  TheBooks (store3 putOID deed ds)
        historyOfParcel p (TheBooks ds) =  sort1 (<=) (filterR2 c ds )
                           where c d = (parcelAffected d) == p
```

Inheritance of behavior. In a class based object-oriented model, classes serve to describe abstract sets of objects which all have the same operations. Data types describe the objects and it is the programmer who links the two together, stating that objects of this data type are in this class (e.g. all instantiation for TheBooks are in the class Registries).

Inheritance of Implementation. From an implementation point of view objects which support the same operation must have the same representation. Therefore in programming languages objects of a subclass are constructed with the operations, representation and implementation of the superclass included.

Multiple inheritance permits that a subclass inherits properties from multiple superclasses. A navigable river inherits from ‘waterway’ and ‘river’. Such situations were assumed to be very important in GIS (Egenhofer, and Frank 1987), but in practice, the relations are seldom perceived in this form at the application level and it is the analyst who must discover them. They are more obvious

for the data processing aspects of a system: a *parcel* is a *TextObject* (which can be printed), is a *GeometricObject* (which can be graphically rendered), a *DatabaseObject* (which can be stored) etc.

3.1 Polymorphism

Polymorphism means that the same operation (i.e. a name) can be applied to different types of objects. The operation *registerDeed* can be applied to *Registry* or to *TheBooks* having different effects in each case. If all objects to which an operation can be applied result from a general type and are produced by different parameter values, then this polymorphism is called parametric polymorphism (Cardelli and Wegner 1985). The *Registry* and *TheBooks* are both produced from the class *Registries*, replacing the parameter r. Ad hoc polymorphism allows the application of operations on a number of types, which are not otherwise related.

Parametric polymorphism as used here forces the designers to search for the common superclass if they intend to use the same operation name. This helps to detect commonalties and leads to more similarities in the design, more coherence in the user interface and less code.

C++ provides 'templates' to achieve the same effect, which is a domesticated form of the C macro facilities. "A template defines the layout and operations for an unbounded set of related types" (Ellis, and Stroustrup 1990, p. 341).

To construct classes for data structures and use them broadly, simple parameters are not sufficient and so called constructor classes are necessary. Haskell has recently added this concept [draft Haskell report 1.3], which was pioneered in Gofer (Jones 1991, Jones 1995). This is not used in the example here, but constructor classes were used to build the simple storage tools used for storing the deeds.

If a class is parametrized and for the operation specified another operation is required then the instantiation must check that the operation is available for the data type with which the class is instantiated. Such dependencies are often overlooked, e.g. the need to have an equality test for the individuals in order to build a set over them. In the formalism here, such dependencies are expressed in the class heading before a '=>' symbol.

4. APPROACH: PROTOTYPICAL CASE FIRST

4.1. How to Start?

The initial step in an object-oriented design of an application is often insurmountable: where should one commence? Two classical approaches are often propagated: top-down or bottom-up design. Neither is perfect, thus leading to a 'middle out' design philosophy as a compromise.

The bottom-up method leads to a design where the low-level parts of the application are elaborated before an overall picture what the application should achieve is gained. The top-down method has the tendency that a very complex picture of the application is developed which is so comprehensive that it cannot be implemented. Both methods are based on the assumption that an application is naturally structured in a hierarchy of task which are subdivided in subtasks. But this is not the common structure of radial categories human cognition follows: successive levels of subdivisions do not form hierarchies, but are independent partitions. A division of the scale 'good-bad' in three 'good-OK-bad' shows this clearly.

For designing the database schema of the property registration system we considered all the different aspects of the relations between persons and real estate property. Half a dozen different types of persons (natural person, estates, companies, public bodies etc.) can hold an equal number of different rights in real estate. This lead to a complex design from the middle out, where much revision was necessary as the top and bottom was reached.

These methods for modeling a property registration system failed not because it was not possible to model the aspect of the situation covered, but simply because the design document became to complicated. It became difficult to see how it would work. When we proceeded to implement the database schema we ended up with more than ten pages of unreadable (automatically produced) SQL code.

4.2. Prototype First

I propose to use the radial category concept to guide the approach to application design. Categories are organized around a prototype, which shows the properties best and less prototypical elements

deviate to some degree from the central prototype (a duck/or a penguin are birds, but less typical ones).

Humans form categories following their experience and interacting with the world. Repeated experience shapes our view of the world and leads to concepts (Neisser 1976). Our concepts follow therefore the more often encountered case and special cases are then dealt with in an *ad hoc* fashion.

Identifying the prototypical skeleton of an application reduces the clutter of the innumerable special cases and focus attention on the overall picture. It helps identify the basic entities involved and their properties. For the prototypical function of an application domain, the prototypical objects are often easy to identify and have good names. The nouns used in the description of the central operation become the classes, the verbs the operations.

A description of a property registry may read:

> The property registry registers documents about sales of parcels (deeds). In a deed a seller transfers ownership of a parcel to a buyer. The deed is registered with date and time received and an abstract is entered in the book. Persons can inquire about the history of a parcel and who (probably) owns it.

This is clearly only the most important aspect - there are many different methods of transferring a parcel and the determination of ownership is a much more subtler issue (Al-Taha 1992) then covered in this simple description, but it provides a good start.

4.3. Translation to a formal design

The informal description gives major clues for the formalization. The nouns indicate which classes will be needed and the verb indicate the operations. The nouns are: registry, sales, parcels, deed, seller, buyer, ownership, date and time, abstract, book, history of parcel, owner. The verbs are: register, transfer, inquire.

This can be grouped as

- registry with operations registerDeed, inquire history of parcel, inquire present owner
- person (buyer, seller)
- deed with operations sign, timeStamp, abstract,
- parcel with inquire about owner, inquire about history
- history of parcel

This leads to the classes Registries, Deeds (shown above), Persons, and Parcels. The later classes contain only operations for their display (not shown). History is a simple list of all Deeds registered for this Parcel.

The data types are:

```
data Person = Person String
data ParcelID = ParcelID Int
data Parcel = Parcel ParcelID
data Deed = Deed Person Person ParcelID Time Time OID
--     seller, buyer, parcel,  signing time, registration time,  register id
type DeedRel = Rel2 Deed     -- an indexed collection of Deeds
data TheBooks = TheBooks DeedRel
data Registry = Registry Time TheBooks
                      -- the Time field represents the wall clock of the registry
```

The implementation of the operations for the Registry and TheBooks were shown earlier.

In about 6 hours this prototype was designed and coded using a minimal data storage package - a total of 3 pages (most of it to create readable output). Test cases were created and run to demonstrated to the client (i.e. the domain specialist) and discussed. Feedback for correction was received quickly.

5. REFINEMENT STEPS

Several refinement steps are possible and they will lead the design in different directions. With the tools discussed, they can be carried out independently and the resulting changes in the code are local.

5.2. Refinement of object class without interference with remainder

For example, one can refine the person into different types of persons (companies, public entities, etc.). This creates only the more classes for specific persons and data definitions for their

representation. The interaction between the remainder of the registry is with the operations provided by the general class and does not depend on the particular type of person. Similarly, differentiation between different types of transfers (sales, gift, order of a court of law, inheritance etc.) does not affect other parts.

The persons were differentiated in natural persons and corporations with the code:

```
class NatPersons p where
        makeNatPers :: Name -> Time -> Int -> p
        birthDay :: p -> Time
        socSec :: p -> Int
class Corps p where
        makeCorp :: Name -> String -> p
        location :: p -> String

data Person = NatPerson Name Time Int |
              Corp Name String
instance Persons Person where
        name ( NatPerson n b s ) = n
        name ( Corp n s) = n
instance NatPersons Person where
        makeNatPers n b s = NatPerson n b s
        birthDay ( NatPerson p b s ) = b
        socSec ( NatPerson p b s ) = s
instance Corps Person where
        makeCorp n l = Corp n l
        location ( Corp n l) = l
```

The transfer of ownership was differentiated in deeds resulting from sales and court orders:

```
class Sales d where
        closeSale :: Person -> Person -> ParcelID  -> Time -> Money ->  d
        amount :: d ->Money

class CourtTransfer d where
        orderTransfer :: Person -> Person -> ParcelID  -> Time -> String ->  d
        court :: d -> String   -- the court which ordered
                               -- how to deal with registration time?
data Deed' = Sale Deed Money |
             CourtTransfer Deed String
```

This step adds bulk to the design documents without revealing much about the application. It is wise to include two or three different kinds from a general class to see how it interact with the rest of the design and to have examples for further expansion later.

5.3. Refinement with Observer operations

Additional operations can be added which do not require additional domain information; these are essentially 'views' in the database sense. For example, one can determine all the parcels owned by a person by computing the difference of all the parcels he bought and all he sold (without going into legal particulars). This does not affect the overall structure of the design but may help the application specialist to understand the prototype and point out shortcomings of the analysis.

```
instance Registries TheBooks where
     ...
    parcelsOfOwner n (TheBooks ds) = (bought n ds) \\ (sold n ds) -- set
difference
          where
           bought n ds = map parcelAffected (filterR2 buyerName   ds)
                         where buyerName  = (n ==) . buyer
           sold n ds = map parcelAffected (filterR2 sellerName   ds)
                         where sellerName  = (n ==) . seller
```

5.4. Refinement of the internal structure

Multiple internal structures represent possible trade-offs between storing information or recomputing it. In the traditional form of an application, using paper documents and index card files, the physical storage structure was important. In a perfect world, data should be stored free from redundancy and different reports and queries produced.

In the imperfect real world data includes errors and omissions which we cannot correct always and the applications may have evolved to accommodate such errors and become resilient. In property registers persons are represented by names (and sometimes address, birthday etc. but seldom are these available consistently). The fiction of database normalization rules that a person can be replaced in all tables with a key to the person tuple in a particular relation, may not capture the logic of the property

registry, where there is often no method to find out, if two names spelled slightly differently, mean the same person or not.

In property registers, indices are maintained to permit rapid access to the data, which is chronologically registered, typically one for person names and one for parcels. Such index structures help the designer to properly understand the application and the description given by the domain specialist. The design can be streamlined later, if it becomes clear that the traditional data structuring is not necessary.

For this change two newdata types for index books are added. One index is by parcels, listing all the deeds affecting this parcel and one indexed by person, showing all the deeds with which he acquired or sold a parcel.

```
data TheBooks = TheBooks DeedRel HoldingPerson HoldingParcel
instance Registries TheBooks where
        registerDeed deed (TheBooks  ds hs ps) =   TheBooks ds1 hs2 ps1
                     where ds1 = (store3 putOID deed ds)
                           hs1 = acquire (parcelAffected deed) (name (buyer
deed)) hs
                           hs2 = release (parcelAffected deed) (name (seller
deed)) hs1
                           ps1 = acquire (name (buyer deed)) (parcelAffected
deed) ps
        ownerOfParcel p (TheBooks  ds hs ps fs) = makePerson( head (findx p ps))
```

To store a deed it must be entered, once in the parcel index and twice in the name index (with the name of the seller and the buyer).

```
type HoldingRel n p  = Rel n [p]
type HoldingPerson = HoldingRel Name ParcelID

class Holdings v i s where
        acquire, release :: v -> i -> s -> s

instance Holdings ParcelID Name HoldingPerson where
        acquire p n hs = update (put p) n hs'
                                where hs' = if  (isInx n hs) then hs else store n
hs
        release p n hs = if isInx n hs then update (remove p) n hs else hs
                         -- if a parcel is sold by an unknown, it cannot be
updated

-- the list of parcels and the owners
type HoldingParcel = HoldingRel  ParcelID Name

instance Holdings Name ParcelID HoldingParcel where
        acquire p n hs = update (put p) n hs'
                                where hs' = if  (isInx n hs) then hs else store n
hs
```

This refinement step leads to questions about the operation of the registry: how to proceed with the registration of a deed when the seller does not correspond to the current owner? how to register a deed when the parcel is not known to the registry? These questions are relevant legal questions and have to be answered by the domain specialist; for property registration, the rules for the interpretation of the legal history of a parcel is typically found in real estate law.

5.5. Refinement of Data Flow

The flow of data in an organization is often crucial. The elaboration of some data for a step needs manpower and time and cannot therefore not proceeded synchronously with other the processing steps. Tasks which require this data must wait till it becomes available. The processing of data is not anymore ‘instantaneous’ but a process in time, which must be modeled; ‘database’ time becomes important (Snodgrass 1992).

Computational models of an organization which do not include the flow of information and its elaboration in time fail. In the case of the registry, customers apply for registration of deeds, sometimes in rapid succession. Clerks abstract carefully from the complicated contract text the necessary elements for registration and enter the essence in the registers. The effect of registration is dated back to the application date. It is thus necessary to timeStamp documents as they enter the registry and store them for later processing - and this must become part of the application model. It may become useful to keep track when a deed was actually processed (database time), but the order of processing does not matter, as always the order of registration is reestablished.

Assuming we do not maintain the indices of the previous refinement step, then the code is simply:

```
registerDeed deed (TheBooks  ds hs ps fs) =
                        TheBooks ds hs ps (store3 (putOID) deed fs)
processForms (TheBooks ds  fs) =  if null fs then (TheBooks ds fs)
                                  else (TheBooks ds1 fs1)
                          where deed = head fs
                                fs1 = tail fs
                                ds1 = (store3 putOID deed ds)
```

6. CONCLUSION

The design method shown here centers around the notion of 'prototypical case'. Prototypes and the corresponding radial categories are currently accepted as the way human form categories of the things in the world (Lakoff 1987). Application design and programming is modeling how humans or society understands the world (Watzlawick 1976). This is especially true for legal or administrative aspects which are often central in GIS applications. Property registration, the case studied here, is a good example.

In this paper some critical points of the today dominant object-oriented software engineering strategy are revisited and compared to the human cognitive abilities. It argues that the differences make the task of the designer of application software. This may be one of the reasons for the software crisis (Joch 1995). The current paradigm of object-oriented programming languages was contrasted with perspective of a designer. Class bases concepts of object-oriented, as found in modern functional programming languages seem to serve better.

Starting the application analysis and design is a crucial step. A method respecting the importance of prototypes for human cognition is presented. It starts with the most simple prototype, which is then refined. The tools presented support this method well. The prototype is then refined.

Refinement steps can refine the objects or the operations of the application, can introduce new operations or refine the inner organization of the work. Each refinement step is a few hours work and adds a page or so to the code. It can be tested immediately to see if it captures the intended semantics. The refinement step changed code only locally. This indicates that the approach to design and the tools used are appropriate.

It is important to read and re-read the code as it grows and in our experience about half of the time is spent in revisions of the code to maintain it readable. In particular, factoring out common parts of the code is a major method to detect communality in the tasks. Its effect is not only reduction of code, but much more important, conceptual clarity.

NOTE

The code is available Over anonymous Ftp (Ftp://www.....)

ACKNOWLEDGMENTS

REFERENCES

Al-Taha, Khaled. "Temporal Reasoning in Cadastral Systems." Ph.D., University of Maine, 1992.

Bancilhon, François, Delobel, Claue, and Kanellakis, Paris. *Building an Object-Oriented Database System - The Story of O_2*. San Mateo: Morgan Kaufmann, 1992.

Bird, Richard, and Wadler, Philip. *Introduction to Functional Programming*. 1989.

Birkhoff, G., and Lipson, J. D. "Heterogeneous Algebras." *Journal of Combinatorial Theory* 8 (1970): 115-133.

Bobrow, Daniel G., Mittal, Sanjay, and Stefik, Mark J. "Expert Systems: Perils and Promise." *Comm. of the ACM* 29 (9 1986): 880-894.

Cardelli, Luca, and Wegner, Peter. "On Understanding Types, Data Abstraction, and Polymorphism." *ACM Computing Surveys* 17 (4 1985): 471 - 522.

Egenhofer, Max J., and Frank, Andrew U. "Object-Oriented Databases: Database Requirements for GIS." In *International Geographic Information Systems (IGIS) Symposium: The Research Agenda in Crystal City, VA*, edited by Aangeenbrug, Robert T., and Schiffman, Yale M., NASA, 189 - 212, 1987.

Ellis, Margaret A., and Stroustrup, Bjarne. *The Annotated C++ Reference Manual*. Reading, Mass.: Addison-Wesley, 1990.

Frank, Andrew U. "The Prevalence of Objects with Sharp Boundaries in GIS." In *Geographic Objects with Indeterminate Boundaries*, ed. Burrough, P. A., and Frank, A. U. London: Taylor & Francis, 1996.

Guttag, J.V., Horning, J.J., and Wing, J.M. *Larch in Five Easy Pieces*. Digital Equipment Corporation, Systems Research Center, 1985. Report

Guttag, John V., Horowitz, Ellis, and Musser, David R. "Abstract Data Types and Software Validation." *Comm. ACM* 21 (12 1978): 1048-1064.

Hudak, P *et al.* . "Report on the functional programming language Haskell, Version 1.2." *SIGPLAN Notices* 27 (1992):

Joch, Alan. "How Software Doesn't Work." *Byte* 20 (12 (Dec.) 1995): 48-60.

Jones, Mark P. *An Introduction to Gofer*. Yale University, 1991. (available by anonymous ftp)

Jones, Mark P. *Gofer - Functional Programming Environment*. Geoinformation, TU Vienna, 1994. Script

Jones, Mark P. "Functional Programming with Overloading and Higher-Order Polymorphism." In *Advanced Functional Programmin*, ed. Jeuring, Johan, and Mejer, Erik. 331. 925. Berlin: Springer, 1995.

Lakoff, George. *Women, Fire, and Dangerous Things: What Categories Reveal About the Mind*. Chicago, IL: University of Chicago Press, 1987.

Lakoff, G., and Johnson, M. *Metaphors We Live By*. Chicago: University of Chicago Press, 1980.

Langacker, Ronald W. *Foundations of Cognitive Grammar*. Vol. 1 Theoretical Prerequisites. Stanford, Cal.: Stanford University Press, 1987.

Liskov, Barbara, and Guttag, John. *Abstraction and Specification in Program Development*. Cambridge, MA: MIT Press, 1986.

Neisser, Ulric. *Cognition and reality*. San Francisco: W. H. Freeman & Co., 1976.

Shaw, M. "The impact of modelling and abstraction concerns on modern programming languages." In *On Conceptual Modelling, Perspectives from Artificial Intelligence, Databases, and Programming Languages*, ed. Brodie, M., Mylopolous, J., and Schmidt, J. W. New York: Springer Verlag,, 1984.

Snodgrass, R.T. "Temporal Databases." In *Theories and Methods of Spatio-Temporal Reasoning in Geographic Space*, ed. Frank, A.U., Campari, I., and Formentini, U. 22-64. 639. Heidelberg-Berlin: Springer-Verlag, 1992.

Watzlawick, Paul. *Wie Wirklich ist die Wirklichkeit*. 1995 ed., Vol. 174. München: Piper, 1976.

Worboys, Michael. "Unifying the Spatial and Temporal Components of Geographical Information." In *Six International Symposium on Spatial Data Handling in Edinburgh*, edited by Waugh, T. C., and Healey, R. G., AGI, 505 - 517, 1994.